

程序员指南

硬件接口开发系列

The  
PROGRAMMER'S  
GUIDE TO SCSI

SCSI

程序员指南

【美】 Brian Sawert 著  
袁潮 韩永彬 译

ADDISON-WESLEY



中国电力出版社  
[www.cepp.com.cn](http://www.cepp.com.cn)

电力出版社

硬件接口开发系列

The Programmer's Guide To SCSI

# SCSI 程序员指南

【美】 Brian Sawert 著  
袁潮 韩永彬 译

中国电力出版社



北航 C0531762

## 内 容 提 要

SCSI(小型计算机系统接口)作为高性能计算机外围设备的接口,已经被人们广泛接受并应用到各种规模的计算机设备中。

本书对 SCSI 的介绍偏重于软件开发方面。在介绍了 SCSI 的基本概念后,介绍了 SCSI 编程的程序化方法,并在 DOS 和 Windows 下研究了 ASPI(高级 SCSI 编程接口),在 Windows 和 Windows NT 下研究了 ASPI32 的扩展,在介绍 SCSI 在 UNIX 平台的应用时,把重点放在了 Linux 平台上。

本书结构严谨、条理清晰、语言简洁、内容丰富,是那些为 SCSI 外围设备编写支持软件的程序员的一本指导书和参考书,也适合作为大中专院校教材。

JS474/40 08

## 图书在版编目(CIP)数据

SCSI 程序员指南/(美)萨维特著;袁潮,韩永彬译.—北京:中国电力出版社,2000.12

ISBN 7-5083-0478-0

I.S… II.①萨…②袁…③韩… III.电子计算机—接口—指南  
IV.TP334.7

中国版本图书馆 CIP 数据核字 (2000) 第 75027 号

中国电力出版社出版、发行

(北京三里河路 6 号 100044 <http://www.infopower.com.cn>)

三河市实验小学印刷厂印刷

各地新华书店经售

\*

2001 年 3 月第一版 2001 年 3 月北京第一次印刷  
787 毫米×1092 毫米 16 开本 15.75 印张 349 千字  
定价 35.00 元

版 权 所 有 翻 印 必 究

(本书如有印装质量问题,我社发行部负责退换)

# 前　　言

自从 SCSI（小型计算机系统接口）首次出现以来，在短短的几年中，作为高性能计算机外围设备的接口，它已经被人们所广泛接受了。SCSI 设备曾经只局限于大型机和高档工作站上使用，但是现在它已经被运行于个人电脑上的大多数操作系统所支持。

SCSI 接口受到如此广泛的支持的原因之一是它是为许多种设备类型所设计的高性能接口。磁盘驱动器、光盘驱动器、磁带驱动器、扫描仪、打印机等设备都配备了 SCSI 接口。另外，计算机的速度越来越快，所以要求外围设备的速度也更快，而 SCSI 技术正好为此提供了解决办法。

尽管 SCSI 已经受到了设备制造商和最终用户的欢迎，但是目前关于 SCSI 设备编程方面的书籍仍然很少。本书是从一个程序员的视角来介绍 SCSI 的，希望它能够填补这方面的空白。

## □ 读者对象

本书试图成为那些为 SCSI 外围设备编写支持软件的程序员的一本指导书和参考书。不管你是在为 SCSI 设备驱动编写低层代码还是为某个应用程序编写高层代码，你都能在本书中找到有用的信息。

也许你已经仔细阅读了 ANSI 规范说明文档的每个细节，也许你正在设法阅读别的程序员编写的源代码，毫无疑问，通过试验和在试验中出现的错误来学习 SCSI 编程基础会遭到无数次的失败。我们的目的是在你学习的道路上树立起一些路标，以使你能够沿着正确的方向前进，使你在学习这一复杂但令人着迷的技术的过程中避开弯路，踏上捷径。

本书对 SCSI 的介绍偏重于软件开发方面。关于信号特征、时序协议和硬件细节的信息只有当它们直接和编程任务有关的时候才出现。我们假设在读本书的时候，你已经有了一些计算机方面的知识，而且你已经有了一些使用 C、C++、汇编语言进行编程的经历。

## □ 本书的组织方式

本书的开始部分是对 SCSI 的一个概述。在介绍完 SCSI 的标准之后，我们将介绍它的设计原则以及它是如何发展演化，并把一些新的特性和功能加入进去的。我们还将谈到 SCSI 标准的某些变体，它们提供了更快的传输速率、更宽的数据通道和其他一些特性。

任何一个致力于 SCSI 方面工作的人都必须明白一些基本概念。SCSI 设备如何进行通信？命令如何执行？数据如何传输？驱动器（initiator）和目标器(target)分别担当什么角

色？我们将通过对 SCSI 事务处理的某些描述来回答这些问题。以上这些为进一步详细讨论 SCSI 事务处理的元素提供了基础。

接着，我们介绍了 SCSI 编程的一个层次化的方法，它从高层编程接口开始。我们将在 DOS 和 Windows 下研究 ASPI（高级 SCSI 编程接口），在 Windows95 和 Windows NT 环境下研究 ASPI32 扩展。

Windows NT 提供了它自己内建的 SCSI 支持。我们将通过分析 Windows NT 的设备模型来研究它是如何工作的，并研究 ASPI 层是如何使用它的。

然后，当我们考虑用普通的 SCSI 处理器和类似于 Symbios Logic's SCRIPTS 这样的脚本语言进行低层编程的时候，我们将涉及到更多的高级内容。我们将演示对启动器和目标器的操作。

SCSI 在 UNIX 系统下得到了广泛的支持。但不幸的是，在不同的 UNIX 系统下对 SCSI 支持的详细说明书是迥然不同的。所以 UNIX 一章强调了 SCSI 在不同的 UNIX 系统下的支持，然后就把注意力集中在 Linux 系统上。因为 Linux 平台目前应用得很广，而且它的开放性允许在它上面进行试验。

最后，我们开发了一个 SCSI 类库，而且使用它开发了一个 Windows 环境下的应用程序实例。我们鼓励你为自己的项目使用和扩展这个类库。

最后一章为你提供了故障查找和调试方面的建议。附录 B 提供了 SCSI 各种形式的资源，包括各种书籍和许多网址。

## □ 你需要什么

本书中例程的代码是可移植的。我们的大多数代码是用 Microsoft Visual C++ 开发的。SCRIPTS 例程的代码要用 Borland's C++ 和 Turbo Assembler 来编译。每一个例程都无需你做许多的工作。应用程序例子用的是微软的基础类库（Microsoft's Foundation Classes library），它也适用于其他编译器。Linux 代码用的是操作系统自带的编译器。

我们推荐使用一个 Iomega Zip 驱动器作为例程的试验设备。某些例子演示了磁盘驱动器的操作。当你做试验的时候，在你的系统磁盘上最好有一些东西协同工作。

Zip 驱动器以 SCSI 或并行端口的形式出现。使用并行端口的 Zip 驱动器使用 ASPI 兼容的驱动程序。

低层代码使用 Symbios Logic SCRIPTS 编译器，而且主机适配器配备了 53C8XX 系列的处理器。这个编译器可以从 Symbios Logic FTP 站点获得。这些代码应该是和该系列的其他适配器兼容的。如果你的适配器用的是另外一个厂家的处理器，你就不能使用该代码。如果你想和 Symbios Logic 公司联系，请参考附录 B 中的制造商列表。

ASPI 代码几乎适用于所有的主机适配器，只要该主机适配器使用 ASPI 兼容的驱动程序。如果要进行更高级的开发工作，你可以从 Adaptec 公司购买 ASPI 开发人员工具包。若想和 Adaptec 公司联系，请参考附录 B 中的制造商列表。

如果你想用 SCSI 进行一些重要的工作，你就需要一份 ANSI SCSI-2 规范说明文档。尽管我们在本书中的讨论涵盖了 SCSI 的主要部分，并对一些编程的问题进行了深入的探讨，但是本书还是不能覆盖规范说明文档中的全部细节。本书是对 ANSI 文档的一个补充，而 ANSI 文档则可以从 *Gloable Engineering Documents* 中得到。这些文档已经在附录 B 中列了出来。

## □ 致谢

本书的完成是和很多人的贡献分不开的。Addison-Wesley 出版社的前任编辑 Kathleen Tibbetts 一开始就坚信本书的成功，这给了我莫大的鼓励。

Earle Associates 的 Pamela Thompson 和在 Symbios Logic 的 Lauren Unddenberg 慷慨地提供了关于 Symbios 产品的支持程序和信息。

感谢 Adaptec 公司的 Mike Berhan 和 Dan Polfer，他们审阅了和 ASPI 相关的内容。还要感谢 T10 技术委员会的主席 John Lohmeyer，他对 SCSI 规范说明部分的内容进行了审阅和注释。

特别要感谢的是本书的另外两个作者 Larry Martin 和 Gary Field。Larry 在关于 ASPI、Windows 设备支持和 SCSI 目标器模式的章节中提供了他宝贵的编程经验。Gary 负责维护 comp.periph.scsi 新闻组的 SCSI FAQ 部分，并为关于 SCSI 设备的 UNIX 支持的章节提供了大量的资料。

最重要的是感谢我的妻子，在本书的写作过程中我得到了她的鼓励和支持，她使我获得了很多灵感。

# 作 者

## □ Brian Sawert

Brian Sawert 在 Northern Arizona University 获得物理学位。他致力于小型计算机接口的研究已有多年了，专门为 SCSI 设备开发应用程序和驱动程序。这些 SCSI 设备范围广泛，包括光盘驱动器和扫描仪等。他在 Dr. Dobb's Journal 和 DOS Developer's Journal 上发表了多篇论文，其中一篇标题为“高级 SCSI 编程接口”的论文研究了用 ASPI 进行 SCSI 编程的问题。

在生活中，Brian 喜欢骑车、收集 Jan 和 Dean 的磁带、陪他的妻子和他们的两只哈叭狗 Poco 和 Rocky。

## □ Larry Martin

Larry Martin 在 1982 年拥有了他自己的第一台 IBM PC，从那时起，他就开始编程。他用这台机器给当地的企业编写软件，以此挣钱完成了他的大学学业。从那时开始，他就已经把精力集中在硬件和软件的接口问题上，特别是在嵌入式系统中的软硬件接口。Larry 自 80 年代后期工作于 Flagstaff Engineering 公司后，就致力于 SCSI 接口的研究。他主要为适配卡、扫描仪、磁盘驱动器和磁带驱动器编写驱动程序，也为不同的 SCSI 外围设备编写目标器模式的代码，甚至还写了一些 ASPI 兼容的驱动程序，用非 SCSI 设备来模拟一些流行的 SCSI 设备。Larry 目前正致力于新兴的 IEEE 1394 “FireWire” 串行总线接口的研究，而且他正和 3A International 合作，开发 1394 测试设备，该设备对于程序员是很有用的。

Larry 爱好滑雪、潜水，当他咒骂不精确的数据表时，他的脸就会涨得通红。

## □ Garry Field

Garry Field 在 Northeastern University 获得计算机科学的学位。他从 1978 年开始致力于设备层软件的研究。1985 年，他在 Wang Laboratories 开始接触 DOS 环境下的 SCSI。后来，他领导了针对几个 UNIX 平台的一个 ANSI CAM 子系统的开发工作。另外，他负责维护新闻组网络系统 comp.periph.scsi 的 FAQ 部分已经有好几年了。自从 1996 年以来，Garry 一直在数字设备公司 (DEC) 的 UNIX I/O 开发小组工作。

在家里时，他是一个侦察技术方面的爱好者。在业余时间，他喜欢搞无线电、电子设备的修理、摄影，以及和妻子一起野营、划船、钓鱼等。

## 绪 论

对 SCSI 外围设备进行编程不仅是一门科学，而且还是一门艺术。很多编程细节是很模糊的，在标准文档上也找不到，这就迫使新手向其他的程序员学习，很多细节性的东西就是这样通过口头相传的，例如如何处理一个特殊的消息？为什么一些命令不按照你所希望的那样去工作？这时候，只有一个在 SCSI 编程方面积累了无数经验的高手才能给你的这些问题提供答案。

对那些以前从来没有接触过 SCSI 的人来说，我们希望本书能为他们提供以上所说的帮助；对那些在 SCSI 设备编程方面已经有经验的人来说，也可以从本书中找到新的信息或不同的侧重点，也许它们将使你在以后的编程工作上取得更高的效率。

新手可能希望能够较为容易地看懂本书的内容，希望在开始的时候介绍一下 SCSI 的规范说明书，并且对 SCSI 的工作原理做一个概括性的简单介绍。如果你已经对 SCSI 很熟悉了，并且对以上这些内容的介绍不感兴趣，就请跳过前面的几章，直接到专题章节中去，像 ASPI、SCRIPTS 编程、在 UNIX 环境下的 SCSI 等章节。本书既可以作为教材，也可以作为工作参考手册。

如果你对 SCSI 产生了兴趣，你就可以去研究列于附录 A 和附录 B 中的其他资源。随着 SCSI 的日益普及，你每天都可以获得更多有关 SCSI 的信息。

祝学习愉快。

# 目 录

前言	
作者	
绪论	
<b>第一章 SCSI 技术概述</b>	<b>1</b>
SCSI 的解决方案	1
SCSI-1	1
SCSI-1 的特性	3
SCSI-2 的诞生	4
SCSI-2 的新特性	4
改进的 SCSI-1 特性	6
正在升起的 SCSI-3	6
<b>第二章 SCSI 基 础</b>	<b>11</b>
SCSI 事务处理	11
<b>第三章 SCSI 阶 段</b>	<b>13</b>
SCSI 阶段	13
<b>第四章 SCSI 消 息</b>	<b>18</b>
消息类型	18
扩展消息	19
其他通用消息	21
<b>第五章 SCSI 命 令</b>	<b>24</b>
命令结构	24
字节顺序	25
强制 SCSI 命令	26
可选命令	30
设备类型特定命令	30
读和写	35
其他命令	36
<b>第六章 状态、检测和错误</b>	<b>37</b>
状态码	37
检测数据	38

<b>第七章 ASPI.....</b>	42
什么是 ASPI.....	42
为什么要用 ASPI.....	43
ASPI 概念.....	43
ASPI 命令.....	54
ASPI 错误和状态码 .....	72
附加的 Win32 下的 ASPI 功能.....	79
<b>第八章 用 SCRIPTS 进行低级 SCSI 编程.....</b>	81
使用 SCRIPTS.....	81
SCRIPTS 概览.....	82
在 C 代码中嵌入 SCRIPTS.....	84
例程 .....	95
<b>第九章 SCSI 目标模式编程 .....</b>	101
硬件 .....	101
处理 SCSI 阶段 .....	102
目标模式 API.....	109
连接 TSPI 控制器 .....	124
使用 TSPI 接口 .....	126
<b>第十章 Windows 环境下的 SCSI.....</b>	132
Windows 3.x 下的 ASPI.....	132
Win 32 (Windows 95 和 NT) 下的 ASPI.....	133
Windows 95 和 NT SCSI 模型 .....	133
Windows NT 中的 SCSI Pass-Through Interface .....	135
<b>第十一章 Unix 环境下的 SCSI .....</b>	144
UNIX 设备驱动程序的简单描述 .....	145
UNIX 形式比较.....	147
Linux SCSI 磁盘驱动器.....	154
Linux SCSI Pass-Through 驱动程序.....	176
总结 .....	192
感谢 .....	193

<b>第十二章 常用问题及解决方法</b>	194
从硬件层开始	194
慎用电缆	195
不要简单地对待说明书	195
要注意平台的依赖性	196
测试工具	197
记录	198
<b>第十三章 应用举例：SCSI Snooper</b>	199
SCSI Snooper 的概观	199
ASPI 类库	203
使用 ASPI 类库	217
<b>附录 A 缩写词汇表</b>	222
通用词汇	222
SCSI-2 定义	222
SCSI-3 定义	223
SCSI 软件接口	224
<b>附录 B SCSI 资源</b>	225
书籍	225
杂志和日报	226
在线信息	226
制造商联系信息	229
<b>附录 C 在 Windows NT 环境下安装 ASPI32 服务</b>	230
<b>附录 D 本书附带光盘的内容</b>	232
例程代码	232
SCSI 规范说明书	233
SCSI 常见问题解答	233
Symbios SCRIPTS 支持	233
Linux SCSI 文献	233

# 第一章 SCSI 技术概述

在 SCSI 出现以前，计算机外围设备领域是十分混乱的，特别是小型计算机上装备的磁盘驱动器、打印机和其他外围设备所采用的接口和通信协议的分类十分混乱。ST506 和 ESDI 接口都在争夺磁盘驱动器的市场，希望自己能获得主导地位。并行接口和串行接口拥有各自的标准，引出了广泛的兼容性问题。对软件开发者来说在小型计算机市场上，每引入一种新的设备，都意味着要做一个新的软件来支持它，工作量很大。

## ■ SCSI 的解决方案

小型计算机系统接口（SCSI）就是试图为计算机的外围设备建立一个标准的接口和通信协议。SCSI 定义了电缆敷设要求、电气信号标准、一个事务处理协议和一个通用命令集。提出 SCSI 的目的是为了实现一种设备接口，它能够适用于多种外围设备，这些外围设备包括从磁盘驱动器、磁带驱动器等存储设备到打印机之类的输出设备。SCSI 规范说明书包含了该总线上的各种设备，而且提供了对它们在不同的硬件平台和操作系统上的设备兼容性的展望。

实际上，早期的 SCSI 设备并没有达到这种理想情况。在规范说明书中有很多定义得比较含糊或者松散的地方，不同的制造商对这些地方有不同的解释。所以早期的 SCSI 设备只能够和某些特定的主机适配器协同工作。把不同类型的设备放到同样的总线上的做法仅仅适用于那些最可靠的系统集成器，而且还需要大量的时间和专门的技术来使他们协同工作。

## ■ SCSI-1

SCSI 也被称为 SCSI-1，它首先于 1986 年被定义在 ANSI 规范说明书 X3.131-1986 中。这份 200 多页的说明文档概述了在几个层上的新的接口。它提出了电缆敷设要求、连接器、信号电压的电气要求、时序以及在物理传输层的总线终端。除了以上这些对物理接口的规范说明之外，它还对 SCSI 设备采用的通信协议作了概述。最后，它还定义了一套强制实现和可选实现的设备命令。

图 1.1 是对小型计算机系统接口的一个图解表示。

SCSI 规范说明书并没有定义一个编程接口。后来，出现了软件层的标准，例如 Adaptec 的高级 SCSI 编程接口（ASPI）和通用存取方法（CAM），它们填补了这一空白（在图 1.1

中这一层用虚线边框画出)。许多实际的标准其实是市场选择的结果。CAM 已经随着 SCSI-3 一起更新了,但是人们往往忽略这一点,因为在 PC 市场上大家更加偏爱 ASPI(高级 SCSI 编程接口)。但 CAM 仍然在许多 UNIX 操作系统上得到支持。



图 1-1 小型计算机接口

在实现 SCSI 的过程中有很多问题,它们集中在接口的物理特性和电气特性上,像电缆长度和信号端子这样的问题是常见的。不同的制造商在相同的连接器上使用不同的管脚引出线。例如,Future Domain 和 Apple 计算机都使用 DB25 连接器来布线。但是因为他们的布线方式不同,所以如果布线不匹配的话,将有可能损坏 SCSI 设备。

SCSI-1 的规范说明书在这些层的定义上并不是很明确。例如,它虽然定义了终端负载的电源特性,但是并未指出哪些设备需要提供这个特性。由此产生的后果就是:某些 SCSI 主机适配器提供一个 TERMPWR 信号,但另外一些则不提供这一信号。当主机适配器和附属设备都不提供终端负载电源的时候,就会出现问题。

总线终端负载是一个经常被误解的功能部件。这可能是因为规范说明书的说明较为含糊,也可能是读者的误解。不管是出于哪一种原因,最普遍的问题常常集中在 SCSI 安装中的故障查找和处理 SCSI 链上的设备对终端负载的支持上。

在命令层,SCSI-1 也有很多没作解释的东西。它定义了一个最小的命令集,声明了一些强制实现的、可选的、和厂商自定的命令。强制实现的命令要处理设备辨认、状态报告、错误报告、以及错误恢复。其他的命令基本上都属于可选的或厂商自定义的命令一类。虽然对制造商来说不同的设备类型缺乏标准命令集这一情况有利于他们在自己的设备上实现 SCSI,但是这对软件开发人员来说却意味着大麻烦。如果一个程序员想在一个应用程序中支持 SCSI 扫描仪,那么他就必须知道每个扫描仪的命令集;如果他想使用任何一条可选命令,他就必须知道哪些设备支持这条命令。SCSI 简化了硬件设计的工作,但是程

程序员还不能从这个标准设备接口中获益。

缺乏一个标准命令集并不是 ANSI X3T9.2 委员会方面的疏忽所至。在这个规范说明书出现之前，SCSI 的外围设备出现在市场上已经好几年了。这就为完整地宣布这个标准制造了压力，但是这不会在未来的修订中造成阻碍。一个过渡期工作小组开发了一个通用命令集 (CCS)，但是它仅仅面向磁盘驱动器。很多其他类型 SCSI 设备的命令集在 SCSI-2 的规范说明书中才出现。

## SCSI-1 的特性

尽管 SCSI-1 有它的缺点，但是它还是为小型计算机带来了从其他接口所不能获得的特性。

### 智能设备

在 SCSI 模型中，外围设备使用智能型的板载控制器。这就把命令处理的任务从系统处理器转移到了设备自身。每个设备都要负责错误报告和错误恢复。直接存取设备（例如磁盘驱动器）的命令使用逻辑块寻址。设备自己负责把逻辑地址映射为物理地址，补偿缺陷扇区和异常的几何结构。在 DOS 下把它和老式的 ST506 接口作比较，ST506 强迫操作系统维护有坏扇区的磁道，而且它仅能支持对磁盘驱动器的有限的配置。

### 多任务 I/O

SCSI-1 在它对多任务 I/O 的支持上为小型计算机提供了一些新的东西。例如，断开连接/重新连接的特性对磁盘驱动器的操作来说就特别有效，它使设备能够在长时间的操作过程中从总线上断开，在完成该操作之后再进行重新连接。当磁盘驱动器收到一个读请求的时候，在寻址和读取数据的过程中，它可以从总线上断开。在此过程中，总线是空闲的，可以被其他设备使用或用于其他操作。当设备已经为传输数据作好准备时，它就重新连接启动器，并完成这个操作。在多任务操作系统中，该特性能够极大地提高吞吐量。

### 同步数据传输

在 SCSI-1 下，默认的数据传输方式是异步传输，它靠握手（handshaking）的方式来确认每个字节的传输。在异步传输模式下，8 位宽的 SCSI 总线上的传输速率约为 1.5MB/s。SCSI-1 定义了一种同步传输选项，它可以使传输速率激增至 5MB/s。也许从今天的标准来看，它显得有些慢，但是它代表了一种超越当时主流技术的技术进步。

## ■ 多种设备类型使用一种接口

SCSI-1 定义的接口在链式配置下可以最多支持 8 个设备。系统设计者可以用 SCSI-1 在一个主机适配器上最多连接 7 个外围设备，而且，这些外围设备并不局限于一种设备类型。SCSI-1 的一个独有的特性是它能够用一个通用接口支持磁盘驱动器、磁带驱动器、扫描仪和其他设备。实际上，由于设备兼容性和其他因素在 SCSI-1 下要达到这个目标是很困难的。

## ■ SCSI-2 的诞生

像 SCSI-1 一样，SCSI-2 也有很多的创新，但也有其局限性。SCSI-2 标准允许在奇偶校验、消息处理、命令处理上有些例外。利用这些例外的制造商经常会发现其产品和别人的不兼容。

在计算机工业的标准中，速度是一个很重要的因素。当工业界在个人计算机上采用 16 位的体系结构时，原来的 8 位标准看上去就有些过时了。在使用了更新的体系结构后，处理器的性能提高了，以至于 5MB/s 的同步传输速率甚至也不能跟上新的处理器的速度。

在 SCSI-1 成为官方认可的正式标准之前，在市场上人们已经对产品提出了进一步的要求，并且部分已经被实现了。这些技术进步被放在标准中，并且成为 SCSI-2 标准的一部分。在一个经过官方正式认可的 SCSI-2 标准出现之前，人们在购买设备时注意发票上一定要写着“SCSI-2 兼容”，因为这一点是很重要的。很多不兼容的原因是因为设备制造商使用了一些标准草案中的特性，而这些特性最后没有被批准。当许多人使用新的标准的时候，他们会发现这一点。

## ■ SCSI-2 的新特性

SCSI-2 的新规范主要定位在一些 SCSI-1 的缺点上，它提供了一些改进，并且包含了一些新的特性。这些特性包括以下几个方面。

### ■ Fast SCSI

SCSI-2 用一个可选的快速传输率来提高异步传输速度。这项被称为 Fast SCSI（快速 SCSI）的特性把最大异步数据传输速率提高到 10MB/s。但是同时它也对时序提出了更加严格的要求，从而 Fast SCSI 对电缆和电气特性要求也就更高了。

从第一份 SCSI 规范说明书开始，就有两种可选的配线方案。单端的配线方案适用于短程线路，它的最大电缆长度为 6 米。差分 SCSI 则是为远距离通信设计的，它的最大电

缆长度为 25 米。一般情况下，差分配线方案能够提供更好的信号完整性和时序，对快速传输来说，它是一个较好的选择。即便如此单端 SCSI 设备仍然比差分 SCSI 设备用得更为广泛。

### ■ Wide SCSI

SCSI-2 引入了 16 位和 32 位宽的 SCSI 数据总线选项。Wide SCSI（宽带 SCSI）突破了 SCSI-1 的单字节传输限制，使传输速率变为原来的 2 到 4 倍。使用 Wide SCSI 选项，必须定义新的电缆才能提供这种功能。老式的 SCSI-1 电缆指定使用 A 电缆，在 SCSI-2 中附加了一条 B 电缆，作为附加的信号线，从而把数据总线的宽度变为 16 位或者 32 位。

增加数据线的结果是可以对更多的设备进行编址。从原理上来说，16 位的总线可以支持 16 个设备，32 位的总线可以支持 32 个设备。然而，电气因素和其他因素限制了一条总线实际可以支持的设备数目。实际上，只有 16 位的 Wide SCSI 产品已经进入了市场，很少有制造商做出 32 位的 Wide SCSI 产品。这项产品的空白使得我们可以假定 Wide SCSI 这一术语一般指的是它的 16 位版本。

### ■ Fast Wide SCSI

当我们把 Fast SCSI 和 Wide SCSI 结合在一起的时候，将会发生什么事呢？SCSI-2 的规范说明书定义了一个 Fast Wide SCSI 高速宽带选项。使传输速率飚升至 20MB/s 或 40MB/s，具体使用哪种传输速率取决于总线宽度。它对电缆的要求和 Fast SCSI 一样。

和 Wide SCSI 的情况一样，市场上有很多 16 位的 Fast Wide SCSI 产品，32 位的 Fast Wide SCSI 更多的只是作为书面标准而不是实际的产品出现。

### ■ 排队 I/O 进程

SCSI-2 为无标签的和有标签的两种类型的 I/O 进程队列增加了一个选项。无标签队列允许设备在从一个启动器 (initiator) 那里执行 I/O 进程的时候，从另一个启动器处接收命令。标签队列使目标器能够从同一个或者不同的启动器那里接收一系列的 I/O 进程。它们的执行顺序是由一个队列管理算法而来的，或者是按照一个指定的顺序。

排队和命令连接是不同的，命令连接是 SCSI-1 所支持的。在一个单独的 I/O 操作中使用命令连接，可以使几个命令按照一定的顺序执行。对一个磁盘驱动器来说，一个连接的命令可能包含一个 seek 命令，后接一个 read 命令。连接的命令在一个 I/O 进程队列中被作为一个整体存储。

### ■ 新的命令集

SCSI-2 为没有包含在原始规范说明书中的设备定义了命令集。对 CD-ROM 驱动器、

扫描仪、光驱、介质转换器和通信设备的支持都出现在 SCSI-2 的文档中。所有的命令集都被扩展和增强，反映了人们对更复杂特性的要求。

## 改进的 SCSI-1 特性

SCSI-1 几年来的实践，除了显示出它的很多优点之外，也逐渐暴露出了一些缺点。在这些实践经验的指导下，ANSI X3T9.2 任务小组在原来的标准中提出了一些改进方案。几个新的低层的要求就出现了。

### 数据奇偶校验

SCSI-2 要求使用数据奇偶校验，但在 SCSI-1 中它是可选的。

### 要求支持消息

SCSI-2 要求支持消息作为事务处理协议的一部分。在 SCSI-1 中，它是可选的，但是在 SCSI-2 中它是高级特性，要求在启动器和目标器之间进行协商。

SCSI-2 使用 Identify 消息在目标器和启动器之间协商断开连接的权力。异步和宽数据传输、排队操作和其他特性也通过在启动器和目标器之间传递消息进行协商。

### 启动器为终端负载供电

SCSI-2 解决了终端负载的供电问题，它强制要求启动器为终端负载供电。它也概述了一种活动终端负载的方法，活动终端负载比通常使用的无源终端负载更为有效。

总的来说，新的 SCSI 规范说明书弥补了原来的漏洞，加强了更为严格的兼容性需求，增加了对新的设备和特性支持。

## 正在升起的 SCSI-3

SCSI 规范说明书不是固定不变的，它在不断被修改以适应计算机外围设备的更新和更快的速度需求。随着 SCSI-2 在 1994 年最终完成并成为 ANSI 文档 X3.131-1994，关于它的下一个版本的修订工作也就马上开始了。虽然整个 SCSI-3 的规范说明书还没有被通过，但是 SCSI-3 的一些特性以及与 SCSI-3 兼容的硬件已经在市场上出现了。

就像从 SCSI-1 到 SCSI-2 一样，SCSI-2 和 SCSI-3 之间的划分也并不是很明显，SCSI-3 的一些特征是 SCSI-2 没有及时出现而成为老的标准的一部分的那些特性的扩展。其他特性被标记为 SCSI 体系结构的新方法，这就保证了这种接口在未来的几年内仍然将富有活力。改进再一次把焦点集中在提高传输速率上，同时还扩展了 SCSI 的体系结构。就像它

的老版本一样，SCSI-3 和它的老版本之间保持硬件和软件的兼容性。

在 SCSI-2 下，电缆是一个容易令人混淆的问题。宽数据通道靠增加一条附加的数据电缆提供，它为 16 位和 32 位的传输提供额外的信号线。SCSI-3 为 Wide SCSI 的实现定义了一条 16 位的电缆。这些非正式的特性已经被许多外围设备厂商所采用，并且比正式的 SCSI-2 配置的使用要普遍得多。

## ■ Fast-20 和 Fast-40 SCSI

SCSI-3 提供了新的 Fast SCSI（快速 SCSI）协议。Fast-20 也被称为 Ultra SCSI，在 8 位数据传输宽度的条件下，把同步传输速度提高到了 20MB/s，是 SCSI-2 在相同传输条件下的 2 倍。Fast-40 也被称为 Ultra2 SCSI，它把同步传输速度提高到了 40MB/s，是 SCSI-2 的 4 倍。就像在 SCSI-2 中一样，这些快速协议的使用也可以和宽数据总线相结合。Wide Ultra SCSI 和 Wide Ultra2 SCSI 在 16 位传输宽度的条件下可以达到 40MB/s 和 80MB/s 的传输速率。如此高的传输速率对维护信号完整性的时序和电气特性提出了更高的需求标准。就象旧的 Fast SCSI 一样，它们使用差分 SCSI 的方案保证更可靠地工作。实际上，Fast-40 要求使用新的低电压差分（LVD）选项。现在的很多设备都提供对多种模式的支持，包括 LVD，但是要真正做到 Fast-40 的速度则要求所有的连接设备都支持 LVD。差分选项的另外一个重要优点是它可以支持更多的设备和更长的电缆连接。

表 1-1 对不同的并行 SCSI 做了一个比较。

表 1-1 并 行 S C S I 标 准

名 称	传 输 速 率	数 据 宽 度
异步 SCSI	1.5MB/s	8 位
同步 SCSI	5MB/s	8 位
Fast SCSI	10MB/s	8 位
Wide-16 SCSI	10MB/s	16 位
Wide-32 SCSI	20MB/s	32 位
Fast Wide-16 SCSI	20MB/s	16 位
Fast Wide-32 SCSI	40MB/s	32 位
Fast-20 ( Ultra SCSI )	20MB/s	8 位
Fast-40 ( Ultra2 SCSI )	40MB/s	8 位
Wide Ultra SCSI	40MB/s	16 位
Wide Ultra2 SCSI	80MB/s	16 位

## ■ 串行 SCSI 标准

尽管 Wide Ultra SCSI 提供了 80MB/s 的传输速率，是旧标准所提供的传输速率的 8 倍，但是当它和 SCSI-3 中定义的一套新的标准所宣布的传输速率相比就黯然失色了。总

的来说，它们被称为串行 SCSI 标准。与当前的 SCSI 标准不同的是，它不属于并行数据接口，这三个标准定义了 SCSI 在串行接口上的不同实现。

串行接口的几个优点是很明显的。并行数据传输方式使总线变得更宽，电缆变得更加复杂，也不易弯曲，一个 Wide SCSI 的安装需要一条 50 针和一条 68 针的电缆；而串行标准大大减少了所需的导线数。

串行数据传输还有一些其他优点。采用串行方式可以使用更长的连接电缆。对使用差分方式的并行 SCSI 连接来说，电缆长度限制大约为 25 米。而在串行标准下，若使用光纤作为传输介质的话，传输长度可以达到几公里。

数据的完整性也得到改进，并行接口依赖于奇偶校验和握手（handshaking）方式。而串行标准则都使用循环冗余校验（CRC）来保证可靠的数据传输。

然而，最大的改进还在传输速率上。在规范说明书的草案中提出了 100MB/s 的传输速率。若使用扩展，则传输速率可以达到 100MB/s 的 2 倍或 4 倍。此外若使用保留带宽的选项将可以进一步提高它的数据吞吐量。

串行 SCSI 标准由三个分立的标准组成，每一个都有自己的优点。它们是：光纤通道、串行存储体系结构和 P1394。

## □ 光纤通道

光纤通道（FC）标准是为灵活性而设计的。尽管它的名字叫光纤通道，但是它同时支持光纤介质和铜介质。可能的话，光纤通道 SCSI 能够使电缆与光纤局域网或双绞线共享。

协议也定义了不同的服务层。专用服务层为连接的 SCSI 设备保留连接的所有带宽。框架切换服务层提供了多路复用服务，它可以和其他协议共享。另有一层提供了多路复用服务，但不用确认数据包的接收。

光纤通道 SCSI 从数据网络上引入了很多的概念。最复杂的是，这个标准定义了一个光纤通道交换结构，它类似于一个交换网络，提供同步的活动连接。这种拓扑结构原理上支持几百万个设备的连接。一个被称为仲裁环的较为简单的拓扑结构允许最多 127 个设备彼此通信，但一次只能在两个设备之间通信。光纤通道提出了一个极快的传输速率的串行协议，它提供了每秒 100MB/s 或者更高的全双工传输速率。

## □ 串行标准体系结构

串行标准体系结构（SSA）源自 IBM 的串行 SCSI 标准。它采用的导线是双绞铜线电缆，但也可以用光纤。它的优点在于支持全双工通信和空间复用。

像光纤通道的交换式实现一样，SSA 允许在同一介质上同时进行多个连接。这项设计消除了在一个时刻只能有一个启动器/目标器可以占用数据总线的限制。

## □ P1394

P1394 因为它在苹果机上的实现而闻名，在苹果机上它叫“FireWire”。它的设计目的在于简单性和多媒体功能。它提供了一种等时传输模式，能够为对定时要求敏感的传输（例如视频和音频数据传输）保留带宽，以确保它们的传输。P1394 还没有对光介质传输做出定义。

## □ 分层体系结构

随着串行 SCSI 标准的加入，SCSI 规范说明书正变得越来越庞大而不可控制。在最后一次修订中，X3T10 委员会决定把该标准分为更小的单元。对其中某些单元的修订工作移交给了其他委员会，由此带来的好处是整个标准可以比以前更快地得到批准。

X3 委员会本身也经历了一些变化，它变成了信息技术国家委员会（NCITS）。随着它的改变，X3T10 委员会变成了 T10 技术委员会。从技术上说，NCITS 和它下属的委员会不属于 ANSI 组织，虽然他们开发 ANSI 公布的标准。

SCSI-3 标准采用了和用于网络数据传输的 OSI 模型相似的分层模型。这个结构有助于把接口的硬件和软件功能分离开来。命令集和编程接口在不同的系统结构上是很相似的，这就简化了程序员把程序从一个平台移植到另外一个平台的工作。

图 1-2 所示的就是 SCSI-3 标准的体系结构。

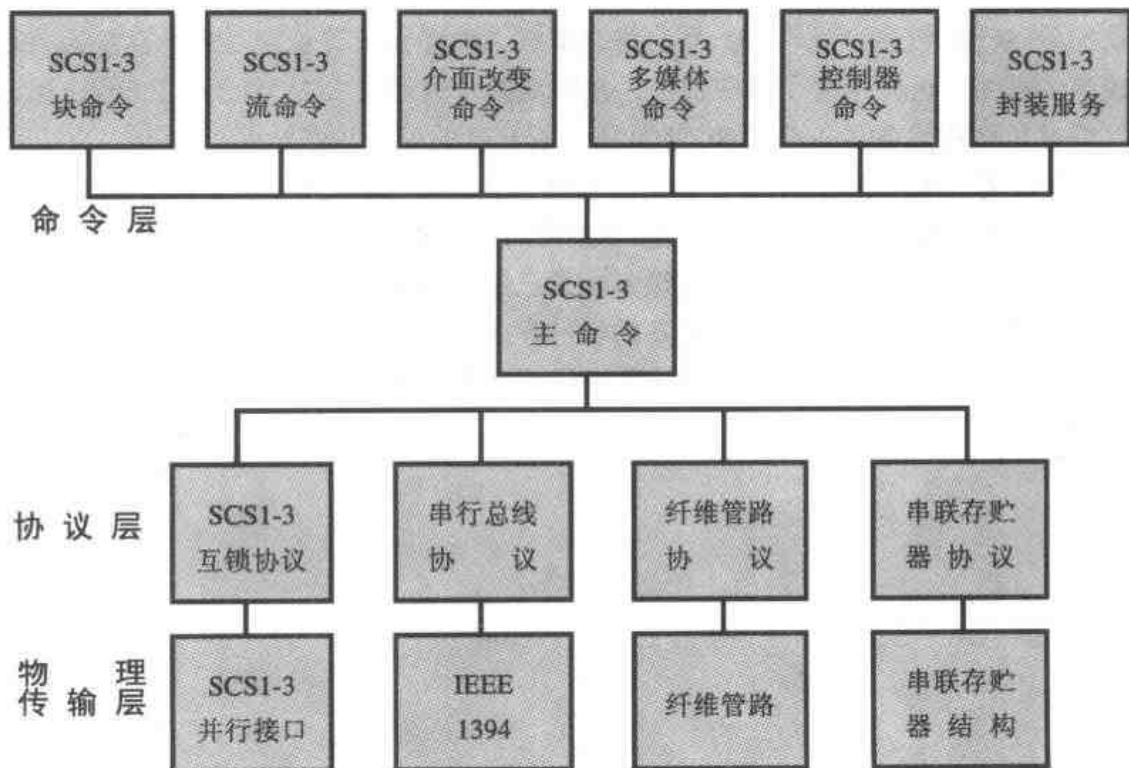


图 1-2 SCSI-3 层次化结构

该标准定义了三层。命令层包含了 SCSI 设备新的和扩展的命令集。协议层和物理层包含了一个并行协议，一个类似于 SCSI-2 并行模型的物理层和新的串行 SCSI 标准。

这个称为 SCSI 体系结构模型 (SAM) 的模型不仅定义了 SCSI 的物理实现，还定义了它们的传输机制和协议。对所有这些命令集、传输机制和协议，有一个缩写字母表来标识它们。完整列表见附录 A 的术语表。

对该模型的修订工作已经在进行了。SCSI-3 并行接口-2 (SPI-2) 将被加入并且取代原有的 SCSI-3 互锁协议 (SIP)、SCSI-3 并行接口 (SPI)、SCSI-3 Fast-20 和 Fast-40。串行总线协议将划分为串行总线协议-2 (SBP-2) 和 SCSI 经由 SBP-2 的传输协议 (STS)。这个变化允许非 SCSI 命令集在 IEEE1394 硬件上进行传输。

正如本章前面所提到的，CAM 提供了对命令集的访问，它是按照功能分组的。CAM 已经为 SCSI-3 做了更新，现在称为 CAM-3。但在个人计算机世界，它还没有得到广泛的支持。

## 口 即插即用 SCSI

由于 PC 体系结构和操作系统的进步，SCSI 工业转向外围设备的自我配置仅仅是一个时间上的问题。SCSI-3 标准草案的一个附录定义了 SCSI 自动配置 (SCAM)。主机适配器使用这个协议就可以自动地在系统启动的时候为 SCSI 设备动态分配 SCSI ID 号。但是它也能为老式的 SCSI 设备提供固定的地址。这项特性是即插即用 (PnP) SCSI 的基础。这项标准是由几家 SCSI 设备制造商和微软公司合作提出来的。

手动配置 SCSI 设备、处理 ID 号、终端负载、电缆要求等长期以来一直被认为是一项难度较高的技术。PnP 接管了这些责任，在启动的时候通过 SCAM 分配 ID 号，保证正确的电气终端负载和设备加入或从 SCSI 总线上断开时的信号时序。

PnP 标准定义了两层服务：第一层负责基本的即插即用操作，第二层支持对热插拔 SCSI 设备的多启动器和设备的 ID 分配。尽管操作系统对 SCAM (SCSI 自动配置) 的支持滞后，但是即插即用 SCSI 主机适配器和外围设备还是已经在市场上出现了。当然分配驱动器字母或者分配基于一个 SCSI ID 的启动设备的软件在改变它们地址的时候还是会遇到问题。但是即使这样，早期并不认同 SCAM 标准的微软公司也开始把对它的支持放到了其操作系统产品内。

## 第二章 SCSI 基 础

所有的事务处理，不管是在人与人之间，还是在计算机的外围设备之间，都依赖于一套规则或协议，以使事务处理更为有效和高效。人和人之间的相互作用有时是很混乱的，就像是在纽约证券交易所里繁忙的交易日的情景。但在另一个极端，例如在美国参议院内的争论则可以作为有序的相互作用的模型，当然要假设那年不是一个选举年。

它们之间有什么不同呢？对后者来说，有一套严格的规则来指导这些参与者。参与者要得到主席的认可，而主席保证他们按照一定的顺序发言。国会的议程和罗伯特排序规则为这些事务处理定义了协议，就像 SCSI 规范说明书为 SCSI 设备之间的的数据交换定义了协议一样。

也许这个类比有些牵强，但是它能够解释 SCSI 事务处理是按照定义好的步骤有序执行的。很多程序员在第一次看 SCSI 协议的时候会产生困惑，特别是当他们面对复杂的阶段图表的时候。我们将在后面讨论阶段图表。首先我们来看一下 SCSI 的基础部分。

### SCSI 事 务 处 理

当 SCSI 总线上没有活动时，就产生了总线空闲条件。作为启动器的 SCSI 设备可以通过一个称为仲裁的过程来要求总线控制权。它申明它的要求，并且检查是否有其他设备同时也在竞争总线控制权，如果参与竞争的设备的 SCSI ID 号比它大，那就意味着它的优先权不如别的设备高。所以 SCSI ID 小的设备只能等待下一次的申请。

一旦某个启动器获得了总线的控制权，它就会选择一个目标器进行事务处理。如果那个目标器存在，并且已经为接收命令做好了准备，它就会对启动器的选择做出确认。

在目标器和启动器之间已经建立了连接之后，目标器就通过指示事务处理的阶段来控制进程。在某些情况下，启动器可以请求一个特殊的阶段，但是由目标器来控制确定阶段的总线信号。

选择阶段之后是消息阶段。请求这个阶段是由启动器发出信号，然后目标器对之做出响应。在启动器和目标器之间传递的信号使设备之间能够彼此确认对方，并且为接下来的事务处理协商参数和程序规则。

消息阶段几乎可以在 SCSI 事务处理过程中的任何时候发生。协议使用消息来报告错误、命令状态和各种其他信息，也可以使用消息来发送控制信息。

在消息阶段之后，事务处理就进入了命令阶段，启动器发送一个带有命令指令和参数的数据块给目标器。如果目标器要报告命令块格式或参数的错误，那么事务处理就又进入了消息阶段。

执行一个命令通常要花费一定的时间。这可能是由于设备不得不对它自己的一些机械元件进行倒带或重新定位。如果启动器和目标器之间已经对断开连接的优先权进行过了协商，目标器就将断开连接，释放启动器用于其他操作。当目标器为下一步工作做好准备后，它就重新选择阶段再和启动器连接。重新选择阶段和连接阶段相似，但是目标器是活动元素，而不是启动器。

接下来，是否产生数据阶段取决于所发送的命令。命令决定了数据传输的方向，而不管启动器是发送命令还是接收命令。目标器为传输的正确方向设置 SCSI 总线。

在数据阶段之后也可能有消息阶段，但是状态阶段标志着命令执行的结束。状态代码表示了命令执行的结果。如果发生错误，状态代码就表示出扩展信息是否是可利用的，例如检测数据就是扩展信息。

一个正常的事务处理的最后阶段也是消息阶段。目标器给启动器发送一条消息，告诉它命令已经执行完了，并且发送一个状态代码。只要这个消息被发送出去了，目标器就释放数据总线，回到总线空闲状态。

以上是对事务处理的一个简单的描述。在下一章，我们将深入讨论这些阶段的细节。

## 第三章 SCSI 阶段

在一个 SCSI 事务处理的过程中，启动器和目标器将经历几个阶段。这些阶段如何编排，在它们之间又有哪些信号呢？

SCSI 的事务处理有点像人的求爱过程。当处于这种关系中的甲方准备好以后，就向乙方发出信号。值得庆幸的是，在 SCSI 事务处理中的阶段和信号比人类中的这种关系解释起来要容易。

在一个简单的 SCSI 事务处理中，启动器为取得总线的控制权而进行仲裁。一旦它获得了对总线的控制权，它就选择一个目标器进行连接。目标器做出响应，然后启动器和它进行消息交换，为即将到来的事务处理建立程序规则。断开连接的优先权、数据传输宽度、和同步传输的时序都通过消息来协商。当协商完成以后，启动器就可以给目标器发送命令了，然后就有可能发生数据传输。命令执行结束以后，目标器就给启动器发送一个状态码指示结果。在总线被释放以前，还可以进行一些其他消息发送。

这个过程按照一定的顺序发生，从一个阶段到另一个阶段。虽然由启动器发起这个过程，但是当前总线阶段却是由目标器来控制的。启动器可以要求把当前阶段改变到一个特殊的阶段，但是最终是由目标器决定从一个阶段改变到另外一个阶段。那么具体都有哪些阶段？它们又分别代表什么？

### SCSI 阶段

SCSI 协议定义了 8 个不同的阶段：

- ◆ 总线空闲阶段
- ◆ 仲裁阶段
- ◆ 选择阶段
- ◆ 重新选择阶段
- ◆ 命令阶段
- ◆ 数据阶段
- ◆ 状态阶段
- ◆ 消息阶段

这些阶段不一定按照以上列出的顺序发生。像以前提到过的一样，消息阶段几乎可以在任何时候发生。是否支持重新选择取决于目标器设备是否有断开连接的权力。数据阶段仅仅应用于传输数据的操作。

这些阶段中的每一个都被 SCSI 总线上的不同信号的结合和所交换的明确的数据类型

标识出来。在 SCSI 总线上的 6 个信号决定了它处于哪一个阶段，以及传输的数据的类型和方向。这些信号是：

- BSY (忙) 表示总线正在使用的信号
- SEL (选择) 表示选择一个目标器或者重选一个启动器的信号
- C/D (控制/数据) 表示控制或数据信息的信号
- I/O (输入/输出) 表示数据相对于启动器的传输方向的信号
- MSG (消息) 表示消息阶段的信号
- ATN (注意) 表示启动器用来请求消息阶段的信号

## ■ 总线空闲阶段

在总线空闲阶段，在总线上没有活动。没有 I/O 进程处于等待状态，没有设备提出使用总线的要求，总线随时可以被占用。上面列出的信号没有一个是活动的。

## ■ 仲裁阶段

在仲裁阶段，启动器为取得总线的控制权进行协商。它声明一个 BSY 信号，表示总线正在被使用，并驱动对应于它的 SCSI ID 的数据线。对一个 8 位的数据总线来说，数据线的编号是从 DB (0) 到 DB (7)。一个 ID 号为 7 的启动器声明 DB (7) 为真，以此提出使用总线的要求。

在一个规定的时间延迟后，启动器检查是否有另外一个设备正在试图获得总线控制权。它怎么知道这一点呢？如果和另外一个设备 ID 对应的数据线被声明为真，那么这个设备就在试图获取总线的控制权。最后，ID 号最大的设备在这次仲裁中获得胜利，得到总线的控制权。一般说来，SCSI 主机适配器被分配给较高的 ID 号，以保证它们在仲裁中获得许可。

一旦启动器获得总线的控制权，它就声明 SEL 信号来结束仲裁阶段，并进入选择阶段。

## ■ 选择阶段

在 BSY 信号仍然为真的时候，声明 SEL 信号就使启动器选择了一个目标器。然后它就声明对应于它自己的 SCSI ID 的数据线和对应于它所选择的目标器的 SCSI ID 的数据线。这里的 I/O 信号是反相的，以把这个阶段区别于重新选择阶段。

启动器也设置 ATN 信号为真来请求消息输出阶段，这个阶段在选择阶段之后。这一步在 SCSI-1 下是可选的，但是在 SCSI-2 下是强制实现的。

最后，启动器释放 BSY 信号。

对一个目标器来说，当 SEL 信号和它的 ID 位都为真，并且 BSY 和 I/O 信号都为假的时候，它就可以断定自己被选中了。它用在一个指定的时间段内设置 BSY 信号为真来响应。

启动器通过释放 SEL 信号来确认它的选择。

## □ 重新选择阶段

当目标器正在执行命令的时候，如果它从总线上断开，就会切换到重新选择阶段来重新建立连接。这个阶段和选择阶段是镜像对应的关系，但是在仲裁和设置总线信号的过程中，目标器是主动方，而不是启动器。在重新选择阶段目标器也声明 I/O 信号和 SEL 信号。启动器用声明 BSY 信号的办法来对重新选择做出响应。一旦目标器声明 BSY 信号和释放 SEL 线来做出响应，启动器就释放 BSY 信号，连接继续进行。

## □ 消息输出阶段

因为启动器在选择阶段设置了 ATN 信号，目标器接下来就进入了消息输出阶段。它依靠声明 MSG 和 C/D 信号来做到这一点。在 SCSI-2 中，在设备选择阶段之后必须跟随消息输出阶段。

在初始的消息输出阶段，只有一小部分消息是有效的。在正常的情况下，启动器将会发送一条 Identify 消息。这条消息在启动器和目标器设备内的逻辑单元之间建立起了连接。这个连接被称为一个 I\_T\_L 连接。对于支持目标器例行程序而不是逻辑单元的设备，可以建立一个 I\_T\_R 连接。对目标器例行程序的支持，虽然是在 SCSI-2 中被提出的，但是在 SCSI-3 中被逐步淘汰了。

在 Identify 消息之后可能跟随着其他的消息。对标签队列的请求和同步传输或宽数据传输也在这里发生。

在第四章中将更加详细地讨论 SCSI 的消息。

## □ 命令阶段

当消息输出阶段结束之后，目标器进入了命令阶段。它提高 C/D 信号的电平，使 I/O 信号和 MSG 信号反相，以此来表明进入了命令阶段。这就通知了启动器：目标器已经为接收一个 CDB（命令描述块）做好了准备。

CDB 的大小随着其所属的命令组而不同。SCSI-2 的规范说明书定义了 8 个不同的命令组，它们的名称很简单：从 0 号组到 7 号组。在 SCSI-2 规范说明书中定义的命令都属于 0 号、1 号、2 号和 5 号组。3 号和 4 号组被保留。6 号和 7 号组被留出来供厂商自己定义命令。

命令组号告诉目标器在 CDB 中有多少个字节。一旦它收到了整个命令块，可能会发生以下几种情况：如果 CDB 的大小或格式有错误，目标器将切换到消息阶段来报告错误。如果命令要求数据传输，目标器就切换到数据输出或数据输入阶段，这取决于所发出的命令。如果不数据传输，目标器就切换到状态阶段，并且给启动器发送一个状态字节来报告

命令的输出。

下面的部分全面介绍了可能出现的三种情况。

## □ 数据输入和数据输出

如果一条 SCSI 命令要求数据传输，而且参与数据传输的目标器已经为数据接收或者发送做好准备，目标器就将总线设置为数据输出或数据输入状态。在简单的情况下，在命令阶段和数据阶段中启动器和目标器保持连接。在较为复杂的情况下，目标器可以和启动器断开，当数据准备好以后再和启动器重新连接。这要求启动器在选择阶段后发送给目标器的 Identify 消息中已经赋予目标器断开连接的权力。

例如，磁盘驱动器在寻道和读取指定扇区的时候就可以断开连接。这就使得当目标器被占用的时候，启动器可以被释放出来以用于另外一个事务处理。

目标器通过反相 C/D 信号和 MSG 信号电平来进入数据阶段。它根据相对于启动器而言的数据传输方向来提高或降低 I/O 信号的电平。提高 I/O 信号的电平表示数据输入阶段；相反，使该信号反相来表示数据输出阶段。

在异步传输模式下，启动器和目标器通过一系列的 REQ/ACK 握手的方式来协调它们之间的数据传输。如果它们之间协商同步传输，在数据字节和能够被预先发送的 REQ 脉冲序号之间的时序就要预先建立起来。在任何一种情况下，REQ 和 ACK 的脉冲序号必须相等。

## □ 状态阶段

在命令和任何相关数据的传输完成后，目标器就切换到状态阶段。它声明 C/D 和 I/O 信号为真，并把 MSG 信号反相。状态阶段之后命令就结束了，除非命令是非正常终止的。一条消息会引起进程异常终止，意外的设备复位或断开连接，或者总线复位都会引起命令的非正常终止。

状态码只有一个字节，表示命令执行的成功或失败。它可能会指出目标器忙或者目标器被一个启动器保留；也可能指出一系列命令的中间某一条命令执行成功，它也可以告诉启动器哪些被称为检测数据的扩展信息是可以利用的。

关于状态码和检测数据的细节可以参考第六章。

## □ 消息输入阶段

消息输入阶段是 SCSI 事务处理的最后一个阶段。目标器声明 C/D 和 MSG 信号为真，从而进入了消息阶段，并且声明 I/O 信号为真表示消息对于启动器来说是输入的。

经常，目标器将发送一条 Command Complete 消息来表示命令处理已经结束了，而且已经发送了一个状态字节。它也可以发送消息表示错误状态或者在命令处理的过程中，在

它从总线上断开连接之前给启动器发出警告。

## ■ 阶段序列

从一个阶段转移到另外一个阶段是被严格定义的，而且是复杂的。上面的描述概括了一个典型的事物处理，虽然简单了一点。实际上，一旦启动器选择了一个目标器，阶段序列可以按照一定的路径进行。

图 3-1 说明了在阶段之间可能的状态转换。

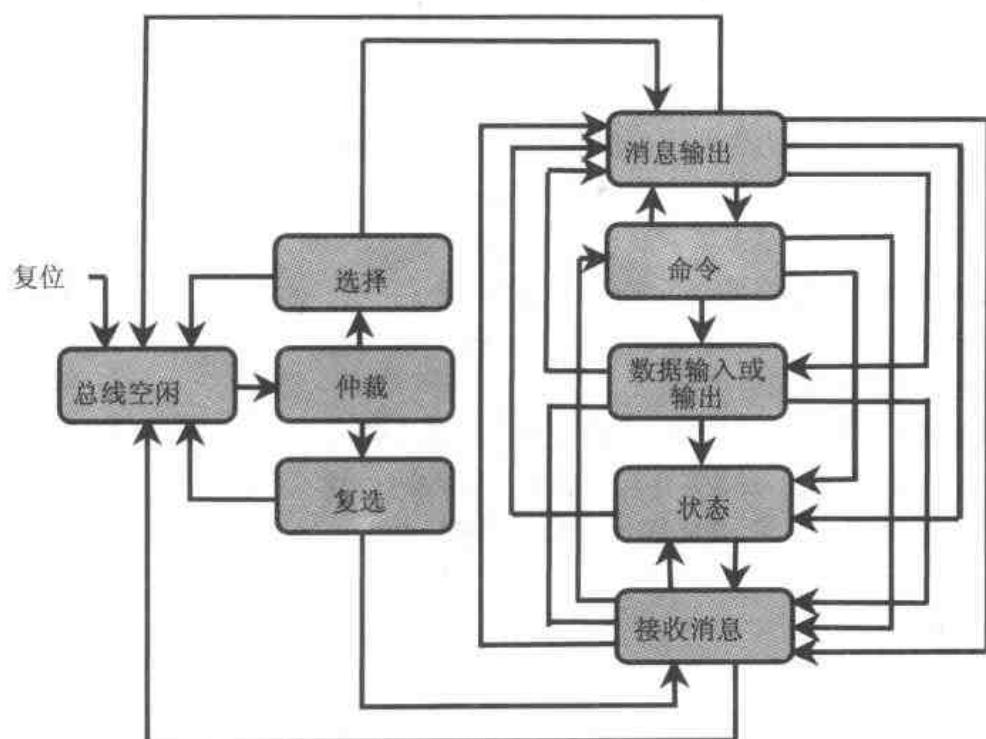


图 3-1 SCSI 状态转换

## 第四章 SCSI 消息

SCSI 事务处理是一个复杂的过程。一个保持事情正常进行的消息系统被指定为 SCSI 协议的一部分。这个消息系统靠提供一种错误报告和恢复机制、协商数据传输的参数、管理进程队列以及其他的功能来处理接口管理的细节。

消息阶段可以在设备选择阶段之后的任何时候发生。启动器给目标器发送一条 Identify 消息，来建立一个 I\_T\_L 连接，并指出断开连接的权力。它也可以发送消息来协商同步传输或宽数据传输。如果目标器支持标签队列。启动器就发送消息来管理这个队列。

目标器也可以把消息发送给启动器，宣布它想从总线上断开连接。在这个消息之前会有一个消息请求，请求启动器为当前进程保存指向命令、数据和状态信息的指针。

目标器也可以用消息通知启动器命令执行的结果。它可以发送一条 Command Complete 消息告诉启动器一个进程已经结束了，或者可以发送一条 Linked Command Complete 消息来报告一系列连接的命令中间的某一条命令已经执行完了。

启动器和目标器都可以通过消息来指出错误。Message Reject 消息表示一条以前的消息没有被实现或者是不适当的。

目标器控制数据总线的状态，并能够转换到消息输入阶段，而不管它是否需要给启动器发送消息。另一方面，当启动器需要和目标器通信的时候就不能强制目标器转换到消息输出阶段。取而代之的是，它使 ATN 信号为真来请求目标器转换到消息输出阶段。

### 消息类型

大多数的消息仅仅由一个或两个字节组成。扩展消息包含 3 个或 3 个以上的字节，很多还包含参数。SCSI 规范说明书支持一些强制消息，其他命令是可选的。

表 4-1 列出了启动器和目标器的强制消息。

表 4-1

强 制 型 SCSI 消 息

消息代码	消息名称	提供者
00H	Command Complete	目标器和启动器
05H	Initiator Detected Error	目标器和启动器
06H	Abort	目标器
07H	Message Reject	目标器和启动器
08H	No Operation	目标器和启动器
09H	Message Parity Error	目标器和启动器
0CH	Bus Device Reset	目标器
80H~FFH	Identify	目标器和启动器

消息 00H 即 Command Complete, 是 SCSI 消息集的主要消息。在正常的情况下, 目标器在 SCSI 事务处理结束时发送这一消息。消息 02H 到 1FH 是单字节消息。消息 20H 到 2FH 是 2 字节消息, 在这些消息代码后面有一个参数字节。

### Identify 消息

Identify 消息是一个单字节消息, 它有位编码选项。它是一个强制消息, 让我们来仔细地研究一下。

表 4-2

标 识 信 息

位 7	位 6	位 5	位 3~4	位 0~2
标识	DiscPriv	LUNTAR	保留位	LUNTRN

Identify 消息或者由启动器发送, 或者由目标器发送, 通常在选择阶段之后发送。这条消息通过在 LUNTRN 字段内标识一个逻辑单元或者目标器例行程序来建立了一个 L\_T\_L 或 L\_T\_R 连接。如果 LUNTAR 位没有被设置, LUNTRN 就表示一个逻辑单元号。DiscPriv 位表示启动器已经为目标器保证了断开连接的权力。当目标器发送消息时, 该位不定义。

### 扩展消息

01H 的消息代码表示一个扩展消息。表 4-3 列出了一条扩展消息的一般结构。

表 4-3

扩 展 信 息

字 节 #	字 段 描 述
0	扩展信息标志
1	扩展信息发展
2	扩展信息代码
3-(n+1)	可选参数

扩展消息的第二个字节是扩展消息码和参数的长度。在这里, 0 代表 256 字节的长度。

在 SCSI-2 下只定义了三个扩展消息。他们中的两个, Synchronous Data Transfer Request 和 Wide Data Transfer Request 值得进一步讨论。

### Synchronous Data Transfer Request 消息

Synchronous Data Transfer Request 消息是由启动器或目标器发送的, 目的是为同步数据传输协商时序。在 SCSI-2 中, 时序限制扩大到 Fast SCSI (高速 SCSI)。

表 4-4 列出了 Synchronous Data Transfer Request 消息的结构。

表 4-4

同步数据传输请求

字节 #	字 段 描 述
0	扩展信息标志
1	扩展信息长度
2	同步数据传输请求代码 (01H)
3	传输周期因子
4	REQ/ACK 偏移

在建立起同步数据传输的过程中，有两个关键的信息。传输周期因子是传输周期的四分之一。传输周期定义为两个连续的 REQ 脉冲的上升沿之间的时间间隔。这个时间就是数据字节或数据字在总线上的停留时间，它的单位是纳秒。

同步传输允许 REQ 的脉冲序号领先于 ACK 的脉冲序号一定的数量，这个数量在 REQ/ACK 的偏移量字段中给出。在异步传输的过程中，当数据被发送时，单个的 REQ 和 ACK 脉冲相互交替。在同步传输过程中，一系列的 REQ 脉冲被一系列相应的 ACK 脉冲确认。REQ 脉冲的数目和 ACK 脉冲的数目是相等的。但是 REQ 的脉冲数量可以在 ACK 脉冲被发出以前领先 REQ/ACK 偏移量字段中给出的数值。这个偏移量有效地确定了在一个同步脉冲串期间所传输的数据块的大小。在这个字段中，0 表示异步传输；FFh 则表示偏移量没有限制。

发报设备发送 Synchronous Data Transfer Request 消息。如果响应设备对这些参数表示同意，它就返回相同的消息；否则，它就把传输周期因子和 REQ/ACK 偏移量设置为它所能够支持的值。

在响应一个同步请求的时候，任何一方的设备都不能发送 Message Reject 消息。如果发送 Message Reject 消息，设备将回到异步传输方式。一旦它们之间已经协商好了同步传输的参数，他们就保持有效的状态一直到设备或总线复位，或者直到两个设备中的一方要求重新协商。

#### ■ Wide Data Transfer Request 消息

Wide Data Transfer Request 消息随着 Wide SCSI 的出现而出现。这个消息为那些支持 16 位或者 32 位数据通道的设备设置传输宽度。

表 4-5 列出了 Wide Data Transfer Request 消息的结构。

表 4-5

宽 数据 传 输 请求

字节 #	字 段 描 述
0	扩展信息标志 (01H)
1	扩展信息长度 (02H)
3	宽数据传输请求代码 (03H)
4	数据传输宽度指数

以字节为单位的传输宽度等于以 2 为底、以数据传输宽度指数字段中的数值为指数的计算结果。所以当指数字段中的数值分别为 0、1、2 时，传输宽度就是 8 位、16 位、32 位。你可能已经猜到，该字段中大于 2 的值没有被定义。

协商传输宽度的过程和协商同步传输的过程一样。发报设备发送这条消息，指数字段的值被设置为它所支持的值，对象设备可以接受或者拒绝这个值。如果协商失败，两个设备就都回到窄的 8 位传输方式。协商后的值保持有效状态一直到设备或总线复位，或者直到两个设备中的一方要求重新协商。

## 其他通用消息

除了 Command Complete、Identify 和扩展消息之外，SCSI 规范说明书还定义了其他的消息。这些消息都属于 1 字节和 2 字节的命令。消息码 02H 到 1FH 都是 1 字节命令，消息码 20H 到 2FH 都是 2 字节命令。

在本章的其余部分，我们将讨论这些消息中的较为有用或者较为有趣的消息。这些消息的完整列表可以参见 SCSI-2 规范说明文档。

### No Operation 消息

当需要一条消息但是又没有有效消息的时候，设备可以发送一条 No Operation 消息 (08H)。

### Abort 消息

启动器给目标器发送一条 Abort 消息 (06H) 来为一个 L\_T\_L 或 L\_T\_R 连接清除所有的 I/O 活动进程。它不影响目标器内的其他逻辑单元或者例行程序的进程。

### Bus Device Reset 消息

发送一条 Bus Device Reset 消息 (0CH) 来清除所有的错误状态，这对于程序员来说是很具有吸引力的，但是这是一种过激的手段。这条消息告诉目标器清除所有的 I/O 进程，并且迫使设备硬复位。它还会产生一个必须进行处理的 Unit Attention 状态。

### Disconnect 消息

在 I/O 进程中，目标器需要有一种方法来通知启动器它想断开连接，Disconnect 消息 (04H) 提供了这一方法。启动器也可以发送这条消息去指示目标器发回一条 Disconnect 消息。

目标器在发送一条 Disconnect 消息之前，可以发送一条 Save Data Pointers 消息 (02H)，

指示启动器存储当前缓冲区的偏移量；当重新连接时，一条 Restore Pointers 消息（03H）告诉启动器从前面的偏移量开始恢复数据传输。

### ■ Ignore Wide Residue 消息

如果在宽数据传输的过程中，要求的数据字节数目是奇数将会发生什么情况呢？一个 2 字节的消息可以处理这种情况。目标器发送 Ignore Wide Residue 消息指出在宽数据传输中有多少个字节被丢弃了。

表 4-6 列出了这条消息的结构。

表 4-6 空字符残留信息

字节 #	字段描述
0	信息代码 (23H)
1	空字符数量

Ignore Count 字段指出了在前面的数据传输中有多少字节被丢弃了。有效的值是 1、2 和 3。在一个数据输入阶段之后，目标器立即发送这条消息。

### ■ Queue Tag 消息

如果设备支持标签队列，它就必须支持标签排队消息。有 5 个消息用于管理排队操作。它们中有两个是单字节消息，分别是：Abort Tag (0DH) 和 Clear Queue (0EH)。Abort Tag 消息将会异常终止当前的 I/O 进程而不会影响在队列中的其他进程；Clear Queue 消息清除所有的排队 I/O 进程，并且异常终止活动进程。

其他的标签消息，Simple Queue Tag、Head of Queue Tag 和 Ordered Queue Tag 都是双字节命令。它们告诉目标器怎样对一个要执行的 I/O 进程排队。

表 4-7 列出了这些双字节排队标签消息的结构。

表 4-7 队列标记信息

字节 #	字段描述
0	队列标记信息代码 (20H~22H)
1	队列标记 (00H~FFH)

排队标签字段包含建立一个 I\_T\_L\_Q 连接的标识符。它为目标器内的一个逻辑单元唯一地标识出了一个排队后的进程。

在同一个消息输出阶段，在 Identify 之后，启动器立即发送一个排队标签消息。如果目标器不支持标签队列，它可以拒绝这条消息。当目标器想继续一个标签 I/O 进程的时候，它也能够发送这条消息。

标签命令队列不应该和连接的命令相混淆。连接的命令是作为一个单独的 I/O 进程来执行的。

排队标签消息有三个变体。

### ☒ Simple Queue Tag 消息

Simple Queue Tag 消息 (20H) 告诉目标器根据当前的排队管理算法来执行命令。关于排队算法的限定在发送给目标器设备的命令中说明。

### ☒ Head of Queue Tag 消息

Head of Queue Tag 消息 (21H) 告诉目标器把一条命令放在它的命令队列的最前面。它对活动的 I/O 进程没有影响。

### ☒ Ordered Queue Tag 消息

Ordered Queue Tag 消息 (22H) 告诉目标器按照接收到的顺序执行命令。目标器简单地把它放在队列中其他还未执行的命令之后。

## 第五章 SCSI 命令

通常认为，支持 SCSI 外围设备最困难之处在于寻找特定设备所支持的命令集的相关文档。SCSI-1 的规范说明文档为查询设备信息和报告错误定义了一个最小的强制命令集。而几乎所有的其他命令都被归入了“厂商自定义”命令那一类。

但 SCSI-2 做了一件令人称赞的好事，它清楚地说明了被不同的设备类别和设备类型所支持的命令集。相比于 SCSI-1 的规范说明书，SCSI-2 的规范说明书的长度增加了不少。增加的部分大多数用来说明新增的命令和描述这些命令的细节。它们可能使这个文档看来有些笨重，但正是因为有了标准命令集，程序员的工作才会得到简化。

本章是对 SCSI 命令集的概述，包括怎样使用这些命令，他们返回什么类型的数据。本章并不想把 SCSI 规范说明书需要用几百页来说明的内容浓缩为几页。若想得到不同的设备类型的命令集的一个完整的参考，那么你还是应该去看 SCSI 的规范说明书。

### 命令结构

SCSI 的命令和参数被封装在称为 CDB 命令描述块的结构中。和消息的情况一样，不同的命令组对应于不同长度的命令描述符。0 号组 CDB 是 6 字节长；1 号组和 2 号组的 CDB 是 10 字节长；5 号组的 CDB 占用 12 个字节；其他组为将来的使用或者厂商自定的命令保留。

### 多任务 I/O

CDB 的第一个字节是描述该命令的操作码。该字节的高三位表示命令所属的命令组，它的取值可以从 0 到 7；低 5 位表示命令码。

表 5-1 显示了操作码的结构。

表 5-1

操作 代 码

位 5~7	位 0~4
组码	命令代码

### 逻辑单元号

CDB 的第二个字节的高三位表示一个 LUN（逻辑单元号）；低 5 位可能是保留的，也可能是以后字段的一部分，这取决于命令所属的命令组。

表 5-2

逻辑设备号

位 5~7	位 0~4
逻辑设备号	保留位或其他数据

## 口 命令参数

在逻辑单元号字段之后就是命令参数字段，它通常包含命令参数。对直接存取设备来说，它包含的是逻辑块地址；对于传输数据的命令来说，包含的是传输长度，也可以包含和特定的命令或设备类型相关的值。

## 口 控制字段

每个命令描述符的最后一个字段是 Control (控制) 字段。这个字节包含了在连接命令选项中使用的位标志。Link (连接) 标志指出该 CDB 是否是一系列连接的命令的一部分。Flag (标志) 位决定一条连接的命令被成功执行以后目标器所返回的状态码。在本章的后面部分，我们将对命令连接作进一步的讨论。

表 5-3 列出了控制字段的结构。

表 5-3

控 制 字 段

位 6~7	位 2~5	位 1	位 0
厂商自定	保留位	标志位	连接位

## 口 参数列表

一些命令要求在命令描述符块之后跟随一个参数列表。如果是这样的话，在 CDB 中的一个参数字段可以指出后面所接数据的长度。

## 口 字节顺序

应该注意到，多字节字段中的值是按照大末尾 (big-endian) 顺序规定的，具体来说就是最高位的字节出现在高位，而低位的字节出现在低位。这会引起工作在 Intel 平台上的程序员的迷惑，因为在 Intel 的平台上，数据是按照小末尾 (little-endian) 顺序存储的，高位的字节出现在低位。

## 强制 SCSI 命令

SCSI 规范说明书为所有的设备定义了强制命令和可选命令。每个设备类型都有它自己的一套命令，其中有一些是强制的，而其他一些是可选的。我们首先来看一下那些适用于所有设备类型的命令。

有 4 个所有设备都必须支持的强制命令。表 5-4 列出了这些命令。

表 5-4

强制性 SCSI 命令

代 码	命 令 名 字
00H	Test unit Ready
03H	Request Sense
12H	Inquiry
1DH	Send Diagnostic

这些命令都要处理设备标识符、状态和错误报告。很多可选命令执行与设备配置和报告相关功能。

### Test Unit Ready 命令

Test Unit Ready 命令是强制命令中最简单的，它只是简单地报告设备是否已经为执行命令做好了准备。表 5-5 显示了这条命令的 CDB。

表 5-5

Test Unit Ready

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码							
1	逻辑设备号		保留位					
2	保留位							
3	保留位							
4	保留位							
5	控制字段							

这个命令的结构并不复杂。对控制字段的描述和前面一样，尽管对于这个命令来说，Link 字段几乎没有作用。

如果设备已经为执行命令做好了准备，Test Unit Ready 命令就返回一个 Good 状态码。如果没有做好准备，它就返回一个 check condition 状态。在这种情况下，可以利用检测信息来获得进一步的细节。对状态和检测信息的解释是它自己的工作，我们将在第 6 章讨论它们。

## ■ Inquiry 命令

Inquiry 命令引起一个 SCSI 设备发送关于该设备的制造商、模型信息、和被支持的特性的信息。Inquiry 命令的 CDB 如表 5-6 所示。

表 5-6

Inquiry 命令

字节#	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	操作代码							
1	逻辑设备号			保留位				EVPD
2	页代码							
3	保留位							
4	分配长度							
5	控制字段							

EVPD 标志告诉目标器设备返回一个重要的产品数据，而不是标准查询信息。页代码字段表示返回哪一个重要的产品数据。对 EVPD 标志的支持是可选的。分配长度字段告诉目标器为返回数据分配了多少空间。

标准查询命令产生的信息用表 5-7 中列出的格式返回。

表 5-7

Inquiry 数据结构

字节#	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	外设限定符			设备类型代码				
1	RMB	设备类型修改器						
2	ISO 模型		ECMA 模型		ANSI 增强模型			
3	AENC	TrnIOP	保留		响应数据格式			
4	附加数据长度							
5	保留							
6	保留							
7	RelAdr	WBus32	WBus16	Sync	Linked	保留	CmdQue	SftRe
8~15	厂商标识字符串							
16~31	产品标识字符串							
32~35	产品修正级字符串							
36~55	厂商自定信息字符串							
56~95	保留字							
96~end	厂商自定数据							

查询数据包含了有关设备的信息的重要部分。开始部分是 Peripheral qualifier (外围设备限定字符) 字段, 它指出设备是否和查询的逻辑单元相连接。Device type (设备类型) 字段指出这个地址的设备的类型。SCSI 规范说明书列出了几种设备类型码, 如表 5-8 所示。

表 5-8

外 围 设 备 类 型 代 码

代 码	设 备 描 述
00H	直接存贮设备 (磁盘驱动器)
01H	顺序存贮设备 (磁带驱动器)
02H	打印机设备
03H	处理器设备
04H	一次写入设备
05H	CD-ROM 设备
06H	扫描仪设备
07H	光存贮设备 (光驱)
08H	介质转换设备 (自动电唱机)
0AH~0BH	图形预处理设备
1FH	未知设备类型

如果没有设备与指定的目标器和 LUN 相连, 查询数据的第一个字节就被设置为 07H, 它对应于外围设备限定符 03H 和设备类型 01H。

RMB 标志表示设备是否支持可移动介质。Device type modifier (设备类型修饰符) 仅仅是为了和 SCSI-1 规范说明书向后兼容而定义的。

第三个字节包含了关于该设备和其他标准的兼容性的信息。其中最有用的可能就是 ANSI 版本字段, 它说明 SCSI 标准的哪一个版本被支持。

AENC 和 TrnIOP 标志表示对异步事件通知和 Terminate I/O Process 消息的支持。Response date format (响应数据格式) 值 02H 在此表示 SCSI-2 的一致性。字段表示查询数据的结构是否和 SCSI-1、SCSI-2 标准或它们的中间版本相符合。

Additional date length (附加数据长度) 字段表示在标准查询数据头后面跟随着多少数据。注意, 它表示多少数据是可以使用的, 而不是实际传输了多少数据。

7 号字节比较有趣。一系列的位标志表示这个设备支持哪些功能。按照出现的顺序, 它们依次支持相对地址 (RelAdr)、32 位 Wide SCSI (Wbus32)、16 位 Wide SCSI (WBus16)、同步数据传输 (Sync)、命令连接 (Link)、命令排队 (CmdQue) 和软复位 (SftRe)。

后面三个字段包含了设备商和产品的信息。它们都是 ASCII 字符, 都带有空格, 以适合它们各自的字段的宽度。保留字段是为将来使用或者放置厂商自定的信息。

## 口 例子：Iomega Zip 驱动器

让我们来看一下某个外围设备返回的查询数据。查询的设备是 Iomega Zip 驱动器。这个驱动器是一个 SCSI 直接存取设备，它使用可移动存储介质。从 Inquiry 命令返回的数据见表 5-9。

表 5-9 Iomega Zip 驱动器查询数据

字节#	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0									
0	外围放大器			设备类型代码													
1	RMB(1)	设备类型修改器															
2	ISO 模型		ECMA 模型			ANSI 增强模型											
3	AENC (0)	TrnIOP (0)	保留位		响应数据格式												
4	附加数据长度																
5	保留字节																
6	保留字节																
7	RelAdr (0)	Wbus32 (0)	WBus16 (0)	Sync (0)	Linked (0)	保留	CmdQue (0)	SftRe (0)									
8~15	厂商标识字符串																
16~31	产品标识字符串																
32~35	产品修正级字符串																
36~55	厂商自定信息字符串																
56~95	保留字节																
96~end	厂商自定数据																

所列出的查询数据指出它是一个使用可移动存储介质的直接存取设备。它符合 SCSI-2 的 ANSI 标准，而且数据格式符合 SCSI-2。它不支持 32 位或 16 位的 Wide SCSI，也不支持同步数据传输、连接的命令、命令排队。

厂商和产品标识字段确认制造厂家和模型，修正级字符串表示产品修订级别。厂商自定字段包含制造商数据和版权通知。

设置了 EVPD 标志并且页代码为 00H 的查询应该返回一个该驱动器所支持的重要产品信息代码页的列表。然而，这个命令的执行并不成功，因为错误码指出它是一个非法的请求。这个设备不支持重要产品数据特性。

## 可选命令

其他可以应用于所有设备的命令是可选的。表 5-10 列出了这些命令。

表 5-10 可选 SCSI 命令

代 码	命 令 名
18H	Copy
1CH	Receive Diagnostic Results
39H	Compare
3AH	Copy and Verify
3BH	Write Buffer
3CH	Read Buffer
40H	Change Definition
4CH	Log Select
4DH	Log Sense

Copy 命令、Copy and Verify 命令和 Compare 命令提供了在一个目标器的两个逻辑单元之间拷贝和比较数据的方法。这些命令的参数随着设备类型的不同而不同，但是都要指出源地址、目标地址和传输长度。

Log Sense 命令和 Log Select 命令提供了一种为目标器设备的管理统计信息机制。SCSI 规范说明书为记录开始值和累积值定义了页代码参数，但是没有指定记录数据的类型。

Read Buffer 和 Write Buffer 命令用来检测设备的缓冲内存。除了这个诊断功能之外，它也能显示某个设备有多少内存。

Change Definition 命令允许启动器改变目标器的操作模式，只要这个命令被支持就行。启动器可以要求目标器为兼容性而采用 SCSI-1、SCSI-2 或通用命令集的定义。

## 设备类型特定命令

有 4 个命令是为所有的设备定义的，它们仅对某些设备来说是要求强制支持的。Mode Sense 和 Mode Select 命令有 6 字节和 10 字节两个版本。

表 5-11 设备类型标识 SCSI 命令

代 码	命 令 名
15H	Mode Select(6~byte)
1AH	Mode Sense(6~byte)
55H	Mode Select(10~byte)
5AH	Mode Sense(10~byte)

这些命令让启动器可以读取或者设置许多设备参数。参数以页的方式组织，用页代码标识出来。很多页代码是设备特定的，但是有一些是和通用的 SCSI 参数相关的，像断开连接控制和命令排队算法。

## ■ Mode Select 命令

Mode Select 命令让驱动器为一个目标器或者目标器上的一个逻辑单元设置操作参数。6 字节 Mode Select 命令的 CDB 见表 5-12。

表 5-12

Mode Select (6-byte)

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作代码 (15H)							
1	逻辑设备号		PF		保留位		SP	
2	保留字节							
3	保留字节							
4	参数列表长度							
5	控制字段							

PF 标志指出了参数列表的格式。这个标志应该被设置为指出该页符合 SCSI-2 的规范说明书。清除这个标志表示页格式是厂商自定的，就像在 SCSI-1 中一样。

SP 标志请求目标器把页存储到内存中去。实际被保存的页会有改变，但是可以使用 Mode Sense 命令来查询。

参数列表长度字段指出所有的模式页和后面的信息的总长度。在研究 Mode Sense 命令的时候，我们将仔细地研究返回数据的格式。

## ■ Mode Sense 命令

Mode Sense 命令请求获取描述目标器的操作参数的信息。这个命令能够查询默认参数、当前参数或其他可以用 Mode Select 命令改变的参数。

6 字节 Mode Sense 命令的 CDB 列于表 5-13 中。

表 5-13

Mode Sense(6-byte)

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码 (1AH)							
1	逻辑设备号		保留位	DBD	保留位			
2	PC	页代码						
3	保留字节							
4	分配长度							
5	控制字节							

DBD 标志告诉目标器在返回数据中禁用块描述符（块描述符和数据格式将在后面讨论）。PC 值表示所请求的参数类型。它的取值可以从 0 到 3，依次为当前值、可变值、默认值和保存值。

最重要的字段可能就是页代码字段。它的值表示返回参数类型。3FH 表示目标器返回所有可用的模式页。

Mode Sense 命令返回什么类型的信息？SCSI 规范说明书为具体的设备类型定义了模式页。例如，直接存储设备可以返回关于存储介质类型和磁盘几何结构的信息、高速缓存参数、错误恢复和很多其他的操作细节。

## ■ 模式检测数据参数

以页的格式返回模式信息的设备遵循一种 SCSI 文档中描述的数据结构。一个模式参数头标在该数据的最前面，后接块描述符和模式页。6 字节和 10 字节命令的头标是不同的。6 字节 Mode Sense 命令的头标如下所示。

表 5-14

模式参数磁头

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	模式数据长度							
1	介质类型							
2	设备标识参数							
3	区块描述符长度							

模式数据长度字段给出其余数据的全部长度。介质类型字段中的代码随着设备类型的不同而改变，它一般描述介质密度和格式。块描述符长度字段给出了后接的块描述符的全部长度，以 8 个字节为单位。

块描述符给出了设备所支持的当前和默认介质的更多的具体信息，如表 5-15 所示。

表 5-15

模式参数块描述符

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	密度代码							
1~3	区块数目							
4	保留字节							
5~6	区块长度							

密度代码字段随着设备类型而改变。区块数目字段表示长度有多少字段。如果区块长度为整个介质，这个字段就被设为 0。

在块描述符后面是一个模式页的列表。表 5-16 显示了一个模式页的一般格式。

表 5-16

模式页格式

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	PS	保留位	页代码					
1	页长度							
2~end	模式参数							

PS 标志表示了一个模式页是否能够被存储。页代码字段标识出返回的信息。其中一些代码可应用于所有的设备，但另外一些是厂商自定的。页长度字段给出了在该页其余部分的模式参数的长度。

### 口 例：Iomega Zip 驱动器

让我们来重新看一下 Iomega Zip 驱动器，把它作为模式页信息的一个简单的例子。我们将使用 6 字节的 Mode Sense 命令来查询该驱动器的默认的信息和驱动器的所有可用的模式页。它的 CDB 如下所示。

表 5-17

Iomega Zip 驱动器模式读出

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码 (1AH)							
1	逻辑设备号 (0)		保留位	DBD(1)	保留位			
2	PC(2)	页代码 (3FH)						
3	保留字节							
4	分配长度							
5	控制字段							

命令的返回数据列于表 5-18。

表 5-18

Zip 驱动器模式参数磁头

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	模式数据长度 (25H)							
1	介质类型 (0)							
2	设备标识参数 (0)							
3	区块描述符长度 (8)							

它意味着什么呢？数据长度字段告诉我们后面有 37 个字节；介质类型字段中的 0 代表这些信息是关于默认介质类型的；块描述符长度字段告诉我们后接一个 8 字节的块描述符。

对于作为直接存取设备的 Zip 驱动器而言，设备特定参数由经过编码的位字段组成。表 5-19 列出了 Zip 驱动器设备标识参数。

表 5-19 Zip 驱动器设备标识参数

位 7	位 5~6	位 4	位 0~3
WP	保留位	DPO/FUA	保留位

WP 位表示设备是否是写保护的。DPO/FUA 标志表示该器件是否支持某些读操作的高速缓存选项。

在头标之后是块描述符，如表 5-20 所示。

表 5-20 Zip 驱动器区块描述符

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	密码代码 (0)							
1~3	区块数目 (0)							
4	保留字节							
5~6	区块长度 (200H)							

直接存取设备没有定义密度码，所以该字段没有包含有用的信息。因为块数目字段是 0，所以我们知道在这个描述符中的信息适用于该介质上所有剩下的块。

根据块描述符可知，块长度是 512 字节。这个字段以高位在后的顺序出现，在这里 20H、00H 分别在第 5 和第 6 字节。在 Intel 平台上进行 SCSI 编程的一个常见的错误是当读数据字段的时候忘记修正字节顺序。

在描述符后有三个模式参数页。第一个是错误恢复页 (01H)，后面是断开连接-重新连接页 (02H) 和厂商自定页 (2FH)。让我们看一下错误恢复页，如表 5-21 所示。

表 5-21 Zip 驱动器错误恢复模式页

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	PS(0)	保留	页代码 (01H)					
1	页长度 (6)							
2	AWRE (1)	ARRE (1)	TB (0)	RC (0)	EER (1)	PER (0)	DTE (0)	DCR (0)
3	重读数量 (0)							
4	纠正范围 (0)							
5	头偏移范围 (0)							
6	数据选通偏移数量 (0)							
7	保留字节							

PS 字段中的值是 0，它表示这页不能被存储在非易失性存储器（nonvolatile memory）内。Zip 驱动器不支持保存页，用 Mode Sense 命令查询请求的保存页将返回一个错误。

错误恢复数据的正常页长度是 10 字节。在这里，它仅有 6 字节长。

该页的第 3 个字节包含一组位标志用于错误恢复选项。自动读和自动写重新分配（AWRE 和 ARRE）用于缺陷数据块。允许及早恢复（EER）标志表示设备将使用可获得的最便利的错误恢复方法。其他字段告诉我们该设备不报告被恢复的错误，当错误发生时终止数据阶段，或者使用纠错码进行错误恢复。

从程序员的观点来看，这是重要的信息。知道一个设备如何处理错误就能够把你建立的这种错误恢复码放入你的软件中去。

## 读和写

在本章将要结束的时候，让我们来看一下 SCSI 命令集中的主要函数——Read 和 Write 命令。这些命令随着设备类型的不同而不同，所以我们将分析一下直接存取设备是如何使用它们的。

和其他命令一样，Read 和 Write 命令有 6 字节和 10 字节的版本，分别属于 0 号组和 2 号组。10 字节版本允许命令描述块中在地址和传输长度字段中指定更大的数字。它们还包含额外的字段用于规定高速缓存处理和相对寻址。

用于直接存取设备的 Read 和 Write 命令是面向块的。它们以一个逻辑块地址开始，以块为单位来表示传输长度。你需要 Mode Sense 命令或其他命令报告的块长度来确定数据传输的字节数。

### Read 命令

6 字节的 Read 命令很简单。它的字段标识出要读的第一个逻辑块和要传输的逻辑块的数目。表 5-22 列出了 Read 命令的 CDB。

表 5-22

Read(6-byte)

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码 (08H)							
1	逻辑设备号		逻辑块地址					
2	逻辑块地址 (接#1 字节)							
3	逻辑块地址 (接#2 字节)							
4	传输长度							
5	控制字段							

逻辑块地址字段跨越了三个字节，最高有效字节（MSB）最先出现。传输长度字段占用一个单独的字节。

### Write 命令

Write 命令的 CDB 几乎和 Read 命令的 CDB 一样。只是在操作码上有所不同。表 5-23 列出了 Write 命令的 CDB。

表 5-23

Write(6-byte)

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码 (OAH)							
1	逻辑设备号		逻辑块地址					
2	逻辑块地址 (接#1 字节)							
3	逻辑块地址 (接#2 字节)							
4	传输长度							
5	控制字段							

### 其他命令

到目前为止，我们几乎接触了被不同的 SCSI 设备类型支持的所有的命令。SCSI-2 规范说明书为那些希望试验的人列出了完整的命令列表。

## 第六章 状态、检测和错误

如果一切都是那么完美，那么就不必从 SCSI 设备处接收反馈。它们总是处于预备通信状态，每一个命令都会被成功执行，对 SCSI 外围设备的编程将是一项简单的工作。事实上，命令可能被完全地执行，也可能只执行了一部分，甚至也可能一点都没被执行。设备会耐心地等待离线，这是因为纸用完了，或者由于磁带盒是写保护的而引起操作失败。即使其他每一部分都正常工作，SCSI 总线上产生的错误也有可能阻止通信。

SCSI 规范说明书是一个全面的文档，它涉及到了很多不同类型的设备。每个设备类型都有它自己的发生故障的事件、报告的错误、影响设备对命令响应方式的条件改变。如果在 SCSI 下的错误处理看上去复杂或者令人混淆，那么这就是原因。

SCSI 文档的作者做了一个绝妙的工作，他们统一了对不同设备类型的支持。他们在说明书中定义了不同级别的反馈。从总体上来说，状态码报告发送给设备的命令的执行结果。详细地讲，一个详细的检测信息集指出了错误是在哪里发生的。

每个正常结束的命令阶段通常后接一个状态阶段。在状态阶段，目标器给启动器回送一个字节，指出命令执行的结果。表 6-1 列出了状态字节的结构。

表 6-1 状 态 字 节

位 6~7	位 1~5	位 0
保留位	状态代码	保留位

这个状态字节用起来可能是麻烦的，因为最低位是保留的，很容易忘记实际的状态码是从 1 号位开始的。

### 状态码

在 SCSI-2 的规范说明书中只定义了 9 个状态码。所有其他的值都被保留了。

表 6-2 状 态 代 码

状态 代 码	说 明
00H	良好
01H	检验状态
02H	满足条件
04H	忙
08H	介质
0AH	介质条件满足

续表

状态代码	说 明
0CH	限制冲突
11H	命令结束
14H	队列满

在命令被成功执行后，目标器返回 Good（良好）状态。但是一些数据查找或预取命令返回 Condition Met（满足条件）状态。

Busy（忙）状态表示设备正在参与其他的进程，或者现在不能够接收命令。当试图访问被另外一个启动器保留的设备时，将产生 Reservation Conflict（限制冲突）状态。如果设备支持标签命令队列，它可以返回 Queue Full（队列满）状态表示它的命令队列不能再接收任何登录。

一些状态码和连接的命令有关，Intermediate（介质）和 Intermediate-Condition Met（介质条件满足）在每个连接的命令之后返回，对于一般的命令，相应的返回状态是 Good 或 Condition Met 状态。

最有趣的状态码是 Check Condition（检验状态）。这个状态码和 Command Terminated（命令结束）一起出现，表示存在一个偶然事件通信状态。当目标器有可利用的扩展错误信息时，就会发生偶然事件通信状态，这种扩展错误信息被称为检测数据。它将一直保留这个检测数据，直到启动器把它读取出来，或者有另外一个动作清除偶然事件通信状态。如果设备仅能为一个启动器维持检测数据，它就给其他试图访问它的设备返回 Busy 状态。

## 检测数据

检测数据包含了关于错误状态的详细信息。它主要被组织为一个检测键的主类和附加检测码（ASC）的子类以及附加检测码标识符的形式。这些数据结合起来，就能够精确地传递关于错误状态的详细信息。

在解释这些数字之前，我们必须检索它们。Request Sense（请求读出）命令完成了这项功能。对所有的设备类型来说，Request Sense 命令是强制实现的。表 6-3 是这个命令的 CDB。

表 6-3

请 求 读 出

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	操作码（03H）							
1	逻辑设备号		保留位					
2	保留字节							
3	保留字节							
4	分配长度							
5	控制字段							

分配长度字段告诉目标器将返回多少检测数据。

该命令所返回的数据后接一张 SCSI 规范说明书定义的表。这张表对于所有的设备都是相同的，但是某些字段仅仅和一定的设备类型有关。对于目标器来说，检测数据使用厂商自定的格式也是允许的。表 6-4 是读出数据格式列表。

表 6-4 读出数据格式

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	合法	错误代码 (70H 或 71H)						
1	段号							
2	文件标记	EOM	ILI	保留	检测链			
3~6	信息							
7	附加检测长度							
8~11	命令标志信息							
12	附加检测代码 (ASC)							
13	附加检测代码限定符 (ASCQ)							
14	字段可替换单元代码							
15	SKSV	检测链标识						
16~17	检测链标识 (接 15 字节)							
18~end	附加检测数据							

这个数据传递关于已经发生的错误的信息。Valid (合法) 位表示下面的数据遵循上面定义的结构。在错误码字段只定义了两个值：70H 表示当前的错误；71H 表示延迟的错误。对延迟的错误的支持是可选的。段序号、文件标志、介质末尾 (EOM) 字段应用于特殊的命令或设备类型。

如果错误长度指示符 (ILI) 位被置“1”，那么命令请求的数据数量就和从目标器处获得的数据数量不匹配。信息字段包含了直接存取设备或磁带设备的块的不同点，或者不是面向块的设备的字节的不同点，如扫描仪。

信息字段可能包含不同的数据，这依赖于和它相关的命令或设备类型。对命令特定信息字段也是这样。

## ❑ Sense Key

(SenseKey) 检测键指出错误属于哪一类。这些类报告硬件错误、写保护错误、非法请求和别的种类的状态。SCSI 规范说明书定义了表 6-5 所示的检测键，其他值被保留。

表 6-5

检 测 链

状态代码	说 明
00H	没有意义
01H	恢复错误
02H	未准备好
03H	介质错误
04H	硬件错误
05H	非法请求
06H	单元注意
07H	数据保护
08H	空检测
09H	厂商自定
0AH	拷贝失败
0BH	命令失败
0CH	等于
0DH	超出容量
0EH	比较错误

它们中的一些需要进一步解释。例如，非法请求（Illegal Request）键表示一个 CDB 包含了一个非法的字段或参数。如果这样的话，在检测数据内的检测键特定值（SKSV）标志就被置“1”，并且检测键特定字段包含了指向 CDB 或参数中损坏值的指针。当调试 SCSI 软件的时候，很容易获得这些信息。

对恢复错误键、硬件错误键和介质错误键来说，检测键特定字段包含一个重复计数器，如果 SKSV 标志被置“1”的话。

## ► 单元注意键

当某些可能改变设备操作参数的事件发生时，单元注意键就会被使用。一次复位或一个加电（wp power）周期，一次介质变化或者模式参数的一个改变都能够触发一个单元注意状态。在这种状态下，设备仅仅能够对 Inquiry 命令和 Request Sense 命令做出响应，而对其他的命令则返回一个 Check Condition 状态。在这种状态下，仅有 Request Sense 命令可以清除单元注意状态，如果还存在一个偶然事件通信，那么任何来自启动器的命令都将清除单元注意状态。

在加电的时候，设备会进入这个状态，它意味着发送第一个命令会返会一个 Check Condition 状态。这对于那些对 SCSI 的工作方式不熟悉的程序员来说是特别麻烦的。

## ■ ASC

在很多情况下，单独靠检测键就可以为错误恢复提供足够多的信息。当它做不到的时候，ASC（附加检测码）和 ASCQ 就提供补充信息。ASC 给出了关于错误来源的附加信息，ASCQ 给出了更多的具体细节。ASC 和 ASCQ 一起确切地指出在 SCSI 的操作中发生了什么错误。这些错误中的一些仅仅应用于特定的设备类型。

ASC 和 ASCQ 分配的完整列表在 SCSI 文档中长达几页。你可以浏览一下这张表，对这些错误有一个了解，以便于你的软件可以处理它们。

## ■ 例：Iomega Zip 驱动器

我们再来看一下 Iomega Zip 驱动器，把它作为检测数据的一个例子。我们用改变磁盘的方法来强制产生一个单元注意状态，并且来分析一下 Request Sense 命令的结果。

表 6-6 Iomega Zip 驱动器 检测数据

字节#	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	合法 (0)	错误代码 (70H)						
1	段号 (0)							
2	文件标记 (0)	EOM (0)	LLI (0)	保留	检测链 (06H)			
3~6	信息 (0)							
7	附加检测长度 (17)							
8~11	命令标志信息 (0)							
12	附加检测代码 (ASC) (28H)							
13	附加检测代码限定符 (ASCQ) (0)							
14	字段可替换单元代码 (0)							
15	SKSV (0)	检测链标识 (0)						
16~17	检测链标识 (接#15 字节) (0)							
18~end	附加检测数据 (FFH、FEH、01H、02H、1CH、0、0)							

检测键对应于单元注意键，这正是我们所希望的。ASC 和 ASCQ 指向一个消息“未准备好转换，介质可能已经改变”，这表示在线设备是不可用的，这也是我们所希望的。

有些奇怪的是 Valid 位没有被置“1”，并且附加检测数据字段看上去包含厂商自定的数据。它很好地提醒了我们不是所有的 SCSI 设备都完全遵循标准。当使用某个特定设备的时候，来自制造商的信息可能是关键的。

## 第七章 ASPI

在 SCSI 的早期，大多数的软件开发者把支持 SCSI 适配器卡的硬编码直接加入他们的应用程序。为一个新的 SCSI 适配器或者芯片加入支持程序是一项乏味并且容易出错的工作，这项任务主要由某些程序员承担，这些程序员的精力主要集中在应用程序本身，而不是 SCSI 接口代码。因此应用程序往往要求主机适配器和外围设备之间的特别约定，这通常要求负责 SCSI 接口代码的程序员来做。因为这些代码与特定的设备和适配器如此紧密地相关，所以需要持续地维护和升级来保证它能够跟上硬件的变化。

在开发 SCSI 代码的过程中，常常会遇到的问题主要集中在两个方面：为不同的外围（目标器）设备产生正确的 SCSI 命令；通过和 SCSI 主机适配器对接，把这些命令发送给设备。如同在第一章中所描述的，第一件事被 CCS 和 SCSI-2 对 SCSI 标准的改进执行了。这些改进帮助设备制造商实现一个命令集，这些命令适合于那些设备，从而允许程序员为一个给定的设备编写一般的代码，并且有理由认为这些代码可以适用于这个类型的大多数其他的设备。第二个问题由标准编程接口的定义来解决，标准命令接口把命令传递给设备，而不考虑使用哪一种主机适配器。在本章中，我们将叙述在 PC 平台上的接口中最流行的一种接口：ASPI（高级 SCSI 编程接口）。

### 什么是 ASPI

ASPI 高级 SCSI 编程接口由 Adaptec 公司开发，该公司是主机适配卡的主要生产厂家。Adaptec 公司公布了它们的 Adaptec SCSI 编程接口，把它重新命名，并鼓励其他的生产厂商支持它。ASPI 被定义和在大多数基于 PC 的操作系统上使用，这些操作系统包括 DOS、Windows、OS/2 和 Novell Netware，并且 Adaptec 自己的 SCSI 主机适配器产品直接对它进行支持。当程序员开始接受 ASPI 时，大多数其他的 SCSI 设备的生产厂商开始把 ASPI 兼容的接口引入到他们自己的硬件中，特别是对 MS DOS 操作系统而言更是如此。ASPI 不久变成了一个基于 PC 的 SCSI 编程的标准，并且使用于 PC 平台上的 SCSI 设备得到了人们更为广泛的接受。

ASPI 的成功很大程度上归功于它的相对简单性。它放弃了一些 SCSI 的特性，例如标签队列和异步事件通知，支持一个较为简单的接口模型和相对容易的实现（在许多外围设备和操作系统中就是这样）。另外，ASPI 主要集中在应用程序上，它仅仅处理启动器或主机适配器的请求，这意味着 ASPI 能够把命令发送到一个 SCSI 设备，但是它不能够处理从别的启动器（主机适配器）处收到的命令。大多数基于 PC 的应用程序把注意力放在对外围设备的控制上，所以缺乏对目标器模式的支持对 ASPI 来说并不是一个障碍。

## 为什么要用 ASPI

让我们来看一下 ASPI 实际上为程序员做了些什么。首先，它把你和 SCSI 主机适配器的硬件接口隔离开来。从原理上来说（大多数实际情况也的确是这样），你可以用一种 ASPI 管理器和主机适配器编写代码，但是这些代码可以在任何一个支持兼容 ASPI 管理器的主机适配器上运行。所有这些硬件特定的接口代码由 ASPI 管理器负责处理。每个不同的 SCSI 主机适配器的设计都有一个相应的 ASPI 管理器，通常，它是由生产厂商自己开发的。

除了提供硬件独立性之外，ASPI 还提供了大量的操作系统独立性。使用 ASPI 的应用程序员不必担心中断处理程序和其他的与设备驱动程序相关的操作系统的具体细节。页面锁定和页面交换在旧的操作系统中还会涉及到，例如在 DOS 和 Windows 3.1 中，但是在 Windows 95 和 Windows NT 中就很少涉及到它们了。除了像这样的小细节之外，ASPI 接口本身在所有的操作系统平台上是很一致的，尽管它们的实现细节不同。从一个程序员的观点来看，在各个操作系统平台上的 ASPI 之间的主要不同在于用来连接 ASPI 管理器本身的机制。例如，在 MS-DOS 系统中，ASPI 通常作为设备驱动程序由 CONFIG.SYS 文件装载来实现；然而，在 Windows 下，ASPI 是通过动态链接库（DLL）来实现的。在它的 16 位和 32 位实现之间，它们在传递给 ASPI 管理器的结构内字段的安排和顺序上略有不同。下面的例子用的是 ASPI 的 Win32 的结构和定义，它可用于 Windows 95 和 Windows NT 系统。如果你在另外一个平台上使用 ASPI，一定要确认你所采用的结构定义适合于那个系统。

## ASPI 概念

下面的一些部分叙述了一些主要的 ASPI 概念，在我们研究 ASPI 命令和结构之前，我们先来研究一下这些概念。这些概念包含设备寻址、发出 SCSI 命令和等待命令完成，还有一些你需要了解的一些适配器特定的细节。当读这些内容的时候，你会觉得 ASPI 比实际要复杂得多。我们将在后面提供大量的例子和代码。现在，你应该注意到，有一个主要的例行程序，通过对它的调用就可以发送大多数的 SCSI 命令。这个例行程序使用一个指向结构的指针，该结构包含了执行一个给定的 SCSI 命令所需要的全部信息。在下面的例子中，这个例行程序被称为 `SendASPI32Command()`，然后，我们将向你演示如何使用它。如果你对这些命令或结构有很强的好奇心，那么你可以先看后面的相关内容。

在 ASPI 规范说明书最近的一次修订中，已经加入了其他的例行程序来支持即插即用 SCSI 和大缓冲区。

### 适配器和设备编址

SCSI 规范说明书用一个 SCSI 设备 ID 来标识设备，用一个 LUN 逻辑单元号来标识一

个设备上特定的子单元。典型的 SCSI 设备的 ID 号可以从 0 到 7, LUN 号也是这样。根据这两条信息，在一条 SCSI 总线上的任何单元或子单元都可以用 ID 号独一无二地标识出来。然而，可以把两个或更多的 SCSI 主机适配器连接到一个系统上，所以 ASPI 管理器还要求提供一条信息——HA（主机适配器号）。把主机适配器看作一个 I/O 通道是很有帮助的，因为某些主机适配器卡能够支持不止一个通道。

HA（主机适配器号）是连续的，从 0 开始。第一个（或者仅有的）主机适配器的 ID 应该为 0，第二个是 1，依此类推。所以在 ASPI 下，SCSI 设备可以被一个主机适配器号、一个 SCSI ID、和一个 LUN 一起独一无二地标识出来。涉及特定 SCSI 设备的每一个 ASPI 命令都包含这三条信息 (HA: ID: LUN)。

## ■ 发送 SCSI 命令

因为我们能够对 SCSI 设备寻址，所以我们通常想通过发送命令使设备做一些有用的事情。它包含 4 个基本的步骤：建立命令，把命令发送到 ASPI 管理器，等待命令完成，最后对 ASPI 管理器返回的任何命令或者状态码作出解释。

所有的 ASPI 命令都使用一个称为 SRB SCSI 请求块的数据结构。SRB 开头的几个字段对于所有的 ASPI 命令都是通用的，但是剩下的字段则依赖于具体被执行的命令。在 SRB 建立完成后，它就包含了 ASPI 管理器或者目标器设备返回的所有数据和状态码。

## ■ 建立 SRB

发送一条 ASPI 命令的第一步是初始化 SRB 结构。在很多情况下，你只要在要求的字段中填入值就可以了。我们将在本章的后面描述这些字段，但是现在，让我们先迅速地来看一下一个常见的例子——把一条 SCSI 命令 Inquiry 发送给一个目标器设备。你要做的第一件事情是分配一个 SRB，并指出这个 SRB 描述了哪一个 ASPI 命令。我们还为从设备接收查询数据分配了一个小的数据缓冲区。

```
SRB_ExecSCSICmd srb;                      // 分配SRB
unsigned char buf[128];                     // 分配数据缓冲区
memset(&sr, 0, sizeof(srb));                // 清除所有的域
sr.SRB_Cmd = SC_EXEC_SCSI_CMD;              // 指定ASPI命令
```

下一步，你需要提供一个将要接受这个命令的 SCSI 设备的地址。假设你知道主机适配器的编号、该设备的 SCSI ID 和逻辑单元号，那么一切都很简单。

```
sr.SRB_HaId = HostAdapterNumber;
sr.SRB_Target = TargetScsiId;
sr.SRB_Lun = 0;
```

此外我们还需要提供 SCSI CDB 命令描述块。对这个 Inquiry 命令的例子来说，这一步如下所示。

```
srb.SRB_CDBLen = 6;                      // 查询命令为6字节
srb.SRB_CDBByte[0] = 0x12;                 // 查询命令操作码
srb.SRB_CDBByte[1] = 0;                   // LUN和页标志
srb.SRB_CDBByte[2] = 0;                   // 查询页代码
srb.SRB_CDBByte[3] = 0;                   // 保留字节
srb.SRB_CDBByte[4] = sizeof(buf);         // 分配长度
srb.SRB_CDBByte[5] = 0;                   // 控制字节
```

读者若想获得关于 CDB 的信息，请回到第 5 章，参见对 SCSI 命令 Inquiry 的叙述。

接下来，由于 Inquiry 命令把数据返回给了我们，我们必须告诉 ASPI 管理器把这些数据放在哪里。另外还要设置 SRB\_DATA\_IN 标志，告诉 ASPI 管理器我们希望从设备那里接收到数据。如果我们正在把数据发送给设备，那么我们就设置 SRB\_DATA\_OUT 标志。

```
srb.SRB_BufLen = sizeof(buf);           // 缓冲区长度
srb.SRB_BufPointer = buf;               // 地址
srb.SRB_Flags |= SRB_DATA_IN;          // 传输方向
```

最后，我们还要告诉 ASPI 管理器，在 SCSI 的 Check Condition 状态下，返回的检测数据的数目。SRB\_ExecSCSICmd 结构包含了为检测数据准备的一个缓冲区，但我们仍然要指定这个缓冲区的大小。这个值有可能是不真实的，因为 Windows 95 和 Windows NT 的 Bug 把这个缓冲区的实际大小限制为 14 个字节。我们一般使用已定义的值 SENSE\_LEN，它被置为 16 字节。

```
Srb.SRB_SenseLen = SENSE_LEN;
```

通过以上步骤，现在我们已经为把 SRB 发送到 ASPI 管理器做好了准备。如果你已经看过了前面 SRB 的定义，你可能已经注意到了我们的讨论跳过了 SRB\_PostProc 字段。我们将在后面讨论它。

## ■ 给 ASPI 管理器发送一个 SRB (SCSI 请求块)

在 SRB 已经被正确地初始化后，你就可以简单地把它的地址传递给 ASPI 管理器，ASPI 管理器将会处理实际的 SCSI 命令的所有阶段和数据传输。这是十分简单的——你只需调用 ASPI 管理器的入口点，把 SRB 的地址作为一个参数传递过去。

```
SendASPICommand((LPSRB) &srb);
```

ASPI 使命令的实际执行和请求一个功能调用一样简单。是的，在执行方面就是这么简

单, ASPI 管理器通常只是把 SRB 放在一个队列中, 并且立即返回调用值, 甚至在命令开始执行之前。这就允许你的应用程序在一个命令还在执行的时候就可以为下一个命令做好准备。ASPI 管理器在后台处理执行这些命令过程中的所有细节, 然后在命令完全执行完成后更新 SRB 中的状态字段。

## ■ 等待 SRB 完成

由于 ASPI 管理器可能在一个 SRB 完成之前返回, 所以在 SRB 完成之前, 你的应用程序一定不能分配 SRB 或依赖于任何返回数据。保证这一点最简单的方法是设置一个循环, 等待 SRB\_Status 字段指示出 SRB 已经完成。

```
while (srb.SRB_Status == SS_PENDING)           // 如果部分未解决, 就不做任何处理
```

对状态字段进行轮询(Polling)来等待一个命令完成是一种效率很低的方法。在 MS-DOS 环境下, 这并不算什么, 因为在任何一个给定的时间段内, 只有一个应用程序在运行。然而, 在多任务操作系统中, 查询方式浪费了 CPU 的时间, 这些时间本来可以用于运行其他的线程和应用程序。幸运的是, ASPI 提供了另外一种等待的方法, 叫做记录(Posting), 它就是对你在 SRB 中指明的例行程序的一次回调。当 ASPI 管理器处理完 SRB, 它将调用你的例行程序, 把 SRB 的地址送入堆栈内, 这个回调例行程序能够检查 SRB 的状态字段, 并且可能立即执行另外一个 SRB。这个回调例行程序在 SRB 完成之后被尽快调用, 在某些情况下, 甚至从 ASPI 管理器的中断处理程序内也可以调用。正是因为这样, 你就要保证你的回调例行程序简单短小。而且, 因为回调可能发生在中断时间, 所以你不能使用任何不完全可重入的操作系统服务。这意味着你的回调例行程序常常只限于检查刚完成的 SRB, 并且可能为处理发送一个新的 SRB 给 ASPI 管理器。这是一个接受 ASPI 通知的简单而高效的方法, 但是在 Win32 下, 它要求 ASPI 管理器建立一个后台线程来对这个回调进行管理。这是你在这种环境下可以使用的另外一种方法, 我们将在后面对它进行讨论。

当使用回调例行程序涉及到 ASPI 管理器使用的调用约定的时候, 会有一个常见的问题。Windows 95 和 Windows NT 的 ASPI 管理器希望回调例行程序使用标准的 CDECL 调用约定, 它等价于大多数 Win32 编译器中的\_stdcall 调用约定。在 Windows 3.x 环境下, 你应该使用 FAR PASCAL 约定, 就像大多数的 16 位 Windows 回调和入口点一样。在 MS-DOS 环境下, 情况就更加复杂了。ASPI 管理器把刚完成的 SRB 地址放到堆栈中, 然后对回调例行程序作一个 FAR 调用, 如下面所示:

```
push [SrbOffset]          ; 保存SRB的偏移地址
push [SrbSegment]         ; 保存SRB的段基址
call dword ptr [SRB_PostProc] ; 调用复查例程
add sp, 4                 ; 清除栈
```

这对应于大多数 MS-DOS 下的编译器使用的 C 语言的调用约定。然而, MS-DOS 的 ASPI

规范说明书也要求回调例行程序保留所有的寄存器的值。因而，一个在 MS-DOS 环境下的回调例行程序可能是这样的：

```
CallbackRoutine:
    push    bp
    mov     bp, sp
    pusha
    push    ds
    push    es
    les    bx, dword ptr [bp+6]
    ; Now ES:BX has the address of the SRB

    pop     es
    pop     ds
    popa
    pop     bp
    retf
```

幸运的是，大多数的 MS-DOS 编译器已经实现了扩展，从而允许你自动地指定对编译器的需要。关于细节内容，可以查阅一下你的编译器文档，其中大多数的编译器支持下面的形式：

```
void __cdecl __saverregs __loadss CallbackRoutine( SRB _far *P )
{
}
```

如果你使用的是另外一种编程语言，或者你的编译器不支持这种扩展，那么你可以用汇编语言写一个小的子程序，就像上面的那样，简单地把调用转换为你的编译器所需要的格式。

让我们看一下在 Windows 95 和 Windows NT 下怎么使用回调程序。我们的例子将简单地给一个 Win32 EVENT 对象发信号。这是一个为此专门编写的例子，但是它的确显示了如何建立一个回调。

我们要做的下一件事情是建立事件对象和回调程序本身：

```
HANDLE EventHandle;
Void CallbackRoutine( SRB *P )
{
    SetEvent(EventHandle);
}
```

让我们假设我们已经初始化了一个 SRB 来执行一个 SCSI 的 Inquiry 命令，就像前面所说的，简单地填满丢失的部分并把它发送到 ASPI 管理器。

```

Srb.PostProc = CallbackRoutine;
Srb.Flags |= SRB_POSTING;
ResetEvent(EventHandle);
dwStatus = SendASPI32Command((LPSRB) &srb);
if (dwStatus == SS_PENDING)
    WaitForSingleObject(EventHandle, INFINITE);

```

我们用 Windows 95/NT 的 WaitForSingleObject() 服务来阻塞 (block) 我们的线程，直到 SRB 完成。在那时 ASPI 管理器将调用我们的回调程序，它将依次给我们正在阻塞的线程发信号。净结果和查询的结果一样，但是我们并没有浪费处理器的时间。

当控制像磁带驱动器和 CD-ROM 记录器这样的流式设备的时候，回调程序也是很有用的，在流式设备中，存储介质是经常移动的，你必须在一个很短的时间周期里发出下一条 Read 或 Write 命令。如果下一条命令发送得太晚，磁带驱动器就需要重新定位（花费一定的时间），或者 CD-ROM 记录器将会用完要写的数据（浪费光盘）。处理这种情况的一种常用方法是准备多个 SRB，把它们放入前台的队列里，然后用一个回调程序发送下一个要执行的 SRB 给后台的 ASPI 管理器，只要前面的命令完成。这就提供了一种简单而又有效的多任务形式。

回调程序可能难度很高，而且是难以调试的，所以人们并不经常使用它们，除非是对性能要求很高的情况。许多应用程序仍然用查询的方法检查命令是否完成，虽然这很浪费 CPU 的时间。为了使事情对程序员来说变得更容易，Windows 95 和 Windows NT 下的 ASPI 的实现提供了第三种方法用于等待，称为事件通知。不是指定一个回调程序的地址，而是提供一个 Win32 事件对象的句柄，当命令完成后，该事件对象能被自动发信号通知。在开始 SRB 后，你就能够简单地等待这个事件，并且你的线程将不花费 CPU 的时间。我们修改一下上面的例子：

```

Srb.PostProc = EventHandle;
Srb.Flags |= SRB_EVENT_NOTIFY;
ResetEvent(EventHandle);
dwStatus = SendASPI32Command((LPSRB) &srb);
if (dwStatus == SS_PENDING)
    WaitForSingleObject(EventHandle, INFINITE);

```

注意到回调程序和事件通知是相互排斥的，不要在 SRB\_Flags 字段内同时设置 SRB\_EVENT\_NOTIFY 和 SRB\_POSITION 位。在回调和事件通知之间要作出一个选择，后者更简单和有效。

## ■ 处理返回状态信息

在处理 SRB 过程中的最后一步涉及到检查状态字段。对 SCSI I/O 命令来说，它可以包含至少四个独立的字段：SRB\_Status、SRB\_HaStat、SRB\_TargStat 和 SRB\_SenseArea 缓冲

区。我们在本章后面的 ASPI 错误和状态码部分叙述在这些字段中的返回值。到目前为止，你应该记得事情可能会出错，而这些字段将告诉你发生了什么。

另外一点需要注意的是，如果你使用回调函数或事件通知，在 SRB 的执行完成之前，状态字段不一定是有效的。这意味着在执行回调或者触发事件以前，你不应该查询状态字段来获得信息。

## ■ 适配器特定属性

ASPI 接口努力为你管理大多数的硬件特定细节，但是一些适配卡有数据缓冲区调整和最大长度限制，它们必须由应用程序来调整。例如，许多基于 PC 的 SCSI 适配器能够在一个命令里传输最长为 65536 字节的数据。应用程序必须识别出这种限制，并且保证它们在所限范围内。不幸的是原来的 ASPI 规范说明书没有提供一种方法来为特定的适配卡确定这些信息。一份 1994 年 2 月的 ASPI 规范说明书的附录详细描述了收集硬件特定信息的方法，但是你必须注意确定一个特定的 ASPI 管理器支持这种扩展。Win32(Windows 95 和 Windows NT) 规范说明书的 ASPI 为适配器特定信息的标准集定义了主机适配器标识符字段的一部分，它简化了检测过程，但是在其他操作系统下不一定和旧的 ASPI 管理器兼容。更多的信息可以参见对主机适配器查询命令的描述。

## ■ 和 ASPI 管理器的连接

正如前面所提到的，ASPI 管理器的实际实现随着操作系统的不同而不同。在 MS-DOS 和 OS/2 环境下，ASPI 管理器是作为设备驱动程序实现的；而在 Windows 环境下，ASPI 管理器是作为 DLL 实现的。不同的 Windows 实现也可以使用附加的组件 (VxD 或设备驱动程序) 来处理低层操作系统和硬件接口函数。对大多数来说，实现上的不同对应用程序员来说是不重要的，除了连接到 ASPI 管理器的方法不同。在 Windows 系统下，你只要简单地把你的应用程序连接到 ASPI 管理器的引入库 (WINASPLIB 或 WNASPI32.LIB)，并且依赖于 Windows 的动态连接机制来连接。对 MS-DOS 和 OS/2 系统来说，你必须打开设备驱动程序并且发送一个 IOCTL 调用来获得 ASPI 管理器的入口点的地址。

不要担心，这些听起来很复杂，但实际上很简单。在 16 位的 Windows 系统中，你可以用程序清单 7-1 中的代码来连接 ASPI 管理器。

---

### 程序清单 7-1 在 16 位 Windows 中与 ASPI 连接

---

```
WORD AspiStatus;
BYTE NumAdapters;

AspiStatus = GetASPISupportInfo();

switch (HIBYTE(AspiStatus))
```

```

{
case SS_COMP:
    /* ASPI已经进行了正确的初始化并正在运行状态标志的低字节中 */
    /* 保存了在ASPI下的主机适配器的数量 */
    NumAdapters = LOBYTE(AspiStatus);
    break;

case SS_ILLEGAL_MODE:
    /* ASPI不能在当前运行的Windows模式中
       正常使用(实模式、标准模式、增强模式) */
    NumAdapters = 0;
    break;

case SS_OLD_MANAGER:
    /* 安装的MS-DOS ASPI管理器不支持Windows, 有些 */
    /* MS-DOS下的ASPI管理器可以在Windows下使用, 但是 */
    /* 大多数并不支持虚拟DMA服务或者是与Windows不兼容。 */
    NumAdapters = 0;
    break;

default:
    /* 产生了某种错误, 所以不再尝试 */
    NumAdapters = 0;
}

```

在这里最常见的问题是忘记把 WINASPLIB 引入库连接到你的应用程序内。所以连接程序说它不能找到 GetASPISupportInfo() 程序。这个处理对 32 位的 Windows 系统是一样的，除非你连接了 WNASPI32.DLL 引入库，称为 GetASPI32SupportInfo()，并且你不必担心兼容的 MS-DOS ASPI 管理器。程序清单 7-2 是在 32 位 Windows 中连接 ASPI 的代码。

---

### 程序清单 7-2 在 32 位 Windows 中连接 ASPI

---

```

DWORD AspiStatus;
BYTE NumAdapters;

AspiStatus = GetASPI32SupportInfo();

switch (HIBYTE(LOWORD(AspiStatus)))
{
case SS_COMP:
    /* ASPI已经正确初始化并正在运行, 状态标志的 */
    /* 低字节中包括所安装的主机适配器的数目 */
    NumAdapters = LOBYTE(AspiStatus);
}

```

```

break;

default:
/* 当前并未安装ASPI管理器 */
NumAdapters = 0;
}

```

这些例子假设你正在把 WINASPI.LIB 或 WNASPI32.LIB 引入库连接到你的应用程序内。如果不这么做，你就不能使用 ASPI 管理器，但是你必须使用 WindowsLoadLibrary( )程序显式地把它装入，并用 GetProcAddress( )程序获得 GteASPISupportInfo( )程序的地址。这个过程有点复杂，但是即使在 ASPI 管理器没有被安装的情况下你的程序也能够被启动，从而允许你给用户提供更加具体的信息，而不仅仅是标准的 Windows 消息“couldn't find a required DLL”。你也可以像程序清单 7-3 一样动态地装入 Win32 ASPI 管理器。

---

### 程序清单 7-3 在 16 位 windows 中动态装载 ASPI

---

```

DWORD (*GetASPI32SupportInfo)();           /* 函数指针 */
DWORD (*SendASPI32Command(LPSRB lpSrb));  /* 函数指针 */
HANDLE WnAspiHandle;
DWORD AspiStatus;
BYTE NumAdapters;

WnAspiHandle = LoadLibrary("WNASPI32.DLL");
if (WnAspiHandle)

{
    GetASPI32SupportInfo = GetProcAddress("GetASPI32SupportInfo");
    SendASPI32Command = GetProcAddress("SendASPI32Command");
    if (GetASPI32SupportInfo & SendASPICommand)
    {
        AspiStatus = GetASPI32SupportInfo();
        switch (HIBYTE(LOWORD(AspiStatus)))
        {
            case SS_COMP:
                /* ASPI已经正确安装并正在运行，状态标志的*/
                /* 低字节中包括了安装的主机适配器的数目 */
                NumAdapters = LOBYTE(AspiStatus);
                break;

            default:
                /* 当前并未安装ASPI管理器 */
                NumAdapters = 0;
        }
    }
}

```

```

    }
else
{
    /* 不能查找到GetASPI32SupportInfo() */
    /* 或SendASPI32Command的地址, 可能是由于ASPI没有正确安装 */
}
}

else
{
    /* 不能装载WNASPI32.DLL */
    /* 确保该文件已经正确安装 */
}

```

在 MS-DOS 系统下, 你必须首先打开 ASPI 管理器的设备驱动程序并得到 ASPI 管理器的入口点地址。这可以用程序清单 7-4 中的代码做到:

---

#### 程序清单 7-4 在 MS-DOS 中对 ASPI 进行初始化

---

```

.DATA
AspiEntryPoint DD 0          ; 入口点的地址
AspiHandle      DW 0          ; 文件处理器
AspiDriverName  DB "SCSIMGR$",0 ; ASPI设备的名称

.CODE
GetAspiEntryPoint PROC
    push ds                ; 保存当前的数据段
    mov ax,@DATA             ; 装载当地的数据段
    mov ds,ax
    lea dx,AspiDriverName   ; 装载驱动器名称的偏移地址
    mov ax,3D00h              ; 以MS-DOS方式打开文件
    iny                      ;
    jc failed                ;
    mov [AspiHandle],ax       ; 保存文件处理器

    mov bx,[AspiHandle]        ; 装载文件处理器
    lea dx,AspiEntryPoint     ; 缓冲区的地址
    mov cx,4                  ; 长度=4字节
    mov ax,4402h              ; MS-DOS下的IOCTL读命令
    int 21h                   ;
    jc failed                ;

    mov bx,[AspiHandle]        ; 装载文件处理器
    mov ax,3E00h              ; MS-DOS下关闭文件
    int 21h                   ;

```

```

    mov ax,word ptr [AspiEntryPoint] ; 返回ASPI入口点的地址
    mov dx,word ptr [AspiEntryPoint+2] ;
    pop ds
    ret

failed:
    mov ax,0                         ; 发生错误返回NULL
    mov dx,0
    pop ds
    ret
GetAspiEntryPoint ENDP

```

接着应用程序就能用程序清单 7-5 中的顺序来连接 ASPI 管理器。

---

### 程序清单 7-5 在 MS-DOS 下连接 ASPI

---

```

BYTE NumAdapters;
WORD (FAR *SendASPICommand)(void FAR *pSrb); /* 函数指针 */

SendASPICommand = GetAspiEntryPoint();
if (SendASPICommand)
{
    SRB_HA_Inquiry HostAdapterInfo;
    memset( &HostAdapterInfo, 0, sizeof(HostAdapterInfo) );
    HostAdapterInfo.SRB_Cmd = SC_HA_INQUIRY;
    HostAdapterInfo.SRB_HaId = 0; /* 第一个主机适配器 */
    SendASPICommand( (LPSRB) &HostAdapterInfo );
    switch (HostAdapterInfo.SRB_Status)
    {
        NumAdapters = HostAdapterInfo.HA_Count;
        break;

        default:
            /* 产生了某种错误 */
            NumAdapters = 0;
    }
}

else
{
    /* ASPI管理器尚未安装 */
    NumAdapters = 0;
}

```

上面的 MS-DOS 的例子实际上使用 ASPI 管理器来执行一个主机适配器查询命令接收安装的主机适配器的数目。这是用 MS-DOS 代码来模拟 Windows GetASPISupportInfo( )程序，它也返回已经安装的主机适配器的数目（在返回值的最低有效字节）。

## ASPI 命令

正如上面所提到的，所有对 ASPI 管理器的请求通过一个 SendASPICommand( ) 函数发送，该函数使用一个指向一般 SRB SCSI 请求块的指针作为唯一的参数。在 SRB 中的前几个字段对所有的 ASPI 命令都是通用的。这些字段包含一个通用代码、一个主机适配器号、一个状态字段和一个有不同控制标志的字段。在 SRB 中剩下的字段对那个命令是特定的，并且包含了对那个命令适用的信息和值。通用的 SRB 字段如下。

```
BYTE    SRB_Cmd;
BYTE    SRB_Status;
BYTE    SRB_HaId;
BYTE    SRB_Flags;
DWORD   SRB_Hdr_Rsvd;
/* 后面是命令代码段 */
```

SRB\_Cmd 字段定义了 ASPI 命令和任何命令特定数据的格式；SRB\_Status 字段返回命令的状态，不管它是有待执行的、完成的还是遇到了错误，在 SRB\_Status 状态字段中可能的返回值将在后面的 ASPI Error 和 Status Code 部分叙述；SRB\_HaId 字段表示哪个主机适配器将会处理这个请求；SRB\_Flags 字段包含了应用于该命令的所有的位标志。

在 SRB\_Cmd 字段中可能的 ASPI 命令值如表 7-1 所示。

表 7-1 ASPI 命令值

值	命 令	说 明
0	SC_HA_INQUIRY	取得关于已经安装的主机适配器硬件的信息，包括安装的主机适配器的数目
1	SC_GET_DEV_TYPE	标识 SCSI 总线上可利用的设备
2	SC_EXEC_SCSI_CMD	执行一个 SCSI I/O 命令
3	SC_ABORT_SRBC	请求异常中止一个有待执行的 SRB
4	SC_RESET_DEV	把 SCSI 总线设备复位信号发送给目标器设备
5	SC_SET_HA_INFO (在 Windows95 或 NT 上无效)	设置主机适配器特定信息。（这个命令对某个主机适配器/ASPI 管理器的结合是特定的，并且不由应用程序编制者发送）

续表

值	命 令	说 明
6	SC_GET_DISK_INFO (在 WindowsNT 上无效)	取得关于一个磁盘设备的 INT 13 驱动器和几何结构映射的信息。(这个命令只有在与 MS-DOS 兼容的操作系统中才可以用, 这些操作系统支持磁盘驱动器的 Int 13 服务)
7	SC_RESCAN_SCSI_BUS	重新扫描与给定的主机适配器相连的 SCSI 总线。这就使 ASPI 管理器能够知道新连接的和移除的设备, 并据此来更新它的内部表(这条命令只能用于 Windows 95 和 Windows NT 环境下的 ASPI 的 Win32 实现)
8	SC_GETSET_TIMEOUTS	设置或取得 SCSI 命令对于一个指定的目标器设备的超时值(这条命令只有在 Windows 95 和 Windows NT 的 ASPI 的 Win32 实现下才是有效的)

SC\_HA\_INQUIRY 和 SC\_GET\_DEV\_TYPE 命令用于获得关于安装的主机适配器和 SCSI 设备的信息, 它们典型地被用来确定在一个系统上哪些设备是可利用的; SC\_EXEC\_SCSI\_CMD 命令用于所有的 SCSI 命令, 也是使用频率最高的命令; 当试图从超时或错误状态中恢复的时候, 就可以使用 SC\_ABORT\_SRQ 和 SC\_RESET\_DEV 命令; SC\_SET\_HA\_INFO 和 SC\_GET\_DISK\_INFO 命令一般不被应用程序编制人员使用, 所以我们不对它进行进一步的讨论。

### ■ 主机适配器查询命令 (SC\_HA\_INQUIRY)

主机适配器查询命令用于获得关于安装的 SCSI 主机适配器和 ASPI 管理器的信息。你对 ASPI 管理器的调用首先应该是主机适配器查询命令以确定安装的 SCSI 主机适配器的数目及它们的能力。主机适配器用数字指定, 数字的最小值是 0, 它被传递到 SRB\_HaId 字段, 信息被返回到 HA\_\* 字段, 如程序清单 7-6 中所示。

---

#### 程序清单 7-6 主机适配器查询 SRB

---

```
typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_HA_INQUIRY
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器号(0 ~ N)
    BYTE    SRB_Flags;         // 请求标志, 为 0

    DWORD   SRB_Hdr_Rsvd;      // 保留, 必须为0
    BYTE    HA_Count;          // 主机适配器的总数
    BYTE    HA_SCSI_ID;        // 特定主机适配器SCSI ID
}
```

```

BYTE    HA_ManagerId[16];      // ASPI管理器的ASCII描述字串
BYTE    HA_Identifier[16];     // 主机适配器的ASCII描述字串
BYTE    HA_Unique[16];         // 主机适配器的唯一参数
BYTE    HA_Rsvd1;             // 保留
} SRB_HAInquiry;

```

你所发送的第一个主机适配器查询命令应该指定主机适配器号 (SRB\_HaId) 为 0。这将读取关于第一个 SCSI 主机适配器的信息，它还将给出你安装在这个系统中的所有的主机适配器的数目。如果存在一个以上的主机适配器，你可以像程序清单 7-7 一样对别的主机适配器发出另外的主机适配器查询命令。

---

### 程序清单 7-7 主机适配器查询调用

---

```

SRB_HAInquiry HostAdapterInfo;
memset( &HostAdapterInfo, 0, sizeof(HostAdapterInfo) );
HostAdapterInfo.SRB_Cmd = SC_HA_INQUIRY;
HostAdapterInfo.SRB_HaId = 0; /* 第一个主机适配器 */
SendASPICommand( (LPSRB) &HostAdapterInfo );

switch (HostAdapterInfo.SRB_Status)
{
    case SS_COMP:
        /* ASPI管理器已经正确安装并正在运行 */
        NumAdapters = HostAdapterInfo.HA_Count;
        break;
    default:
        /* 产生了某种错误 */
        NumAdapters = 0;
}

```

让我们仔细地看一下 SRB 结构的字段，如表 7-2 所示。

表 7-2 主机适配器查询字段

字 段	说 明
SRB_Cmd	ASPI 命令码 该字段必须包含 SC_HA_INQUIRY 以获得后面的信息

续表

字 段	说 明
SRB_Status	<b>ASPI 命令状态</b> 这个命令用来保存有待执行的和完成的 ASPI 命令的状态。在返回时，这个字段将包含以下值之一： SS_COMP：无错误完成命令 SS_INVALID_HA：无效主机适配器号 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	<b>主机适配器索引</b> 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器；1 表示第二个主机适配器，依此类推。有些主机适配器支持多 SCSI 总线，在这种情况下，这个字段中的值实际上指的是一个逻辑总线索引，而不是一个物理主机适配器的索引。如果你指定的索引在范围之外，那么 SRB_Status 字段将返回一个 SS_INVALID_HA 状态代码。你可以确定在 ASPI 管理器下的可用的主机适配器的总数目，只要把这个字段置为 0，并察看在 HA_Count 字段中的返回值
SRB_Flags	<b>ASPI 命令标志</b> 对这个命令，没有标志是有效的，这个字段应该被置为 0
SRB_Hdr_Rsvd	这个字段是保留的，应该被置为 0
HA_Count	<b>主机适配器的数目</b> 这个字段返回在 ASPI 管理器下的主机适配器的总数目。有些主机适配器支持多 SCSI 总线，在这种情况下，这个字段中的值实际上指的是安装的逻辑 SCSI 总线数目，而不是物理主机适配器卡的数目。在大多数情况下，这种区别并不重要，因为你只要把在一块卡上的多个 SCSI 总线看作它们是在不同的卡上就可以了
HA_SCSI_ID	<b>主机适配器的 SCSI ID</b> 这个字段返回指定的主机适配器的 SCSI ID
HA_MangerId	<b>ASPI 管理器标识符字符串</b> ASPI 管理器在这个字段返回一个 ASCII 字符串，它描述了这个 ASPI 管理器。典型代表是，这个字符串描述了这个 ASPI 管理器的开发者。注意这个字符串不能以空字符结束
HA_Identifier	<b>主机适配器的标识符</b> 在这个字段中返回一个 ASCII 字符串，它描述了指定的主机适配器。通常情况下，这个字符串描述了这个主机适配器类型或模型号。注意这个字符串不能以空字符结束
HA_Uneque	<b>主机适配器唯一参数</b> 这个字段返回描述指定主机适配器的特性和需求的各种信息。进一步的信息见下文

HA\_Uneque 字段一开始是为某个特定 ASPI 管理器实现的特定的信息保留的。在 Windows 95 和 Windows NT 下，这个字段的一部分用于返回关于主机适配器能力的信息。

表 7-3

主机适配器特征参数

位 #	描述
0~1	<p>缓冲区对齐掩码 该值指出主机适配器的缓冲区对齐要求。</p> <p>0: 无对齐要求 1: 要求字对齐 3: 双字对齐 7: 四字对齐</p> <p>因此,可以通过缓冲区地址和该掩码进行“与”操作的办法来检查缓冲区对齐。如果结果为“0”,表示缓冲区被正确地对齐了。你必须确保发送到 ADPI 管理器的任何数据缓冲区符合对齐要求</p>
2	<p>适配器唯一标志 位 0: 保留 位 1: 支持残留字节计数 位 2~7: 保留</p> <p>如果 1 位被置为“1”,那么适配器就支持残留字节计数报告。这意味着在 SCSI I/O 传输(通过 SC_EXEC_SCSI_CMD 命令)之后,SRB_BufLen 字段可以被更新,用来指出在一次传输完成之后,还剩多少字节没有传输。这在缓冲区不满的时候很有用,因为你可以确定命令实际传输了多少字节。参见 SC_EXEC_SCSI_CMD 命令的描述,以获得更多信息</p>
3	<p>最大数目的 SCSI 目标器支持 该值指出给定主机适配器所支持的 SCSI 目标器最大数目。如果该值为“0”,你就应该假定最多有 8 个目标器</p>
4~7	最大值表示一条 SCSI I/O 命令在指定适配器上能传输的最大字节数。字节 7 是最重要的字节
8~15	保留

当可以获得 HA\_Unique 时,使用它的信息是很重要的。一些主机适配器和驱动程序在用一条 SCSI 命令进行数据传输时,它的最大数据传输量是有限制的,这个限制反映在 HA\_Unique 数据的最大传输长度字段。在很多基于 PC 的主机适配器上,常见的限制是 64K,而且,这种限制有可能由于使用的操作系统而进一步减少。你可能不得不把传输分为几个更小的块,这样就可以消除这种限制。

缓冲区队列掩码在新的硬件平台上变得更加重要了。由于对速度永不停止的追求,所以很多系统总线和主机适配器的设计使用复杂的电路来增加在数据被正确地排队后的数据传输次数。很多系统内存的缓冲区和总线是为 64 位、128 位甚至更大的数据突发传输作优化的。当你的数据缓冲区队列定位在自然总线大小的边界上,传输可能是非常快的。然而,不正确定位的数据经常要求附加的总线周期来通过总线传输,它常常会明显地降低传输速度。

## ■ 获取设备类型命令 (SC\_GET\_DEV\_TYPE)

获取设备类型命令用来获取与一个主机适配器相连的指定的目标器设备的基本信息。这个主机适配器由数字指定，这些数字的最小值是 0，放在 SRB\_HaId 字段内，目标器设备的 SCSI ID 在 SRB\_Target 字段中被指定，目标器设备的 LUN 在 SRB\_Lun 字段中指定。如果指定的逻辑单元在目标器上存在，它的设备轮询标识符将返回到 SRB\_DeviceType 字段。该命令使你可以快速确定一个指定的目标器设备是否可用。

---

### 程序清单 7-8 获取设备类型 SRB

---

```
typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_GET_DEV_TYPE
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器编号
    BYTE    SRB_Flags;         // 请求标志应该为0
    DWORD   SRB_Hdr_Rsvd;     // 保留必须为0
    BYTE    SRB_Target;        // 设备的SCSI ID (通常为0~7)
    BYTE    SRB_Lun;           // 设备的逻辑单元号
    BYTE    SRB_DeviceType;    // 返回SCSI设备类型
    BYTE    SRB_Rsvd1;         // 校正保留
} SRB_GDEVBlock;
```

你应该用这条命令确定一个指定的目标器/LUN 设备是否存在。大多数的 ASPI 管理器在第一次被装入时，将会扫描 SCSI 总线，并把在该系统中找到的每一个设备的信息保存下来。Get Device Type 命令典型地使用这些保存的信息，实际上不必再次对这些设备进行查询(在不连接的目标器/LUN 设备上会引起超时)。应用程序能够在一个循环中发送 Get Device Type 命令，为每一个适配器/目标器/LUN 的一次可能的组合建立一个和你的系统相关的设备的列表。

---

### 程序清单 7-9 获取设备类型调用

---

```
SRB_GDEVBlock srb;
int adapter, target;
for (adapter=0; adapter<NumAdapters; adapter++)
{
    for (target=0; target<NumDevices; target++)
        /* 主机适配器请求的设备号 */
```

```

    {
        memset(&srb, 0, sizeof(srb));
        srb.SRB_Cmd = SC_GET_DEV_TYPE;
        srb.HaId = adapter;
        srb.Target = target;
        srb.Lun = 0;
        SendASPI32Command((LPSRB)&srb);
        if (srb.SRB_Status == SS_COMP)
        {
            /* 发现一个设备，它的类型在Srb.SRB_DeviceType域 */
        }
        else
        {
            /* 设备不存在或没有响应 */
        }
    }
}

```

让我们更加详细地看一下 SRB\_GET\_DEV\_TYPE 命令的 SRB 字段，如表 7-4 所示。

表 7-4

获取设备类型字段

字 段	说 明
SRB_Cmd	ASPI 命令码 这个字段必须包含 SC_GET_DEV_TYPE，以获得下面的信息
SRB_Status	ASPI 命令状态 这个字段用来保存有待执行的和完成的 ASPI 命令状态。在返回时，这个字段将包含下面的值： SS_COMP：没有错误 SS_INVALID_HA：主机适配器号 SS_NO_DEVICE：设备未安装 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	主机适配器索引 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器；1 表示第二个主机适配器，依此类推。参见关于主机适配器查询命令的描述以获得进一步的信息
SRB_Flags	ASPI 命令标志 对这个命令，没有标志是有效的，这个字段应该被置为 0
SRB_Hdr_Rsvd	这个字段是保留的，应该被置为 0
SRB_target	目标器设备的 SCSI ID 这个字段指出了查询设备的 SCSI ID。典型的值是 0~7。一些主机适配器允许多于 8 个目标器在一个总线上，所以一定要检查主机适配器查询命令返回的信息，以确定每个主机适配器的这个字段的最大的可能值。

续表

字 段	说 明
SRB_Lun	目标器设备的逻辑单元号 这个字段指出了查询设备的 LUN 逻辑单元号。某些 SCSI 外围设备在一个 SCSI ID 上有几个单独的设备单元。这些子单元由它们的 LUN 来标识，并且这个字段是可选择的。如果设备只有一个逻辑单元，就把这个字段设为 0
SRB_DeviceType	设备类型 ID 这个字段返回指定的目标器/LUN 的 SCSI 外围设备类型标识符。这个值通常和 SCSI Inquiry 命令返回的外围设备类型一样。你可能想发送一条 Inquiry 命令来确定
SRB_Rsvd1	定位保留 这个字段的值设为 0

### Execute SCSI Command (SC\_EXEC\_SCSI\_CMD)

Execute SCSI Command 命令给一个目标器设备发送一条 SCSI I/O 命令。你只需提供一个 SCSI 命令描述块 (CDB) 和一个数据缓冲区，ASPI 管理器将会完成剩下的事情。所有的 SCSI 总线的复杂性都被隐藏在这条 ASPI 命令之后。

---

#### 程序清单 7-10 执行 SCSI 命令 SRB

---

```

typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_EXEC_SCSI_CMD
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 请求的主机适配器号 (0 ~ N)
    BYTE    SRB_Flags;         // 请求标志，参考下文
    DWORD   SRB_Hdr_Rsvd;      // 保留必须为 0
    BYTE    SRB_Target;        // 目标器的SCSI ID (通常为0 ~ 7)
    BYTE    SRB_Lun;           // 设备的逻辑单元号
    WORD    SRB_Rsvd1;         // 校正保留
    DWORD   SRB_BufLen;        // 数据缓冲区长度
    BYTE    *SRB_BufPointer;   // 数据缓冲区地址
    BYTE    SRB_SenseLen;       // 检测缓冲区长度
    BYTE    SRB_CDBLen;         // CDB 长度
    BYTE    SRB_HaStat;         // 主机适配器状态 (返回的)
    BYTE    SRB_TargStat;       // 目标器状态 (返回的)
    void(*SRB_PostProc)(LPSRB); // 发送例程地址
    BYTE    SRB_Rsvd2[20];      // 保留
    BYTE    SRB_CDBByte[16];     // SCSI CDB
    BYTE    SRB_SenseArea[16];   // SCSI 检测数据缓冲区
} SRB_ExecSCSICmd;

```

我们来仔细地研究一下它的字段，如表 7-5 所示。

表 7-5 执行 SCSI 命令字段

字 段	说 明
SRB_Cmd	ASPI 命令代码 这个字段必须包含 SC_EXEC_SCSI_CMD，以执行一个 SCSI I/O 命令
SRB_Status	ASPI 命令状态 这个字段用来保存有待执行的和完成的 ASPI 命令状态。在返回时，这个字段将包含下面的值中的一个： SS_PENDING：仍在处理中 SS_COMP：没有错误 SS_INVALID_HA：主机适配器号 SS_ABORTED：被异常终止 SS_ERR：出错结束 SS_INVALID_SRIB：SRB 字段或标志是无效的 SS_INVALID_PATH_ID：目标器 ID 或 LUN 是无效的 SS_BUFFER_TOO_BIG：ASPI 管理器不能处理指定的缓冲区长度 SS_SECURITY_VIOLATION：在指定的设备上调用程序没有安全的优先权来执行 SCSI I/O 命令 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	主机适配器索引 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器；1 表示第二个主机适配器，依此类推。参见关于主机适配器查询命令的描述以获得进一步的信息
SRB_Flags	ASPI 命令标志 这个字段包含一套标志，它们控制 SRB 的执行。对这个命令有效的标志是： SRB_DIR_IN SRB_DIR_OUT (对于数据传输命令来说，SRB_DIR_IN 或 SRB_DIR_OUT 必须被置“1”) SRB_EVENT_NOTIFY SRB_POSTING SRB_ENABLE_RESIDUAL_COUNT
SRB_Hdr_Rsvd	这个字段是保留的，应该被置为 0
SRB_Target	目标器设备的 SCSI ID 这个字段指出了该命令访问的设备的 SCSI ID。一般的值是 0~7。一些主机适配器允许多于 8 个的目标器在一个总线上，所以一定要检查主机适配器查询命令返回的信息，以确定每个主机适配器的这个字段的最大的可能值
SRB_LUN	目标器设备的逻辑单元号 这个字段指出了该命令访问的设备的 LUN 逻辑单元号。某些 SCSI 外围设备在一个 SCSI ID 上有几个单独的设备单元。这些子单元由它们的 LUN 来标识，并且这个字段是可选择的。如果设备只有一个逻辑单元，就把这个字段设为 0

续表

字 段	说 明
SRB_Rsvd1	为定位保留 将该字段置“0”可兼容以后的 ASPI 版本
SRB_BufLen	数据缓冲区长度  这个字段中的值是为 SCSI I/O 命令设置的数据缓冲区大小，对于 Write 命令来说，是将被发送到目标器设备的数据字节数目；对 Read 命令而言，就是将要从目标器设备中读出的数据的最大字节数。如果 SCSI I/O 命令不包括数据传输，把这个字段置为 0。如果支持剩余字节数报告功能被支持并允许执行，这个字段就返回没有被 SCSI I/O 命令传输的字节数。你也可以把这个字段看作一个计数器，经过每次的数据传输后，不管传到还是传出目标器，它的数值都会减小。如果你想从目标器设备处读取 100 个字节，但是这个设备仅仅返回了 10 个字节，这个字段中的返回值将会是 90（如果剩余字节数报告功能被支持和允许）
SRB_BufPointer	数据缓冲区指针  这个字段中的值是 SCSI I/O 命令使用的数据缓冲区的地址，对于 Write 命令来说，数据将从这里发送到目标器设备；对于 Read 命令来说，数据将从目标器设备被发送到这个缓冲区
SRB_SenseLen	检测数据长度  这个字段的值为 SRB_SenseArea 字段的大小，后面的字段在本结构的最后。当一个目标器设备产生一个 Check Condition 状态后，ASPI 管理器自动发送一个 Request Sense 命令，并把检测数据放到 SRB_SenseArea 字段。大多数应用程序简单地把这个字段置为 14，调整后为 16 字节的缓冲区
SRB_CDBLen	CDB 的长度  这个字段的值为包含在 SRB_CDBByte[ ] 字段中的 CDB 的大小。这个值随发送的 SCSI I/O 命令的不同而变化，但是一般为 6、10 和 12。在 SCSI 事务处理的命令阶段 ASPI 管理器从 SRB_CDBByte[ ] 字段发送很多命令字节到目标器设备
SRB_HaStat	主机适配器状态  这个字段返回一个主机适配器的状态值。它指出在 SRB 的执行过程中主机适配器所遇到的错误状态。可能值如下：  HASTAT_OK HASTAT_SEL_TO HASTAT_DO_DU HASTAT_BUS_FREE HASTAT_PHASE_ERR HASTAT_TIMEOUT HASTAT_COMMAND_TIMEOUT HASTAT_MESSAGE_REJECT HASTAT_BUS_RESET HASTAT_PARITY_ERROR HASTAT_REQUEST_SENSE_PAILED  进一步的信息可以参考 ASPI 的错误代码和状态代码的有关部分

续表

字 段	说 明
SRB_TargStat	<p>目标器状态 这个字段返回由目标器设备发送的状态值，在 SCSI 命令结束的时候。在这个字段中可能的返回值在 SCSI 规范说明书中都作了定义</p> <p>0x00 正常 0x02 状态检查 0x04 状态符合 0x08 忙 0x10 中间 0x14 中间条件符合 0x18 保留冲突 0x22 命令中止 0x28 任务集满 0x30 偶然事件自动激活 进一步的信息可以参考 ASPI 的错误代码和状态代码的有关部分</p>
SRB_PostProc	<p>记入程序 如果你正在用命令记入，这个字段设置记入程序的地址；如果你在用事件通知，那么它就设置 Win32 事件信号的句柄。如果使用了以上两种方式中的任何一种，你就必须在 SRB_Flags 字段设置合适的位（SRB_POSITION 或 SRB_EVENT_NOTIFICATION）</p>
SRB_Rsvd2	这个字段是保留的，应该被置为 0
SRB_Rsvd3	这个字段是保留的，应该被置为 0
SRB_CDBByte	<p>命令描述符块 这个字段包含 SCSI 命令描述块(CDB)，在 SCSI 传输的命令阶段，它被发送到目标器设备</p>
SRB_SenseArea	<p>检测数据缓冲区 当目标器设备返回 Check Condition 状态时，ASPI 管理器自动取得设备检测数据，并把它放在这个字段内。检测数据描述了引起 Check Condition 状态的错误或异常。读者可以参考第 6 章以获得更多的信息</p>

## 口 终止 SRB 命令 (SC\_ABORT\_SRБ)

Abort SRB 命令用来异常终止一个有待执行的 SRB。如果一个命令在一个合理的时间内还没有完成，你就要用到这条命令。老版本的 ASPI 不直接实现或支持超时操作，但是最近的 Win32 实现有了一个 Get/SetTimeout 函数可以完成这项功能。

SCSI 命令的执行时间可能变化很大。如果你不采用超时功能，那么你的应用程序就不得不自己想办法来确定命令的执行时间是否已经过长。当你觉得某条命令的执行用去时间已经足够长以后，你可以发送 SRB 命令请求 ASPI 管理器有效地终止这条命令并且释放它使用的一切资源。

---

### 程序清单 7-11 终止 SRB 命令的 SRB

---

```
typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_ABORT_SRB
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器编号
    BYTE    SRB_Flags;         // 请求标志, 应该为0
    DWORD   SRB_Hdr_Rsvd;     // 保留, 必须为0
    void *SRB_ToAbort;        // 将要终止的SRB地址
} SRB_Abort;
```

SRB\_HaId 字段必须设置为和 SRB 中指定的主机适配器号一样, 该 SRB 就是你希望终止的, 并且 SRB\_ToAbort 字段必须指向那个 SRB。虽然你正在终止的 SRB 可能要到以后才能够完成, 但是在 ASPI 管理器返回到你的应用程序之前, 这条命令一定会完成。如果原来的 SRB 被成功地终止, 那么它的 SRB\_Status 字段将会被最终设置为 SS\_ABORTED。注意你仍旧必须等候原来的 SRB 完成。小心不要释放或者重用它, 直到它的 SRB\_Status 字段不再是 SS\_PENDING 为止。还要记住, 除非你正在使用查询, 否则在命令完成之前你不应该检查这个字段。

---

### 程序清单 7-12 终止 SRB 调用

---

```
HANDLE EventHandle;
SRB_ExecSCSICmd OriginalSrb;

/* 假设OriginalSrb已经进行了初始化 */
OriginalSrb.PostProc = EventHandle;
OriginalSrb.Flags |= SRB_EVENT_NOTIFY;
ResetEvent(EventHandle);
SendASPI32Command((LPSRB)&OriginalSrb);
if (WaitForSingleObject(EventHandle,timeout) == WAIT_TIMEOUT)
{
    SRB_Abort AbortSrb;
    AbortSrb.SRB_Cmd = SC_ABORT_SRB;
    AbortSrb.SRB_HaId = OriginalSrb.SRB_HaId;

    AbortSrb.Flags = 0;
    AbortSrb.Hdr_Rsvd = 0;
    AbortSrb.SRB_ToAbort = &OriginalSrb;
    SendASPI32Command((LPSRB) &AbortSrb);
    if (AbortSrb.SRB_Status != SS_COMP)
```

```

{
/* 产生了不可预知的错误 */
}
else
{
    while (OriginalSrb.SRB_Status == SS_PENDING)
    {
        /* 等待OriginalSrb完成应该设定一个附加的 */
        /* 超时值作为产生灾难性错误的一种安全阈值 */
    }
}
}

```

注意上面的代码不能够保证在 Windows 95 或 Windows NT 下工作。一旦在这些平台上把一个请求传送到小型端口驱动程序那里，它就不能被异常终止。

表 7-6 终止 SRB 字段

字 段	说 明
SRB_Cmd	ASPI 命令代码 这个字段必须包含 SC_ABORT_SRB 来执行 Abort SRB 命令
SRB_Status	ASPI 命令状态 这个命令用来保存有待执行的和完成的 ASPI 命令的状态。在返回时，这个字段将包含以下值之一： SS_COMP：无错误完成命令 SS_INVALID_HA：无效主机适配器号 SS_INVALID_SRB：一个 SRB 字段或标志是无效的 如果这个字段返回 SS_COMP，ASPI 管理器将会试图终止指定的 SRB；如果返回的是别的值，那么 ASPI 管理器将不会试图异常终止 SRB
SRB_HaId	主机适配器索引 这个字段必须包含和 SRB 中指定的同样的主机适配器号，该 SRB 是你希望终止的
SRB_Flags	ASPI 命令标志 对这个命令来说，没有标志是有效的，应该置为 0
SRB_Hdr_Rsvd	该字段被保留，并且应该置为 0
SRB_ToAbort	要终止的 SRB 这个字段包含你希望终止的 SRB 的地址。它应该是和传递到 SendASPICommand() 同样的指针，后者启动原来的 SRB

## ■ SCSI 设备复位命令 (SC\_RESET\_DEV)

SCSI 设备复位命令被用来发送一条 SCSI Bus Device Reset 消息给一个目标器设备。这引起目标器停止所有的有待执行的 I/O 进程，并且把所有的操作参数复位到它们原来加电时的值。注意这个命令对某个特定的设备是特殊的，而且不包括选通 SCSI 总线的 RST 信号。此外，主机适配器将设法通过一个一般的 SCSI 总线事务处理发送总线设备复位消息。此时，在 Windows 95 和 Windows NT 下这条命令将不会正常工作，但是支持这条命令可以和其他的 ASPI 保持兼容性。

---

程序清单 7-13 复位 SCSI 设备 SRB

---

```
typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_RESET_DEV
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器编号
    BYTE    SRB_Flags;         // 请求标志应该为0
    DWORD   SRB_Hdr_Rsvd;     // 保留，必须为0
    BYTE    SRB_Target;
    BYTE    SRB_Lun;
    BYTE    SRB_Rsvd1[12];
    BYTE    SRB_HaStat;
    BYTE    SRB_TargStat;
    void(*SRB_PostProc)(LPSRB);
    BYTE    SRB_Rsvd2[36];
} SRB_BusDeviceReset;
```

注意这个字段和 SRB\_ExecSCSICmd 结构中的字段很相似。对于 SCSI 设备复位命令和执行 SCSI 命令 (Execute SCSI) 来说，它们有相同的含义。ASPI 管理器将会对这些命令排队，并返回一个 SS\_PENDING 状态。你必须等待 SRB 完成，并且你能够指定一个回调程序或者事件句柄，就像你对 Execute SCSI Command 序列所做的一样。

---

程序清单 7-14 复位 SCSI 设备调用

---

```
HANDLE EventHandle;
SRB_BusDeviceReset srb;
memset(&sr, 0, sizeof(srb));
sr.SRB_Cmd = SC_RESET_DEV;
sr.SRB_HaId = HostAdapterNumber;
```

```

srb.SRB_Target = TargetScsiId;
srb.SRB_Lun = 0;
srb.PostProc = EventHandle;
srb.Flags = SRB_EVENT_NOTIFY;
ResetEvent(EventHandle);
dwStatus = SendASPI32Command((LPSRB) &srb);
if (dwStatus == SS_PENDING)
    WaitForSingleEvent(EventHandle, INFINITE);

```

表 7-7

复位 SCSI 设备字段

字 段	说 明
SRB_Cmd	ASPI 命令代码 这个字段必须包含 SC_RESET_DEV 来执行这条命令
ARB_Status	ASPI 命令状态 这个命令用来保存有待执行的和完成的 ASPI 命令的状态。在返回时，这个字段将包含以下值之一： SS_PENDING：仍在处理中 SS_COMP：没有错误 SS_INVALID_HA：主机适配器号 SS_ABORTED：被异常终止 SS_ERR：出错结束 SS_INVALID_SRB：SRB 字段或标志是无效的 SS_INVALID_PATH_ID：目标器 ID 或 LUN 是无效的 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	主机适配器索引 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器；1 表示第二个主机适配器，依此类推。参见关于主机适配器查询命令的描述以获得进一步的信息
SRB_Flags	ASPI 命令标志 这个字段包含一套标志，它们控制 SRB 的执行。对这个命令有效的标志是： SRB_EVENT_NOTIFY SRB_POSTING
SRB_Hdr_Rsvd	这个字段是保留的，应该置为 0
SRB_Target	目标器设备的 SCSI ID 这个字段指出了将被该命令复位的设备的 SCSI ID
SRB_Lun	目标器设备的逻辑单元号 这个字段是为这个命令定义的，但是事实上它并没有被使用。SCSI 总线复位操作是对 SCSI 目标器设备本身执行的，包括在目标器设备上的所有的逻辑单元
SRB_Rsvd1	保留，你应该把这个字段中的所有字节置为 0
SRB_HaStat	主机适配器状态 这个字段返回一个主机适配器的状态值。它指出在 SRB 的执行过程中主机适配器所遇到的错误状态。可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息

续表

字 段	说 明
SRB_TargStat	目标器状态 在 SCSI 命令结束的时候这个字段返回由目标器设备发送的状态值。可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_PostProc	记入程序 如果你正在用命令记入，这个字段设置记入程序的地址；如果你在用事件通知，那就设置 Win32 事件信号的句柄。如果你使用了以上两种方式中的任何一种，你必须在 SRB_Flags 字段设置合适的位 (SRB_POSTING 或 SRB_EVENT_NOTIFICATION)
SRB_Rsvd2	该字段被保留，并且应该置为 0
SRB_Rsvd3	保留，你应该把这个字段中的所有字节置为 0

## 口 重新扫描 SCSI 总线命令 (SC\_RESCAN\_SCSI\_BUS)

重新扫描 SCSI 总线命令引起 ASPI 管理器因为任何改变而检查指定的主机适配器的 SCSI 总线。新增加到总线上的设备将被识别出来，并得到 ASPI 管理器的支持。这条命令只有在 Windows 95 或 Windows NT 平台上、在 ASPI 的 Win32 实现下才能用。在 Windows NT 下，ASPI 管理器将会检查到新的设备，如果它们被禁止或者调换了的话，它们也不会被删除。在 Windows 95 下，这条命令和即插即用服务一起工作，用以加入或者删除设备。因而，在这条重新扫描命令发出以后，需要过几秒钟，设备才会出现。

---

### 程序清单 7-15 重新扫描 SCSI 总线 SRB

---

```
typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_RESCAN_SCSI_BUS
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器编号
    BYTE    SRB_Flags;         // 请求标志应该为0
    DWORD   SRB_Hdr_Rsvd;      // 保留，必须为0
} SRB_RescanPort;
```

---

### 程序清单 7-16 重新扫描 SCSI 总线 SRB 调用

---

```
int adapter;
for (adapter=0; adapter<NumAdapters; adapter++)
{
    SRB_RescanPort srb;
    memset(&srp, 0, sizeof(srp));
```

```

srb.SRB_Cmd = SC_RESCAN_SCSI_BUS;
srb.SRB_HaId = adapter;
SendASPI32Command((LPSRB) &srb);
}
Sleep(10000L); // 等待设备出现达10秒钟
// 现在可以使用 SC_GET_DEVICE_TYPE 查找新设备

```

表 7-8

重检 SCSI 总线字段

字 段	说 明
SRB_Cmd	ASPI 命令代码 这个字段必须包含 SC_RESCAN_SCSI_BUS 来执行这条命令
SRB_Status	ASPI 命令状态 这个命令用来保存有待执行的和完成的 ASPI 命令的状态。在返回时，这个字段将包含以下值之一： SS_COMP：没有错误 SS_INVALID_HA：主机适配器号 SS_INVALID_SRБ：SRB 字段或标志是无效的 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	主机适配器索引 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器，1 表示第二个主机适配器，依此类推。参见关于主机适配器查询命令的描述以获得进一步的信息
SRB_Flags	ASPI 命令标志 对这个命令，没有标志是有效的，这个字段应该被置为 0

## □ 获取/设置超时命令 (SC\_GETSET\_TIMEOUTS)

获取/设置超时命令允许你为发送到特定的设备的 SCSI 命令设置或取得超时值。该命令仅仅在 Windows 95 或 Windows NT 下的 ASPI 的 Win32 实现才可用。超时的单位是 0.5 秒，最大值是 108000 (30 小时)。SRB\_Flags 字段确定你是否正在获取或设定超时值。

---

### 程序清单 7-17 获得/设置

---

```

typedef struct {
    BYTE    SRB_Cmd;           // 命令代码 = SC_GETSET_TIMEOUTS
    BYTE    SRB_Status;        // 命令状态字节
    BYTE    SRB_HaId;          // 主机适配器编号或都设定为 0xFF
    BYTE    SRB_Flags;         // SRB_DIR_IN 或 SRB_DIR_OUT
    DWORD   SRB_Hdr_Rsvd;      // 保留，必须为 0
}

```

```

BYTE    SRB_Target;           // 目标器ID或都设定0xFF
BYTE    SRB_Lun;             // 目标器LUN或都设定为0xFF
DWORD   SRB_Timeout;         // 超时设定值，以1/2秒为单位
} SRB_GetSetTimeouts;

```

### 程序清单 7-18 获得/设置暂停时间调用

```

SRB_GetSetTimeouts srb;
DWORD old_timeout = 0;
memset(&srb, 0, sizeof(srb));
srb.SRB_Cmd = SC_GETSET_TIMEOUTS;
srb.SRB_HaId = HostAdapterNumber;
srb.SRB_Target = TargetScsiId;
srb.SRB_Lun = 0;
srb.SRB_Flags = SRB_DIR_IN;           // 读取当前的超时设备
SendASPI32Command((LPSRB)&srb);
if (srb.SRB_Status == SS_COMP)
    old_timeout = SRB.Timeout;
srb.Flags = SRB_DIR_OUT;              // 设置新的超时值
srb.SRB_Timeout = 10;                 // 5秒
SendASPI32Command((LPSRB)&srb);

```

超时对设备和应用程序都是特定的。一个应用程序对不同的设备可以设置不同的超时值，其他的应用程序也可以为同一个设备设置其他的超时值。一旦一个超时值被设定，它就适用于所有后继的发送给 ASPI 管理器的 SC\_EXEC\_SCSI\_CMD 命令。

当为你的应用程序设置超时值的时候，要特别小心，因为当一条 SCSI 命令超时以后，整个 SCSI 总线将会复位。注意这是通过 RST 信号进行真正的 SCSI 总线复位，而不仅是一个发送给设备的 Reset Device 消息。在每一个和那个总线相连的设备上的有待执行的命令和 SRB 将被异常终止，而不仅仅是超时命令。因而，你的超时设置应该足够长，从而保证不会在正常操作中发生超时；还应该注意到你的 SCSI 命令可能被一个超时打断，而这个超时可能来自于另外一个应用程序的 SRB。你的应用程序应该能够处理这种情况，必要时要重新进行这一操作。

表 7-9

获取 / 设置 超时 字段

字 段	说 明
SRB_Cmd	ASPI 命令代码 这个字段必须包含 SC_GETSET_TIMEOUT 来执行这条命令

续表

字 段	说 明
SRB_Status	<b>ASPI 命令状态</b> 这个命令用来保存有待执行的和完成的 ASPI 命令的状态。在返回时，这个字段将包含以下值之一： SS_COMP：没有错误 SS_INVALID_HA：主机适配器号 SS_INVALID_SRB：SRB 字段或标志是无效的 SS_INVALID_PATH_ID：目标器 ID 或 LUN 是无效的 可以参考 ASPI 错误和状态代码部分以获得关于这个字段的进一步的信息
SRB_HaId	<b>主机适配器索引</b> 这个字段指出命令将访问哪一个被安装的主机适配器。0 表示第一个主机适配器；1 表示第二个主机适配器，依此类推。你也可以指定一个特殊的通配符值 0xFF 来指定将超时应用于指定的目标器/LUN 并结合在所有的主机适配器上。参见关于主机适配器查询命令的描述以获得进一步的信息
SRB_Flags	<b>ASPI 命令标志</b> 这个字段指出你在获得还是在设置超时值。对给定的设备，使用 SRB_DIR_IN 来获取当前的超时值，或者用 SRB_DIR_OUT 来设置一个新的超时值。当设置一个超时值时，SRB_HaId、SRB_Target 和 SRB_Lun 字段可以包含 0xFF 值的通配符，它表示将超时应用于所有独立匹配的适配器、目标器或 LUN
SRB_Hdr_Rsvd	这个字段是保留的，应该被置为 0
SRB_Target	<b>目标器设备的 SCSI ID</b> 这个字段指出了受到超时影响的设备的 SCSI ID。如果你正在设置一个超时值，这个字段可以被置为 0xFF，它表示将超时应用于所有的 SCSI 设备和相应的适配器和 LUN 号
SRB_Lun	<b>目标器设备的逻辑单元号</b> 这个字段指出了受到超时影响的设备的逻辑单元号 (LUN)。如果你正在设置一个超时值，这个字段可以被置为 0xFF，它表示超时应用于设备上的所有的逻辑单元
SRB_Timeout	<b>超时值</b> 这个字段返回或者指出超时值，它以 0.5 秒为单位。它的值从 0 到 108000 (30 小时)。0 被看作一种特殊情况，表示最大超时可能。为了和老的应用程序兼容，默认设置就是允许的最大值

## ■ ASPI 错误和状态码

如果你仔细观察 SRB\_ExecSCSICmd 的结构，你将发现三个独立的状态字段：SRB\_Status、SRB\_HaStat 和 SRB\_TargStat。他们中的每一个都包含了状态信息，这些状态信息和执行一个 SCSI I/O 命令过程中的不同的阶段有关。SRB\_Status 表示 SRB 本身的处理状态，包括在 SRB 的结构、字段或执行中的任何错误。SRB\_HaStat 返回从主机适配器返

回的状态，描述了在 SCSI 总线传输中的问题。SRB\_TargStat 是由目标器设备返回的状态，描述了由于 SCSI I/O 命令或它在目标器设备上执行而产生的问题。

## ■ ASPI SRB 状态 (SRB\_Status)

SRB\_Status 包含了 SRB 的处理状态。在这个字段中的返回值和 SRB 本身的处理有关，并且通常是独立于主机适配器和目标器设备的。例如，SRB\_Status 能够指出一条命令是处于挂起、完成、异常终止还是无效状态。一个 SS\_ERR 值表示主机适配器由于发送到目标器设备的 SCSI I/O 的命令遇到了问题。在这种情况下，你必须查看 SRB\_HaStat 和 SRB\_TargStat 字段，以确定错误的原因。如果 SRB\_Status 字段中的值是 SS\_COMP，那么 SRB\_HaStat 和 SRB\_TargStat 字段中的值就不一定是有用的，不要从它们那里获得有关正常完成的命令的信息。

在 SRB 的执行过程中，SRB\_Status 字段将包含一个 SS\_PENDING 值，表示 ASPI 管理器还没有完成 SRB 的处理。在 SRB 完成后，ASPI 管理器将会把 SRB 的完成状态写到这个字段。这个被写入的值指出 ASPI 是否遇到了和 SRB 本身有关的任何问题。

### ■ SS\_PENDING

这个值表示 SRB 还没有完成，只有执行 SC\_EXEC\_SCSI\_CMD 命令和 SC\_RESET\_DEV 命令才会返回这个状态值，它指出 SRB 已经被排队或者启动了，但是还没有被完成。一般而言，当一条 SCSI I/O 命令完成时，主机适配器将会产生中断，并且 ASPI 管理器将会捕获那个中断来完成它的处理。这包含对 SRB 中所有的相关返回字段进行更新，还可能从目标器设备获得 SCSI 检测数据。然后，ASPI 管理器将设置 SRB\_Status 字段为另外一个值，表示 SRB 的完成状态。

### ■ SS\_COMP

这个值表示 SRB 已经完成，而且没有出错。你应该知道，当它从目标器设备读取数据时，某些 ASPI 管理器和主机适配器的驱动程序不会考虑一个数据缓冲器欠载运行的错误。这是因为对某些设备类型来说，数据缓冲器欠载运行是一种常见状态。例如，当读取一个包含长度不同的块的磁带时，你通常会发送一条 SCSI 的 Read 命令，这条命令指定的数据缓冲区的大小应该足以容纳你认为的可能遇到的最大数据块。如果实际读取的数据块较小，你就会在 SRB\_HaStat 字段得到一个数据缓冲器欠载运行指示符 (HASTAT\_DO\_DU)，你还一定会在 SRB\_TargStat 字段中得到一个 check condition 指示。同样，这些字段中的信息对于成功完成的命令是不代表任何含义的。正式认可的用于数据检查的方法允许进行剩余数据报告。

## ▣ SS\_ERR

这个值表示 SRB 已经完成了，但是遇到了一个错误或异常状态。返回这个值可能有几种原因，你必须检查 SRB\_HaStat 和 SRB\_TargStat 字段来确定究竟发生了什么。对一个 SRB 来说，出现 SS\_ERR 状态并不一定意味着发生了什么可怕的事。通常这只是意味着目标器设备返回了一个 check condition 状态，对很多的设备类型来说，这是一个普通的指示。读者可以参考对 SRB\_TargStatus 的 check condition 值的描述，以获得进一步的信息。

## ▣ SS\_INVALID\_CMD

如果 SRB\_Cmd 字段包含了一个无效的 ASPI 命令的值，那么就会返回这个值。如果你看见了这个状态码，那么你几乎一定没有正确地初始化这个 SRB。

## ▣ SS\_INVALID\_HA

如果 SRB\_HaId 字段表示一个不存在的主机适配器号，就会返回这个值。主机适配器的编号是连续的，它从 0 开始。你可以发送一条 SC\_HA\_INQUIRY 命令来确定安装的主机适配器的编号，该命令的 SRB\_HaId 字段要设为 0。在返回时，通过 ASPI 管理器，HA\_Count 字段将包含可利用的主机适配器的编号。

## ▣ SS\_NO\_DEVICE

这个值表示在 SRB\_Target 字段中指定的 SCSI ID 在主机适配器的 SCSI 总线上是不可用的。这通常意味着没有目标器设备在使用那个 SCSI ID。可以使用 SC\_GET\_DEVICE\_TYPE 命令来确定在一个给定的主机适配器的 SCSI 总线上是否有一个特定的 SCSI ID 可以用。

## ▣ SS\_INVALID\_SRБ

这个值表示 SRB 在一个或多个字段中包含一个无效的值。如果看见了这个状态代码，你应该仔细地检查你的 SRB 初始化代码。在 SRB\_Flags 字段中相互设置独占位标志是引起这种错误的通常原因。

## ▣ SS\_FAILED\_INIT

如果 ASPI 管理器没有被正确地初始化，就会返回这个值。在某些操作系统（Windows）下，如果 ASPI 管理器不能和一个要求得到正确操作的基本设备驱动程序相连，那么 ASPI 管理器将返回这个错误。如果看到这个状态代码，你应该用你安装的 ASPI 管理器检查问题。

### ❑ SS\_ASPI\_IS\_BUSY

如果 ASPI 管理器不能接收用于处理的 SRB，那么将返回这个代码。如果你启动大量的 SRB 请求，并且 ASPI 管理器用完了给它们排队的空间，那么就会发生这种情况。大多数应用程序只有一个或两个挂起的 SRB 请求，所以这几乎不会造成什么影响。但是如果看到这个状态码，你就应该减少你的请求。一个可能的解决办法是维护你自己的 SRB 请求队列，并且只有在其他的请求完成之后才把它们发送给 ASPI 管理器。

### ❑ SS\_BUFFER\_TOO\_BIG

这个值表示主机适配器不能处理 SRB，因为它的数据缓冲区太大 (SRB\_BufLen)。如果看到这个状态码，你可以把你的传输数据分为几个小块，分别传输。

### ❑ SS\_BUFFER\_ALIGN

这个值表示在 SRB\_BufPointer 字段中的数据缓冲区地址对于主机适配器来说没有被正确地排列。某些主机适配器要求数据缓冲区被安排在某些特定的硬件限制的边界上。如果看到这个状态码，你就应该改变你的数据缓冲区的定位，把它设置为由 SC\_HA\_INQUIRY 命令返回的定位值。

### ❑ SS\_SECURITY\_VIOLATION

这个值表示不允许你对指定的目标器设备进行存取操作。如果你把命令发送给一个由操作系统控制的 SCSI 硬盘，这种情况就可能发生。

### ❑ SS\_ABORTED

这个值表示在 SRB 被正常完成之前就被异常终止了。如果你发送一条 SC\_ABORT\_SRB 命令来异常终止这个 SRB 时，你就会见到这个状态；在 SRB 由于 SCSI 总线复位而异常终止时，也会看到这个状态。你不应该依赖于任何返回字段，因为它们在命令异常终止之前不会被更新。

### ❑ SS\_ABORT\_FAIL

这个值表示一条 SC\_ABORT\_SRB 命令失败了。

### ❑ SS\_NO\_ASPI

ASPI 管理器 DLL 存在，但是它不能够和一个请求的设备驱动程序或 VxD 建立连接。

你应该重新安装 ASPI 管理器。

#### □ SS\_ILLEGAL\_MODE

你正在从 Win32 环境下运行 ASPI 的 Win32 实现，但是它并不被支持。在 Windows 3.x 环境下，你仅仅可以运行作为 Windows 组件的 16 位 ASPI。

#### □ SS\_MISMATCHED\_COMPONENTS

ASPI 管理器 DLL 存在，但是请求的设备驱动程序或 VxD 的版本号不匹配。你应该重新安装 ASPI 管理器。

#### □ SS\_NO\_ADAPTERS

如果在一个系统上没有安装 SCSI 主机适配器，GetASPI32SupportInfo( )就会返回这个值。老版本的 ASPI 把它看作一个致命错误，并且拒绝装载；但是在即插即用的情况下，一个 SCSI PCMCIA 适配器可能在以后被插入。

#### □ SS\_INSUFFICIENT\_RESOURCES

这个值表示 ASPI 管理器不能分配足够的系统资源来正确地初始化。这通常表示系统的内存不足。

#### □ 主机适配器状态 (SRB\_HaStat)

SRB\_HaStat 返回主机适配器的状态。SRB\_HaStat 字段是仅为 SC\_EXEC\_SCSI\_CMD 和 SC\_RESET\_DEV 命令而定义的，因为这是实际上用主机适配器操纵 SCSI 总线的仅有的两个命令。在 SRB\_HaStat 字段中的返回值将会告诉你把命令或数据传输到目标器设备的过程中所发生的任何问题。这些问题通常包括硬件事件或超时。

#### □ HASTAT\_OK

这个值表示 SCSI 事务处理正常完成。你还必须为可能的目标器错误检查 SRB\_TargStat 字段。HASTAT\_OK 意味着 SCSI 总线传输是成功的。SRB\_TargStat 字段包含了 SCSI 命令本身的状态。例如，主机适配器将会成功地传输一个无效的 CDB 到一个目标器设备，但是目标器将拒绝接收它。在这种情况下，SRB\_HaStat 字段将会包含 HASTAT\_OK，但是 SRB\_TargStat 字段将包含 0x22，表示一个 check condition 状态。对检测数据的进一步检查将会找到产生这个问题的确切原因。

### □ HASTAT\_SEL\_TO

这个值表示对在 SCSI 总线上的一个选择不做出响应。这常常意味着没有指定 SCSI ID 的设备。它也可能表示 SCSI 总线本身的问题，例如丢失或不正确的终止。在任何一种情况下，你的应用程序都不能对此做些什么。如果这个设备不能够对一个选择作出响应的话，它就不能接受任何 SCSI 命令，你也就被迫停止。你可以尝试给 ASPI 管理器发送一条 SC\_RESET\_DEV 命令，但是这样一般只是给目标器设备发送一条“device reset”消息。由于目标器不对 SCSI 总线选择做出响应，所以它不能获得这条复位消息。然而，一些 ASPI 管理器将会发现一个前面响应的设备已经消失了，并且有可能发送一条 SCSI 总线复位信号（RST 信号），以尝试找回丢失的设备。

### □ HASTAT\_DO\_DU

这个值表示实际的 SCSI 数据传输的长度比在 SRB\_BufLen 字段中定义的长度要大。例如，Write 命令的 CDB 可能指定一个 1024 字节的长度，但是你的数据缓冲区的长度只有 512 字节。在这种情况下，目标器将请求主机适配器传输所有的 1024 字节，但是主机适配器只有 512 个字节发送。大多数的 ASPI 管理器将会在 SRB\_HaStat 字段中返回 HASTAT\_DO\_DU 来警告你。如果发现这个状态，你应该仔细检查你的缓冲区长度和 CDB。此外你还应该检查 SRB\_TargStat 字段，因为可能还有这个 SCSI 命令的一个 Check Condition 状态。

### □ HASTAT\_BUS\_FREE

如果目标器设备意外地和 SCSI 总线失去连接，就返回这个值。这可能是由于电缆或信号的问题，并且很可能在选择过程中或选择后发生。如果目标器遇到 SCSI 总线或阶段改变的问题，它一般就会终止这个事务处理，并放开 SCSI 总线。这个状态被主机适配器发现，并用 HASTAT\_STAT\_BUS 状态码报告。你可以期待这是一个暂时的问题而重试这一命令，但是如果频繁地看到 HASTAT\_STAT\_BUS 错误，你就应该检查你的 SCSI 电缆和终端。

### □ HASTAT\_PHASE\_ERR

如果目标器设备进入一个 SCSI 总线阶段，但是主机适配器并不希望如此，那么就会返回这个值。这可能是一个暂时的状态，或者它可以表示主机适配器和目标器设备之间不兼容。SCSI-2 定义了允许的总线阶段转换，所以这对于新的适配器和目标器设备将不是一个  
问题。

### □ HASTAT\_TIMEOUT

如果在一个过程中的某个事务处理超时了，就会返回这个值。它表示在等待总线事务

处理时超时了，这可能是由于阶段或协议错误。一个 ASPI 管理器或主机适配器的驱动程序为 SRB 实现它们自己的超时机制，这个状态码用来反映超时状态。

#### ▣ HASTAT\_COMMAND\_TIMEOUT

如果主机适配器发现一个 SRB 已经过期了，就会返回这个值。它可能表示一个设备错误，或者一个阶段或协议错误。这个返回值和 HASTAT\_TIMEOUT 不同。因为 HASTAT\_COMMAND\_TIMEOUT 代码常常表示 SCSI 事务处理已经启动，但是在给定的时间段内没有完成。

#### ▣ HASTAT\_MESSAGE\_REJECT

这个值表示目标器发送了一条 SCSI Message Reject 消息码给主机适配器。发送这个消息代码表示目标器不能够接受一个主机适配器发送的消息代码，或者消息代码没有被哪个目标器执行。它也可以表示主机适配器和目标器设备之间不兼容。

#### ▣ HASTAT\_BUS\_RESET

这个值表示 SCSI 检测到了总线复位。

#### ▣ HASTAT\_PARITY\_ERR

当在 SCSI 总线上发现奇偶校验错误时，就会返回这个值。这意味着传输的命令或数据被打断了。和以前一样，你应该检查你的电缆和终端。

#### ▣ HASTAT\_REQUEST\_SENSE\_FAILED

这个值表示主机适配器或 ASPI 管理器在从目标器处收到 check condition 状态后，不能够获得目标器设备的检测数据。如果看到了这个状态，你就应该忽略在 SRB 的 SRB\_SenseArea[ ] 字段中的所有数据。

#### ▣ 目标器设备状态 (SRB\_TargStat)

SRB\_TargStat 字段包含了目标器设备返回的 SCSI 状态值，它一般在 SCSI 命令的最后状态阶段返回。这些值由 SCSI 规范说明书定义，而不是 ASPI，但是我将在这里讨论它们，因为它们在逻辑上适合于上面所讨论的错误和状态码。我将仅仅讨论 ASPI 管理器可能返回的值。如果遇到一个在表 7-10 中没有讨论的值，你就应该查阅最近的 SCSI 规范说明书，以获得细节信息。

表 7-10

目标器设备状态码

值	说 明
0x00	Good 当没有错误或异常情况发生时就返回这个值
0x02	Check Condition 这个值表示发生了偶然事件自动报告。在人类语言里，这意味着某些你应该知道的事情已经发生了。你可以通过检查检测数据来确定到底发生了什么。当 ASPI 管理器发现 Check Condition 状态后，它将自动从目标器设备处获得检测数据，并把它拷贝到 SRB 中的 SRB_SenseArea[ ] 字段。你应该检查检测码、检测键以及在检测数据中的 ASC/ASQ 值，从而确定产生这个 Check Condition 状态的原因。注意 Check Condition 状态不一定代表错误，但是它的确表示你应该检查一些什么
0x08	Busy 这个状态值表示目标器设备（实际上是逻辑单元）正被使用，而且不能接收命令。如果前面的一个 SCSI 命令已经启动但是还没有完成，就会发生这种情况。你可以周期性地重发这条命令，直到目标器接收它
0x18	Reservation Conflict 如果你试图存取的逻辑单元已经为另外一个启动器所保留，那么就会返回这个值。只有在多个启动器（主机适配器）连接到同一条 SCSI 总线上时，才会发生这种情况。（SCSI 定义了 Reserve 和 Release 命令来获得或释放对一个逻辑单元的独占模式的存取。）只有别的启动器已经保留了这个逻辑单元的时候，你才会看到目标器状态码

## ■ 附加的 Win32 下的 ASPI 功能

Win32 下的 ASPI 规范说明书最近被修订了，使它在更多的主机适配器上支持大数据缓冲区（大于 64K）。Win32 下的 ASPI 规范说明书一直允许大数据缓冲区，但是因为特殊的缓冲区安排和分页限制，很多主机适配器和驱动程序不能支持它们。（因为使用虚拟内存和 Windows 的分页，用户分配的缓冲区对于大多数主机适配器来说都太分散了，从而不能使用。）新的 GetASPI32Buffer( ) 和 FreeASPI32Buffer( ) 程序允许应用程序分配一个数据缓冲区，它能够适合这些主机适配器所有必要的需求。这些程序由 Win32 的 DLL 形式的 ASPI 输出，它使用和 GetASPI32SupportInfo( ) 和 SendASPI32Command( ) 程序相同的方式。

```
BOOL GetASPI32Buffer( ASPI32BUFF *P );
BOOL FreeASPI32Buffer( ASPI32BUFF *P );
```

它们每一个都使用一个指向某个数据结构的指针，这个数据结构描述了被分配的数据缓冲区。

---

### 程序清单 7-19 ASPI32BUFF 结构

---

```
typedef struct {
    LPBYTE AB_BufPointer;      // 指向分配的数据缓冲区的指针
    DWORD AB_BufLen;          // 数据缓冲区长度(字节)
    DWORD AB_ZeroFill;        // 如果为1, 缓冲区用0填满
    DWORD AB_Reserved;        // 保留, 必须为0
} ASPI32BUFF;
```

当分配一个缓冲区时, 你就把它填在 `AB_BufLen` 和 `AB_ZeroFill` 字段中, 并把这个结构传递给 `GetASPI32Buffer()` 程序。当释放一个数据缓冲区时, 你就在 `AB_BufPointer` 和 `AB_BufLen` 字段中填入分配操作的返回值, 并把这个结构传递给 `FreeASPI32Buffer()` 程序。

缓冲区的最大容量为 512K。如果 ASPI 管理器不能分配要求的数量, 它就返回 FALSE。你应该假定这个调用可能失败, 而你的应用程序应该准备好把传输数据分成更小的块。

ASPI 的 Win32 规范说明书还有一个函数——`TranslateASPI32Address()`— 把 Windows 95 的 DE VNODEs 和 ASPI 的适配器/设备/逻辑单元转化为 SCSI 设备地址。这个功能对于确定和即插即用事件相关联的 ASPI 目标器地址是很有用的。

```
BOOL TranslateASPI32Address( DWORD *aspi_path, DWORD *devnode );
```

第一个参数是一个指针, 它指向一个代表 ASPI 设备地址的 `DWORD`。最低字节包含了 LUN; 第二个字节包含了 SCSI ID; 第三个字节包含了主机适配器的号。或者用 C 语言来表示为: (`(adapter<<16) | (target<<8) | lun`)。第二个参数也是一个指针, 它指向一个 `DWORD`, 这个 `DWORD` 包含应该被传输的 Windows 95 DEVNODE ID。

在 `TranslateASPI32Address()` 函数返回的时候, 第一个参数指明的 `DWORD` 将被更新为 Windows 95 DEVNODE ID 指定的 ASPI 地址。你也可以为第一个参数指定一个有效的 ASPI 地址, 并为第二个参数使用 0 值的 DEVNODE ID, 从而执行相反的操作。在这种情况下, 对应于所给的 ASPI 地址的 Windows 95 DEVNODE ID 将被放进第二个参数。

# 第八章 用 SCRIPTS 进行低级 SCSI 编程

使用早期 SCSI 协议芯片编程的程序员喜欢谈论关于它有多难这样的话题，这就好像关于你的祖父在暴风雪中跋涉到学校的故事一样，这些故事都表现了讲故事者的遭遇：同汇编语言进行斗争，操作寄存器和 I/O 端口，在代码中建立严格的定时限制。值得感谢的是，从那时开始，我们已经走过了一条漫长的路。对编程人员来说，现在芯片级可以利用的编程工具使那些老的工具看起来就像是石刀。

因为 SCSI 协议芯片和 I/O 控制器变得愈来愈复杂，所以它们也就更加容易使用。它们中许多都有内建的处理器和高级语言编程使用的用来处理 SCSI 协议的微小细节的脚本引擎。在本章中，我们将分析这些工具中较为流行而且功能强大的 Symbios Logic's SCRIPTS 语言。

Symbios Logic 是 NCR 微电子学的继承者，后者是 SCSI 硬件制造者的先驱。如果你理解 SCSI 协议的基础（如果读到这里的话，你应该已经理解了），那么 SCRIPTS 的程序清单将是简单的。

## 使用 SCRIPTS

低级编程对编制大多数应用软件是不适合的。它要求对硬件端口进行存取和对物理内存进行寻址。大多数的现代操作系统把硬件与表现拙劣的程序屏蔽开来。直接存取方式被硬件驱动程序或具有比一般的应用程序更高优先级的运行代码保留。

我们在本章中使用的代码运行于 DOS 下，它没有这样的限制。这更好地演示了怎样使用 SCRIPTS 而保持系统调用和内务操作尽可能地简单。如果你在 Windows 下进行开发，你就只能够以命令行提示的模式启动，或者退出和重新启动到 MS-DOS 模式下。这些代码在 Windows 下的 DOS 框内不能工作。

当使用 SCRIPTS 时，你需要装备两个主要的工具：芯片集的编程向导和软件开发包（SDK）。两个都可以通过 Sysbios Logic 的发行商获得。

编程向导包含关于 SCRIPTS 语言和 SCRIPTS 的 NASM 编译器的扩展文档，它还列出了芯片的寄存器和它们的函数、特征集和能力以及很多其他的信息。对不同的芯片集，该向导有不同的版本，所以要确保你的硬件所用的版本是正确的。

SDK 和编程向导一起放在磁盘上。它由例子代码、工具包和 NASM 编译器组成。SDK

也可以由 Symbios Logic 的 ftp 站点获得，还可以从那里得到很多的例程和工具。

本章中使用的例程和实用程序可以在 SDK 中找到。目标器硬件是一个 Symbios Logic SYM8251S SCSI 主机适配器。这是一个 SCSI 支持的 PCI 适配器。代码使用内嵌的 80386 汇编代码来为 PCI 调用存取扩展 CPU 寄存器。SDK 和例程代码要求使用 Borland C++ 和 Turbo Assembler。因为它们支持这种类型的内嵌代码。如果你正在使用别的编译器，那么你就要把这些源代码分开，然后分别建立它们。

## SCRIPTS 概览

SCRIPTS 的基本原理很简单：从 SCSI 控制器内核和支持电路开始，加入一个专用的 RISC 处理器用于实现编程功能；建立一个支持仲裁、阶段管理、比较、接口控制和逻辑功能的编程语言；在 SCRIPTS 引擎下执行编译程序，把处理负担从 CPU 处转移。

SCRIPTS 处理器对 SCSI 操作是专用的，它独立于操作系统工作。当你需要把缓冲区的地址传送给它的时候这有可能是不便的，SCRIPTS 仅仅处理物理内存的地址，而不是分段的或者虚拟地址。

Symbios Logic 为 SCSI 程序提供了 NASM 编译器。程序的输出是一个文件，该文件包含 C 语言的长 16 进制整数数组，它们代表 SCRIPTS 的操作码和常量。把文件作为头标包含在你的 C 程序的源代码内，这些源代码把这些数组声明为全局变量。

这些数组是 SCSI 操作的小程序单元。为了执行它们，你可以把这些数组的物理地址写到芯片上的寄存器内，这样，你就把这些数组传递给 SCRIPTS 引擎了。一些更为高级的芯片配备了板载 RAM 用于 SCRIPTS 存储。这些芯片可以执行 SCRIPTS 程序而不需要花费从系统内存取指令这样的系统开销。

## SCRIPTS 指令

SCRPTS 语言有一些指令，它们用于 I/O、传输控制、内存移动和其他功能。I/O 函数处理基本的 SCSI 操作。例如，

```
SELECT ATN scsi_id, REL (do_reselect)
```

在 scsi\_id 中选择目标器编码，引发一个 ATN 标志，用于以后请求一个消息输出阶段。如果启动器被一个目标器选择或重选，执行就跳到了相关地址 do\_reselect。它是一个用于复合操作的十分简单的单线命令。

```
MOVE l, msg_buf, WHEN MSG_OUT
MOVE FROM cmd_buf, WHEN CMD
MOVE FROM msgin_buf, WHEN MSG_IN
```

在第一个指令中，处理器将会一直等待，直到它发现消息输出阶段，然后把一个字节从 msg\_buf 移动到 SCSI 总线上。在第二个命令中，处理器等待命令阶段，然后在 cmd\_buf 那里读取一个字节计数和缓冲区地址。最后一个指令等待消息输入阶段，从 SCSI 总线上读取数据，并把它们存储在表的入口 msgin\_buf 指向的位置。

一些指令把控制传输给脚本的其他部分。

```
JUMP REL(handle_phase)
JUMP send_cmd
CALL get_data WHEN DATA_IN
```

JUMP 指令把控制传输到指定的位置。这个位置可能和当前的指令相关，或者是一个绝对地址。CALL 指令如你希望的那样工作，执行一个子程序，它把控制返回给下一条指令。它也支持相对或绝对寻址。

一些指令执行特定的 SCSI 操作。

```
SELECT FROM scsi_id, reselect_addr
WAIT RESELECT, select_addr
WAIT DISCONNECT
CLEAR ATN
```

第一条指令设法选择在 scsi\_id 中的设备，如果它首先被另外一个设备选中，就跳到 reselect\_addr。第二条指令是相反的，告诉芯片等待重选，如果它首先被另外一个设备选择，就跳到 select\_addr。第三条指令等待直到设备从 SCSI 总线上断开连接。最后一条指令简单地清除 ATN 标志。

另外一套指令处理寄存器操作。

```
LOAD SCNTL3 1, def_scntl3
STORE ISTAT 1, cur_istat
MOVE SCNTL3 | 0x08 TO SCNTL3
MOVE SWIDE TO SFBR
```

LOAD 和 STORE 指令在寄存器和内存之间传输数据。第一个例子把一个字节从 def\_scntl3 装入 SCNTL3 寄存器，而第二个把 ISTAT 中的内容存入在 cur\_istat 中的缓冲区。

MOVE 命令对于读、修改和回写寄存器内容的操作很有用。第三个例子演示了在 SCNTL3 寄存器中设置第三位。

最后一个例子演示了一个特殊的情况。在寄存器之间的移动只有在一个寄存器是 SFBR 的时候才有效，SFBR 是 SCSI 第一字节寄存器的英文缩写。这个寄存器受到特殊的对待，因为它有另外一个目的——在条件指令下对存储在 SFBR 中的数据执行比较操作。

另外有一种这个寄存器命令的变体，用于传输数据、位操作或执行数学运算。当 SCRIPTS 程序运行时，它们提供了寄存器操作的唯一方法。在一些例外的情况下，在 SCRIPTS 执行期间，你不能够从你的 C 代码中对寄存器进行存取。

## 口 逻辑操作和状态检测

大多数 SCRIPTS 指令支持对 SCSI 阶段、数据或其他状态的逻辑测试。

```
JUMP address WHEN DATA_IN
JUMP address IF DATA_OUT
JUMP address IF ATN
JUMP address IF 0x01
JUMP address IF 0x0 MASK 0xFF
```

**WHEN** 操作等待直到给定的条件为真，而 **IF** 操作执行一个即时的比较。你将很少用 **WHEN** 来测试 SCSI 的阶段。你可以测试的状态包括 SCSI 阶段、标志和数据值。数据比较测试 SFBR 寄存器的内容，它包含了在最近的 I/O 操作中收到的第一个字节。这有可能是一个消息字节，在一个命令描述符块中的操作码，或者数据块的第一个字节。**MASK** 操作符使你在执行比较之前进行数据的过滤。

我们将使用 **JUMP** 操作来演示逻辑测试，但是大多数的控制指令和很多的移动指令也支持它们，例如：

```
MOVE FROM data_buf, WHEN DATA_IN
CALL address WHEN DATA_OUT
INT err_bad_phase IF NOT MESSAGE_OUT
```

## ■ 在 C 代码中嵌入 SCRIPTS

在编译完你的 SCRIPTS 代码并建立了一个输出文件以后，会发生什么呢？事实上，你已经告诉了 SCRIPTS 引擎去执行它，并告诉它代码在哪里，这或多或少有些莫名其妙。

NASM 输出文件包含了一个 DWORD 数组和编译过的 SCRIPTS 代码。默认时，它将调用这个 SCRIPT 数组，它看上去有点像这样：

```
ULONG SCRIPT[] = {
    (array of DWORD values...)
};
```

执行这个代码是很简单的。Symbios Logic 芯片有一个 DMA SCRIPTS POINTER (DSP) 寄存器。当你把这个寄存器设置为 SCRIPTS 代码的寄存器地址时，它就开始执行了，仅有一个小的要求——地址必须是双字节的。

用你的 C 代码做到这一点是十分容易的。只要把缓冲区分配得比那个 SCRIPTS 数组大一点，在缓冲区内找到第一个 DWORD 对应的地址，并把这个数组拷贝到新的地址。

---

程序清单 8-1 SCRIPTS 代码调整

---

```

DWORD *my_script;           // 指向 Script 的指针

DWORD *alloc_script(WORD size)
{
    BYTE *ptr;           // 临时指针
    WORD seg, off;       // 指针内容
    DWORD *newptr = NULL;

    ptr = malloc(size + 4);

    if (ptr) {
        // 分配Script内存
        // 按照双字节分配缓冲区
        seg = FP_SEG(ptr);
        off = FP_OFF(ptr);
        off += (4 - (off & 0x03));
        newptr = (DWORD *) MK_FP(seg, off);
    }

    return newptr;
}

my_script = alloc_script(sizeof(SCRIPT));

if (my_script != NULL) {
    // 分配Script内存
    // 拷贝script 数组
    memcpy(my_script, SCRIPT, sizeof(SCRIPT));
}

```

为了把这个新地址传输给 SCRIPTS 引擎，还要写入 DSP 寄存器。

```

IOWrite32(io_base + DSP, getPhysAddr(my_script));

```

**◆ 改变运行时间参数**

SCRIPTS 是一种独立的语言。它运行于一个专用的处理器上，对系统资源的仅有存取是通过内存总线和中断控制器执行的。你不能把参数传递给 SCRIPTS 程序，或者像你在 C 语言中一样调用它。一旦一个 SCRIPTS 程序被编译以后，它就在内存里了，在你的数据

段内嵌着一个不连续的代码单元。你如何和它进行通信呢？你如何指导它的操作呢？

### 修补 (Patching)

NASM 产生的输出文件除了编译后的 SCRIPTS 数组外还包含几个其他组件。NASM 还列出了关于绝对数值、入口点、地址和其他命名的元素的信息。例如你可以在你的 SCRIPTS 源代码中声明并使用一个称为 DATA\_COUNT 的值。

```
ABSOLUTE DATA_COUNT = 6
```

输出文件将包含与下面相似的代码：

```
#define A_DATA_COUNT      0x00000006L

ULONG A_DATA_COUNT_Used[] = {
    0x00000011L,
    0x00000018L
};
```

A\_DATA\_COUNT\_Used 数组列出了编译 SCRIPTS 代码中的偏移量，在那里数值被实际使用了。为了在 SCRIPTS 代码中改变它，只要简单地把偏移量作为数组的索引。例如，把 DATA\_COUNT 的值从 6 改为 10：

```
SCRIPT[A_DATA_COUNT_Used[0]] = 10L;
SCRIPT[A_DATA_COUNT_Used[1]] = 10L;
```

这个处理过程被称为修补。除了改变绝对数值之外，你也能够改变指向外部缓冲区的指针、相对地址和其他的公共存取元素。

输出文件还列出了进入 SCRIPTS 代码的入口点。如果你的代码包含入口点 TEST\_UNIT\_READY、INQUIRY 和 RESET\_DEVICE，它们将以如下的形式出现。

```
#define Ent_TEST_UNIT_READY      0x00000110L
#define Ent_INQUIRY                0x00000192L
#define Ent_RESET_DEVICE           0x00000210L
```

当你告诉 SCRIPTS 处理器从哪里开始执行后，你就能够把 SCRIPTS 数组加上偏移量的地址传给指定的程序。

### 表间接寻址 (Table Indirect addressing)

修补是方便的，但是要经常改变就比较麻烦。大多数的 Symbios Logic 芯片还支持表间接操作。这种芯片提供一个额外的寄存器，在你的 C 程序中用一个表的地址来设置它。表的入口包括设备选择或数据传输操作的信息。

这些入口实际上是结构，它们包含两个 DWORD 元素。

```
Typedef struct {
    DWORD count;
    DWORD address;
} table_entry;
```

对于设备选择来说，count 是一个编码的数值，它存储着 SCSI 控制参数、同步数据传输的定时因子和目标器设备的 SCSI ID；address 是保留的，应该置为 0；对于其他的操作来说，这个地址指向一个物理内存中的缓冲区，count 指出缓冲区的大小。

表间接操作起来很有趣，因为它们要求保留两套内存块。你的 SCRIPTS 代码保存了一个表的声明和对它入口点的引用。然而，这不会产生任何代码或分配任何的内存。这个表只是一个占位符，引用表示进入这个表的偏移量。

实际的工作是在你的 C 代码中完成的。你再次声明这张表，保证入口点相同，并有和在 SCRIPTS 表中相同的顺序。你也可以定义一些助记值作为表的索引，它们对应于在 SCRIPTS 代码中使用的名字。

这也是你实际为这个表分配内存的地方，并且把它对应在一个 **DWORD** 的边界上。一旦你获得了内存，你就可以在表中填入所希望的数值，并把表的地址置入正确地寄存器。下面的代码分别表示了用 SCRIPTS 和用 C 开始的处理。

### 程序清单 8-2 SCRIPTS 表声明

```
TABLE my_table \
    select_info = ??, \
    cmd_buf = ??, \
    msg_buf = ??, \
    data_buf = ??
```

### 程序清单 8-3 C 表声明

```
typedef struct {           // 表格入口定义
    DWORD count;
    DWORD address;
} table_entry;

table_entry *my_table;      // 指向表格的指针
BYTE command_buf[6];        // 命令缓冲区
BYTE message_buf[2];        // 信息缓冲区
BYTE data_buf[32];          // 数据缓冲区
BYTE targ_id;               // 目标器 SCSI ID
```

```
#define TABLE_SIZE 4          // 表格入口点的数目

enum table_offsets {
    SELECT_INFO = 0,
    CMD_BUF,
    MSG_BUF,
    DATA_BUF
}

table_entry *alloc_table(WORD nentries)
{
    BYTE *ptr;           // 临时指针
    WORD seg, off;       // 指针内容
    table_entry *newptr = NULL;

    ptr = malloc(nentries * sizeof(table_entry) + 4);

    if (ptr) {
        // 分配的表格内存
        // 按照双字节分配缓冲区
        seg = FP_SEG(ptr);
        off = FP_OFF(ptr);
        off += (4 - (off & 0x03));
        newptr = (table_entry *) MK_FP(seg, off);
    }
}

return newptr;
}

my_table = alloc_table(TABLE_SIZE);

if (my_table != NULL) {
    // 分配的表格内存
    // 实现表格入口点
    my_table[SELECT_INFO].count = (0x00000300L & targ_id) << 16;
    my_table[SELECT_INFO].address = 0L;

    my_table[CMD_BUF].count = 6L;
    my_table[CMD_BUF].address = getPhysAddr(command_buf);

    my_table[MSG_BUF].count = 2L;
```

```

my_table[MSG_BUF].address = getPhysAddr(message_buf);

my_table[DATA_BUF].count = 32L;
my_table[DATA_BUF].address = getPhysAddr(data_buf);
}

```

为了使这个表对于 SCRIPTS 代码可用，把在芯片上的 Data Structure Address (DSA) 寄存器设置为表的物理地址。

```
IOWrite32(io_base + DSA, getphysAddr(my_table));
```

因为表间接操作的灵活性，你可以在你的 C 代码中很方便快捷地改变地址、字节数或目标器设备信息，而不是修补 SCRIPTS 数组。这使你为不同的命令或函数重用代码变得容易了。

## 检测 SCRIPTS 程序的完成

知道 SCRIPTS 程序何时完成是很有用的。有不同的方法来检测这一点，它们都集中在芯片上的 ISTAT 寄存器上。这个寄存器包含发生在 SCSI 操作中的中断信息，特别是，它会告诉你中断源。

一般你用一个 INT 指令来结束你的 SCRIPTS 程序，它会终止 SCRIPTS 代码的执行。它为一个参数获得一个值，把它存储在寄存器里，你可以在以后获取它。这就允许你从 SCRIPTS 代码中返回一个值到你的 C 程序中。

SCSI 错误也可能引起你的 SCRIPTS 代码不正常结束。一次异常断开、复位或阶段不匹配，都可能终止你的程序，如果它没有准备好处理这些状态的话。

ISTAT 寄存器包含两个重要的标志，指示你进一步的信息。在 0 号位的 DMA 中断挂起(DIP)标志告诉你检查 DMA 的状态(DSTAT)寄存器，获得中断源。如果中断来自 SCRIPTS 代码中的 INT 指令，那么寄存器的 2 号位被设置，如果是这种情况，那么 DMA SCRIPTS 指针保存 (DSPS) 寄存器保存由 INT 指令返回的数值。

如果 1 号位的 SCSI 中断挂起 (SIP) 标志被设置，那么就是 SCSI 错误引起中断。两个其他的寄存器 SCSI 中断状态 0 和 1 (SIST0 和 SIST1) 保存发生错误的信息。

很多这种中断状态可能通过设置别的寄存器而被屏蔽。你必须保证对有效状态进行测试。你还必须知道哪些 SCSI 状态是致命的，而哪些不是。

### 轮询是否完成

ISTAT 寄存器仅有一个，因此，你可以从你的 C 程序那里存取它当你的 SCRIPT 代码执行时。通过在一个循环中轮询寄存器，你就可以检测当 SCRIPTS 代码完成时，位 0 或位 1 是否被设置。这对程序来说最简单，但是轮询会浪费 CPU 的周期。

你也可以在你的 C 代码中轮询数据缓冲区的内容或状态字节，从而确定是否完成。这

有一定的风险，因为当由于 SCSI 错误而结束时，它可能会检测不到。

### 硬件中断或完成

一个更加优美但是也更复杂的检测完成方法是通过硬件中断。如果你在 DMA 控制 (DCNTL) 寄存器中允许执行中断，当中断发生时，芯片将会产生一个 IRQ。如果你很喜欢写硬件中断，你就可以用这种方式。在你的处理程序中，通过检查上面所述的寄存器来确定中断源。

IRQ 的级别依赖于芯片被如何配置。如果你正在使用别的 PCI SCSI 适配器，你就能够通过 PCI BIOS 调用获取 IRQ 的级别信息。当我们讨论初始化和建立芯片的时候，我们将讨论怎么做这些事情。

在你考虑运行一个 SCRIPT 程序之前，还有一些你要注意的内务处理事情。你要询问控制器它被如何配置以及它支持什么特性，你需要复位 SCSI 功能单元并为这些寄存器选择默认值。

### PCI BIOS 功能

SYM8251S 主机适配器是一个基于 Symbios Logic SYM53C825 SCSI I/O 控制器芯片的 PCI 板。通过 INT 0x1A 和 PCI BIOS 功能 (0xB1) ID 来使用各种功能，你就可以定位已安装的电路板，并对它们的配置进行查询或配置。但是，首先你应该知道正在运行程序的机器是否安装了当前的 PCI BIOS。

如果你现在用的是第二版的 BIOS，调用 PCI 中断子功能 0x01 就把标识字符串“ICP”返回到 EDX 寄存器。老一点的版本把它返回到 CX: DX 寄存器。存取扩展寄存器需要 80386 的指令。很多 C 编译器在内嵌的汇编代码中不支持这种指令，所以你可能需要把它们建立在一个单独的汇编代码中。程序清单 8-4 表示了如何检查 PCI BIOS 的存在。

---

#### 程序清单 8-4 检测 PCI BIOS 版本

---

```
WORD PCI_GetPCIBIOSVersion(pci_bios *ppcibios)
{
    WORD r_ax, r_bx, r_cx, r_dx; // 寄存器变量
    DWORD r_edx;
    DWORD pci_sig; // PCI 特征标记
    WORD retval = PCI_NO_BIOS;
    pci_sig = 0x20494350L; // ICP 特征标志
    // 调用 BIOS 的检测 PCI 功能
    r_ax = ((PCI_FUNCTION_ID << 8) | PCI_BIOS_PRESENT);
    asm {
        .386
        mov ax, [r_ax]
        int PCI_BIOS_INT
    }
}
```

```

    mov DWORD PTR [r_edx], edx
    mov [r_dx], dx
    mov [r_cx], cx
    mov [r_bx], bx
    mov [r_ax], ax
}

if (r_dx == LOWORD(pci_sig)) {
    // 包含 PCI BIOS
    if (r_cx == HIWORD(pci_sig) &&
        (r_bx & 0xff00) == 0x0100) {
        // PCI BIOS 版本为 1.x
        retval = PCI_BIOS_REV_1X;
    }
    else if (r_edx == pci_sig) {
        // PCI BIOS 版本为 2.x
        retval = PCI_BIOS_REV_2X;
    }
    else {
        // 未知的版本
        retval = PCI_UNKNOWN_BIOS;
    }
    if (ppcibios != NULL) {
        // 实现 BIOS 的信息结构
        ppcibios->access = (r_ax & 0xff);
        ppcibios->version = r_bx;
        ppcibios->lastbus = (r_cx & 0xff);
    }
}
return retval;
}

```

把 PCI BIOS 的信息存储在一个结构中，以待后用。尽管这个函数提供了关于 BIOS 版本和总线数目的信息，但是它的主要用途还是告诉我们有一个 PCI 总线在机器上。

子功能 0x02 在 PCI 总线上寻找一个指定的设备。每个设备由一个厂商的 ID、设备的 ID 和设备索引来标识。在程序清单 8-5 中，我们用一个结构来保存 PCI 的设备信息。我们把设备 ID 设为 53C825 芯片，就是 0x003。Symbios Logic 的厂商 ID 是 0x1000。

---

### 程序清单 8-5 定位 PCI 设备

---

```

int PCI_FindDevice(pci_device *ppcidevice)
{
    // REGPACK 结构寄存器;

```

```
WORD r_ax, r_bx, r_cx, r_dx, r_si;
DWORD config;
int retval = 0;

// 保证有一个 PCI BIOS
if (PCI_GetPCIBIOSVersion(NULL) != PCI_NO_BIOS) {
// 包含 PCI BIOS
    // call PCI function to find device
    r_ax = ((PCI_FUNCTION_ID << 8) |
        (PCI_FIND_DEVICE));
    r_cx = ppcidevice->dev_id;
    r_dx = PCI_SYM_VENDOR_ID;
    r_si = ppcidevice->dev_index;
    asm {
        .386
        mov ax, [r_ax]
        mov cx, [r_cx]
        mov dx, [r_dx]
        mov si, [r_si]
        int PCI BIOS INT
        mov ax, 0
        adc ax, 0
        mov [r_bx], bx
        mov [r_ax], ax
    }
    if (r_ax == 0) {
        // 进位标志位清除成功
        // 保存设备总线数
        ppcidevice->bus_num = ((r_bx & 0xFF00) >> 8);
        // 保留设备号
        ppcidevice->dev_num = (r_bx & 0x00FF);
        // 保留设备功能
        ppcidevice->function = (r_bx & 0x0007);
        // 获得命令寄存器
        ppcidevice->command =
            (WORD) PCI_GetConfigRegister(
                ppcidevice, PCI_CONFIG_REG_CMD);
        // 获得版本 ID
        ppcidevice->rev_id =
            (BYTE) PCI_GetConfigRegister(
                ppcidevice, PCI_CONFIG_REG_REVID);
        // 获得子系统和开发商 ID
        config = PCI_GetConfigRegister(
            ppcidevice, PCI_CONFIG_REG_SUBV);
        ppcidevice->sub_ven_id = (WORD) config;
```

```

ppcidevice->sub_id = (WORD) (config >> 16);
// 获得 I/O 基地址
config = PCI_GetConfigRegister(
    ppcidevice, C8XX_CONFIG_REG_IOB);
ppcidevice->io_base = (config & 0xFFFFFFFEL);

// 获得 ROM 基地址
ppcidevice->rom_base = PCI_GetConfigRegister(
    ppcidevice, PCI_CONFIG_REG_ROM);
// 获得中断号
config = PCI_GetConfigRegister(
    ppcidevice, PCI_CONFIG_REG_INTL);
ppcidevice->intl = (BYTE) config;
retval = 1;
}
}

return retval;
}

```

如果这个功能成功，它就通过总线号和设备号返回适配器的地址。我们用它来从总线上获得进一步的信息。修补、ID 数目、I/O 基地址和使用的中断级别，它们都可以通过 PCI 查询获得。

`PCI_GetConfigRegister` 函数使用子功能 0x0A 来读取配置寄存器。我们再次使用一个结构来传递和返回 PCI 设备信息。偏移量参数指向一个我们所指定的寄存器。

---

#### 程序清单 8-6 查询 PCI 设备配置情况

---

```

DWORD PCI_GetConfigRegister(pci_device *ppcidevice,
    WORD offset)
{
    WORD r_ax, r_bx, r_di, r_dx, r_cx;
    DWORD r_ecx;
    WORD pci_version;
    DWORD retval = 0L;

    // 获得 PCI 版本
    pci_version = PCI_GetPCIBIOSVersion(NULL);
    if (!(pci_version == PCI_NO_BIOS ||
        retval == PCI_UNKNOWN_BIOS)) {
        // PCI BIOS 存在
        // 调用 PCI 函数读取寄存器
        r_ax = ((PCI_FUNCTION_ID << 8) |
            (PCI_READ_CONFIG_DWORD));

```

```

// 设置总线数和要查找的设备号
r_bx = ((ppcidevice->bus_num & 0xff) << 8) |
(ppcidevice->dev_num & 0xff);
// 设置配置寄存器的偏移地址
r_di = offset;
asm {
    .386
    mov ax, [r_ax]
    mov bx, [r_bx]
    mov di, [r_di]
    int PCI BIOS INT
    mov DWORD PTR [r_ecx], ecx
    mov [r_dx], dx
    mov [r_cx], cx
}
if (pci_version == PCI BIOS REV_1X) {
// PCI 版本 1.x
    retval = r_dx;
    retval = (retval << 16) | r_cx;
}
else if (pci_version == PCI BIOS REV_2X) {
// PCI 版本 2.x
    retval = r_ecx;
}
}
return retval;
}

```

这个函数返回请求的信息在 ECX 或 CX: DX 寄存器内，如前所述，这完全取决于 PCI 的版本。

由于我们有了控制器的基本 I/O 地址，我们就可以初始化控制寄存器。

## ■ 初始化 SCSI 控制寄存器

当芯片被加电时，控制寄存器被设置为硬件默认值。对大多数情况来说，这就足够了。如果你的控制器板或主板配备了 SCSI BIOS，在启动时可能会改变一些值。如果对这些值不满意，你可以自由地改变它们。

在试图调整你的寄存器设置的时候，绝对有必要获得关于你的控制器芯片的文档。这些设置中很多都是位编码写入寄存器的。其他的和你的控制器可能不支持的特性有关，例如 Wide SCSI 或 Fast-20 SCSI。没有文档而进行操作就像没有地图而进行野外旅行一样。虽然你可以到你想去的地方，但是很有可能会在路上迷失方向。

## 例程

为了演示怎样把前面的这些代码连在一起，让我们用它们来建立一个简单的程序。从一个 SCRIPTS 模块开始处理普通的 SCSI 函数，我们将加入支持 C 的代码来建立一个程序，该程序将会查询 SCSI 总线并打印出它找到的设备的信息。如果它遇到一个直接存取设备，它将会读取并显示第一个块的内容。

这仅仅需要少量的 SCSI 函数。Test Unit Ready、Device Inquiry 和 Request Sense 对所有的设备都适用。对于直接存取设备来说，我们还要使用 Read Capacity 以及 Read (6 字节版本) 命令。

一般的 SCRIPTS 模块、GENSCSI.SS、处理选择、消息阶段和数据阶段也处理断开连接/重新连接序列。对于更多的高级应用来说，你还需要加入同步传输或宽传输的协议。在 Symbios Logic 的 SDK 中的例程包含了一些这样的例子。但为简单起见，我们将忽略这些特征，使这些代码容易跟踪。

C 的模块被分解成函数，SPCL.C 包含了 PCI BIOS 接口代码。53C800 系列芯片的初始化代码被单独放在 S8XX.C 文件中。标准 SCSI 函数的实现实际被包含在 SSCSI.C 中。工具函数在 SDK 文件 GENTOOLS.C 中。

在主模块 SQUERY.C 中，我们通过用 PCI 调用来初始化主机适配器。我们获取关于主机适配器的配置细节、使用 I/O 端口的信息、对 Wide SCSI 和 Fast SCSI 的支持等信息。Wide SCSI 支持是重要的，因为它告诉你要寻找多少设备。

接下来我们要为 SCRIPTS 数组建立缓冲区，并为 SCSI 调用建立数据缓冲区，分配消息缓冲区、命令描述符块和数据缓冲区，并使用 DWORD 的格式。当我们使用它们的时候，已经填入了正确的数值。

最后，我们搜索对 Test Unit Ready 命令做出响应的设备。它们中很多都以 Unit Attention 状态做出响应，所以我们要用 Request Sense 命令来读取检测数据。当设备在加电后第一次被访问时，它就会产生 Unit Attention 状态。因此，这段代码包含了一个重试循环。

一个设备查询将跟随对 Test Unit Ready 命令做出响应的 SCSI ID。程序指出了设备的类型、标识符字符串和其他参数。

如果它找到一个直接存取设备，程序就发出一个 Read Capacity 命令。它指出在存储介质上的数据块的大小和数目。如果你正在使用一个可移动介质驱动器，就像 Iomega Zip 驱动器，如果没有磁盘存在，这个调用将返回失败。在你插入介质后的第一次调用，将会返回检测数据，表示介质已经被改变了。我们期待出现这种状态，然后恢复它。

对直接存取设备的最后一个调用读取介质上的第一个块，并以十六进制和 ASCII 码的形式打印出它的内容。在一个格式化的 Zip 磁盘上的 0 号块在前几个字节中包含一个 Iomega 的标记。

## 四 类属 SCRIPTS 代码

我们例程的中心是 GENSCSI.SS 中的 SCRIPTS 代码。让我们更加仔细地来研究一下。我们从声明正在编译的程序的体系结构开始。在这里，它是一个 53C825 芯片。

```
;----- set architecture for 53C825
ARCH 825
```

下面我们声明一些常量，它们是这些SCRIPTS代码将返回到主程序的值。

```
;----- set constant values
ABSOLUTE err_cmd_complete =      0x00000000
ABSOLUTE err_not_msgout =        0x00000001
ABSOLUTE err_bad_reselect =      0x00000002
ABSOLUTE err_bad_phase =         0x00000004
```

现在到了一个关键的部分。我们声明了一个缓冲区的表，因为我们的数据传输采用的是表间接寻址。我们调用的是什么并不重要，因为一个 SCRIPTS 模块只能包含一个表。

记住，简单地声明一个表并不能够为它分配内存。那是在你的支持 C 代码中做的。在下面的例子中，跟在每个表元素后面的数值只是简单地用于调试的占位符。

在 SCRIPTS 的语法中，表的声明是一行简单的代码，末尾是回车符。我们已经用反斜杠作为行连接符来分离表元，并使代码更具可读性。

```
;----- set up table definitions
TABLE table0 \
  scsi_id = ID {0x33, 0x00, 0x00, 0x00}, \
  msgout_buf = {0x80, 0x00}, \
  cmd_buf = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, \
  stat_buf = ??, \
  msgin_buf = 2 {??}, \
  exmsgin_buf = 4 {??} \
  datain_buf = 0x40 {??}
```

在表的声明之后，我们为代码声明了一个入口点，并为脚本声明了一个名字。脚本的名字将会被用来作为处于被编译的代码中的 SCRIPTS 数组名。

```
;----- entry point for general SCSI script
ENTRY start_scsi
```

```
PROC GEN_SCRIPT:
```

最后，我们用一个 SELECT ATN 指令来开始实际的代码。我们的 C 代码已经用我们正在选择的关于目标器的信息填满了 scsi\_id 缓冲区。如果在这步中，主机适配器本身被选择或重选，执行将转移到 bad\_reselect 表。

```
;----- start SCSI target selection
start_scsi:

; select device from encoded SCSI ID
; set ATN for message out after select
SELECT ATN FROM scsi_id, REL(bad_reselect)
```

因为我们在选择过程中设置了 ATN 标记，后接一个消息输出阶段。我们将发送 Identify 消息，告诉目标器它是否有断开连接的权力。我们在 C 代码中建立了 msgout\_buf 缓冲区。

```
;----- send identify message
; exit if not message out phase
INT err_not_msgout, WHEN NOT MSG_OUT

; send identify message
MOVE FROM msgout_buf, WHEN MSG_OUT
JUMP REL(handle_phase)
```

在发送了 Identify 消息后，我们跳到了一个阶段处理程序。这个程序简单地检查总线阶段，并跳到相应的程序。在这里，在消息输出阶段之后是命令阶段。

```
;----- send SCSI command
send_cmd:

; send command block to target
MOVE FROM cmd_buf, WHEN CMD
JUMP REL(handle_phase)
```

在命令阶段之后，再次回到了阶段处理，发送命令确定下面将会发生什么。对一个像 Test Unit Ready 这样的命令，不会发生数据传输——我们将直接进入状态阶段。在本例中，我们只是把状态字节读入缓冲区。

```
;----- get SCSI status
get_status:

; read status byte from data bus
MOVE FROM stat_buf, WHEN STATUS
JUMP REL(handle_phase)
```

在状态阶段之后是消息输入阶段。我们读取消息字节，并清除 ACK 位。如果一切顺利的话，那么这个消息应该是 Command Complete，并跳到我们的退出代码。如果在命令的执行中一些别的点到达了这里，我们就必须处理其他消息。

我们要检查的其他消息是 Disconnect 和扩展消息。如果我们检测到一个 Disconnect 消息，我们就会转移到一个等待断开连接发生的程序。为了检查扩展信息，我们在跳入阶段处理程序之前，从总线上再读取一个字节。任何其他消息都被忽略。

```
;----- get SCSI message input
get_msgin:

; read message byte from data bus
MOVE FROM msgin_buf, WHEN MSG_IN
CLEAR ACK

; handle Command Complete message
JUMP REL(cmd_complete), IF 0x00

; handle Disconnect message
JUMP REL(wait_disconnect), IF 0x04

; handle extended message
JUMP REL(ext_msgin), IF 0x01
JUMP REL(handle_phase)

;----- handle extended message
ext_msgin:

; read extended message from data bus
MOVE FROM exmsgin_buf, WHEN MSG_IN
CLEAR ACK
JUMP REL(handle_phase)
```

如果我们发送一条命令来读取或返回数据，阶段处理程序将会把我们派送到数据输出程序。要读的字节数和目标地址包含在 data\_buf 内。我们在 C 代码内再一次建立它们。

这个处理对于把数据发送到设备或从设备收到数据都是很相似的。数据的方向由命令决定。

```
;----- get data input
get_datain:

; read data from bus
MOVE FROM datain_buf, WHEN DATA_IN
JUMP REL(handle_phase)
```

这里是我们用来处理不同的总线阶段的派送表。意外的阶段将伴随错误代码退出。

```
;----- handle SCSI phases
handle_phase:

; jump to appropriate handler for phase
JUMP REL(get_status), WHEN STATUS
JUMP REL(get_msgin), WHEN MSG_IN
JUMP REL(get_datain), WHEN DATA_IN
JUMP REL(send_cmd), WHEN COMMAND
; unhandled phase
INT err_bad_phase
```

这时我们正常退出程序，会产生一个 **Command Complete** 消息。我们希望总线断开连接，所以我们把在断开连接时产生错误的寄存器的位清零了。当断开连接发生时，我们将返回一个成功码。

```
;----- SCSI command execution complete
cmd_complete:

; command complete - wait for disconnect
MOVE SCNTL2 & 0x7F to SCNTL2
CLEAR ACK
WAIT DISCONNECT
INT err_cmd_complete
```

如果我们收到一个 **Disconnect** 消息，我们就一直等待，直到断开连接发生。在断开连接后，我们清除所有的在 SCSI 总线和 DMA 寄存器中的数据，并等待重新选择。一旦被重选，就等待来自目标器的 **Identify** 消息，并进入错误处理程序。

到达断开连接/重新连接的途径是简单的。为了获得更强大的程序，你就要处理 **Save Data Pointers** 消息，它领先于 **Disconnect** 消息，并存储其他的信息，以保证在数据传输中没有间隔或重叠。

```
;----- handle disconnect before reselect
wait_disconnect:

MOVE SCNTL2 & 0x7F to SCNTL2
CLEAR ACK
WAIT DISCONNECT

; clear DMA and SCSI fifos
MOVE CTEST3 | 0x04 to CTEST3
```

```
MOVE STEST3 | 0x02 to STEST3

; wait for reselect
WAIT RESELECT REL(bad_reselect)

; expect identify message
MOVE FROM msgin_buf, WHEN MSG_IN
CLEAR ACK

; shortcut to update sync and wide options
SELECT FROM scsi_id, REL(handle_phase)
```

分析例程和 SCRIPTS 程序。对它进行实验，为别的命令或消息加入处理程序。

Symbios Logic 的 ftp 站点是获得例程的好地方，还可以从那里获得 NASM 编译器，用来建立 SCRIPTS 代码。在你使用它以后，你将会看到 SCRIPTS 对于低级的 SCSI 开发来说是多么强劲。

## 第九章 SCSI 目标模式编程

在本书的前面几章里，所讨论的大多数 SCSI 编程都是从利用主机适配器来控制外围设备的角度来叙述的。但是在本章中我们要换一个角度，专门从作为目标器的外围设备的观点来介绍 SCSI 编程的方法。长期以来，很少的中小型工业把目标模式的 SCSI 编程视为一种很特殊的功能。但是要知道使用目标模式进行 SCSI 编程，就像一个孤独的程序员埋头在 R&D 实验室中一直工作到深夜，并把私酿的威士忌酒从酒瓶中倒出来，放到 I.V. 管中慢慢地汲取营养。不用多说，本章显然不是专门为那些“精英”们所写的，相反，我们是想让我们这些非“精英”对各个方面的 SCSI 编程有所了解，并捎带着学习把 PC 用作 SCSI 目标器设备的方法。

对于一个程序员来说，典型的 SCSI 设备应该包括三个完全不同的组件：硬件、驱动程序（包括 ASPI）和应用程序。硬件主要用来对 SCSI 各项事务的物理接口进行控制，包括总线仲裁、选择和数据传输；驱动程序可以对硬件进行管理，响应总线阶段的变化，也会处理有关消息字节的各项事务。另外还提供了与应用程序的接口；应用程序主要用来完成编程的工作——在较高层次上使用和编制 SCSI 命令。不过在对目标模式的 SCSI 编程进行讨论的过程中，我会尽量把这两种不同的工作作为一个类别进行介绍。

在我首次准备编写基于 PC 的目标模式的 SCSI 应用程序时，我非常想找到一个有关 SCSI 总线低层次的细节管理的软件库或程序包。当时我最想得到的是一种“反 ASPI”驱动程序，因为这种“反 ASPI”驱动程序可以提供给我引入的 CDB 代码和数据传输，这样我们就可以进行真正的工作。我对诸如 SCSI 总线相位的改变、消息字节之类并不感兴趣，我只想把我手头上的工作做完。尽管 Common Access Method (ANSI CAM) 为目标模式的 SCSI 编程制订了规范，但是还没有发现一种简单地基于 PC 的 SCSI 适配器能够提供 CAM 驱动程序的功能。不仅如此，也没有发现任何对目标模式的 SCSI 编程有所帮助的软件库或程序包，所以我只好自己在上述软件的基础上开发自己的程序代码。不幸中的万幸，我对主机模式的 SCSI 编程还算比较熟悉，所以只要对这些命令设置进行“仿制”就可以了。何况 SCSI-2 规范已经相当完善，其中包括了几乎所有我可能会用到的信息。从这些因素看来，好像进行目标模式的 SCSI 编程已经万事大吉了，但是，年轻人，我宁愿能够得到哪怕是一点儿有关目标模式 SCSI 编程的忠告或例子。我之所以把目标模式 SCSI 编程的代码和技术都在本章中进行介绍，就是不想让那些和我同样遭受挫折的人们再次重蹈覆辙。

### 硬件

并不是所有的 SCSI 主机适配器都可以使用目标模式的 SCSI 编程方法。另外还专门设计了一些接口卡作为 SCSI 的启动器，但是它们并不能响应来自另一个启动器的选择请求。

一些内嵌固件的接口卡尽管对于主机模式的事务处理来说是最优的，但是对于目标模式的 SCSI 编程来说却是有害无利的。在某些情况下，我们可以下载特殊的目标模式的固件到这些接口卡中，但是在大多数情况下我们是不会如此幸运的。到目前为止，在目标模式的 SCSI 编程中最大的障碍是在大多数的适配器中缺少相应的说明文档。现在仅有极少的 PC SCSI 卡能够支持目标模式的操作，当然只有更少产品中包含使用方法的说明文档。我还不止一次发现很有必要直接从半导体制造商那里选择一个 SCSI 芯片组，然后又回过头来找寻使用这种特殊芯片组的 PC 卡。这样做很可能会失去板卡制造商的支持，但是至少我还曾经直接从半导体制造商的数据单中获得过 SCSI 芯片组的说明文档。

在我使用的 SCSI 芯片组中有一种称之为 Symbios Logic 53C400A。这种芯片组是一种比较古老、相对低端的 SCSI 芯片，它使用 NCR 的 53C80E SCSI 内核和 ISA 的总线接口。之所以选择这种芯片作为例子，是因为它使用的是—种初级的 SCSI 总线编程。在这种芯片组中使用了最少的硬件来完成 SCSI 的总线仲裁、选择和信号交换，而且并没有因此省略 SCSI 事务处理中的某些细节。这就使我们能够在大多数应用程序中既可以获得可观的传输速率，而且还可以来学习 SCSI 总线的细节内容。

## 处理 SCSI 阶段

在我们继续进行下一步之前，先来简单介绍一下 SCSI 事务处理的不同阶段。SCSI 的事务处理都是以总线仲裁和选择阶段开始的，这为启动器获得 SCSI 总线、建立与特定的目标器之间的连接创造了条件。在选择阶段结束之后，目标设备就会获得 SCSI 总线的控制权，并对 SCSI 总线阶段的改变时序进行监控。如果启动器发送了 ATN 信号，目标器就会进入消息输出阶段，并同时从启动器中读取消息字节。这些消息通常可以用来确定目标器的 LUN 和断路优先级，但是也可以发送其他的消息。目标器会在接收并处理完这些消息之后进入命令阶段，同时对来自主机的 SCSI 的命令描述块（CDB）进行监督。该过程一结束，目标器就可以开始对命令进行处理。

如果目标器执行的是一个比较长的命令（只有 Identify 消息可以用来进行断路处理），此时它就可以决定是否从 SCSI 总线中断开。如果想要从总线中断开，接下来将会进入消息阶段。在这个消息阶段里，目标器会发送 Save Data Pointer 消息和 Disconnect 消息，随后进入总线空闲状态，并释放 SCSI 总线的控制权。当目标器想要重新进行连接时，它就会进入总线仲裁和重选阶段，并发送 Identify 消息来确定将要进行连接的 LUN（逻辑单元号），然后就会进行数据或状态的传输。如果在数据传输的过程中产生了奇偶校验错误，目标器将会向主机发送一个 Restore Data Pointer 消息，并重新对数据进行传输。最后，目标器会进入状态阶段，并向启动器发送一个 command 状态来表明过程的结束，然后将进入总线空闲阶段，同时释放 SCSI 总线的控制权。

在下面例子的代码中，我们可以看到 53C400A 是如何对选择作出响应，以及如何对 SCSI 事务处理中的各种阶段进行控制的。而且在下面的代码中，我们假设已经存在一个 WriteReg

例程，该例程可以向 53C400A 的寄存器中写入数据，还存在一个 ReadReg 例程，用来读取特定的 53C400A 寄存器中的数据。尽管下面的代码中使用的是 53C400A 中特定的寄存器名称和数据，但我们能够从注释中获得更广泛意义上的帮助。当然，尽管其他的 SCSI 芯片组使用的是不同的低层次的代码时序，但是我们还是可以从下面例子的代码中获得处理各种 SCSI 总线阶段的方法。

我们首先要做的是对 53C400A 进行初始化和选通。这些操作可以通过重置芯片、选通目标模式操作（但是会危及通常的主机模式操作）、清除终端状态来完成。

---

### 程序清单 9-1 芯片初始化

---

```
WriteReg (CR0, 0x80);      // 复位 53C400A 芯片
WriteReg (MR, 0x40);      // 允许目标模式操作
WriteReg (SER, 1<<ID);   // 确定目标器的 ID SCSI
ReadReg (RIR);            // 读取/复位 SCSI 中断信号
WriteReg (CR0, 0x10);      // 允许 SCSI 中断信号
```

现在，我们只需静静地等待中断的产生，以表明另一个主机适配器已经选通了 53C400A 芯片。实际上我们有时可以仅仅像本例中一样进行简单设置，但是在大多数情况下还需要一个真正的中断处理程序。一旦我们发现中断信号，应该检测是否产生了总线重置或奇偶校验错误。如果我们被选为 SCSI 事务处理中的目标器，那么应该对启动器的 SCSI ID 作出判断。需要注意的是，在我们对选择位进行双检验 (double-check) 时，一定要保证选择正确的目标器。在实际应用中，由于低频干扰经常会出现选择的错误，另外在一些老式的主机适配器上，在总线仲裁或选择处理时还会产生一些奇怪的现象。当我们确认已经被正确选取时，应该插入 BSY 信号以通知启动器已经接收了 SCSI 总线的控制权。然后启动器会降低 SEL 信号的电平作为响应。大多数新式的 SCSI 芯片组会自动地为我们处理这一切，但是在 53C400A 芯片中应该对这些操作熟练掌握，因为这需要人工干预才能完成。

---

### 程序清单 9-2 响应总线选择阶段

---

```
while (!(ReadReg (ISR) & IRQ))
// 等待中断的产生
;

irq_cause = ReadReg (ISR);      // 保存中断源
ReadReg (RIR);                  // 复位 53C400A 中断

if (irq_cause & BUS_RESET)      // 总线复位中断?
{
    ProcessBusReset();          // 在其他地方处理该中断
```

```

    return;
}

if (irq_cause & PARITY_ERROR)
// 是否是奇偶校验错
{
    ProcessParityError();      // 在其他地方处理该错误
    return;
}

// 由以上判定可知，这一定是一个选择中断，我们可以从SCSI数据
// 总线中获取选择ID，并确保该ID位处于置“1”的状态（SCSI总
// 线上的低频干扰可能会导致一个非法的选择中断产生）

selection_id = ReadReg(DATA);
if (!(selection_id & (1 << ID)))
{
    // 由于此时 SCSI ID 位并没有置“1”，
    // 所以并没有真正选择，所以忽略该选择中断信号
    return;
}

// 现在通过检测所设置的ID位来判断启动器的SCSI ID

selection_id &= ~ID;          // 首先清除ID位
for (initiator=0; initiator<8; initiator++)
{
    if (selection_id & 1)
        break;
    selection_id >>= 1;
}

// 现在产生的是合法的选择中断，插入“BSY”信号并
// 等待主机将“SEL”信号拉低以结束选择阶段

WriteReg(ICR,BSY);
while (ReadReg(CSC) & SEL)
;

```

到此为止，我们已经完成了 SCSI 总线仲裁和选择阶段，对于目标器来说，也已经获得了 SCSI 总线的控制权。在 SCSI-2 规范中，下面将会进入信息输出阶段，并在这个阶段中至少发送一个 Identify 消息，但是对于老式的 SCSI-1 启动器来说，这个步骤就可以省略了。我们可以通过检测 ATN 信号的状态断定启动器是否向我们发送了什么消息。在对下面的代码进行检验时，要注意我们是通过设置或清除各种信号来对 SCSI 总线的阶段进行控制的，

另外我们通过 REQ 和 ACK 信号来选通每个消息字节。53C400A 芯片组可以自动进行 128 字节的数据块传输，但是对于较小的字节块来说还是要靠我们自己来完成。

### 程序清单 9-3 阶段中的消息

```
while (ReadReg(ISR) & ATN) // 主机信息等待
{
    WriteReg(TCR,MSG|CD); // 设置MSG和C/D信号以进入MSG OUT 阶段

    WriteReg(TCR,MSG|CD|REQ); // 现在插入REQ信号以向主机请求消息字节

    while (!ReadReg(ISR) & ACK);
    // 等待主机的确认信号，否则无法获取数据

    msg_byte = ReadReg(DATA); // 首先读取消息字节

    WriteReg(TCR,MSG|CD); // 释放REQ信号，保留MSG和CD信号

    if (msg_byte & 0x80) // 是否为确认消息？
    {
        // 摘取出确认消息
        ok_to_disconnect = msg_byte & 0x40;
        luntar = msg_byte & 0x20;
        lun = msg_byte & 0x07;
    }
    else if ((msg_byte == 0x06) || (msg_byte == 0x0C))
    {
        // Abort or Bus Device Reset message
        WriteReg(TCR,0); // 释放MSG和CD信号
        WriteReg(ICR,0); // 释放BSY信号
        return;
    }

    else if ((msg_byte == 0x08))
    {
        // 空指定，也就是忽略不处理
    }
    else
    {
        // 本例不对任何消息字节进行处理，故返回给主机一个拒绝消息
        WriteReg(TCR,MSG|CD|IO); // 进入 MSG IN 阶段
        WriteReg(DATA,0x07); // 拒绝消息
    }
}
```

```

WriteReg(ICR,BSY|DB);           // 启动数据总线
WriteReg(TCR,MSG|CD|IO|REQ);   // 插入 REQ 信号
while (!ReadReg(ISR) & ACK)    // 等待主机的确认信号
{
;
WriteReg(TCR,MSG|CD|IO);       // 释放 REQ 信号
WriteReg(ICR,BSY);             // 关闭数据总线
}
}

```

尽管上述的事务处理似乎很多，不过需要我们处理的只有很少的几条消息。实际应用中通常还要处理不少其他的消息，其中包括 Synchronous 和 Wide Data Transfer Request 消息。在通常情况下，我会创建一个 I/O 子程序来对代码进行一些简化，但是在本例中我觉得最好把总线阶段的改变介绍得更加清楚一些。

在消息输出阶段（从主机向目标器发送消息）之后，将会有 一个目标器进入消息阶段，并从主机收集有关的命令描述块（CDB）。这一切是相当简单明了的，唯一还有点复杂的就是要通过第一个命令字节对 CDB 的长度进行判断。

#### 程序清单 9-4 命令阶段

```

WriteReg(TCR,CD);               // 选择 COMMAND 阶段
WriteReg(TCR,CD|REQ);          // 插入 REQ 信号
while (!ReadReg(ISR) & ACK)    // 等待主机的确认信号
;
cdb_byte[0] = ReadReg(DATA);   // 获得COB的第一个字节
WriteReg(TCR,CD);              // 撤销 REQ 信号
switch (cdb_byte[0] >> 5)
// 取出命令组代码以获取CDB 码的长度信息

{
case 0:
    cdb_len = 6;
    break;
case 1:
case 2:
    cdb_len = 10;
    break;
case 5:
    cdb_len = 12;
    break;
default:
// 保留或厂商自定，作出错误进行处理，并提前终止
    cdb_len = 0;
}

```

```

// Now read the remaining CDB bytes
for (i=1; i<cdb_len; i++)
{
    WriteReg(TCR, CD|REQ);           // 保持 REQ 信号有效
    while (!ReadReg(ISR) & ACK)     // 等待主机确认信号
    ;
    cdb_byte[i] = ReadReg(DATA);    // 取得下一个 CDB 字节数据
    WriteReg(TCR, CD);             // 清除 REQ 信号
}

```

此时我们已经获得了一个完整的 SCSI 命令描述块，为下一步的处理做好了准备。由于我们已经接收了主机的 Identify 消息，所以很容易判断将要接收 CDB 的 LUN。（如果我们将从 SCSI-1 启动器接收该消息，那么可以假设 0 号 LUN 接收 CDB，也可以从 cdb\_byte[1] 的前三位获得 LUN。在 SCSI-2 标准之前，对于多 LUN 来说经常会由于缺少 Identify 消息而产生很多问题。）

如果将要执行的命令需要进行数据传输，那么接下来将会进入数据输出阶段（从主机向目标器）。与前面类似，这也是通过 REQ/ACK 信号选通每个命令字节的。就像我们在前面处理 Message Reject 一样，我们还可以进入一个接收消息阶段，来向主机发送其他的消息（从目标器到主机），选通需要的消息数据。如果需要的数据尚未准备好，该命令必须等待一段时间，那么此时我们就可以断开与总线的连接，在数据准备好之后再重新进行连接。在下面叙述的例子中，假设我们执行的是标准的 Inquiry 命令。

### 程序清单 9-5 查询命令的响应

```

if (cdb_byte[0] == INQUIRY)
{
    // 返回请求的数据。假定正确的请求数据保存在inq_byte[]数组中
    nbytes = cdb_byte[4];           // 取得清除数据的长度

    if (nbytes > sizeof(inq_byte)) // 将请求数据长度修整至准确值
        nbytes = sizeof(inq_byte);

    WriteReg(TCR, IO);             // 选择 DATA IN 阶段

    WriteReg(ICR, BSY|DB);         // 启动数据总线

    for (i=0; i<nbytes; i++)
    {
        WriteReg(DATA, inq_byte[i]); // 写入下一个字节内容
        WriteReg(TCR, IO|REQ);      // 设置 REQ 信号
        while (!ReadReg(ISR) & ACK) // 等待主机的确认信号
    }
}

```

```

    WriteReg(TCR, IO);           // 清除 REQ 信号
}

WriteReg(ICR, BSY);          // 关闭数据总线
}

```

此时我们可以检测是否出现了来自主机的有效 ATN 信号，如果出现了有效的 ATN 信号，表明还有尚未传输的消息字节。在实际应用中，我们需要在每次总线阶段改变的时候以及在数据传输过程中每次数据块传输结束之后进行同样的检验，检查是否还要进一步进行数据传输。在我们的例子中，我们将跳过检验这一步，直接进行最后一步，向主机发送状态字节和命令完成消息，并释放总线的控制权。

### 程序清单 9-6 发送状态和命令完成

```

status_byte = 0x00;           // "GOOD" 状态
WriteReg(TCR, CD|IO);        // 选择 "STATUS" 阶段
WriteReg(ICR, BSY|DB);       // 启动数据总线
WriteReg(DATA, status_byte); // 发送状态字节数据
WriteReg(TCR, CD|IO|REQ);    // 设置 REQ 信号
while (!ReadReg(ISR) & ACK) // 等待主机的确认信号
;

WriteReg(TCR, CD|IO);        // 清除 REQ 信号
WriteReg(TCR, MSG|CD|IO);    // 进入 "MSG IN" 阶段
WriteReg(DATA, COMMAND_COMPLETE); // 发送 Command complete 信号
WriteReg(TCR, MSG|CD|IO|REQ); // 设置 REQ 信号
while (!ReadReg(ISR) & ACK) // 等待主机的确认信号
;
WriteReg(ICR, BSY);          // 关闭数据总线
WriteReg(TCR, 0);            // 释放所有的 SCSI 数据线
WriteReg(ICR, 0);            // 进入 "BUS FREE" 阶段

```

正如我们所见到的，对 SCSI 总线进行管理和控制并不是一件轻而易举之事。尽管我们已经处理了一个典型的事务，但是还没有真正地处理过消息和差错等繁琐的细节。而且，我们也一直参与硬件的直接管理和控制。进行细节处理是相当烦人的，比较好的方法是把它留给芯片组的数据单来做，尤其是在我们使用一个完全不同的 SCSI 适配器来进行目标模式的编程时。与其等待相当长的时间来处理 53C400A 编程的细节，不如把它隐藏在前述的目标模式编程内部。如果哪位读者对这些低层次的编程细节感兴趣的话，最好参考本书附带的 CD-ROM 中有关目标模式编程的源代码，相信会有很大帮助。

## 目标模式 API

我们首先考虑目标模式的 SCSI 接口的需求分析。首先也是最主要的就是，就像 ASPI 接口一样，它是一种容易理解和使用的主机模式的编程方法。ASPI 最大的优势就是它的简单易行性，在 ASPI 中还没有出现标志序列和异步事件通知技术，它还仅仅是一种简单、易于理解的接口。通常我们所说的目标模式接口也是这样。我们不必关心 SCSI 目标模式编程的每一个特性，只要掌握与我们的工作相关的几个特性就可以了。而且，目标模式的 SCSI 编程应该使应用程序与硬件脱离直接的关系。因为这样可以使我们在不同的目标模式的接口上运行同一个应用程序，同样，ASPI 的应用程序可以运行在任何 ASPI 的设备上。

需要大家注意的是，在此我并没有试图创建一个标准规范，我只是想把我最近开始使用而且已经颇有心得的应用程序接口（API）进行一下介绍。希望在大家编写基于 PC 的目标模式的应用程序的时候，能够对大家有所帮助。我们可以从本书附带的光盘中获得有关 53C400A TSPI 驱动器完整的源代码。当然，我们使用的不一定就是这一种芯片组，我们还可以在目标模式的应用程序中使用其他的芯片组。如果我们使用的是其他类型的芯片组，我们必须对 TSPI 驱动器的源代码进行修改，以适应我们所选择的芯片组。我曾经试图把硬件规范代码从更多的通用目标模式例行程序中分离开来，以使这个过程变得相对简单一些。

我是通过无数次艰苦的试验和错误的设计过程之后，才得出这个特殊的应用程序接口（API）的。正是这些试验和过程提供了一个相当完全和容易理解（我希望是）的例行程序，这个例行程序把应用程序从 SCSI 总线事务处理的细节中分离开来。该应用程序接口（API）所要完成的功能中包括：

- 为该过程向应用程序传输完整的 CDB
- 为应用程序提供传输数据缓冲区的读/写例行程序
- 尽可能透明地处理 SCSI 的消息字节
- 为断开/重接提供时序
- 允许多 LUN 同时工作

现在我们开始检验包含应用程序接口（API）的中心数据结构和例行程序，这些数据结构和例行程序就是所谓的目标模式的 SCSI 编程接口或者是 TSPI。我已经使用与 ASPI 相同的程序为该接口制定了模型。该接口具有一个唯一的入口点 `tspi_SendCommand()`，这个命令需要传输一个指针到该命令描述的数据结构中，因为大家在第七章中已经对 ASPI 比较熟悉，所以在学习这种接口时也不会产生什么障碍。

在编写该应用程序时，我尽量使源代码简单一些，但又尽量使它具有足够的灵活性以适应大多数的主机适配器。TSPI 接口的工作方式与 ASPI 接口非常相似，它也具有一个称之为 `tspi_SendCommand (void *)` 的唯一的入口点，执行一个给定的 TSPI 命令时，要传递一个指针到包含所有必需信息的数据结构中。后面我们很快就会介绍执行 TSPI 命令的方法。下面我们先来看一下用来在 TSPI 接口中传递信息的命令的结构。

---

**程序清单 9-7 TSPI 命令结构**

---

```
typedef struct TSPI_EVTNT_s
// 保持输入的 SCSI CDB 和总线事件
{
    unsigned char CommandCode;          // 命令类型
    unsigned char Error;                // 返回错误状态

    unsigned short Flags;               // TSPI_FLAG_xxxx
    unsigned char AdapterIndex;         // 适配器编号
    unsigned char InitiatorId;         // 信号源
    unsigned char Lun;                  // 所选择的 LUN
    unsigned char Reserved1[15];        // 预留给 API 使用
    unsigned long Timeout;              // 以毫秒为单位
    unsigned char Reserved2[3];         // 保留（校正）
    unsigned char CdbLength;            // CDB 码的长度
    unsigned char CdbByte[16];           // CDB 的数据字节
} TSPI_EVENT;

typedef struct TSPI_CMD_
// 通用的 TSPI 命令
{
    unsigned char CommandCode;          // 命令类型
    unsigned char Error;                // 返回错误状态
    unsigned short Flags;               // TSPI_FLAG_xxxx
    unsigned char AdapterIndex;         // 适配器编号
    unsigned char InitiatorId;         // 主机 SCSI ID
    unsigned char Lun;                  // 所选择的 LUN
    unsigned char Reserved1[15];        // 预留给 API 使用
    unsigned long Parm[6];              // 通用参数
} TSPI_CMD;

typedef struct TSPI_XFER_s
// 数据传输命令
{
    unsigned char CommandCode;          // 命令类型
    unsigned char Error;                // 返回错误状态
    unsigned short Flags;               // TSPI_FLAG_xxxx
    unsigned char AdapterIndex;         // 适配器编号
    unsigned char InitiatorId;         // 主机 SCSI ID
    unsigned char Lun;                  // 所选择的 LUN
    unsigned char Reserved1[15];        // 预留给 API 使用
    unsigned long TransferLength;       // 读/写字节
```

```
void * TransferAddress; // 数据缓冲区地址
unsigned long ResidualLength; // 非发送字节数据
unsigned long Reserved2[3]; // 保留（校正）
} TSPI_XFER;

// 标志
#define TSPI_FLAG_BusReset 0x0001
#define TSPI_FLAG_DeviceReset 0x0002
#define TSPI_FLAG_HostMsgWaiting 0x0004
#define TSPI_FLAG_SaveDataPointers 0x0008

// 命令代码
#define TSPI_CMD_AdapterInfo 0
#define TSPI_CMD_AttachLUN 1
#define TSPI_CMD_DetachLUN 2
#define TSPI_CMD_GetEvent 3
#define TSPI_CMD_ReadFromHost 4
#define TSPI_CMD_WriteToHost 5
#define TSPI_CMD_CompleteCommand 6
#define TSPI_CMD_SendMessage 7
#define TSPI_CMD_GetMessage 8
#define TSPI_CMD_Disconnect 9
#define TSPI_CMD_Reconnect 10

// 错误代码
#define TSPI_ERR_None 0
#define TSPI_ERR_InvalidCommand 1
#define TSPI_ERR_Busy 2
#define TSPI_ERR_InvalidAdapter 3
#define TSPI_ERR_InvalidTarget 4
#define TSPI_ERR_InvalidLUN 5
#define TSPI_ERR_LunNotAvailable 6
#define TSPI_ERR_Timeout 7

.
```

大家一定都已经看到，在每个命令结构的前几个参数域中几乎完全相同。TSPI 控制器使用 CommandCode 参数来解释其余的域。TSPI\_EVENT 命令用来在它们到达目标器的时候对 SCSI 命令和事件进行检索。TSPI\_XFER 命令对 SCSI 总线上的缓冲区数据传输进行管理。最后，TSPI\_CMD 命令用来控制 SCSI 事务处理的其余部分，包括断开/重接的时序以及命令的执行过程。而 TSPI\_CMD 也可以用来控制 TSPI 接口本身。下面我们将进一步介绍 TSPI 的命令。

## ■ Adapter Inquiry (TSPI\_CMD\_AdapterInfo)

该命令可以获得由 TSPI 驱动器管理的目标模式的适配器的详细信息。执行适配器查询命令时，将会以参数 TSPI\_CMD\_AdapterInfo 取代形式参数 CommandCode，并将用所查询的目标模式的适配器的“0”号基本索引取代 AdapterIndex 形式参数。如果指定的适配器并不存在，那么参数 Error 将会返回 TSPI\_ERR\_InvalidAdapter 返回值。否则，数组 parm[] 中将会返回与该适配器相关的信息。

**入口点：**

CommandCode	TSPI_CMD_AdapterInfo
AdapterIndex	适配器“0”号基本索引
Flags	为“0”

**返回值：**

Error	错误代码
Parm[0]	由 TSPI 控制器控制的目标模式的适配器的数量
Parm[1]	该适配器所支持的 TSPI 版本号 0x00000104 (1.04)
Parm[2]	目标器适配器的 SCSI ID
Parm[3~5]	预留参数，通常返回“0”

---

### 程序清单 9-8 TSPI 适配器查询

---

```
TSPI_CMD info;
info.CommandCode = TSPI_CMD_AdapterInfo;
info.AdapterIndex = 0;
info.Flags = 0;
tspi_SendCommand(&info);
NumAdapters = info.Parm[0];
printf("%lu target-mode adapters available\n",
      NumAdapters);
while (info.AdapterIndex < NumAdapters)
{
    if (info.Error)
        printf("Adapter %lu, error %u\n",
               info.AdapterIndex
               info.Error);
    else
        printf("Adapter %lu, ID=%lu version %lu.%02lu\n",
               info.AdapterIndex
               info.ID
               info.Version
               info.SubVersion);
```

```

info.AdapterIndex,
info.Parm[2],
info.Parm[1] >> 8,
info.Parm[1] & 0xFF);

info.AdapterIndex++;
}

```

我们还可以从这个例子中看到 TSPI 的另一种表现形式。其中除了特别标明要返回一个返回值的参数之外，所有的非预留的参数都舍弃不用，因为这样做不必对命令结构重新进行初始化，所以可以大大简化程序代码的长度。

### ■ Attach LUN (TSPI\_CMD\_AttachLUN)

该命令用来通知 TSPI 控制器应用程序将要执行的 SCSI 命令所涉及到的 LUN。在使用某个 LUN 之前，必须首先使用该命令进行选择，而后才可以执行其他的命令。如果该命令执行完毕，没有出现任何异常错误，那么在应用程序中可以执行 TSPI\_CMD\_GetEvent 命令来检索即将到来的 SCSI 命令和总线事件。

**入口点：**

CommandCode	TSPI_CMD_AttachLUN
AdapterIndex	Adapter (0~n)
Lun	LUN (0~7)
Flags	为“0”

**返回值：**

Error	错误代码
-------	------

---

### 程序清单 9-9 TSPI 选用 LUN

---

```

TSPI_CMD attach;
attach.CommandCode = TSPI_CMD_AttachLUN;
attach.AdapterIndex = 0;
attach.Lun = 0;
attach.Flags = 0;
tspi_SendCommand(&attach);
if (!attach.Error)
    printf("LUN %u is now enabled\n", attach.lun);

```

后面我们还将详细介绍这个复查例行程序。

### ☒ Detach LUN (TSPI\_CMD\_DetachLUN)

该命令可以用来通知 TSPI 控制器应用程序将不再对 SCSI 命令作出响应的 LUN。在应用程序结束之前，必须使用该命令来断开所有原来连接上的 LUN。

**入口点：**

CommandCode	TSPI_CMD_DetachLUN
AdapterIndex	将要断开的适配器，与 TSPI_CMD_AttachLUN 相同
LUN	将要断开的 LUN，与 TSPI_CMD_AttachLUN 相同
Flags	为“0”

**返回值：**

Error	错误代码
-------	------

---

#### 程序清单 9-10 TSPI 解除 LUN

---

```
TSPI_CMD attach;      // 首次调用 TSPI_CMD_AttachLUN
TSPI_CMD detach;
detach.CommandCode = TSPI_CMD_DetachLUN;
detach.AdapterIndex = attach.AdapterIndex;
detach.Lun = attach.Lun;
detach.Flags = 0;
tspi_SendCommand(&detach);
if (!detach.Error)
    printf("LUN %u is now disabled\n", detach.Lun);
```

### ☒ Get Event (TSPI\_CMD\_GetEvent)

该命令用来检索执行的 SCSI 命令。TSPI 控制器将会在缓冲区中为每个连接的 LUN 准备一个即将执行的默认命令，而在应用程序中就是通过使用 TSPI\_CMD\_GetEvent 命令来收回该缓冲区中的命令。大多数的应用程序都是通过循环等待的方法，一直到事件发生再进行处理。需要注意的是该命令的结构必须是 TSPI\_EVENT 类型。

**入口点：**

CommandCode	TSPI_CMD_GetEvent
AdapterIndex	适配器的标号，与 TSPI_CMD_AttachLUN 相同
Lun	LUN，与 TSPI_CMD_AttachLUN 相同

---

Flags	为“0”
Timeout	等待输入的 SCSI 命令或总线复位的毫秒数，如果该参数的值设定为“0”，表示如果没有发现待解决的命令或事件，则立即返回。

**返回值：**

Error	错误代码
Flags	在该命令完毕之后，标志位会根据当前 SCSI 总线的状态进行设置。如果 SCSI 总线被复位，那么 TSPI_FLAG_BusReset 标志位将会被置“1”。如果 TSPI 控制器从目标器接受到了一个设备复位消息，则 TSPI_FLAG_DeviceReset 标志位置“1”。如果启动器插入了 ATN 信号，则 TSPI_FLAG_HostMsgWaiting 标志位置“1”，标明将会有一个消息发送过来。然后，可以使用 TSPI_CMD_GetMessage 命令来获取该消息。
CdbLength	数组 CdbByte[] 中的合法 CDB 字节的数量。如果该参数的值为“0”，则表明该事件没有与任何 CDB 字节相联系。发生这种情况的一个例子就是当 TSPI_FLAG_BusReset 或 TSPI_FLAG_DeviceReset 标志位进行了设置的时候，该参数的数值就是“0”。
CdbByte[]	该数组中包括了从启动器获得的 SCSI 的命令描述块。它的长度是由 CdbLength 参数来决定的。

---

**程序清单 9-11 TSPI 获取事件**

---

```

TSPI_CMD attach;      // 首次调用TSPI_CMD_AttachLUN
TSPI_EVENT event;
while (!quit)
{
    event.CommandCode = TSPI_CMD_GetEvent;

    event.AdapterIndex = attach.AdapterIndex;
    event.Lun = attach.Lun;
    event.Timeout = 100;           // 100/毫秒
    event.Flags = 0;
    tspi_SendCommand(&event);
    if (event.Error == TSPI_ERR_NONE)
    {
        if (event.Flags & TSPI_FLAG_BusReset)
            printf("Bus reset detected\n");
    }
}

```

```

    if (event.Flags & TSPI_FLAG_DeviceReset)
        printf("Device reset detected\n");
    if (event.CdbLength > 0)
        ProcessCdb(&event);
    }
    else if (event.Error != TSPI_ERR_Timeout)
    {
        printf("Error %u, quitting...\n", event.Error);
        quit = 1;
    }
}

```

## ■ Read Data From Host ( TSPI\_CMD\_ReadFromHost )

执行该命令可以使 TSPI 控制器从启动器读取一个数据缓冲区的数据。在执行需要向目标设备中写入数据或参数的命令（例如 Write 命令）时通常会使用该指令。给定的适配器将进入一个 Data Out 阶段，同时将从启动器读取给定字节的数据，并将写入到应用程序的缓冲区中。需要注意的是，不必一次读完整个数据缓冲区，我们可以自己把它分成任意多次的传输，但是对于每个数据块来说只能执行一次该命令。当然，大的数据块执行起来更具有优势，所以不要把缓冲区的大小设置得太小。另外，需要注意的是，例程 tspi\_SendCommand() 是属于 TSPI\_XFER 的，而不是 TSPI\_CMD 的结构。

**入口点：**

CommandCode	TSPI_CMD_ReadFromHost
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器 SCSI ID 编号
LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	为“0”
TransferLength	将要传输的字节数
TransferAddress	将要从主机接收数据的缓冲区的指针。该缓冲区必须足够大，以满足需要的字节数目。

**返回值：**

Error	错误代码
Flags	在该命令执行完毕之后，标志位会根据当前的 SCSI 总线状态进行设置。如果复位了 SCSI 总线，则 TSPI_FLAG_BusReset 标志位置“1”。如果启动器插入了 ATN 信号，TSPI_FLAG_HostMsgWaiting 标志位将会置“1”，以表明将会向我们发送一个消息。然后，我们

**ResidualCount** 可以通过使用 TSPI\_CMD\_GetMessage 命令来获取该消息。尚未传输的数据长度。该参数的数值是请求传输的数据长度减去当前已经接收的数据长度。如果所请求的数据已经全部传输完毕，那么该参数的数值就为“0”。

### 程序清单 9-12 TSPI 从主机读取数据

```
TSPI_EVENT *event; // 最初的输入事件结构
TSPI_XFER xfer; // 我们的传输结构
char block_buf[512];
xfer.CommandCode = TSPI_CMD_ReadFromHost;
xfer.AdapterIndex = event->AdapterIndex;
xfer.InitiatorId = event->InitiatorId;
xfer.Lun = event->Lun;
xfer.Flags = 0;
xfer.TransferLength = sizeof(block_buf);
xfer.TransferAddress = &block_buf[0];
tspi_SendCommand(&xfer);
if (!xfer.Error)
    printf("%lu bytes received\n",
        xfer.TransferLength - xfer.ResidualCount);
```

### 写入数据到主机 (TSPI\_CMD\_WriteToHost)

执行该命令可以让 TSPI 控制器向启动器发送一个缓冲区的数据。在执行需要从目标设备中读取数据或参数的命令（例如 Read 命令）时通常会执行该指令。然后给定的适配器将会进入 Data In 阶段，同时将会从应用程序的缓冲区中发送给定长度的数据到主机。需要注意的是，不必一次向主机发送全部数据，我们可以把它分成任意多次的传输，但是对于每个数据块来说只能执行一次该命令。当然，大的数据块执行起来通常更具有优势，所以缓冲区尽量不要设置得太小。需要注意的是 tspi\_SendCommand() 例行程序的结构是属于 TSPI\_XFER 类型的，而不是 TSPI\_CMD 类型的。

入口点：

CommandCode	TSPI_CMD_WriteToHost
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器的 SCSI ID
LUN	原来的 TSPI_EVENT 结构中的 LUN

---

Flags	为“0”
TransferLength	将要传输的数据的长度
TransferAddress	将要发送到主机的数据的缓冲区的指针

**返回值：**

Error	错误代码
Flags	在该命令执行完毕之后，标志位会根据当前的 SCSI 总线的状态进行设置。如果复位了 SCSI 总线，则 TSPI_FLAG_BusReset 标志位会置“1”。如果启动器插入了 ATN 信号，则 TSPI_FLAG_HostMsgWaiting 标志位会置“1”，用以表明将会向我们发送一个消息。然后，我们可以使用 TSPI_CMD_GetMessage 命令获取该消息。
ResidualCount	尚未传输的请求数据长度。该参数的数值是请求的数据长度减去已经发送的数据的长度。如果所有的请求数据都已经发送完毕，则该参数的数值为“0”。

---

**程序清单 9-13 TSPI Data To Host 用主机写入数据**

---

```

TSPI_EVENT *event;           // 最初的输入事件结构
TSPI_XFER   xfer;          // 我们的传输结构
char block_buf[512];
xfer.CommandCode = TSPI_CMD_WriteToHost;
xfer.AdapterIndex = event->AdapterIndex;
xfer.InitiatorId = event->InitiatorId;
xfer.Lun = event->Lun;
xfer.Flags = 0;
xfer.TransferLength = sizeof(block_buf);
xfer.TransferAddress = &block_buf[0];
tspi_SendCommand(&xfer);
if (!xfer.Error)
    printf("%lu bytes sent\n",
           xfer.TransferLength - xfer.ResidualCount);

```

## ▣ Complete Command (TSPI\_CMD\_Complete Command)

该命令是表示通过复查例行程序的 SCSI 命令的结束标志。执行该命令可以使 TSPI 控制器向启动器发送最后的 STATUS 字节，并同时切断与 SCSI 总线的连接。在执行任一个通过复查例行程序接收的命令时，都要通过一个相应的 TSPI\_CMD\_CompleteCommand 命令来终止其执行。一旦为一个给定的通过复查例行程序接收的 TSPI\_EVENT 命令发送了该

命令之后，就不能访问 TSPI\_EVENT 结构中的任何域，因为 TSPI 控制器会立即重新使用它，开始一个新的命令。但是在实际应用中，这并不真正成为一个限制，因为 TSPI\_CMD\_CompleteCommand 通常是给定事件的最后一个执行的命令。

#### 入口点：

CommandCode	TSPI_CMD_CompleteCommand
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器的编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器的 SCSI ID
LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	为“0”
Parm[0]	返回到启动器的状态字节。下面就举一个典型的例子：

- 0x00：良好
- 0x02：检验条件
- 0x08：忙
- 0x18：保留冲突
- 0x22：命令终止

如果想要得到所有可能的状态字节，可以参考 SCSI 的规范说明书。

#### 返回值：

Error	错误代码
-------	------

---

#### 程序清单 9-14 TSPI 命令完成

---

```
TSPI_EVENT *event;           // 最初的输入事件结构
TSPI_CMD cmd;
cmd.CommandCode = TSPI_CMD_CompleteCommand;
cmd.AdapterIndex = event->AdapterIndex;
cmd.InitiatorId = event->InitiatorId;
cmd.Lun = event->Lun;
cmd.Flags = 0;
cmd.Parm[0] = 0x02;          // Check condition
tspi_SendCommand(&cmd);
if (!cmd.Error)
    printf(" Error %u\n",cmd.Error);
```

## ■ Send Message To Host (TSPI\_CMD\_Send Message)

该命令用来向启动器发送一个消息。应用程序可以通过执行该命令向主机发送一个仲裁 SCSI 消息。除了数据必须在 Message In 阶段进行传输之外，它与 TSPI\_CMD\_WriteToHost 命令完全相同。另外，需要注意的是，tspi\_SendCommand() 的例行程序是属于 TSPI\_XFER 结构类型的，而不是 TSPI\_CMD 类型的。

### 入口点：

CommandCode	TSPI_CMD_WriteToHost
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器的 SCSI ID
LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	为“0”
TransferLength	将要传输的数据长度
TransferAddress	将要发送到主机的消息的缓冲区的指针。由应用程序来决定消息字节的形式。

### 返回值：

Error	错误代码
Flags	在该命令执行完毕之后，标志位会根据当前 SCSI 总线的状态进行设置。如果复位了 SCSI 总线，那么 TSPI_FLAG_BusReset 标志位将会置“1”。如果启动器插入了 ATN 信号，TSPI_FLAG_HostMsgWaiting 标志位将会置“1”，用以表明将会向我们发送一个消息（很可能是 Message Reject）。然后，我们可以通过 TSPI_CMD_GetMessage 命令获得该消息。
ResidualCount	尚未传输的请求数据长度。该参数的数值是请求的数据长度减去已经发送的数据长度。如果所有的请求数据都已经发送完毕，则该参数的数值为“0”。需要注意的是，如果启动器请求进入 MESSAGE OUT 阶段，则该例行程序将会提前结束以及时处理主机的消息。该参数还可以表明主机插入的 ATN 所在的消息缓冲区的位置。

---

### 程序清单 9-15 TSPI 用主机发送信息

---

```
TSPI_EVENT *event; // 最初运行的回收例程
TSPI_XFER xfer; // 我们的传输结构
char msg_buf[4];
```

```

msg_buf[0] = RESTORE_POINTERS; // 恢复保存的指针
xfer.CommandCode = TSPI_CMD_SendMessage;
xfer.AdapterIndex = event.AdapterIndex;
xfer.InitiatorId = event.InitiatorId;
xfer.Lun = event.Lun;
xfer.Flags = 0;

xfer.TransferLength = 1;
xfer.TransferAddress = &msg_buf[0];
tspi_SendCommand(&xfer);
if (!xfer.Error)
    printf("%lu bytes sent\n",
           xfer.TransferLength - xfer.ResidualCount);

```

## □ Get Message From Host (TSPI\_CMD\_Get Message)

执行该命令可以从启动器获得消息。当从另一个命令返回的标志位 **TSPI\_FLAG\_HostMsgWaiting** 置“1”的时候，我们就可以使用该命令来获取消息。执行该命令表示启动器已经插入了一个 ATN 信号，并来请求进入 Message Out 状态。通过该命令就可以读取相应的消息。需要注意的是，如果消息是在事务处理的命令阶段之前发送的，那么 TSPI 控制器会自动对到达的信息进行处理。其中包括几乎所有的大多数应用程序中的消息处理，如 Identify Message、Synchronous、Wide Negotiation 和 Bus Device Reset message。但是，如果消息是在数据传输阶段或该阶段之后到达的，那么 TSPI 控制器就不能对其自动进行处理。这些消息可以是 Abort、Disconnect（从主机）和 Initiator Detected Error 消息等。如果所传输的数据产生了错误，那么启动器可以在 Data In 阶段之后发送一个奇偶校验错误消息。如果标志位 **TSPI\_FLAG\_HostMsgWaiting** 置“1”，则表示有这样一则消息。如果设置了该标志位，可以使用 **TSPI\_CMD\_GetMessage** 命令从主机读取相应的消息。如果读取的是奇偶校验错误消息，我们可以发送一个 Restroe Pointers 消息，重新对错误的数据进行传输。

除了数据传输是发生在 Message In 阶段之外，该命令的其余各方面跟 **TSPI\_CMD\_ReadFromHost** 命令的执行方式完全相同。另外，需要注意的是，**TSPI\_SendCommand()** 命令的结构是属于 **TSPI\_XFER** 类型的，而不是 **TSPI\_CMD** 类型的。另外，如果主机在一个 Message Out 阶段发送了多条消息，那么我们也可以读取多条消息到缓冲区中。

**入口点：**

CommandCode	TSPI_CMD_GetMessage
AdapterIndex	原来的 <b>TSPI_EVENT</b> 结构中的适配器编号
InitiatorId	原来的 <b>TSPI_EVENT</b> 结构中的启动器的 SCSI ID

---

LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	为“0”
TransferLength	传输的数据长度
TransferAddress	将要从主机接收的消息所在的缓冲区的指针。该缓冲区必须足够大，以满足请求数据的需要。

**返回值：**

Error	错误代码
Flags	在该命令执行完毕之后，标志位会根据当前 SCSI 总线的状态进行设置。如果复位了 SCSI 总线，TSPI_FLAG_BusReset 标志位将会置“1”。如果启动器又插入了 ATN 信号，TSPI_FLAG_HostMsgWaiting 标志位将会置“1”，以表明将会有另一个消息发送给我们。
ResidualCount	尚未传输的数据长度。它的数值是请求数据的长度减去已经接收的数据的长度。如果所有的请求数据都已经接收到，该参数的数值将会为“0”。我们可以通过该参数计算出实际接收到的消息的字节数。

---

**程序清单 9-16 TSPI 从主机获取信息**

---

```

TSPI_EVENT *event;      // 最初运行的回收例程
TSPI_XFER   xfer;      // 我们的传输结构
char msg_buf[16];

xfer.CommandCode = TSPI_CMD_GetMessage;
xfer.AdapterIndex = event.AdapterIndex;
xfer.InitiatorId = event.InitiatorId;
xfer.Lun = event.Lun;
xfer.Flags = 0;
xfer.TransferLength = sizeof(msg_buf);
xfer.TransferAddress = &msg_buf[0];
tspi_SendCommand(&xfer);

if (!xfer.Error)
    ProcessMessageBytes( msg_buf,
        xfer.TransferLength -
        xfer.ResidualCount)

```

### ☒ Disconnect (TSPI\_CMD\_Disconnect)

执行该命令可以由 TSPI 控制器向启动器发送一个断开消息，然后将断开与 SCSI 总线的连接。当应用程序执行一个长操作时，可以通过该命令释放总线的控制权给其他的事务处理。应用程序在向 TSPI 控制器发送任何其他的命令之前，必须首先执行一个 TSPI\_CMD\_Reconnect 命令重新获得总线的使用权。

**入口点：**

CommandCode	TSPI_CMD_Disconnect
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器的 SCSI ID
LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	如果标志位 TSPI_CMD_SaveDataPointer 置“1”，则表示 TSPI 控制器将会在断开之前向启动器发送一个 Save Data Pointer 消息。如果我们把传输的数据分成了若干数据块，该标志位将会一直处于“1”状态。

**返回值：**

Error	错误代码
-------	------

#### 程序清单 9-17 TSPI 断开连接

```
TSPI_EVENT *event; // 最初输入的事件结构
TSPI_CMD cmd;
cmd.CommandCode = TSPI_CMD_Disconnect;
cmd.AdapterIndex = event->AdapterIndex;
cmd.InitiatorId = event->InitiatorId;
cmd.Lun = event->Lun;
cmd.Flags = TSPI_FLAG_SaveDataPointer;
tspi_SendCommand(&cmd);
if (!cmd.Error)
    printf(" Error %u\n", cmd.Error);
```

### ☒ Reconnect (TSPI\_CMD\_Reconnect)

执行该命令可以使 TSPI 控制器重新选择启动器继续进行 SCSI 事务处理。如果想更详

细地了解该命令，可以参考 TSPI\_CMD\_Disconnect 命令的说明。

#### 入口点：

CommandCode	TSPI_CMD_Reconnect
AdapterIndex	原来的 TSPI_EVENT 结构中的适配器编号
InitiatorId	原来的 TSPI_EVENT 结构中的启动器 SCSI ID
LUN	原来的 TSPI_EVENT 结构中的 LUN
Flags	为“0”

#### 返回值：

Error	错误代码
Flags	如果启动器已经插入了 ATN 信号，那么标志位 TSPI_FLAG_HostMsgWaiting 将会置“1”，以表明将会向我们发送一个消息。然后，可以通过 TSPI_CMD_GetMessage 命令获取该消息。

表 9-18 TSPI 重新连接

```
TSPI_EVENT *event;           // 最初输入的事件结构
TSPI_CMD cmd;
cmd.CommandCode = TSPI_CMD_Reconnect;
cmd.AdapterIndex = event->AdapterIndex;
cmd.InitiatorId = event->InitiatorId;
cmd.Lun = event->Lun;
cmd.Flags = 0;
tspi_SendCommand(&cmd);
if (!cmd.Error)
    printf("Unable to reconnect, error %u\n", cmd.Error);
```

## 连接 TSPI 控制器

我们可以自行设计 TSPI 接口，所设计的接口与使用它的应用程序分开执行。这种情况的典型例子就是设备驱动程序或 Windows DLL（与 ASPI 有些类似）。对于 MS-DOS 的情况，其连接方法与 ASPI 控制器极为类似。我们将会打开 TSPI 控制器的设备驱动程序，然后获得 TSPI 控制器的入口点的地址。这些可以通过如下的代码来完成。

---

**程序清单 9-19 获取 TSPI 入口点**


---

```

.DATA
TspiEntryPoint    DD 0          ; 入口点地址
TspiHandle        DW 0          ; 文件编号
AspiDriverName    DB "TSPIMGR$",0 ; TSPI 设备名

.CODE
GetTspiEntryPoint PROC
    push ds                ; 保存当前数据段
    mov ax,@DATA            ; 装载本地数据段
    mov ds,ax
    lea dx,TspiDriverName   ; 装载驱动程序的偏移地址
    mov ax,3D00h             ; MS-DOS 方式打开文件
    int 21h
    jc failed
    mov [TspiHandle],ax      ; 保存文件编号

    mov bx,[TspiHandle]       ; 装载文件编号
    lea dx,TspiEntryPoint    ; 缓冲区的地址
    mov cx,4                  ; 数据长度-4字节
    mov ax,4402h              ; MS-DOS 中的IOCTL读方式
    int 21h
    jc failed

    mov bx,[TspiHandle]       ; 装载文件编号
    mov ax,3E00h              ; MS-DOS 下关闭文件
    int 21h

    ; 返回TSPI入口点地址
    mov ax,word ptr [TspiEntryPoint]
    mov dx,word ptr [TspiEntryPoint+2]
    pop ds
    ret

failed:
    mov ax,0                  ; 遇到错误返回NULL
    mov dx,0
    pop ds
    ret
GetTspiEntryPoint ENDP

```

然后应用程序可以通过如下的序列获得与 TSPI 控制器的连接。

### 程序清单 9-20 连接到 TSPI 管理器

```
BYTE NumAdapters;

VOID (FAR *tspi_SendCommand)(void FAR *p);

tspi_SendCommand = GetTspiEntryPoint();
if (tspi_SendCommand)
{
    TSPI_CMD tspi_info;
    memset( &tspi_info, 0, sizeof(tspi_info) );
    tspi_info.CommandCode = TSPI_CMD_AdapterInquiry;
    tspi_info.AdapterIndex = 0;
    tspi_SendCommand( &tspi_info );
    if (tspi_info.Error)
    {
        // 有错误发生
        NumAdapters = 0;
    }
    else
    {
        // TSPI 管理器已经安装并且正在运行
        NumAdapters = tspi_info.Parm[0];
    }
}
else
{
    // TSPI 管理器并未安装
    NumAdapters = 0;
}
```

## ■ 使用 TSPI 接口

现在我们为个人电脑定义目标模式的 SCSI 接口。让我们看一下如何在目标模式的应用程序中使用 SCSI 接口。首先要定义应用程序所支持的 SCSI 命令组。如果应用程序模仿的是已经在 SCSI 规范说明书中定义的设备，这一切将会变得相对容易一些，只要按照 SCSI 规范说明书中为该设备所定义的命令组进行设置即可。当我们看到说明书中为数众多的模式读出/选择页、缓冲区和记录选项以及其他特性时，也许会觉得这是相当繁琐的。我们

可以首先做一个大概模仿，然后把要求的特性加上去。如果还要为启动器编写代码，我们可以侥幸只完成一个最小的命令组即可。

如果应用程序并不与任何已定义的 SCSI 设备类型相匹配，那么必须自己对命令组进行定义。这也不是非常困难的事，因为命令组几乎可以通过所需要的功能完全确定下来。你可能会把应用程序中所有的操作设定为厂商自定类型，但是也可以使用各种各样的已经存在的命令来满足我们特定的要求。无论我们采用的是哪种策略，都必须实现所有的强制类型的命令，因为只有这样做，应用程序才可能与所有现存的主机适配器和软件相兼容。另外需要注意的是，一定要保证查询数据中的 Peripheral Qualifier 参数域中的最高位被置为“1”。这表示设备类型是厂商自定的，而且还可以避免其他的应用程序使用。

如果我们自己定义厂家专用的命令，那么应该尽量使它们像 SCSI 命令。例如，其操作码、LUN 和控制字节域要与所有的 SCSI 命令相一致，所以不要对厂家专用的命令重新进行定义。

另外，还要把 CBD 码的长度设定为 6、10 或 12 个字节，如果可能的话，最好使用已经存在的命令组代码。因为一些主机适配器和软件不能处理其他长度的 CBD 码。

应用程序中应该尽可能地使用标准的检测键和 ASC/ASCQ 代码。这样做可以尽量避免主机方的混乱，一些主机应用程序可以根据这些设定值来进行重试和错误修复。

要准备随时对 Inquiry 或 Test Unit Ready 命令作出响应。如果设备所支持的命令可以在操作结束之前完成的话，这一点是非常重要的。举个例子就是，为连续访问设备检测 Rewind 命令。其中包括一个 Immediate 标志位，如果该标志位被置“1”的话，则设备会在完成 Rewind 命令之前不再等待 Rewind 操作。在这种情况下，主机可以再发出一个 Test Unit Ready 命令来确定 Rewind 命令是否已经真正结束。因为执行 Rewind 操作将要花费相当长的时间，所以在涉及磁盘的应用程序中经常利用该特性。另外，需要注意的是，因为有 TSPI 控制器的存在，所以在当前的 SCSI 命令执行完毕之前，我们不必担心会到来另一个新的 SCSI 命令。前面所提到的仅仅适合那些能够较早完成的 SCSI 命令，对于那些不能再接收其他的介质访问磁盘的命令来说，必须首先等待一段时间。

下面我们来看一个简单的使用了几个很常用的 SCSI 命令的目标模式的应用程序。跟原来的例子一样，这些源代码省略了一些细节内容，但是这并不影响大家对源代码的理解。可以在本书附带的 CD-ROM 光盘中获得一个更加完整的例子。

---

### 程序清单 9-21 TSPI 采样应用程序

---

```
#define SENSE_LEN 18
unsigned char SenseData[SENSE_LEN];

#define INQUIRY_LEN 36
unsigned char InquiryData[INQUIRY_LEN] =
{ your inquiry data here };
```

```
void (FAR *tspi_SendCommand)(void FAR *p);

int main(int argc, char *argv[])
{
    TSPI_CMD     cmd;
    TSPI_EVENT   event;

    tspi_SendCommand = GetTspiEntryPoint();
    if (!tspi_SendCommand)
    {
        printf("TSPI manager not installed\n");
        return 3;
    }

    // 连接到适配器 0 的 LUN0 上
    cmd.CommandCode = TSPI_CMD_AttachLUN;
    cmd.Adapter = 0;
    cmd.Lun = 0;
    tspi_SendCommand(&cmd);
    if (cmd.Error)
    {
        printf("Error %u trying to attach LUN\n",
               cmd.Error);
        return 4;
    }

    SetSenseData(SKEY_NoSense, 0, 0);

    while (!kbhit())      // 一直运行一直到有按键发生
    {
        event.CommandCode = TSPI_CMD_GetEvent;
        event.AdapterIndex = cmd.AdapterIndex;
        event.Lun = cmd.Lun;
        event.Timeout = 100;           // 100/毫秒
        tspi_SendCommand(&event);
        if (event.Error == TSPI_ERR_None)

        {
            if (event.Flags & TSPI_FLAG_BusReset)
                printf("Bus reset detected\n");
            if (event.Flags & TSPI_FLAG_DeviceReset)
                printf("Device reset detected\n");
        }
    }
}
```

```
    if (event.CdbLength > 0)
        ProcessCdb(&event);
    }

    else if (event.Error != TSPI_ERR_Timeout)
    {
        printf("Error %u, quitting...\n", event.Error);
        break;
    }
}

// 退回到从 LUN 上断开连接
cmd.CommandCode = TSPI_CMD_DetachLUN;
cmd.Adapter = 0;
cmd.Lun = 0;
tspi_SendCommand(&cmd);
return 0;
}

void ProcessCdb(TSPI_EVENT *event)
{
    int status;
    TSPI_CMD cmd;

    switch (event->CdbByte[0])
    {
        case TEST_UNIT_READY:
            status = TestUnitReady(event);
            break;
        case INQUIRY:
            status = Inquiry(event);
            break;
        case REQUEST_SENSE:
            status = RequestSense(event);
            break;
        default:
            // 无载体命令
            SetSenseData( SKEY_IllegalRequest,
                          ASC_InvalidCommandCode, 0 );
            status = CHECK_CONDITION;
    }
}
```

```
// 通知 TSPI 管理器已经准备就绪并向启动器返回状态字节

cmd.CommandCode = TSPI_CMD_CompleteCommand;
cmd.AdapterIndex = event->AdapterIndex;
cmd.TargetId = event->TargetId;
cmd.Parm[0] = status;
tspi_SendCommand(&cmd);
}

void SetSenseData( int skey, int asc, int asq )
{
    int i;
    for (i=0; i<SENSE_LEN; i++)
        SenseData[i] = 0;
    SenseData[0] = 0x70;           // Current error
    SenseData[2] = skey & 0x0F;   // Sense Key
    SenseData[12] = asc;         // Additional sense code
    SenseData[13] = asq;         // Additional qualifier
}

int TestUnitReady(TSPI_EVENT *event)
{
    // 用一个全局变量“Ready”来表明就绪状态
    if (Ready)
        return GOOD;
    else
        return CHECK_CONDITION;
}
int RequestSense(TSPI_EVENT *event)
{
    // 向主机返回检测数据
    TSPI_XFER xfer;
    xfer.CommandCode = TSPI_CMD_WriteToHost;
    xfer.AdapterIndex = event->AdapterIndex;
    xfer.InitiatorId = event->InitiatorId;
    xfer.TargetId = event->TargetId;
    xfer.Lun = event->Lun;
    if (event->CdbByte[4] > SENSE_LEN)
        xfer.TransferLength = SENSE_LEN;
    else
        xfer.TransferLength = event->CdbByte[4];
    xfer.TransferAddress = &SenseData[0];
```

```
tspi_SendCommand(&xfer);
    if (xfer.Error)
    {
    }
    return GOOD;
}

int Inquiry(TSPI_EVENT *event)
{
    if (event->CdbByte[1] & EVPD_BIT)
    {
        // 本例中不支持关键页数据，所以此处返回 CHECK CONDITION 消息
        SetSenseData( SKEY_IllegalRequest,
                      ASC_InvalidFieldInCdb, 0)
        return CHECK_CONDITION;
    }
    else
    {
        // 向主机返回规范的请求数据
        TSPI_XFER xfer;
        xfer.CommandCode = TSPI_CMD_WriteToHost;
        xfer.AdapterIndex = event->AdapterIndex;
        xfer.InitiatorId = event->InitiatorId;
        xfer.TargetId = event->TargetId;
        xfer.Lun = event->Lun;
        if (event->CdbByte[4] > INQUIRY_LEN)
            xfer.TransferLength = INQUIRY_LEN;
        else
            xfer.TransferLength = event->CdbByte[4];
        xfer.TransferAddress = &InquiryData[0];
        tspi_SendCommand(&xfer);
        return GOOD;
    }
}
```

## 第十章 Windows 环境下的 SCSI

由于大多数的计算机上安装的是 Microsoft Windows 作为基本的操作系统，所以如果我们不对 Windows 环境下的 SCSI 编程作介绍，那就可就是玩忽职守了。首先需要提醒大家的是，ASPI 控制器可以应用在当前所有版本的 Windows 环境中（3.x、95 和 NT）。在理想情况下，我们有充足的理由说，我们可以不必考虑应用程序所运行的操作系统，而只要编写运行 ASPI 的代码即可，完全可以把一些初始化的细节放在一边。不幸的是，我们生活的是一个真实的世界，在 Windows 环境下使用 ASPI 时，我们还必须考虑一些附加的要求和限制。在本章中，我们首先要对这些要求和限制作一下介绍，然后再来看一下在 Windows95 和 NT 环境下，是如何完成 SCSI 的编程的。

### Windows 3.x 下的 ASPI

Windows 3.x 是在 DOS 的基础上开发的真正的图形界面操作系统（我敢打赌以前你没有听说过）。从我们的目的出发来说，Windows 3.x 与其他操作系统的最重要的不同之处就是，在 Windows 3.x 中仍然使用 DOS 来完成它的服务，而且 DOS 中的设备驱动器（例如 ASPI 控制器）仍然可以在 Windows 3.x 中装载和使用。知道这一点是很重要的，因为尽管在 Windows 3.x 的规范说明书中定义了真正的 ASPI，但并不是所有的 SCSI 适配器的制造商都为需要使用 WinASPI 的 Windows 3.x 提供了驱动器。由于大多数的 Windows 的应用程序必须和所有的 SCSI 适配器相匹配，所以就不得不舍弃 WinASPI 的支持，不能使用 DOS 保护模式界面（DPMI），而直接在真模式下使用 ASPI 驱动器。不幸的是，这种技术仍然存在种种限制。例如，任何传送给真模式下的 ASPI 控制器的 SRB 和数据缓冲区必须属于 1M 的真模式寻址范围之内。这就意味着，在应用程序中必须在 DOS 所允许的寻址范围内对数据进行拷贝。我们可以参考 1994 年 3 月“Dr Dobb’s Journal”中 Brian Sawert 的“The Advanced SCSI 编程界面”一文，来获得使用 DPMI 服务访问真模式的 ASPI 控制器的详细说明。

Adaptec 最终发布了支持 Windows 3.x 应用程序的 ASPI 规范说明书和驱动器。它们使用的依然是 DOS 中的 SCSI 驱动器，但是它们有一个 Windows 3.x 的虚拟设备驱动器（VxD），还有一个可以在 Windows 3.x 的应用程序中使用的 DLL（动态链接库）文件。VxD 会自动地对增强模式的内存管理进行处理，所以应用程序不必再从 DOS 所允许的寻址范围内进行数据传输和拷贝。另外，使用 WinASPI 的应用程序在传输 SRB 和数据缓冲区时，不必局限在 DOS 所允许的寻址范围之内。我们可以使用标准的 Windows GlobalAlloc（）例程来对超出 1M 的内存进行分配，然后可以直接发送到 WinASPI 控制器。唯一的限制条件就是，在应用程序中要对内存进行页面锁定，以避免在 Windows 3.x 的增强模式中进行 ASPI 操作的过程中与磁盘进行数据交换。要想把内存进行页面锁定，首先要通过 GMEM\_FIXED 属性

性对内存进行分配，然后调用 GlobalPageLock() 例程来对其进行锁定。如果应用程序中使用了 ASPI 记入法（复查），那么必须对其代码段和数据段同时进行锁定，这样才能保证不会把数据移动或交换到磁盘上。

## Win 32 (Windows 95 和 NT) 下的 ASPI

Windows 95 和 NT 本身已经在操作系统中固化了对 SCSI 的支持。在这样的操作系统中对 ASPI 具有强有力的支持。需要特别指出的是，我们不必再对代码段和数据段进行页面锁定。如果有必要的话，SCSI 驱动器会自动对此进行处理。但是，在 Win 32 中的 ASPI 的规范说明书并没有什么大的改变。首先，SRB 的结构布局产生了一些细微变化。另外域进行了重新分布以构成较好的 32 位分布，数组 SRB\_CDBByte[] 通常为 16 字节长，后面还有一个可以接收任何返回的检测数据的新域。这就意味着在我们查看检测数据时，不必再考虑 CDB 代码的长度。

Win 32 (Windows 95 和 NT) 中的 ASPI 为通知应用程序 SRB 的完成与否提供了一个新的方法，即调用 Event Notification。在使用这种方法的时候，SRB 结束之后，将会有个标准的 Win 32 事件对象来作为响应。一旦开始了 SRB，应用程序就可以调用 WaitForSingleObject() 例程来对其进行终止 (block)，直到 SRB 结束为止。这样就减少了由于要查询 SRB 是否结束而浪费的处理器的时间。在 Windows 95 和 NT 环境下，Event Notification 是等待 SRB 结束的推荐方法。而且用这种方法设置比记入法更加快捷和有效，因为记入法需要 Win 32 的 ASPI 专门对 SRB 进行监视。这都是因为在 Windows 95 和 NT 环境下在中断发生的时候，发送例程不能同时运行（但是却可以在 DOS 环境下运行）。附加的线程提供了 Win32 控制器的 ASPI 对发送例程进行模仿的一种方法。但是这样做是以加重系统负担为代价的。

## Windows 95 和 NT SCSI 模型

在 Windows 95 和 NT 中都提供了一系列的分层次的设备驱动器，这些设备驱动器提供了对 SCSI 的多层次的支持。最低层次的支持也就是控制 SCSI 总线事物处理的硬件驱动器；中间层次提供了对系统中所有的 SCSI 适配器单一的、一致的接口界面；最高层次提供的是完成不同特点的各种各样的 SCSI 设备类型的类驱动器。尽管在两个操作系统上完成该操作的细节有所不同，但是它们提供的 SCSI 的模型在概念上却是非常相似的。

在最底层的软件水平上，SCSI miniport 驱动器实现了对 SCSI 接口适配器的直接控制。miniport 驱动器对 SCSI 适配器进行初始化，并向硬件传输 I/O 请求，处理中断的产生，执行适配器水平的错误修复和记录。简而言之，miniport 驱动器是一种小型的、仿制的 SCSI I/O 模型，它隐藏了 SCSI 适配器硬件水平上的细节内容。它们提供了带有连续、低层次的接口

的高水平的 SCSI 模型，而根本不用考虑实际的硬件接口。

正如我们在图 10-1 中所看到的，只要实现了规定的 SCSI miniport 接口，SCSI miniport 驱动器就不必对传统的 SCSI 适配器进行控制。这就允许外设制造商在他们的硬件上使用不同的总线接口，从而只需要最小数量的驱动器支持。只要能够使它们的接口界面在某种层次上看起来像 SCSI，那么它们就能够依靠更高层次的 Windows 驱动器来完成大部分需要控制设备的工作。例如，ATAPI 设备拥有一套几乎与 SCSI 完全一致的命令，但是它们进行的数据传输却是基于 IDE 总线的。Windows 中的 ATAPI miniport 驱动器接受了低层次的 SCSI 命令，并把它们通过 IDE 总线发送出去。它把接口的 IDE 特有的性质隐藏在了 SCSI 的外壳之中。类似的是，Iomega 公司在他们的 Zip 驱动器上使用了 miniport 驱动技术，来模仿通过并行端口进行通讯的 SCSI 接口。图 10-1 是 Win32 下的 SCSI 支持模型。

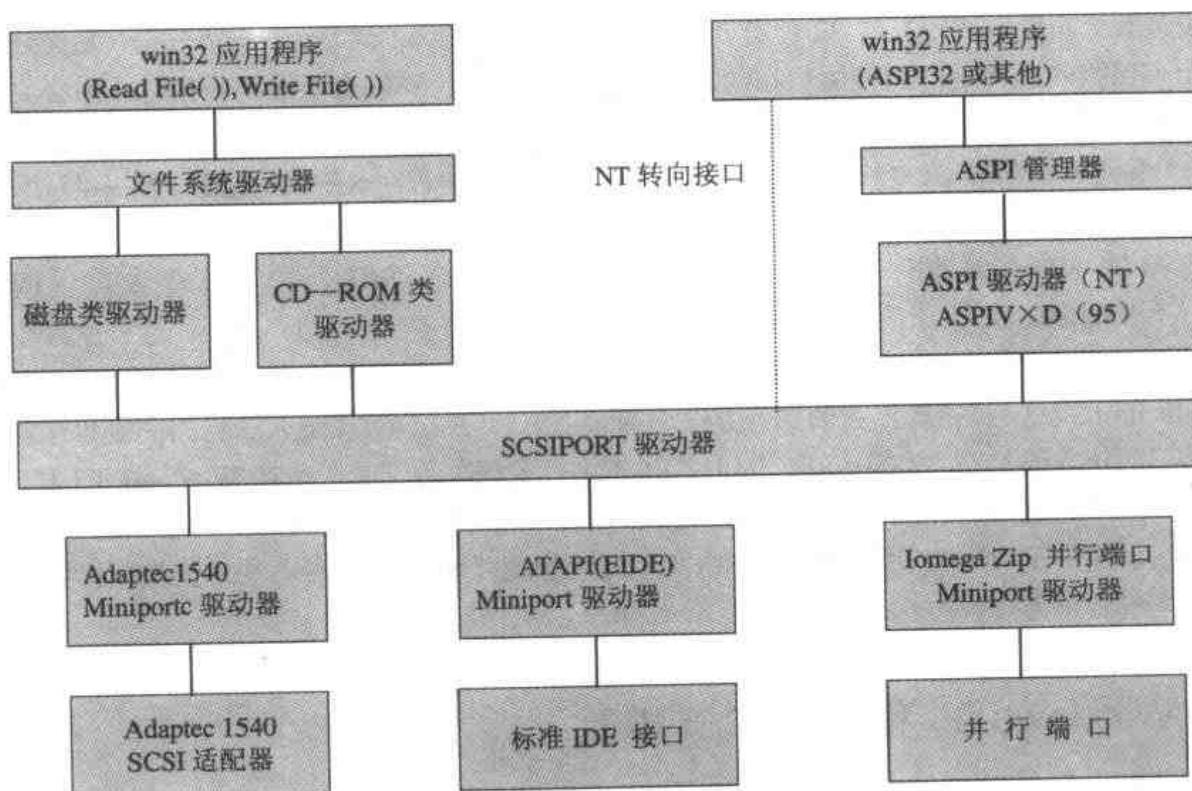


图 10-1 win32 下的 SCSI 支持模型

在这些 miniport 驱动器上面的是称为 SCSIPORT 驱动器的中级驱动器。这种 SCSIPORT 驱动器对于系统中的所有 SCSI 请求提供了唯一的入口点。该驱动器会对系统中各种各样的 miniport 驱动器进行初始化，把系统特有的 SCSI I/O 请求转化成标准的 SCSI 命令描述块 (Command Descriptor Block)，并把这些请求发送给适当的 miniport 驱动器。由于其硬件细节被隐藏在了 miniport 驱动器中，所以高层次的驱动器可以调用 SCSIPORT 驱动器来执行所有的 SCSI I/O 操作，而不必在乎所使用的硬件接口。在 Windows NT 中，应用程序也能够直接通过 DeviceIoControl() 例程来向 SCSIPORT 驱动器发送 SCSI I/O 请求，在后面我们还将对该问题做进一步的讨论。不幸的是，在 Windows 95 中没有为应用程序提供相同

的支持，只有 VxD 和其他的系统组件可以调用 SCSIPORT 驱动器。

在 Windows 分层设备驱动器模型中最高层次的 SCSI 特有的支持是 SCSI 类驱动器(class drivers)。每一种类驱动器负责一种特殊类型的 SCSI 设备的 I/O 请求。在 Windows 中有很多标准的 SCSI 类驱动器，包括用来处理磁盘驱动器、磁带驱动器和 CD-ROM 驱动器的类驱动器。这些类驱动器中的每一种都需要截然不同的高层次的接口界面，但是却都可以使用 SCSIPORT 驱动器来执行低层次的 SCSI I/O 请求。例如，文件系统的驱动器可以调用磁盘类驱动器来执行高层次、面向块的 I/O 请求。磁盘类驱动器会把文件系统的请求转化成一系列的 SCSI I/O 请求，然后它们会被传送到 SCSIPORT 驱动器。磁带类驱动器拥有一个完全不同的高级界面，它适合顺序访问，而不是块访问，并且对文件标记、磁带结束等磁带专用的概念很清楚。磁带类驱动器会把这些高级的磁带请求转化成一个或多个 SCSI I/O 请求，这些请求接着会被传送给 SCSIPORT 驱动器。CD-ROM 和扫描仪的类驱动器非常类似，它们都拥有一个不同的高级界面，调用 SCSIPORT 驱动器来执行 SCSI I/O 的请求。

大家也许会觉得奇怪，ASPI 是如何跟图中相对应的。Windows NT 中的 ASPI 控制器使用一个自定义的设备驱动程序(ASPI32.SYS)直接调用 SCSIPORT 驱动器。而在 Windows 95 中是要通过调用 APIX VxD 来实现其功能的，然后再连接到 SCSIPORT 驱动器上。无论是哪种情况，ASPI 控制器一定要避免应用程序直接对系统使用的特定设备进行访问。在 Windows 进行文件系统的更新时，绝对不要向 SCSI 硬盘驱动器发送命令。Windows 95 和 NT 中的 ASPI 控制器通过在应用程序中隐藏系统保留的设备来对此进行简化。如果想在应用程序中对这些设备进行访问，可以使用标准的文件系统服务。如果真的要在系统使用的过程中对某设备进行访问，那么必须自己编写该设备的驱动程序来进行管理。

## Windows NT 中的 SCSI Pass-Through Interface

在前面我已经提起过，在 Windows NT 中应用程序可以直接发送 SCSI 命令来对 SCSIPORT 驱动器进行访问。这可以通过一个称为 SCSI Pass-Through Interface 或 SPTI 的文件机制来实现。应用程序可以发布各种各样的 SCSI 特有的 IOCTL 来直接对 SCSI 类驱动器或端口驱动器进行访问。然后驱动器会把 SCSI 命令在驱动器的链路上进行传递，并最终发送给设备。如果某类驱动器对一个特殊的设备发出了请求信号，那么我们要做的是向该类驱动器发送 SCSI Pass-Through 命令，而不是发送给 SCSI 端口驱动器。正是由于这种限制的存在，避免了应用程序在没有通知类驱动器的情况下发送命令，而且还可以允许类驱动器保持对设备的控制权。

在使用 SCSI Pass-Through 接口之前，必须首先打开控制相应的设备的类驱动器或端口驱动器。直接存取设备通常是由文件系统驱动器来进行控制的，我们可以通过驱动器号来打开相应的设备。其他的设备都是通过各自的类驱动器来进行访问和控制的。我们可以直接通过设备的 SCSI 适配器驱动器来访问未请求的设备。下面是每种设备名称的例子。

### 设备名称的例子

\\.\C: 硬盘驱动器 C  
 \\.\D: CD-ROM 驱动器 D  
 \\.\Tape0: 磁带驱动器 0  
 \\.\Scsi0: SCSI 适配器 0  
 \\.\Scsi2: SCSI 适配器 2

启动一个驱动器就像打开一个文件一样简单。

### 程序清单 10-1 打开设备驱动器

```

handle = CreateFile( "\\\\"Scsi2:",
  GENERIC_WRITE | GENERIC_READ,
  FILE_SHARE_READ | FILE_SHARE_WRITE,
  NULL, OPEN_EXISTING, 0, NULL );
  
```

一旦我们打开了某个设备，我们就可以使用 IOCTL 调用来获得该驱动器所控制的任何设备上的请求数据，还可以获得主机适配器的权能，执行 SCSI 命令，或对 SCSI 总线进行重新扫描以发现新的设备。所使用的 IOCTL 代码和结构已经定义在了 Windows NT 的设备驱动程序开发包（DDK）中各种各样的头文件里。这些头文件包括：

- ◆ DEVOCTL.H
- ◆ NTDDDISK.H
- ◆ NTDDSCSI.H

可以通过 SCSI Pass-Through 接口使用的调用有：

- ◆ IOCTL\_SCSI\_GET\_INQUIRY\_DATA
- ◆ IOCTL\_SCSI\_GET\_CAPABILITIES
- ◆ IOCTL\_SCSI\_GET\_ADDRESS
- ◆ IOCTL\_SCSI\_RESCAN\_BUS
- ◆ IOCTL\_SCSI\_PASS\_THROUGH
- ◆ IOCTL\_SCSI\_PASS\_THROUGH\_DIRECT

下面我们依次对这些调用进行介绍。

#### ■ IOCTL\_SCSI\_GET\_INQUIRY\_DATA

通过这个 IOCTL 命令可以获取描述每个 SCSI 总线和相应的驱动器所控制的设备的信息。我们先来看一下描述这些信息的结构信息，然后举例说明总线和设备的信息列表。

---

**程序清单 10-2 SCSI (获取查询数据)**

---

```
typedef struct _SCSI_ADAPTER_BUS_INFO
{
    UCHAR NumberOfBuses;           // SCSI总线数目
    SCSI_BUS_DATA BusData[1];     // 数据结构数组
} SCSI_ADAPTER_BUS_INFO, *PSCSI_ADAPTER_BUS_INFO;

typedef struct _SCSI_BUS_DATA
{
    UCHAR NumberOfLogicalUnits;   // 逻辑设备
    UCHAR InitiatorBusId;        // SCSI适配器的ID
    ULONG InquiryDataOffset;      // 请求数据缓冲区
} SCSI_BUS_DATA, *PSCSI_BUS_DATA;

typedef struct _SCSI_INQUIRY_DATA
{
    UCHAR PathId;                // 确定所用的SCSI总线
    UCHAR TargetId;               // 确定所用的SCSI目标器
    UCHAR Lun;                    // 确定所用的SCSI LUN
    BOOLEAN DeviceClaimed;       // 是否被驱动程序请求?
    ULONG InquiryDataLength;     // 请求的数据长度
    ULONG NextInquiryDataOffset; // 下一个 LUN 数据
    UCHAR InquiryData[1];         // 本 LUN 数据
} SCSI_INQUIRY_DATA, *PSCSI_INQUIRY_DATA;

ULONG bus,n;
SCSI_ADAPTER_BUS_INFO *adapter;
char inq_buf[4096];
DeviceIoControl( device_handle, // 来自 CreateFile()
    IOCTL_SCSI_GET_INQUIRY_DATA,
    NULL, 0,
    &inq_buf, sizeof(inq_buf),
    &n, NULL );

// 扫描适配器请求数据输出每一个相关的 BUS/TID/LUN 信息
adapter = (SCSI_ADAPTER_BUS_INFO *) inq_buf;
printf("Bus TID LUN Claimed Inquiry Data\n");
printf("-----\n");
for (bus=0; bus<adapter->NumberOfBuses; bus++)
{
    //
```

```

// 获得第一个逻辑单元请求数据的偏移地址
ULONG inq_offset =
    adapter->BusData[bus].InquiryDataOffset;

while (inq_offset != 0) // end of list?
{
    // 是否到达列表结尾?
    // 获得指向缓冲区中返回的请求数据的指针
    SCSI_INQUIRY_DATA *inq;
    inq = (SCSI_INQUIRY_DATA *) (inq_buf + inq_offset);

    // 输出 BUS/TID/LUN, 而不管该设备是否被类驱动器请求
    printf("%3lu %3u %3u %s ",
        bus, inq->TargetId, inq->Lun,
        inq->DeviceClaimed ? " Yes " : " No " );

    // 现在输出该设备的 SCSI 请求数据
    for (int i=0; i<8; i++)
        printf("%02X ", inq->InquiryData[i]);

    printf("%.28s\n", &inq->InquiryData[8]);

    // 获得指向下一个逻辑单元的请求数据的偏移地址
    inq_offset = inq->NextInquiryDataOffset;
}

```

## ■ IOCTL\_SCSI\_GET\_CAPABILITIES

通过该命令可以确定基本的 SCSI 适配器和 miniport 驱动器的权限。其中包括所允许的最大传输长度、传输所跨越的页数以及传送给它的任何数据缓冲区的校准请求。

---

### 程序清单 10-3 SCSI 获取权能

---

```

typedef struct _IO_SCSI_CAPABILITIES
{
    ULONG Length;                                // 该结构体的长度
    ULONG MaximumTransferLength;                 // 最大传输长度
    ULONG MaximumPhysicalPages;                  // 可以传输的物理页

```

```

ULONG SupportedAsynchronousEvents; // 所允许的异步事件
ULONG AlignmentMask; // 队列请求
BOOLEAN TaggedQueuing; // 是否允许要标签队列
BOOLEAN AdapterScansDown; // 适配器可扫描 BIOS
BOOLEAN AdapterUsesPio; // 适配器是否使用编程 I/O
                           // (与总线主控器和DMA传输结应)

} IO_SCSI_CAPABILITIES, *PIO_SCSI_CAPABILITIES;

IO_SCSI_CAPABILITIES caps;

DeviceIoControl( device_handle, // from CreateFile()
    IOCTL_SCSI_GET_INQUIRY_DATA,
    NULL, 0,
    &caps, sizeof(caps),
    &n, NULL );

```

我们必须时常考虑通过该命令所获得的传输限制。大的数据传输可能会产生失败，更严重的时候可能会导致 Windows NT 的崩溃。因为应用程序不能访问数据缓冲区物理页面，所以我们应该保证传输的数据长度在最大传输长度（MaximumTransferLength）和缓冲区的最坏情况下的页面之内，最坏情况下的页面就是 PAGE\_SIZE\*（MaximumPhysicalPages（最大物理页）-1）。

## ■ IOCTL\_SCSI\_GET\_ADDRESS

通过执行该命令可以获得特定设备的地址信息。需要注意的是，该命令只能应用在类驱动器上。我们可以通过这些地址信息来确定与特定设备相连接的 SCSI 适配器、总线和目标器 ID。

---

### 程序清单 10-4 SCSI 获得地址

---

```

typedef struct _SCSI_ADDRESS
{
    ULONG Length; // 该结构体的长度
    UCHAR PortNumber; // 所控制的SCSI设备号
    UCHAR PathId; // 所使用的总线
    UCHAR TargetId; // 目标器ID
    UCHAR Lun; // 逻辑单元号
} SCSI_ADDRESS, *PSCSI_ADDRESS;

```

```

SCSI_ADDRESS addr;

DeviceIoControl( device_handle, // 来自 CreateFile()
    IOCTL_SCSI_GET_ADDRESS,
    NULL, 0,
    &addr, sizeof(addr),
    &n, NULL );

```

需要注意的是，有关 PathId、TargetId 和 LUN 的信息还可以通过执行 IOCTL\_SCSI\_GET\_INQUIRY\_DATA 命令来获取。PortNumber 域可以用来为端口驱动器创建设备名称。例如，端口号为“2”则表明该设备是通过“\\.\Scsi2”端口驱动器来控制的。

## □ IOCTL\_SCSI\_RESCAN\_BUS

通过执行该命令可以使驱动器重新对 SCSI 总线进行扫描，并查询是否有新的设备出现。它还会为新连接上的设备搜集 SCSI 请求数据，同时又防止任何类驱动器向已经存在的设备发送命令。

---

### 程序清单 10-5 SCSI 重新扫描总线

---

```

DeviceIoControl( device_handle, // 来自 CreateFile()
    IOCTL_SCSI_RESCAN_SCSI_BUS,
    NULL, 0,
    NULL, 0,
    &n, NULL );

```

应用程序可以通过重发 IOCTL\_SCSI\_INQUIRY\_DATA 命令来检验是否有新的设备出现。

## □ IOCTL\_SCSI\_PASS\_THROUGH 和 IOCTL\_SCSI\_PASS\_THROUGH\_DIRECT

这两个命令可以用来向目标设备发送 SCSI 命令。不过这两个命令也有不同之处，IOCTL\_SCSI\_PASS\_THROUGH 对于 SRB 和数据缓冲区使用单一的结构，而 IOCTL\_SCSI\_PASS\_THROUGH\_DIRECT 则允许我们对 I/O 传输的数据所在的缓冲区分别设定地址。这两个命令的结构如下所示。

---

程序清单 10-6 SCSI Pass\_Through

---

```

typedef struct _SCSI_PASS_THROUGH
{
    USHORT Length;           // 本结构体的长度
    UCHAR ScsiStatus;        // 返回的目标器的状态
    UCHAR PathId;            // 路径ID(来自from SCSI_INQUIRY_DATA)
    UCHAR TargetId;          // 目标器ID(来自from SCSI_INQUIRY_DATA)
    UCHAR Lun;                // LUN(来自from SCSI_INQUIRY_DATA)
    UCHAR CdbLength;          // SCSI CDB的长度
    UCHAR SenseInfoLength;    // 检测缓冲区长度
    UCHAR DataIn;             // 方向标志 (参考下文)
    ULONG DataTransferLength; // 将要传输的数据字节
    ULONG TimeOutValue;       // 超时设定(单位为秒)
    ULONG DataBufferOffset;   // 数据缓冲区的偏移地址
    ULONG SenseInfoOffset;    // 检测缓冲区的偏移地址
    UCHAR Cdb[16];             // CDB 字节
} SCSI_PASS_THROUGH, *PSCSI_PASS_THROUGH;

typedef struct _SCSI_PASS_THROUGH_DIRECT
{
    USHORT Length;           // 该结构体的长度
    UCHAR ScsiStatus;        // 返回的目标器状态
    UCHAR PathId;            // 路径ID(来自from SCSI_INQUIRY_DATA)
    UCHAR TargetId;          // 目标器ID(来自from SCSI_INQUIRY_DATA)
    UCHAR Lun;                // LUN(来自from SCSI_INQUIRY_DATA)
    UCHAR CdbLength;          // SCSI CDB 的长度

    UCHAR SenseInfoLength;    // 检测缓冲区长度
    UCHAR DataIn;             // 方向标志(参考下文)
    ULONG DataTransferLength; // 将要传输的数据字节
    ULONG TimeOutValue;       // 超时设定(单位为秒)
    PVOID DataBuffer;          // 数据缓冲区的地址
    ULONG SenseInfoOffset;    // 检测缓冲区的长度
    UCHAR Cdb[16];             // CDB 字节
} SCSI_PASS_THROUGH_DIRECT, *PSCSI_PASS_THROUGH_DIRECT;

```

需要注意的是，除了 DataBufferOffset 和 DataBuffer 域之外，它们的结构几乎完全一致。IOCTL\_SCSI\_PASS\_THROUGH 命令通过数据缓冲区相对于 SCSI\_PASS\_THROUGH 结构开始处的偏移地址来进行寻址，对应的 IOCTL\_SCSI\_PASS\_THROUGH\_DIRECT 命令可以允许我们直接通过指针来对数据缓冲区进行寻址。另外，需要注意的是，在每个命令结构中的 SenseInfoOffset 域表明了其相对于结构开始的偏移量。通常在应用程序中，总是把这

些结构嵌入在它们自己的结构中，如下所示。

### 程序清单 10-7 SCSI Pass-Through 请求

```
typedef struct _NT_SCSI_REQUEST
{
    SCSI_PASS_THROUGH spt;
    unsigned char sense[16];
    unsigned char data[1];
} NT_SCSI_REQUEST;

typedef struct _NT_SCSI_REQUEST_DIRECT
{
    SCSI_PASS_THROUGH_DIRECT spt;
    unsigned char sense[16];
} NT_SCSI_REQUEST_DIRECT;
```

我们还可以通过使用 `offsetof()` 宏命令来确定检测缓冲区的偏移地址信息（通过 `IOCTL_SCSI_PASS_THROUGH` 命令来确定数据缓冲区的偏移地址）。

### 程序清单 10-8 SCSI Pass-Through 数据缓冲区

```
NT_SCSI_REQUEST sr;
sr.spt.SenseInfoOffset = offsetof(NT_SCSI_REQUEST,sense);
sr.spt.DataBufferOffset = offsetof(NT_SCSI_REQUEST,data);

NT_SCSI_REQUEST_DIRECT srd;
char data_buffer[1024];
srd.spt.SenseInfoOffset = offsetof(NT_SCSI_REQUEST,sense);
srd.spt.DataBuffer = &data_buffer[0];
```

下面就如何使用 `IOCTL_SCSI_PASS_THROUGH_DIRECT` 命令实现 SCSI I/O 例程举个简单的例子。我们以后可以使用类似的例程来为 SCSI 设备处理所有的 I/O 命令。

### 程序清单 10-9 使用 SCSI Pass-Through

```
int ScsiCommand( HANDLE device_handle,
    SCSI_INQUIRY_DATA *inq,
    void *cdb_buf, unsigned cdblen,
    void *data_buf, unsigned long dlen,
    int direction,
```

```
void *sense_buf, unsigned slen,
long timeout )
{
    NT_SCSI_REQUEST_DIRECT a;
    ULONG returned;

    memset (&a,0,sizeof(a));
    a.spt.Length = sizeof(a.spt);
    a.spt.PathId = inq->PathId;
    a.spt.TargetId = inq->TargetId;
    a.spt.Lun = inq->Lun;
    a.spt.CdbLength = cdblen;
    a.spt.SenseInfoLength = sizeof(a.sense);
    a.spt.DataIn = direction;
    a.spt.DataTransferLength = dlen;
    a.spt.TimeOutValue = timeout;

    a.spt.DataBuffer = data_buf;
    a.spt.SenseInfoOffset =
        offsetof(NT_SCSI_REQUEST_DIRECT,sense);
    memcpy( a.spt.Cdb, cdb_buf, cdblen );

    if (!DeviceIoControl( h,
        IOCTL_SCSI_PASS_THROUGH_DIRECT,
        &a, sizeof(a), &a, sizeof(a),
        &returned, NULL) )
    {
        int x = GetLastError(); // 检查错误来源
        if (sense_buf && slen)
            memset(sense_buf,0,slen); // 清除检测区域
        return 0xFF; // 返回错误状态
    }

    if (sense_buf && slen)
        memcpy( sense_buf, a.sense,
            (slen < sizeof(a.sense)) ? slen :
            sizeof(a.sense) );

    return a.spt.ScsiStatus;
}
```

## 第十一章 Unix 环境下的 SCSI

在本书的以上部分介绍了如何在行的 MS-DOS、Windows、Windows 95 和 NT 环境中编写 SCSI 设备驱动程序和应用程序。好像近来的发展趋势亦是如此，但是 SCSI 在上述的种种平台上并没有获得很大的发展。在一个使用鼠标和键盘的个人桌面系统中，甚至是一个编程者在软件开发环境中，也仅仅是开始开发具有优越性能 SCSI I/O 子系统而已。

由于在服务器系统中运行的是 UNIX 操作系统，这样 SCSI 的多任务性能就能够发挥它的优越性。因为对于多用户来说，访问的都是多磁盘、多 CD-ROM 等等，所以对于具有单线特性的 IDE 来说，就无法满足其要求。正是因为这个原因，实际上所有当前的服务器都采用 SCSI I/O 系统。

在上一个段落里，我提到了运行着 UNIX 的系统。UNIX 并不是一个真正的操作系统。因为自从 U.C Berkeley 从第 7 版本的 AT&T 代码系统独立开发以来，UNIX 出现过多次分支。为了对 UNIX 进行标准化，已经进行了多次尝试，结果产生了具有在应用程序编程接口水平上兼容的源代码的操作系统。但是，每个 UNIX 内核都是完全独立的。不幸的是，这属于 UNIX 设备驱动程序开发领域。

为数众多的 UNIX 开发商从来没有把应用程序层次上的兼容看成一种必要之举。甚至在 PC 制造商准备对 SCSI API 进行标准化的时候，UNIX 的开发商们仍然没有觉悟。其实当我们准备为多个 UNIX 平台编写 SCSI 设备驱动的时候，这种想法是显而易见的。因为每个平台有它自己完全不同的 SCSI API，而没有注意到内核结构已经完全不同，内核所支持的 I/O 功能也不可一概而论。

正是由于这种情况，在 1985 年左右产生了许许多多的驱动器编写人员为它编写驱动程序。之后不久，ANSI（美国国家标准协会）决定创建一个标准的 SCSI API 规范，该规范的名称为 Common Access Method (CAM)，但是直到 1991 年左右，该规范还没有真正成为一个有用的标准。而且直到那时，所有 UNIX 的制造商们还局限在自己专有的 API 上。只有 Digital Equipment Corporation 采用 CAM 作为他们自己的 Unix 的 SCSI API。

UNIX 下的应用程序开发环境与 MS-DOS 和 Windows 的开发环境有很大的不同之处。用户所编写的应用程序中不允许直接对硬件进行访问。应用程序为了执行 I/O 操作，必须进行系统调用，例如执行“read”调用。调用之后 O/S 就可以确定用以控制将要读取请求数据的设备的驱动程序，并为传输请求数据到用户的内存缓冲区发送请求。

还有一些特殊的应用程序使用另一种方法来进行 O/S，这种方法也就是所谓的 SCSI Pass-Through 驱动程序。在这种情况下，应用程序会创建自己的 SCSI 命令，并把这些命令传递给将要控制该设备的驱动器。一旦命令执行完毕，该状态就会返回到调用的应用程序中。播放 CD 音乐的应用程序就是这样的一个应用例子。

在某些 UNIX 系统中，所有的设备驱动程序都连接在一个庞大的内核文件上。在启动

引导系统的时候，所有的 O/S 映像文件被装载到内存中并运行起来。其他的系统采用一种动态过程，也就是在启动时仅仅装载当前会用到的驱动程序，其他的驱动程序可以在需要的时候再装载。这种动态过程越来越得到广泛的应用，因为它能够节省系统的资源。但是，这样对于编写驱动程序来说增加了不少难度。

本章我们不会就编写所有的 UNIX 平台上的 SCSI 驱动程序和应用程序作详细的讨论。如果要这样做，必须对每个 O/S 用一本书的篇幅进行描述。我只想让读者对各种各样的 UNIX 变体上的 SCSI 编程环境有所了解，然后我再挑选出最流行的 UNIX 变体，并对其作出详细的介绍。

## UNIX 设备驱动程序的简单描述

在 UNIX 系统中，通过核心程序的干涉和调节，使所有的应用程序之间互不影响。所有的 I/O 是通过系统调用传递给 UNIX 核心程序的。设备驱动程序就是运行在核心程序优先级别上的一种程序，这种程序对特定类型的 I/O 设备进行 I/O 操作。在 UNIX 系统中，所有的 I/O 对于应用程序来说都像是文件 I/O 一样，即使 I/O 是针对像终端屏幕一样的硬件设备。在文件系统中（通常在“/dev”目录），每个设备都以一个“特殊的文件”（也可以称之为设备节点）出现。它们的命名规则如下：对于终端来说命名类似“/dev/tty0”的形式；对于硬盘来说命名类似“/dev/hd0”。这些名称是由驱动程序编写者来确定的，但是它们可以由系统管理器来进行改变或进行连接。这些特殊的文件实际上所做的就是记录该设备的主要或次要数。主要数是用来做为所有的设备驱动程序列表的索引的，次要数可以应用在计算机的驱动程序编写者所确定的所有方式中。但是它通常应用在 I/O 所确定的单元上。

一共存在两种类型的 UNIX 驱动程序，分别称为 **block** 和 **character**。使用 **block** 的设备可以访问明确的面向块的设备（如磁盘），而使用 **character** 的设备则可以访问所有其他类型的设备。

**Block** 驱动程序拥有三个入口点：**Open**、**Close** 和 **Strategy**。**Character** 的驱动程序要稍微多一些：**Open**、**Close**、**Read**、**Write** 和 **Ioctl**。

让我们看一下图 11-1 就可以理解所有这些块是如何在一起协调工作的。

驱动程序的操作在两种处理机级别内：用户级和内核级。在用户的应用程序打开驱动程序的时候，也就同时发出了一个数据请求，此时驱动程序就作为用户应用程序的一种扩展（用户级别）。一旦用户的请求发送到了设备上，用户的应用程序将会进入睡眠状态，一直到设备通过驱动器的中断处理程序来中断当前运行的过程为止。中断处理程序是运行在内核级上的。当中断处理程序处理完其缓冲区中的请求数据之后，它会发出一个唤醒请求，以通知用户的应用程序来继续开始执行。中断处理程序的缓冲区中的数据要么是由中断处理程序本身通过 I/O 处理的，要么是设备通过 DMA 进行设置的。现在的 UNIX 设备驱动程序也必须清楚它们的中断处理程序可能会运行在与应用程序不同的处理机上。也就是说，必须为某个驱动程序所访问的结构或变量进行一定形式的锁定。

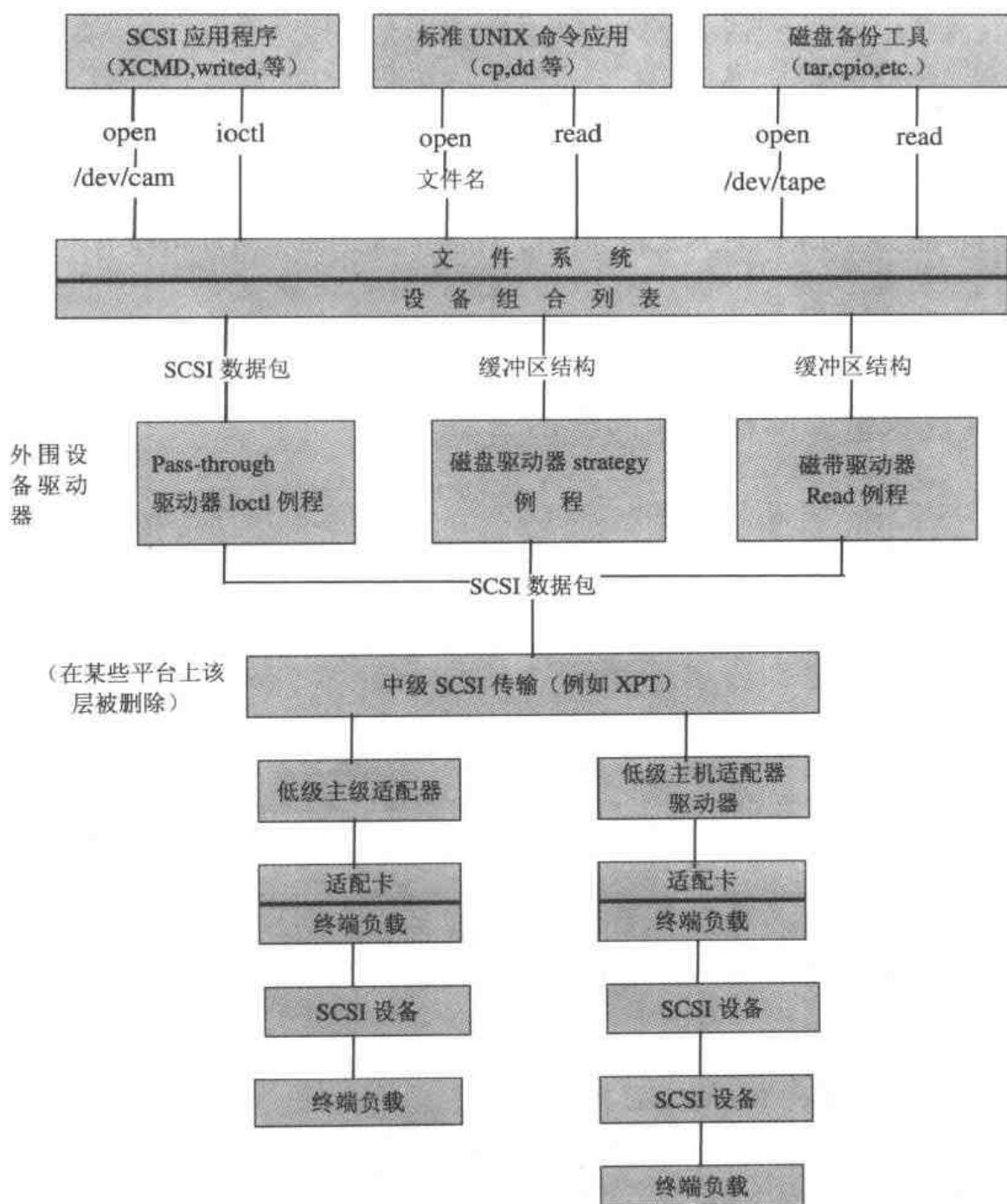


图 11-1 A Typical UNIX SCSI I/O Subsystem

当用户的过程使用标准的文件 I/O 调用来请求数据时，内核的文件系统代码将会调用策略例行程序。通过这种方法所读取的数据都存储在内核的“缓冲存储器”中，这样其他的过程如果需要这些数据，就不必再次进行数据读取的操作。

在进行上述的数据传输操作时所执行的内务操作中会用到一些特殊的内核功能，这些功能名称是不同于 UNIX 的。

## UNIX 形式比较

在这里我希望通过把大多数流行的 UNIX 版本在外观形式上的一些关键点来进行比较来让读者理解得更加深刻。我们还可以从下表中得到在各种不同版本的 UNIX 平台上所开发的驱动程序的有关信息。

表11-1

AIX 4.1 特 性

特    性	描    述
平台	Power RISC和Power PC结构
内核类型	完全动态
内核-内存处理 (allocate/free)	xmattach(),xmalloc(),xmfree()
内核-数据空间转换	copyin(),copyout(),uiomove()
内核-过程分块 (休眠)	e_sleep(),wakeup()
内核-数据控制锁	lock(),unlock(),lock_alloc (),simple_lock_init(),unlock_enable(),lock_free (),disable_lock(),I_enable(),i_disable()
内核-其他功能	fp_opendev(),fp_close(),fp_ioctl().pincode (),unpincode(),pinu(),unpinu(),errsave(),bzero(),uphysio(),dev swadd(),devswdel()
内核-其他结构	buf,sc_buf,sc_iocmd
与SCSI相关的头文件	/usr/include/sys/scsi.h,trcmacros.h,err_rec.h,errids.h,trchkid .h,lock_alloc.h,cdrom.h,scdisk.h
SCSI适配器驱动器接口	devstrat (sc_buf)
必需的驱动器入口点	Drivernameconfig, drivernameopen, drivernameclose, drivernameread, drivernamewrite, drivernameioctl, drivernamestrategy
SCSI pass-through接口	fd=openx("/dev/rct0",O_RDONLY,NULL,SC_DIAGNOS TIC) ioctl(fd, CDIOCMD或DKIOCMD, &sc_iocmd)
驱动器支持命令	Smilt, lsdev, odme, odmadd, odmdelete, odmget, odmchange, odmshow, trace, errpt, errclear, cfgmgr, installp
驱动器支持文件	/etc/drivers/*,/etc/methods/*,/lib/kernex.exp
错误记录	errsave(),errprt
驱动程序代码举例	在3.x版的的Unix中，有一些驱动程序代码的例子放在 O/S的目录中（/usr/lpp/bos/samples/）。还有一些比较好的 例子在下面将要提到的IBM的设备驱动程序手册的附带磁 盘中，其实我们还可以从IBM的ftp站点上来获得一些其他的 的例子 <a href="http://ftp.austin.ibm.com/pub/developer/aix/ddriver/writedd.tar.Z">ftp://ftp.austin.ibm.com/pub/developer/aix/ddriver/writedd.tar.Z</a>

续表

特    性	描    述
调    试	在O/S中还包括一个非常原始的内核调试程序。不过运行该程序需要在串行端口上安装合适的终端负载。只有在开关处于“service”位置时，内核打印功能才有效
适用的驱动程序文件编制	“为AIX 3.2编写设备驱动程序” (IBM P/N GG24-3629-01) “为AIX 4.1编写设备驱动程序” (IBM P/N SC23-2593-03) “内核扩展和设备支持编程概念” (IBM P/N SC23-2611-03)
备    注	AIX的驱动程序比其他的要复杂得多，因为它的驱动程序需要与设备信息的数据库ODM（对象数据管理器）进行信息交互。所有的设备拓扑和标识信息都保存在ODM数据库中  AIX SCSI适配器驱动程序需要能够认识到，发送给它的sc_buf文件中包括一个将要传输的文件列表，或一个文件的名称。如果sc_buf中的buf文件指针不是空(NULL)的，那么表示其中的文件是不止一个。有一个名为odme的编辑器可以直接对ODM数据库进行编辑，这个编辑器仅仅存在于3.x版本中，在4.x版本中并不存在这个工具  lib文件夹中的“export”文件中说明了每个模型中将提供什么功能输出。如果我们不能确定内核功能的确切名称，可以查阅kernex.exp文件，一般都可以解决。如果所使用的驱动器中包括标准入口点之外的入口点，那么必须为该驱动器创建一个输出(export)文件。在这个export文件中包括我们在内核调试器中所需要的任何符号

表 11-2

HP-UX 10.x 特 性

特    性	描    述
平台	HP9000/8xx PA-RISC结构
内核类型	单片电路  注意：自HP-UX 10.xx开始，HP-UX支持两种不同的驱动程序子系统：工作站 I/O (WSIO) 和服务器 I/O (SIO)。一般来说，WSIO驱动程序应用在EISA、HSC和其他一些内嵌式的总线系统中，而SIO驱动程序可以控制HP-IO总线设备
内核-内存处理 (allocate/free)	kmalloc(), kfree(),io_get_mem(),io_rel_mem()
内核-数据空间转换	copyin(),copyout(),bvtospace(),pvtospace(),minphys()
内核-过程分块 (休眠)	biowait(),get_sleep_lock(),sleep(),wakeup()
内核-数据控制锁	alloc_spinlock(),spinlock(),spinunlock()

续表

特    性	描    述
内核-其他结构	physio(),timeout(),biodone(),bzero(),bcopy(),bcmpl()
内核-其他功能	buf,sctl_io,iovec,uioc
SIO-其他功能	io_send(),io_get_trn(),io_get_frame(),io_port_info(),sio_get_pda()
SIO-与SCSI相关的磁头文件	/usr/include/sio/llio.h,sys/sio/drv.h
SIO-SCSI适配器驱动器接口	io_send(io_req) 驱动程序必须跟其将要控制的每个SCSI目标器的设备适配器管理器绑定在一起。然后其他所有的驱动程序都不可在该设备上使用
WSIO-数据控制锁	Scsi_lun_lock(),scsi_lun_unlock()
WSIO-其他功能	Scsi_lun_open(),scsi_lun_close(),scsi_ioctl(),scsi_init_inquiry_data(),scsi_read(),scsi_write(),scsi_enqueue(),scsi_dequeue_bp(),scsi_cmd(),scsi_ddsw_init(),scsi_strategy(),scsi_enqueue_cnt(),scsi_mode_fix(),scsi_mode_select(),scsi_wr_prot(),scsi_action(),scsi_sense_action(),scsi_snooze(),scsi_sleep(),scsi_log_io()
WSIO-与SCSI相关的头文件	/usr/include/sys/wsio.h,sys/scsi_ctl.h
WSIO-SCSI适配器驱动器接口	scsi_start()
与SCSI相关的磁头文件	/usr/include/sys/scsi.h,scsi_meta.h
必须的驱动器入口点	驱动器名称 (I/O信息端口服务器)
SCSI pass-throuth接口	fd=open("/dev/rdsck/c0t5d0,...) 其中有一句需要翻译: Other IOCTL command: 其他的IOCTL命令
驱动器支持命令	Swinstall,swremove,swpackage,ioscan,uxgen
驱动器支持文件	/sbin/rc2.d/Sxxx and Kxxx scripts,/stand/system, /usr/conf/master.d/Drivername,/usr/conf/lib/Driverlib.a
调试	ddb是一个基于dbx的源代码级的内核调试程序。但是使用该调试程序必须保证在同一个子网下存在另一个机器
备注	服务器类的SCSI适配器使用“scsi1(窄)”或“scsi3(宽)”适配器驱动程序。工作站类的SCSI适配器使用“c700(窄)”或“c720(快/宽)”适配器驱动程序 我们可以通过“ioscan -kf”命令来查看SCSI适配器所使用的驱动程序 警告：在某些HP PA RISC文档中以位0为最高位。而且在HP-UX 9.x和10.x中存在着很大的不同。所有的内核子系统通信都是通过消息来完成的

表 11-3

SCO ODT 3 特 性

特 性	描 述
平台	基于PC的intel结构
内核类型	单片电路
内核-数据空间转换	ktop(),ptok(),paddr(),vtop(),copyin(),copyout()
内核-过程分块(休眠)	iowait(),sleep(),wakeup(),iodone()
内核-数据控制锁	sp16(),splx()
内核-其他功能	bcopy(),printf(),delay(),major(),minor(),panic(),physio(),sus er(),signal(),timeout()
内核-其他结构	buf,scsi_io_req,scsi_dev_cfg,u
与SCSI相关的头文件	/usr/include/sys/scsi.h
SCSI适配器驱动器接口	drxinit,drxopen,drclose,drxread,drxwrite,drxioctl,dr xstratgey,drxstart,drxintr
必须的驱动器入口点	fd=open("/dev/rd0,...") ioctl(fd,SCSIUSERCMD,&sc)
SCSI pass-through接口	link_unix,custom
驱动器支持的命令	/etc/conf/cf.d/mscsi,mdevice,sdevice,/etc/conf/pack.d/drxx/d river.o
驱动器支持的文件	cmm_err,/var/adm/messages
错误记录	comes with SCO Driver Book mentioned below.
驱动器源代码例子	在下面将要提到的SCO驱动程序一书附带磁盘中
调试	我们可以从SCO开发商那里得到一个编译器级的内核调试程序
适用的驱动程序文档	Kettle、Peter和Steve Staler所写的SCO UNIX下的设备驱动程序: A Practical Approach, Reading, Massachusetts: Addison Wesley, 1993
备注	以AT&T 系统V为基础, 后来发展称为Microsoft Xenix, 并且1990发展称为SCO UNIX和开放桌面系统

表11-4

Solaris 2.5 特 性

特    性	描    述
平台	Sun SPARC结构
内核类型	完全动态
内核-内存处理 (allocate、free)	scsi_alloc_consistent_buf(),scsi_free_consistent_buf(), kmem_alloc(),kmem_zalloc(),kmem_free()
内核-数据空间转换	copyin(),copyout()
内核-过程分块 (休眠)	sleep(),wakeup(),biowait(),biodone(),bioerror(),cv_init, cv_ wait(),cv_wait_sig(),cv_signal(),cv_broadcast(),cv_destroy()
内核-数据控制锁	Mutex_init(),mutex_enter,mutex_exit()
内核-其他功能	ddi_get_soft_state,ddi_soft_state_fini(),dde_soft_state_zalloc(),ddi_soft_state_free(),mod_remove(),mod_info(),ddi_get_prop(),ddi_get_parent(),ddi_get_instance(),ddi_get_driver_private(),scsi_init_pkt(),ddi_create_minor_node(),ddi_remove_minor_scsi_init_pkt(),ddi+create+minor_node(),ddi_remove+minor_scsi_reset(),scsi_destroy_pkt(),scsi_probe(),scsi_unprobe(),scsi_errmsg(),makecom_g0(),bzero,bcopy(),physio(),getrbuf(),freerbuf()
内核-其他结构	Buf,uscsi_cmd,iovec,uiو
与SCSI相关的头文件	/usr/include/sys/scsi/scsi.h
SCSI适配器驱动程序接口	scsi_transport(scси.h)
必需的驱动程序入口点	scsi_transport(scси_pkt)
SCSI pass-through接口	drivername_open,drivername_close,drivername_read, drivername_write, drivername_ioctl, drivername_strategy, drivername_info, drivername_identify, drivername_probe, drivername_attach, drivername_detach, _init, _fini_info
驱动器支持的命令	Fd=open("/dev/rdsk/c0t210s0",...)
驱动器支持的文件	ioctl(fd,USCSICMD,&uscsi_cmd)
错误记录	Add_drv,rem_drv,drvconfig,pkgmk,pkginfo,pkgtrans,pkgchk,pkgadd,pkgrm,prtconf,sysdef
驱动程序源代码例子	/usr/kernel/drv/driver.conf./etc/devlink.tab,proto.drivername./etc/system./etc/rc2/
调试	在系统的启动PROM中固化了汇编级的内核调试程序， 我们可以通过命令行的方式来执行该调试程序。我们可以输入“L1-A”或“Stop-A”来进入该程序。另外该系统中还提供了另一个名为adb的调试工具
适用的驱动程序文档	ftp://opcom.sun.ca/pub/drivers/svr4_sample_drivers.tar.Z Solaris 2.x DDK (驱动程序开发软件包) “写入设备驱动程序” (Sun P/N 800-6502-06)
备注	在SPARC系统中，在内存管理中零散地使用了DMA技术，所以其数据传输速率不一定能达到16MB 由于驱动程序的装载和卸载不用重新启动系统，所以开发工作非常便利

表11-5 Digital Unix (Formerly OSF/1) 4.x 特 性

特 性	描 述
平台	Digital Alpha结构
内核类型	单片电路（但是可以进行驱动程序的装载和卸载）
内核-内存处理 (allocate/free)	contig_malloc(),contig_free(),MALLOC()
内核-数据空间转换	copyin(),copyout()
内核-过程分块	biodone(),inodone(),mpsleep(),sleep(),spldevhigh(),splx(),wakeup()
内核-数据控制锁	lock_init(),lock_done(),lock_read(),lock_write(),lock_terminate()
内核-其他功能	physio(),devsw_add(),devsw(),devsw_get(),getnewbuf(),brelease(),bzero(),major(),minor()
内核-其他结构	buf,uagt_struct,CCB_SCSIIO,CCB_GETDEV,CCB_PATHINQ,etc.
与SCSI相关的头文件	/usr/include/io/cam/cam.h/scsi_all.h,scsi_cdbs.h,scsi_direct.h,scsi_opcodes.h,uagt.h,dec_cam.h,xpt.h,scsi_status.h,buf.h,ioctl.h
SCSI适配器驱动器接口	xpt_action(&ccb),xpt_ccb_alloc(),xpt_ccb_free()
必需的驱动程序入口点	drivername_open,drivername_close,drivername_read,drivername_write,drivername_ioctl,drivername_strategy
SCSI pass-through接口	fd=open("/dev/cam",...);ioctl(fd,UAGT_CAM_IO,&uagt_struct)
驱动程序支持的命令	uerf, /sbin/doconfig, /sbin/ddr_config, /usr/sbin/sysconfig
驱动程序支持的文件	/etc/ddr.dbase, sysconfigtab, /sys/conf/hostname
错误记录	cam_logger, /var/adm/messages, binary.errlog
调试	dbx-k, dbx-remote, Ladebug (都是完全的代码级的调试程序)
适用的驱动程序文档	Digital UNIX 设备驱动程序编写手册
备注	我们可以通过SCSI外设的驱动程序中的很多例程来完成CAM CCB的发送和接收工作。在CAM驱动程序编写手册中可以找到ccmn xxxx例程（上述驱动程序文档的一部分）

表11-6 Linux (2.0.x版的内核) 特 性

特 性	描 述
平台	基于Intel x86结构的PC, Digital Alpha结构, Sun SPARC结构
内核类型	部分动态
内核-内存处理 (allocate/free)	kmalloc(),kfree(),free_pages(),scsi_init_malloc(),scsi_init_free(),brelease()

续表

特    性	描    述
内核-数据空间转换	put_user(),get_user(),verify_area(),memcpy_tofs(),memcpy_fromfs()
内核-过程分块(休眠)	sleep_on(),wake_up()
内核-数据控制锁	save_flags(),cli(),sti(),restore_flags()
内核-其他功能	printk(),panic(),memset(),suser(),init_module(),register_blkdev(),unregister_blkdev(),register_chrdev(),unregister_chrdev(),scsi_register_device(),scsi_register_host(),scsi_unregister_device(),scsi_unregister_host(),scsi_register_module(),scsi_unregister_module(),scsi_init(),scsi_done(),check_sense(),scsi_abort(),scsi_reset(),scsi_build_commandblocks(),scsi_dev_init()
内核-其他结构	buf,Scsi_Cmnd,Scsi_Device_Template,drx_fops
与SCSI相关的头文件	/usr/src/linux/drivers/scsi/scsi.h,scsi_ioctl.h
SCSI适配器驱动器接口	scsi_do_cmd(SCpnt,cmd,buffer,bufflen,done_rtn,timeout,retries)
必需的驱动程序入口点(字符驱动器)	drx_init,drx_open,drx_release,drx_finish,drx_attach,drx_detach,drx_detect,drx_ioctl,drx_read,drx_write,do_drx_request
必需的驱动程序入口点(区块驱动器)	chexk_media_change,drx_revalidate
SCSI pass-through(字符驱动器)	fd=open("/dev/sg0",...)write(fd,scsi_command,nbytes)read(fd,scsi_results,nbytes)
SCSI pass-through(区块驱动器)	fd=open("/dev/scd0",...)ioctl(fd,SCSI_IOCTL_SEND_COMMAND,buff)
驱动程序支持的命令	Make depend,make zlilo
驱动程序支持的文件	/lib/modules/version/*,/usr/src/linux/kernel/drivers/*
错误处理	/usr/src/linux/kernel/drivers/scsi/sd.*,st.*,sg.c
驱动程序代码例子	logger(),syslog(), printk(),/etc/syslog.conf,/usr/sbin/syslogd,/var/log/messages
调试	Printk(), gdb, /proc/* 我们可以通过/proc文件系统：“cat/proc/scsi/scsi”的一个很方便的特性来查到所有连接的SCSI设备。另外还有一个目录“/proc/scsi/adaptername/”，在这个目录中包括了每个SCSI总线的参数文件
适用的驱动程序文档	Beck, Michael.Linux Kernel Internals.Reading, Massachusetts: Addison-Wesley,1996。可以从Linux附带的手册中查到驱动程序的文档：Linux Kernel Hacker's Guide (Michael K.Johnson); Writing Linux Device Drivers (Michael K.Johnson); Linux Scsi HOWTO (Drew Eckhardt); linux SCSI programming HOWTO (Heiko Ei Felt)
备注	在Linux磁盘驱动程序只有一个界面，不像其他的UNIX磁盘驱动程序那样，不仅提供字符方式还提供区块方式。而且还不包括strategy例程 Linux并不是从AT&T或BSD源代码发展来的

通过对上述的各种 UNIX 平台的内核工作环境的比较，我们现在不难理解为什么在编写 UNIX 驱动程序时要针对某一个平台了。所以就当前的状况来看，编写可以在多平台之间进行移植的 UNIX 设备驱动程序几乎成了不可想象的事。也许我们可以对此做一些变动。现在有很多人正在努力准备把 UNIX 在两个不同级别上进行标准化和模型化。通用设备接口（Universal Device Interface，UDI）就是其中一种标准化的思想，Intel 公司的 I2O 也是一个例子。

如果我们是本着学习的目的来编写第一个 UNIX SCSI 驱动程序的，那么其最佳开发平台非 Linux 莫属了。它为我们提供了一个功能强大、而且免费的 UNIX 驱动程序的开发环境。而且 Linux 几乎可以运行在任何类型的硬件产品上。如果你的机器是 Intel x86 系列的 PC 个人电脑，那么最好首先购买一套多 CD 的 Linux 安装光盘，例如 Slackware 和 Red Hat。对于大多数喜欢动手的用户来说，可以使用 ftp 或 http 在 internet 上下载一个即可，而且这种站点也是不胜枚举。Linux 的网址为 <http://www.linux.org>，而且还链接了不少可以下载 Linux 的站点。

当我们有了可以运行的 Linux 系统的时候，就可以开始安装其内核源文件（假设它还没有安装在 O/S 上）。我们可以在“/usr/src/linux/drivers”目录下得到为 Linux 编写的所有的驱动程序的源代码。学习编写驱动程序的最好方法之一就是直接学习运行的驱动程序就可以了。在开始之前，我先带领大家看一下 SCSI 磁盘驱动程序（sd.c）和 SCSI 机器转移归向驱动程序（还可以参考“SCSI Generic”驱动程序）。

一切准备就绪，让我们开始吧。

## Linux SCSI 磁盘驱动器

下面我想做的是向大家介绍每个驱动程序的入口点的例程，不过这些例程是和可以进行编译和连接到驱动程序的内核的繁琐的代码分离开来的。

下面的摘录都来自于下述源文件：/usr/src/linux/kernel/drivers/sisi/sd.c, sd\_ioctl.c, scsi.c, sd.h。

程序清单 11-1 Linux 下的 sd\_open 例程

```
static int sd_open(struct inode * inode, struct file * filp)
{
    int target;
    target = DEVICE_NR(inode->i_rdev);

    if(target >= sd_template.dev_max || !rscsi_disks[target].device)
        return -ENXIO; /* 该设备不存在 */
```

```
/*
 * 确保同一时刻仅有一个进程执行check_change_disk 操作
 * 这种方法也可以用在当分区表被重多读取时的进一步访问
 */

while (rscsi_disks[target].device->busy)
    barrier();
if(rscsi_disks[target].device->removable) {
    check_disk_change(inode->i_rdev);

    /*
     *如果驱动程序为空，则打开失败
     */
    if ( !rscsi_disks[target].ready )
        return -ENXIO;

    /*
     * 如果设备处于写保护列表中则当用户试图
     * 进行写操作时会产生打开错误
     */
    if ( (rscsi_disks[target].write_prot) && (filp->f_mode & 2))
        return -EROFS;
}

/*
 * 如果要访问一个并不存在分区，则在检测到磁盘改变后进行该操作
 */
if(sd_sizes[MINOR(inode->i_rdev)] == 0)
    return -ENXIO;

if(rscsi_disks[target].device->removable)
    if(!rscsi_disks[target].device->access_count)
        sd_ioctl(inode, NULL, SCSI_IOCTL_DOORLOCK, 0);

rscsi_disks[target].device->access_count++;
if (rscsi_disks[target].device->host->hostt->usage_count)
    (*rscsi_disks[target].device->host->hostt->usage_count)++;
if(sd_template.usage_count) (*sd_template.usage_count)++;
return 0;
}
```

例程 `sd_open` 输入时需要两个参数：指向 `inode` 结构的指针和指向文件结构的指针。该例程所做的第一件事就是检查设备是否真正存在，另外检查驱动器中是否有存储介质。

数组 `rscsi_disks[]` 用来保存系统中每个磁盘的路径信息。如果一切正常的话，该例程将会返回表示成功的“0”。

---

### 程序清单 11-2 Linux 下的 `sd_release` 例程

---

```
static void sd_release(struct inode * inode, struct file * file)
{
    int target;
    fsync_dev(inode->i_rdev);

    target = DEVICE_NR(inode->i_rdev);

    rscsi_disks[target].device->access_count--;
    if (rscsi_disks[target].device->host->hostt->usage_count)
        (*rscsi_disks[target].device->host->hostt->usage_count)--;
    if (sd_template.usage_count) (*sd_template.usage_count)--;
    if (rscsi_disks[target].device->removable){
        if (!rscsi_disks[target].device->access_count)
            sd_ioctl(inode, NULL, SCSI_IOCTL_DOORUNLOCK, 0);
    }
}
```

例程 `sd_release` 输入时需要两个参数：指向 `inode` 结构的指针和指向文件结构的指针。该例程所做的工作就是减少某些计数器变量的数值，并打开可移动磁盘中的存储介质。

---

### 程序清单 11-3 Linux 下的 `sd_init` 例程

---

```
/*
 * sd_init() 函数检查所有存在的SCSI设备，检测
 * 这些设备的大小，并读取这些设备在分区表的入口点
 */

static int sd_registered = 0;

static int sd_init()
{
    int i;

    if (sd_template.dev_noticed == 0) return 0;

    if (!sd_registered) {
        if (register_blkdev(MAJOR_NR, "sd", &sd_fops)) {
```

```
    printk("Unable to get major %d for SCSI disk\n",
           MAJOR_NR);
    return 1;
}
sd_registered++;
}

/* 我们现在仍然不支持连接可装载设备 */
if(rscsi_disks) return 0;

sd_template.dev_max = sd_template.dev_noticed +

rscsi_disks = (Scsi_Disk *)
    scsi_init_malloc(sd_template.dev_max *
                    sizeof(Scsi_Disk), GFP_ATOMIC);
memset(rscsi_disks, 0, sd_template.dev_max *
       sizeof(Scsi_Disk));

sd_sizes = (int *)
    scsi_init_malloc((sd_template.dev_max << 4) *
                     sizeof(int), GFP_ATOMIC);
memset(sd_sizes, 0, (sd_template.dev_max << 4) * sizeof(int));

sd_blocksizes = (int *)
    scsi_init_malloc((sd_template.dev_max << 4) *
                     sizeof(int), GFP_ATOMIC);

sd_hardsizes = (int *)
    scsi_init_malloc((sd_template.dev_max << 4) *
                     sizeof(int), GFP_ATOMIC);

for(i=0;i<(sd_template.dev_max << 4);i++){
    sd_blocksizes[i] = 1024;
    sd_hardsizes[i] = 512;
}
blksize_size[MAJOR_NR] = sd_blocksizes;
hardsect_size[MAJOR_NR] = sd_hardsizes;
sd=(struct hd_struct *)
    scsi_init_malloc((sd_template.dev_max<<4) *
                     sizeof(struct hd_struct), GFP_ATOMIC);

sd_gendisk.max_nr = sd_template.dev_max;
sd_gendisk.part = sd;
sd_gendisk.sizes = sd_sizes;
sd_gendisk.real_devices = (void *) rscsi_disks;
return 0;
}
```

例程 init 中不需要什么入口参数。它首先简单地记录一下内核中的驱动程序，该驱动程序为它在 devswitch 表格中开辟了一个空间。然后它就为 rscsi\_disks[] 数组分配存储空间，还为一对别的数组开辟空间，并对其中的一些参数进行初始化。

---

#### 程序清单 11-4 Linux 下的 sd\_finish 例程

---

```
static void sd_finish()
{
    int i;

    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;

    sd_gendisk.next = gendisk_head;
    gendisk_head = &sd_gendisk;

    for (i = 0; i < sd_template.dev_max; ++i)
        if (!rscsi_disks[i].capacity &&
            rscsi_disks[i].device)
    {
        if (MODULE_FLAG &&
            !rscsi_disks[i].has_part_table) {
            sd_sizes[i << 4] = rscsi_disks[i].capacity;
            /* 执行 MAYBE_REINIT 以便 sd_init_onedisk(i) 重新生效 */
            revalidate_scsidisk(MKDEV(MAJOR_NR, i << 4), 0);
        }
        else
            i=sd_init_onedisk(i);
        rscsi_disks[i].has_part_table = 1;
    }

    /* 如果主机适配器拥有分散-聚集力，则我们增加16个块的读前区(32
     * 扇区)，如果没有这种能力，则使用两个块(4个扇区)的读前区
     */
    if(rscsi_disks[0].device &&
       rscsi_disks[0].device->host->sg_tablesize)
        read_ahead[MAJOR_NR] = 120; /* 120 sector read-ahead */
    else
        read_ahead[MAJOR_NR] = 4; /* 4 sector read-ahead */

    return;
}
```

例程 finish 把驱动程序中的“请求函数”与 blk\_dev[] 数组进行连接，然后读取分配表中的信息，并对磁盘所连接的主机适配器进行初始化。

---

**程序清单 11-5 Linux 下的 sd\_detect 例程**

---

```
static int sd_detect(Scsi_Device * SDp){  
    if(SDp->type != TYPE_DISK && SDp->type != TYPE_MOD)  
        return 0;  
  
    printk("Detected scsi %sdisk sd%c at ",  
          SDp->removable ? "removable " : "",  
          'a'+ (sd_template.dev_noticed++));  
    printk("scsi%d, channel %d, id %d, lun %d\n",  
          SDp->host->host_no, SDp->channel,  
          SDp->id, SDp->lun);  
  
    return 1;  
}
```

例程 detect 仅仅简单地检查一下系统中是否存在硬盘或光盘可以使用，并在引导系统时打印出相应的消息来。

---

**程序清单 11-6 Linux 下的 sd\_attach 例程**

---

```
static int sd_attach(Scsi_Device * SDp){  
    Scsi_Disk * dpnt;  
    int i;  
  
    if(SDp->type != TYPE_DISK && SDp->type != TYPE_MOD)  
        return 0;  
  
    if(sd_template.nr_dev >= sd_template.dev_max) {  
        SDp->attached--;  
        return 1;  
    }  
  
    for(dpnt = rscsi_disks, i=0; i<sd_template.dev_max;  
        i++, dpnt++)  
        if(!dpnt->device) break;  
  
    if(i >= sd_template.dev_max)  
        panic ("scsi_devices corrupt (sd)");  
  
    SDp->scsi_request_fn = do_sd_request;
```

```

rscsi_disks[i].device = SDp;

rscsi_disks[i].has_part_table = 0;
sd_template.nr_dev++;
sd_gendisk.nr_real++;
return 0;
}

```

例程 attach 再次检查系统中是否存在硬盘或光盘，以及是否已经到达了驱动程序所支持的最大的驱动器数。然后它检验 rscsi\_disks[] 数组，来确定其中是否每个磁盘都有一个入口点。如果存在任何不正常的情况，系统就会不知所措。数组 rscsi\_disks[] 中很少一部分的数据进行了初始化，并且所连接的设备数目也相应增加。

---

### 程序清单 11-7 Linux 下的 sd\_detach 例程

---

```

static void sd_detach(Scsi_Device * SDp)
{
    Scsi_Disk * dpnt;
    int i;
    int max_p;
    int start;

    for(dpnt = rscsi_disks, i=0; i<sd_template.dev_max;
        i++, dpnt++)
        if(dpnt->device == SDp) {

            /* 如果我们正从一个磁盘驱动器断开则所有同步事件无效 */
            max_p = sd_gendisk.max_p;
            start = i << sd_gendisk.minor_shift;

            for (i=max_p - 1; i >=0 ; i--) {
                int minor = start+i;
                kdev_t devi = MKDEV(MAJOR_NR, minor);
                sync_dev(devi);
                invalidate_inodes(devi);
                invalidate_buffers(devi);
                sd_gendisk.part[minor].start_sect = 0;
                sd_gendisk.part[minor].nr_sects = 0;
                sd_sizes[minor] = 0;
            }

            dpnt->has_part_table = 0;
            dpnt->device = NULL;
        }
}

```

```

dpnt->capacity = 0;
SDp->attached--;
sd_template.dev_noticed--;
sd_template.nr_dev--;
sd_gendisk.nr_real--;
return;
}
return;
}

```

例程 `sd_detach` 的主要任务就是发送出所有的未写入的数据到物理磁盘上（由 `SDp` 来确定），然后清除所有的变量以表明该设备已经不可用。

---

#### 程序清单 11-8 Linux 下的 `revalidate_scsidisk` 例程

---

```

#define DEVICE_BUSY rscsi_disks[target].device->busy
#define USAGE rscsi_disks[target].device->access_count
#define CAPACITY rscsi_disks[target].capacity
#define MAYBE_REINIT sd_init_onedisk(target)
#define GENDISK_STRUCT sd_gendisk

/* 该例程用来查检所有的改变的SCSI磁盘的分区和分区表
 * 然后重新读取新的分区表，如果是通过介质改变而重新确
 * 认了磁盘，则输出“usage == 0”，而如果我们通过
 * ioctl来实现，则会自动设置“usage == 1”（不过使用
 * ioctl有一定的限制，就是必须使用开放的通道）
*/
int revalidate_scsidisk(kdev_t dev, int maxusage) {
    int target;
    struct gendisk *gdev;
    unsigned long flags;
    int max_p;
    int start;
    int i;

    save_flags(flags);
    cli();
    if (DEVICE_BUSY || USAGE > maxusage) {
        restore_flags(flags);
        printk("Device busy for revalidation (usage=%d)\n",
              USAGE);
    }
}

```

```

        return -EBUSY;
    }

DEVICE_BUSY = 1;
restore_flags(flags);

max_p = gdev->max_p;
start = target << gdev->minor_shift;

for (i=max_p - 1; i >=0 ; i--) {
    int minor = start+i;
    kdev_t devi = MKDEV(MAJOR_NR, minor);
    sync_dev(devi);
    invalidate_inodes(devi);
    invalidate_buffers(devi);
    gdev->part[minor].start_sect = 0;
    gdev->part[minor].nr_sects = 0;
    /*
     * 重置块的大小设置,否则无法读取分区表信息
     */
    blksize_size[MAJOR_NR][minor] = 1024;
}

#endif MAYBE_REINIT
MAYBE_REINIT;
#endif

gdev->part[start].nr_sects = CAPACITY;
resetup_one_dev(gdev, target);

DEVICE_BUSY = 0;
return 0;

```

如果存储介质已经更换,那么将会取消所有未存储的数据,并恢复分区表等等。需要注意的是,如果想要修改数据的结构,首先应该执行一个 `cli()` 例程,之后再执行一个 `restore_flags` 例程。这是 Linux 的数据锁定方法,可以避免另一个驱动器对该数据进行访问。

---

#### 程序清单 11-9 Linux 下的 `do_sd_request` 例程

---

```

/*
 * do_sd_request() 是sd驱动器的请求处理函数,该函数的
 * 作用就是获取块设备的请求,并把这些请求解释成SCSI命令
 */

```

```
static void do_sd_request (void)
{
    Scsi_Cmd * SCpnt = NULL;
    Scsi_Device * SDev;
    struct request * req = NULL;
    unsigned long flags;
    int flag = 0;

    save_flags(flags);
    while (1==1){
        cli();
        if (CURRENT != NULL &&
            CURRENT->rq_status == RQ_INACTIVE) {
            restore_flags(flags);
            return;
        }

        INIT_SCSI_REQUEST;
        SDev = rscsi_disks[DEVICE_NR(CURRENT->rq_dev)].device;

        /*
         * 我也不敢确定在哪做这件事更好一些，所以我们只要
         * 在用户空间中找一个我们喜欢来的地方就可以了。
         */
        if( SDev->was_reset )
        {
        /*
         * 我们必须再将门锁上，不过很可能会进入一个中断请求处理程序中，
         * 所以必须在用户空间做这件事，因为我们不想因为中断而停止工作。
         */
        if( SDev->removable && !intr_count )
        {
            scsi_ioctl(SDev, SCSI_IOCTL_DOORLOCK, 0);
            /* scsi_ioctl 命令可能会导致“CURRENT”的改变，
             * 从而重新开始 */
            SDev->was_reset = 0;
            continue;
        }
        SDev->was_reset = 0;
    }

    /* 在这里我们必须格外小心，因为allocate_device命令会产生一个自由指针而且并不能保证该自由指
     * 针可以进行排列在正常的使用情况下，如果其他的设备类型已经被主机锁死，而且至少有一个对该设备类型的
     * 请求，此时会调用该函数，由于对新命令的调用也会调用该函数，如果我们不仅仅是调用一次
     * allocate_device命令，而且执行的命令不可进行排列，则系统会陷入一种困境。为了安全我们可以执行
     * request_queueable函数，因为主机能够保证在返回指针前可以执行其他的命令。*/
}
```

```

if (flag++ == 0)
    SCpnt = allocate_device(&CURRENT,
                           rscsi_disks[DEVICE_NR(CURRENT->q_dev)].device, 0);
else SCpnt = NULL;

/*
 * 下面的restore_flags 函数会导致潜在的问题.
 * 如果使用 "sti()" 函数可以消除上述潜在问题,
 * 但是会产生竞争状态,并会产生严重的后果.
 */
restore_flags(flags);

/* 这在性能上是一个提高,让我们仔细研究一下请求列表,从中找出一个可排列请求(即设备不忙,而且主机可以接受其他命令),如果我们能找到可排列请求,那么对其进行排列,对于多于一个磁盘的系统,其中存在很大的差异.我们希望在玩弄请求列表的过程中不被其他中断所干扰,如果不这样,内核会错过某个请求.*/
if (!SCpnt && sd_template.nr_dev > 1){
    struct request *req1;
    req1 = NULL;
    cli();
    req = CURRENT;
    while(req){
        SCpnt = request_queueable(req,
                                   rscsi_disks[DEVICE_NR(req->rq_dev)].device);
        if(SCpnt) break;
        req1 = req;
        req = req->next;
    }
    if (SCpnt && req->rq_status == RQ_INACTIVE) {
        if (req == CURRENT)
            CURRENT = CURRENT->next;
        else
            req1->next = req->next;
    }
    restore_flags(flags);
}

if (!SCpnt) return; /* Could not find anything to do */

/* 队列命令 */
requeue_sd_request(SCpnt);
} /* While */
}

```

该例程就是 SCSI 磁盘驱动器的基本调度程序部分。它可以决定是否为另一个命令提供主机适配器以及将要执行的下一个命令。执行该命令之后将会调用 `requeue_sd_request()` 来创建实际的命令请求。

---

### 程序清单 11-10 Linux 下的 `requeue_sd_request` 例程

---

```
static void requeue_sd_request (Scsi_Cmnd * SCpnt)
{
    int dev, devm, block, this_count;
    unsigned char cmd[10];
    int bounce_size, contiguous;
    int max_sg;
    struct buffer_head * bh, *bhp;
    char * buff, *bounce_buffer;

repeat:
    if(!SCpnt || SCpnt->request.rq_status == RQ_INACTIVE) {
        do_sd_request();
        return;
    }

    devm = MINOR(SCpnt->request.rq_dev);
    dev = DEVICE_NR(SCpnt->request.rq_dev);

    block = SCpnt->request.sector;
    this_count = 0;

#ifndef DEBUG
    printk("Doing sd request, dev = %d, block = %d\n",
          devm, block);
#endif

    if (devm >= (sd_template.dev_max << 4) ||
        !rscsi_disks[dev].device ||
        block + SCpnt->request.nr_sectors >
        sd[devm].nr_sects)
    {
        SCpnt = end_scsi_request(SCpnt, 0,
                                 SCpnt->request.nr_sectors);
        goto repeat;
    }
}
```

```
block += sd[devm].start_sect;

if (rscsi_disks[dev].device->changed)
{
    /*
     * 如果发生了磁盘改变，则在改变位重置之前不做任何操作
     */
    SCpnt = end_scsi_request(SCpnt, 0,
        SCpnt->request.nr_sectors);
    goto repeat;
}

#ifndef DEBUG
    printk("sd%c : real dev = /dev/sd%c, block = %d\n",
        'a' + devm, dev, block);
#endif

/*
 * 如果我们的硬件分区大小为1K, 就不能访问512字节的扇区,
 * 从理论上讲我们可以处理上述问题, 而实际上, SCSI的CD-ROM
 * 驱动器必须能进行上述处理, 因为我们平时使用多为1K的分区大小,
 * 而CD-ROM通常使用2K的硬件分区. 但是对于CD-ROM来说需简单一些,
 * 因为它是只读不写的, 为了使用上的方便, 文件系统应该能够处理上述问题,
 * 而不应该由SCSI磁盘驱动器因为这个原因而限制其扇区大小.
 */
if (rscsi_disks[dev].sector_size == 1024)
    if((block & 1) || (SCpnt->request.nr_sectors & 1)) {
        printk("sd.c:Bad block number requested");
        SCpnt = end_scsi_request(SCpnt, 0,
            SCpnt->request.nr_sectors);
        goto repeat;
    }

switch (SCpnt->request.cmd)
{
case WRITE :
    if (!rscsi_disks[dev].device->writeable)
    {
        SCpnt = end_scsi_request(SCpnt, 0,
            SCpnt->request.nr_sectors);
        goto repeat;
    }
    cmd[0] = WRITE_6;
    break;
case READ :
    cmd[0] = READ_6;
    break;
```

```
default :
    panic ("Unknown sd command %d\n", SCpnt->request.cmd);
}

SCpnt->this_count = 0;

/* 如果主机适配器能够处理非常大的分散/聚集请求,
 * 那么会在聚集的过程中浪费很多时间
 */
contiguous = (!CLUSTERABLE_DEVICE(SCpnt) ? 0 : 1);
bounce_buffer = NULL;
bounce_size = (SCpnt->request.nr_sectors << 9);

/* 首先要明确我们是否需要这种请求的散射缓冲区,如果需要
 * 则必须保证我们能分配到一个扇区,如果我们使用的空间
 * 已以超出16MB的限制就不要再分配一个散射缓冲区而浪费空间
 */
if (contiguous && SCpnt->request.bh &&
    ((long) SCpnt->request.bh->b_data)
    + (SCpnt->request.nr_sectors << 9) - 1 >
    ISA_DMA_THRESHOLD
    && SCpnt->host->unchecked_isa_dma) {
    if(((long) SCpnt->request.bh->b_data) >
        ISA_DMA_THRESHOLD)
        bounce_buffer = (char *) scsi_malloc(bounce_size);
    if(!bounce_buffer) contiguous = 0;
}

if(contiguous && SCpnt->request.bh &&
    SCpnt->request.bh->b_reqnext)
    for(bh = SCpnt->request.bh, bhp = bh->b_reqnext;
        bhp; bh = bhp, bhp = bhp->b_reqnext) {
        if(!CONTIGUOUS_BUFFERS(bh, bhp))
            if(bounce_buffer)
                scsi_free(bounce_buffer, bounce_size);
            contiguous = 0;
            break;
    }
}

if (!SCpnt->request.bh || contiguous) {
    /* 页请求的情况(也就是新的设备请求, 或未链接的缓冲区) */
    this_count = SCpnt->request.nr_sectors;
    buff = SCpnt->request.buffer;
    SCpnt->use_sg = 0;
}

} else if (SCpnt->host->sg_tablesize == 0 ||
```

```
(need_isa_buffer && dma_free_sectors <= 10)) {  
  
    /* 这属于主机适配器不能分散/聚集的情况,  
     * 如果通过DMA缓冲区内存的低端运行也属于  
     * 这种情况,我们可以设定一个阀值,当超过该  
     * 值时就要为其他的请求让出空间。虽然我们并  
     * 不需要对每种情况都这么处理,但是为了安全  
     * 起见,我们还是保守一些。因为在低速运行时,  
     * 我们别无选择,只有惊慌失措的份了。  
    */  
  
    if (SCpnt->host->sg_tablesize != 0 &&  
        need_isa_buffer &&  
        dma_free_sectors <= 10)  
        printk("Warning: SCSI DMA buffer space running low.\n");  
        printk(" Using non scatter-gather I/O.\n");  
  
    this_count = SCpnt->request.current_nr_sectors;  
    buff = SCpnt->request.buffer;  
    SCpnt->use_sg = 0;  
  
} else {  
  
    /* 可分散/聚集的主机适配器 */  
    struct scatterlist * sgpt;  
    int count, this_count_max;  
    int counted;  
  
    bh = SCpnt->request.bh;  
    this_count = 0;  
    this_count_max =  
        (rscsi_disks[dev].ten ? 0xffff : 0xff);  
    count = 0;  
    bhp = NULL;  
    while(bh) {  
        if ((this_count + (bh->b_size >> 9)) >  
            this_count_max) break;  
        if (!bhp || !CONTIGUOUS_BUFFERS(bhp,bh) ||  
            !CLUSTERABLE_DEVICE(SCpnt) ||  
            (SCpnt->host->unchecked_isa_dma &&  
            ((unsigned long) bh->b_data-1) ==  
            ISA_DMA_THRESHOLD)) {  
            if (count < SCpnt->host->sg_tablesize)  
                count++;  
            else break;  
        }  
    }  
}
```

```

    this_count += (bh->b_size >> 9);
    bhp = bh;
    bh = bh->b_reqnext;
}

#ifndef O
if(SCpnt->host->unchecked_isa_dma &&
   ((unsigned int) SCpnt->request.bh->b_data-1) ==
   ISA_DMA_THRESHOLD) count--;

#endif
SCpnt->use_sg = count; /* 链的数目 */
/* scsi_malloc 只能按照512字节的块大小来运行 */
count = (SCpnt->use_sg *
         sizeof(struct scatterlist) + 511) & ~511;

SCpnt->sglist_len = count;
max_sg = count / sizeof(struct scatterlist);
if(SCpnt->host->sg_tablesize < max_sg)
    max_sg = SCpnt->host->sg_tablesize;
sgpnt = (struct scatterlist * ) scsi_malloc(count);
if (!sgpnt) {
    printk("Warning - running *really* short on DMA buffers\n");
    SCpnt->use_sg = 0; /* No memory left - bail out */
    this_count = SCpnt->request.current_nr_sectors;
    buff = SCpnt->request.buffer;
} else {
    memset(sgpnt, 0, count);
    /* Zero so it is easy to fill, but only
     * if memory is available */

    buff = (char *) sgpnt;
    counted = 0;
    for(count = 0, bh = SCpnt->request.bh, bhp =
        bh->b_reqnext; count < SCpnt->use_sg && bh;
        count++, bh = bhp) {

        bhp = bh->b_reqnext;

        if(!sgpnt[count].address) sgpnt[count].address =
            bh->b_data;
        sgpnt[count].length += bh->b_size;
        counted += bh->b_size >> 9;

        if (((long) sgpnt[count].address) +
            sgpnt[count].length - 1 >
            ISA_DMA_THRESHOLD &&

```

```

(SCpnt->host->unchecked_isa_dma) &&
!sgpnt[count].alt_address) {
    sgpnt[count].alt_address =
        sgpnt[count].address;
/* 我们应该尽量避免工作在DMA的极端状态,
 * 因为这时还是比较容易控制的,其他情况下
 * 我们很可能会有更需要的地方,而如果我们
 * 真工作在DMA的极端情况,很可能会遇到很大
 * 的麻烦 */
}

if(dma_free_sectors <
    (sgpnt[count].length >> 9) + 10) {
    sgpnt[count].address = NULL;
} else {
    sgpnt[count].address = (char *)
        scsi_malloc(sgpnt[count].length);
}

/* 如果我们开始运行在DMA缓冲区的低端,则会取消
 * 所有的分散/聚集操作并释放所有已经分配的内存,
 * 因为我们必须保证所有的SCSI操作能够进行至少
 * 一种非分散/聚集操作.
 */
if(sgpnt[count].address == NULL) {
/* Out of dma memory */

#ifndef O
    printk("Warning: Running low on \
           SCSI DMA buffers");
    /* 试图返回到一个非分散/聚集操作 */
    while(--count >= 0){
        if(sgpnt[count].alt_address)
            scsi_free(sgpnt[count].address,
                      sgpnt[count].length);
    }
    this_count =
        SCpnt->request.current_nr_sectors;
    buff = SCpnt->request.buffer;
    SCpnt->use_sg = 0;
    scsi_free(sgpnt, SCpnt->sglist_len);
#endif
    SCpnt->use_sg = count;
    this_count = counted -=
        bh->b_size >> 9;
    break;
}
}

```

```
/* 如果我们能保证提供足够大的DMA缓冲器来满足请求,
 * 那么只要簇缓冲器就可以了,如果可能造成突然的DMA
 * 散射缓冲区的使用的话,最好不要聚集新的请求
 */
if(bhp && CONTIGUOUS_BUFFERS(bh,bhp)
   && CLUSTERABLE_DEVICE(SCpnt)) {
    char * tmp;

    if (((long) sgpnt[count].address) +
        sgpnt[count].length +
        bhp->b_size - 1 > ISA_DMA_THRESHOLD &&
        (SCpnt->host->unchecked_isa_dma) &&
        !sgpnt[count].alt_address) continue;

    if(!sgpnt[count].alt_address)
        {count--; continue;}
    if(dma_free_sectors > 10)
        tmp = (char *)
            scsi_malloc(sgpnt[count].length +
                        bhp->b_size);
    else {
        tmp = NULL;
        max_sg = SCpnt->use_sg;
    }
    if(tmp){
        scsi_free(sgpnt[count].address,
                  sgpnt[count].length);
        sgpnt[count].address = tmp;
        count--;
        continue;
    }

    /* 如果我们可以使用另一个分散/聚集链的话,那么
     * 可以增加计数器,这样就可以插入该链,否则就会
     * 由于不足而停止 */

    if (SCpnt->use_sg < max_sg) SCpnt->use_sg++;
} /* 连续的缓冲区 */
} /* 循环 */

/* 这就是我们要传输的数量 */
this_count = counted;

if(count < SCpnt->use_sg || SCpnt->use_sg >
```

```

SCpnt->host->sg_tablesize) {
    bh = SCpnt->request.bh;
    printk("Use sg, count %d %x %d\n",
           SCpnt->use_sg, count, dma_free_sectors);
    printk("maxsg = %x, counted = %d this_count = %d\n",
           max_sg, counted, this_count);

    while(bh) {
        printk("[%p %lx] ", bh->b_data, bh->b_size);
        bh = bh->b_reqnext;
    }
    if(SCpnt->use_sg < 16)
        for(count=0; count<SCpnt->use_sg; count++)
            printk("%d:%p %p %d) ", count,
                  sgpnt[count].address,
                  sgpnt[count].alt_address,
                  sgpnt[count].length);
    panic("Ooops");
}

if (SCpnt->request.cmd == WRITE)

    for(count=0; count<SCpnt->use_sg; count++)
        if(sgpnt[count].alt_address)
            memcpy(sgpnt[count].address,
                   sgpnt[count].alt_address,
                   sgpnt[count].length);
    /* 能够为sgpnt分配空间 */
} /* 具有分散/聚集能力的主机适配器 */

/* 下面计算DMA的地址>16Mb的可能性 */

if(SCpnt->use_sg == 0){
    if (((long) buff) + (this_count << 9) - 1 >
        ISA_DMA_THRESHOLD &&
        (SCpnt->host->unchecked_isa_dma)) {
        if(bounce_buffer)
            buff = bounce_buffer;
        else
            buff = (char *) scsi_malloc(this_count << 9);
        if(buff == NULL) {
            /* 如果处于低端内存区则试图返回一位 */
            this_count = SCpnt->request.current_nr_sectors;
            buff = (char *) scsi_malloc(this_count << 9);
            if(!buff) panic("Ran out of DMA buffers.");
        }
    }
}

```

```
        }

        if (SCpnt->request.cmd == WRITE)
            memcpy(buff, (char *)SCpnt->request.buffer,
                   this_count << 9);
    }

}

#endif DEBUG
printk("sd%c : %s %d/%d 512 byte blocks.\n",
       'a' + devm,
       (SCpnt->request.cmd == WRITE) ?
       "writing" : "reading",
       this_count, SCpnt->request.nr_sectors);
#endif

cmd[1] = (SCpnt->lun << 5) & 0xe0;

if (rscsi_disks[dev].sector_size == 1024){
    if(block & 1)
        panic("sd.c:Bad block number requested");
    if(this_count & 1)
        panic("sd.c:Bad block number requested");

    block = block >> 1;
    this_count = this_count >> 1;
}

if (rscsi_disks[dev].sector_size == 256){
    block = block << 1;
    this_count = this_count << 1;
}

if (((this_count > 0xff) || (block > 0xffff)) &&
    rscsi_disks[dev].ten)
{
    .
    if (this_count > 0xffff)
        this_count = 0xffff;

    cmd[0] += READ_10 - READ_6 ;
    cmd[2] = (unsigned char) (block >> 24) & 0xff;
    cmd[3] = (unsigned char) (block >> 16) & 0xff;
    cmd[4] = (unsigned char) (block >> 8) & 0xff;
    cmd[5] = (unsigned char) block & 0xff;
```

```

cmd[6] = cmd[9] = 0;
cmd[7] = (unsigned char) (this_count >> 8) & 0xff;
cmd[8] = (unsigned char) this_count & 0xff;
}
else
{
    if (this_count > 0xff)
        this_count = 0xff;

    cmd[1] |= (unsigned char) ((block >> 16) & 0x1f);
    cmd[2] = (unsigned char) ((block >> 8) & 0xff);
    cmd[3] = (unsigned char) block & 0xff;
    cmd[4] = (unsigned char) this_count;
    cmd[5] = 0;
}

/*
 * 我们不应该在扇区中间断开,这样会给主机适配器造成
 * 垃圾数据污染,不过我们完全可以假设在每次通断之间
 * 能够传输足够多的数据
 */
SCpnt->transfersize = rscsi_disks[dev].sector_size;

SCpnt->underflow = this_count << 9;
scsi_do_cmd (SCpnt, (void *) cmd, buff,
              this_count * rscsi_disks[dev].sector_size,
              rw_intr,
              (SCpnt->device->type == TYPE_DISK ?
               SD_TIMEOUT : SD_MOD_TIMEOUT),
              MAX_RETRIES);
}

```

例程 `queue_sd_request` 创建了所有的 SCSI CDB，这些 CDB 将会发送到合适的低层次的 SCSI 适配器驱动器。需要注意的是，在编写驱动程序的过程中，必须考虑到 ISA 总线 16M 寻址空间的限制，通过它可以在该限制下使用 `bounce` 缓冲器来中断数据传输。另外驱动程序还要考虑到进行的操作是否会用到“集散”技术。这就意味着由于 DMA 传输可以不通过 CPU 的内存管理单元直接处理物理寻址空间，逻辑上连续的内存空间在物理上可能是分散的。也就是说 SCSI 主机适配器在进行数据传输的时候必须能够找到内存区段的连接信息。在读（`read`）操作的过程中需要把数据物理地址分散到内存空间中，而在写（`write`）操作的过程中需要把数据集中到磁盘上单一的数据流。

---

**程序清单 11-11 Linux 下的 check\_scsidisk\_media\_change 例程**

---

```
static int check_scsidisk_media_change(kdev_t full_dev) {
    int retval;
    int target;
    struct inode inode;
    int flag = 0;

    target = DEVICE_NR(full_dev);

    if (target >= sd_template.dev_max ||
        !rscsi_disks[target].device) {
        printk("SCSI disk request error: invalid device.\n");
        return 0;
    }

    if(!rscsi_disks[target].device->removable) return 0;

    inode.i_rdev = full_dev;
    /* 这里才是我们真正所需要的 */
    retval = sd_ioctl(&inode, NULL,
    if(retval){/*
        * 测试不成功. 该设备单元很可能没准备好, 通常是
        * 驱动器中没有磁盘, 一旦驱动器做好准备, 标志会改变.
        */
        rscsi_disks[target].ready = 0;
        rscsi_disks[target].device->changed = 1;
        return 1;
        /* 如果调用check_disk_change 则会造成溢出 */
    }

    /*
     * 对于可移动SCSI磁盘(可光读的软磁盘), 我们必须确保
     * 驱动器中有磁盘存在. 这些功能已经保存到SCSI磁盘的结构中,
     * 并且每次启动都会进行测试
     */
    rscsi_disks[target].ready = 1; /* FLOPTICAL */

    retval = rscsi_disks[target].device->changed;
    if(!flag) rscsi_disks[target].device->changed = 0;
    return retval;
}
```

该例程中执行了一个 Test Unit Ready 命令，通过执行该命令确定存储介质是否已经发生了改变，并且定义了一个结构变量来表示是否真正进行了改变。

## Linux SCSI Pass-Through 驱动程序

SCSI Pass-Through 驱动程序为应用程序提供了一种简单的方法，通过这种简单的方法来避免某些内核保护机制，还可以向连接的设备发送 SCSI 命令。通常 Pass-Through 驱动程序很少跟没有驻留型内核驱动程序的设备一起工作。很典型的例子就是扫描仪、CD 录音机、自动唱片点唱机等等。

Linux 中存在这样一种驱动程序，它的名字就是 SCSI generic 驱动程序 (sg)。让我们来看一下它是如何工作的。每一个连接在系统主机适配器上的 SCSI 设备都对应着一个设备特殊文件（或者称之为设备节点）。这些特殊的文件名称类似于 “/dev/sga” 和 “/dev/sgb” 等等，在引导系统启动的时候，这些文件与每个 SCSI 设备相对应。例如，如果用户启动的系统中在主机适配器“0”上的 ID 为“1”和“3”端口连接着 SCSI 设备，那么名称为 “/dev/sga”的特殊文件将会与 ID 为 “1”的设备相对应，而名称为 “/dev/sgb”的特殊文件将会与 ID 为 “3”的设备相对应。

命令也是通过打开适当的设备特殊文件来发送给设备的，然后执行一个 write () 系统调用即可。我们可以通过执行一个 read () 调用来获得该命令的执行结果。如果系统调用所返回的值为负数，表明产生了错误。也许你会觉得这很简单。下面让我们来看一下 pass-through 驱动程序是如何为我们创建该接口的。

---

程序清单 11-12 Linux 下的 sg\_ioctl 例程

---

```
***** sg.c *****

static int sg_init(void);
static int sg_attach(Scsi_Device *);
static int sg_detect(Scsi_Device *);
static void sg_detach(Scsi_Device *);

struct Scsi_Device_Template sg_template =
    {NULL, NULL, "sg", NULL, 0xff,
     SCSI_GENERIC_MAJOR, 0, 0, 0, 0,
     sg_detect, sg_init,
     NULL, sg_attach, sg_detach};

#endif SG_BIG_BUFF
static char *big_buff = NULL;
```

```
/* 等待至缓冲器准备的 */
static struct wait_queue *big_wait;
static int big_inuse=0;
#endif

struct scsi_generic
{
    Scsi_Device *device;
    int users; /* 所打开的人数? */
    /* 等待至设备准备好 */
    struct wait_queue *generic_wait;
    /* 等待响应 */
    struct wait_queue *read_wait;
    /* 等待释放缓冲区 */
    struct wait_queue *write_wait;
    /* 设备的当前默认值 */
    int timeout;
    int buff_len; /* 当前缓冲区的长度 */
    char *buff; /* 缓冲区 */
    /* 即将执行的命令头 */
    struct sg_header header;
    char exclude; /* 为专用访问而打开 */
    char pending; /* 此时不能写入 */
    char complete; /* 命令执行完毕可以进行读操作 */
};

static struct scsi_generic *scsi_generics=NULL;
static void sg_free(char *buff,int size);

static int sg_ioctl(struct inode * inode,
    struct file * file, unsigned int cmd_in,
    unsigned long arg)
{
    int result;
    int dev = MINOR(inode->i_rdev);
    if ((dev<0) || (dev>=sg_template.dev_max))
        return -ENXIO;
    switch(cmd_in)
    {
        case SG_SET_TIMEOUT:
            result = verify_area(VERIFY_READ,
                (const void *)arg, sizeof(long));
    }
}
```

```

    if (result) return result;

    scsi_generics[dev].timeout=get_user((int *) arg);
    return 0;
case SG_GET_TIMEOUT:
    return scsi_generics[dev].timeout;
default:
    return scsi_ioctl(scsi_generics[dev].device,
                      cmd_in, (void *) arg);
}
}
}

```

例程 ioctl 可以简单地获得超时的数值或对其进行设置。

---

### 程序清单 11-13 Linux Linux sg\_open 例程

---

```

static int sg_open(struct inode * inode, struct file * filp)
{
    int dev=MINOR(inode->i_rdev);
    int flags=filp->f_flags;

    if (dev>=sg_template.dev_max ||
        !scsi_generics[dev].device)
        return -ENXIO;
    if (O_RDWR!=(flags & O_ACCMODE))
        return -EACCES;

/*
 * 如果我们想进行专用访问，则必须一直等到设备处于
 * 空闲状态，然后设定标志来阻止其他任何形式的访问。
 */
    if (flags & O_EXCL)
    {
        while(scsi_generics[dev].users)
        {
            if (flags & O_NONBLOCK)
                return -EBUSY;
            interruptible_sleep_on(
                &scsi_generics[dev].generic_wait);
            if (current->signal & ~current->blocked)
                return -ERESTARTSYS;
        }
    }
}

```

```

scsi_generic[dev].exclude=1;
}
else
/*
 * 等待一直到没有任何在该设备上的专用访问.
 */
while(scsi_generic[dev].exclude)
{
    if (flags & O_NONBLOCK)
        return -EBUSY;
    interruptible_sleep_on(
        &scsi_generic[dev].generic_wait);
    if (current->signal & ~current->blocked)
        return -ERESTARTSYS;
}

/*
 * 好,此时我们应该已经获取了该设备的使用权,进行必要的
 * 标记以表示现在我们正在使用它,并对状态变量进行初始化.
 */
if (!scsi_generic[dev].users
    && scsi_generic[dev].pending
    && scsi_generic[dev].complete)
{
    if (scsi_generic[dev].buff != NULL)
        sg_free(scsi_generic[dev].buff,
            scsi_generic[dev].buff_len);
    scsi_generic[dev].buff=NULL;
    scsi_generic[dev].pending=0;
}
if (!scsi_generic[dev].users)
    scsi_generic[dev].timeout=SG_DEFAULT_TIMEOUT;
if (scsi_generic[dev].device->host->hostt->usage_count)
    (*scsi_generic[dev].
        device->host->hostt->usage_count)++;
if (sg_template.usage_count)
    (*sg_template.usage_count)++;
scsi_generic[dev].users++;
return 0;
}

```

我们可以通过该例程对将要使用的设备进行预定 (RESERVE)。

---

**程序清单 11-14 Linux 下的 sg\_open 例程**


---

```
static void sg_close(struct inode * inode, struct file * filp)
{
    int dev=MINOR(inode->i_rdev);
    scsi_generic[dev].users--;
    if (scsi_generic[dev].device->host->hostt->usage_count)
        (*scsi_generic[dev].
            device->host->hostt->usage_count)--;
    if(sg_template.usage_count) (*sg_template.usage_count)--;
    scsi_generic[dev].exclude=0;
    wake_up(&scsi_generic[dev].generic_wait);
}
```

该例程用来释放通过 sg\_open 例程所预定的设备。

---

**程序清单 11-15 Linux 下的 sg\_malloc 例程**


---

```
static char *sg_malloc(int size)
{
    if (size<=4096)
        return (char *) scsi_malloc(size);
#ifndef SG_BIG_BUFF
    if (size<=SG_BIG_BUFF)
    {
        while(big_inuse)
        {
            interruptible_sleep_on(&big_wait);
            if (current->signal & ~current->blocked)
                return NULL;
        }
        big_inuse=1;
        return big_buff;
    }
#endif
    return NULL;
}
```

如果请求的数据小于 4K，该例程将会分配一个新的缓冲区；如果请求的数据大于 4K，该例程将会分配一个大的缓冲区。

---

程序清单 11-16 Linux 下的 sg\_free 例程

---

```
static void sg_free(char *buff,int size)
{
#ifndef SG_BIG_BUFF
    if (buff==big_buff)
    {
        big_inuse=0;
        wake_up(&big_wait);
        return;
    }
#endif
    scsi_free(buff,size);
}
```

该例程用来释放所有的由 malloc 例程所分配的缓冲区。

---

程序清单 11-17 Linux 下的 sg\_read 例程

---

```
/*
 * 读回前一个命令的执行结果,我们用未完和完成标志
 * 来表明现在缓冲区是否可用,该命令是否执行完毕.
 */
static int sg_read(struct inode *inode,struct file *filp,
    char *buf,int count)
{
    int dev=MINOR(inode->i_rdev);
    int i;
    unsigned long flags;
    struct scsi_generic *device=&scsi_generics[dev];
    if ((i=verify_area(VERIFY_WRITE,buf,count)))
        return i;

    /*
     * 等待一直到该命令执行完毕.
     */
    save_flags(flags);
    cli();
    while(!device->pending || !device->complete)

    {
        if (filp->f_flags & O_NONBLOCK)
        {
```

```

        restore_flags(flags);
        return -EAGAIN;
    }
    interruptible_sleep_on(&device->read_wait);
    if (current->signal & ~current->blocked)
    {
        restore_flags(flags);
        return -ERESTARTSYS;
    }
}
restore_flags(flags);

/*
 * 现在将执行结果拷贝到用户缓冲区.
 */
device->header.pack_len=device->header.reply_len;

if (count>=sizeof(struct sg_header))
{
    memcpy_toofs(buf,&device->header,
                 sizeof(struct sg_header));
    buf+=sizeof(struct sg_header);
    if (count>device->header.pack_len)
        count=device->header.pack_len;
    if (count > sizeof(struct sg_header)) {
        memcpy_toofs(buf,device->buff,
                     count-sizeof(struct sg_header));
    }
}
else
    count= device->header.result==0 ? 0 : -EIO;

/*
 * 清除并释放设备的使用权,这样可以继续执行其他的命令.
 */
sg_free(device->buff,device->buff_len);
device->buff = NULL;
device->pending=0;
wake_up(&device->write_wait);
return count;
}

```

例程 `read` 首先要检验调用程序所传递的缓冲区是否为可写。然后通过不断地检验数据位来确定该命令是否执行完毕，其中所检验的数据位是通过下面的例程 `command_done` 来进行设置的。当请求的数据已经准备好的时候，可以通过例程 `memcpy_toofs()` 把这些数

据拷贝到用户的空间，并同时释放缓冲区，这样就可以唤醒任何一个等待使用缓冲区的申请。

---

### 程序清单 11-18 Linux 下的 sg\_command\_done 例程

---

```
/*
 * 当一个命令执行完毕时中断处理器调用该函数
 * 改变相应的标志，以表明我们现在已获取命令执行结果
 */
static void sg_command_done(Scsi_Cmnd * SCpnt)
{
    int dev = MINOR(SCpnt->request.rq_dev);
    struct scsi_generic *device = &scsi_generics[dev];
    if (!device->pending)
    {
        printk("unexpected done for sg %d\n", dev);
        SCpnt->request.rq_status = RQ_INACTIVE;
        return;
    }

    /*
     * 检查命令是正常结束，还是产生了某种错误。
     */
    memcpy(device->header.sense_buffer, SCpnt->sense_buffer,
           sizeof(SCpnt->sense_buffer));
    switch (host_byte(SCpnt->result)) {
    case DID_OK:
        device->header.result = 0;
        break;
    case DID_NO_CONNECT:
    case DID_BUS_BUSY:
    case DID_TIME_OUT:
        device->header.result = EBUSY;
        break;
    case DID_BAD_TARGET:
    case DID_ABORT:
    case DID_PARITY:
    case DID_RESET:
    case DID_BAD_INTR:
        device->header.result = EIO;
        break;
    }
```

```

case DID_ERROR:
    /*
     * 看来真应该设定 DID_UNDERRUN 和 DID_OVERRUN 两个错误值, 执行
     * scsi_do_cmd 命令时产生这两个错误值表示产生了不足和溢出错误, 然而
     * 上述错误值还没有正式实施, 所以尽管在其他情况下会返回正确的错误值,
     * 但是还是会由不足错误产生 DID_ERROR 错误值.
     */
    if (SCpnt->sense_buffer[0] == 0 &&
        status_byte(SCpnt->result) == GOOD)
        device->header.result = 0;
    else device->header.result = EIO;
    break;
}

/*
 * 现在激活等待处理结果的进程
*/
device->complete_l;
SCpnt->request.rq_status = RQ_INACTIVE;
wake_up(&scsi_generic[dev].read_wait);
}

```

当 SCSI 适配器由于完成一个命令而产生一个中断请求的时候, 可以通过该例程来获得控制权(可能会好些, 也可能会比较坏)。该例程会把检测数据进行保存, 并检验是否有错误产生。完成上述功能之后, 它就会唤醒最初发送该命令的调用程序。

---

#### 程序清单 11-19 Linux 下的 sg\_write 例程

---

```

static int sg_write(struct inode *inode,
    struct file *filp, const char *buf, int count)
{
    int bsize, size, amt, i;
    unsigned char cmnd[MAX_COMMAND_SIZE];
    kdev_t devt = inode->i_rdev;
    int dev = MINOR(devt);
    struct scsi_generic * device=&scsi_generic[dev];

    int input_size;
    unsigned char opcode;
    Scsi_Cmnd * SCpnt;

    if ((i=verify_area VERIFY_READ, buf, count)))
        return i;

```

```
/*
 * 最小的SCSI命令长度是6个字节,如果见到
 * 任何小于6个字节的所谓的SCSI命令,那很明显一定是假的
 */
if (count<(sizeof(struct sg_header) + 6))
    return -EIO;

/*
 * 如果上一个命令的执行尚未真正结束,那么会一直
 * 等待到上一个命令完全结束才可以执行下一个命令
 */
while(device->pending)
{
    if (filp->f_flags & O_NONBLOCK)
        return -EAGAIN;
#endif DEBUG
    printk("sg_write: sleeping on pending request\n");
#endif
    interruptible_sleep_on(&device->write_wait);
    if (current->signal & ~current->blocked)
        return -ERESTARTSYS;
}

/*
 * 给设备设定新的状态标志
 */
device->pending=1;
device->complete=0;
memcpy_fromfs(&device->header,buf,
              sizeof(struct sg_header));

device->header.pack_len=count;
buf+=sizeof(struct sg_header);

/*
 * 现在我们要从用户缓冲区中提出该命令.
 */
opcode = get_user(buf);
size=COMMAND_SIZE(opcode);
if (opcode >= 0xc0 &&
    device->header.twelve_byte) size = 12;

/*
 * 决定缓冲区的大小.
*/
```

```
/*
input_size = device->header.pack_len - size;
if( input_size > device->header.reply_len)
{
    bsize = input_size;
} else {
    bsize = device->header.reply_len;
}

/*
 * 计算大小时并不包括命令头在内
 */
bsize-=sizeof(struct sg_header);
input_size-=sizeof(struct sg_header);

/*
 * 确保用户已经为该命令的执行传递了足够的字节.
 */
if( input_size < 0 )
{
    device->pending=0;
    wake_up( &device->write_wait );
    return -EIO;
}

/*
 * 分配一个足够容纳所请求的数据的缓冲区,然后近似取一个偶数
 * 的扇区,因为scsi_malloc 每次以512字节为单位分配空间.
 */
amt=bsize;
if (!bsize)
    bsize++;
bsize=(bsize+511) & ~511;

/*
 * 如果我们分配不到缓冲区,则报告错误.
 */
if ((bsize<0) || !(device->buff=
    sg_malloc(device->buff_len=bsize)))
{
    device->pending=0;
    wake_up(&device->write_wait);
    return -ENOMEM;
}
```

```
#ifdef DEBUG
    printk("allocating device\n");
#endif

/*
 * 获取我们将要与之对话的设备的指针, 如果
 * 我们不想中断信息交流, 可以返回适当的信息.
 */
if (!(SCpnt=allocate_device(NJLL,device->device,
    !(filp->f_flags & O_NONBLOCK) )))
{
    device->pending=0;
    wake_up(&device->write_wait);
    sg_free(device->buff,device->buff_len);
    device->buff = NULL;
    return -EAGAIN;
}

#endif DEBUG
printk("device allocated\n");
#endif

SCpnt->request.rq_dev = devt;
SCpnt->request.rq_status = RQ_ACTIVE;
SCpnt->sense_buffer[0]=0;
SCpnt->cmd_len = size;

/*
 * 现在从用户的地址空间拷贝SCSI命令
 *
 */
memcpy_fromfs(cmnd,buf,size);
buf+=size;

/*
 * 如果我们正在写数据, 则拷贝正在写入的数据.
 * 由于 pack_len 中包括了命令和命令头的长度,
 * 所以我们应该从数据长度中减去上述数值.
 */
if (input_size > 0) memcpy_fromfs(device->buff,
    buf, input_size);

/*
 * 设置命令结构中的LUN域
 */
cmnd[1] = (cmnd[1] & 0x1f) | (device->device->lun<<5);
```

```

#ifndef DEBUG
    printk("do cmd\n");
#endif

/*
 * 现在向低端驱动器发送适当的命令，在此我们不必
 * 做任何事，只要在中断产生之后，发送正确的命令即可。
 */
scsi_do_cmd (SCpnt, (void *) cmd,
              (void *) device->buff, amt,
              sg_command_done, device->timeout, SG_DEFAULT_RETRIES);

#ifndef DEBUG
    printk("done cmd\n");
#endif

return count;
}

```

例程 write 用来向 SCSI 设备发送命令。它会等待所有的进行中的命令去完成它，并检验潜在的错误因素，为将要传输的数据分配一个足够大的缓冲区。然后用户的命令（如果需要数据的话，还会传输数据）从用户空间拷贝到内核，并通过例程 scsi\_do\_cmd 来执行。

---

#### 程序清单 11-20 Linux 下的 sg\_select 例程

---

```

static int sg_select(struct inode *inode, struct file *file,
                     int sel_type, select_table * wait)
{
    int dev=MINOR(inode->i_rdev);
    int r = 0;
    struct scsi_generic *device=&scsi_generics[dev];
    if (sel_type == SEL_IN) {
        if(device->pending && device->complete)
        {
            r = 1;
        } else {
            select_wait(&scsi_generics[dev].read_wait, wait);
        }
    }
    if (sel_type == SEL_OUT) {
        if(!device->pending){
            r = 1;
        }
    }
    else

```

```

    {
        select_wait(&scsi_generic[dev].write_wait, wait);
    }
}

return(r);
}

static struct file_operations sg_fops = {
    NULL,           /* 请求 */
    sg_read,        /* 读 */
    sg_write,       /* 写 */
    NULL,           /* 读目录 */
    sg_select,      /* 选择 */
    sg_ioctl,       /* 输入输出控制 */
    NULL,           /* 映射 */
    sg_open,         /* 打开 */
    sg_close,        /* 释放 */
    NULL            /* 同步 */
};


```

该例程可以检验相应的设备上是否会执行一个 write 命令。

---

#### 程序清单 11-21 Linux 下的 sg\_detect 例程

---

```

static int sg_detect(Scsi_Device * SDp){

    switch (SDp->type) {
        case TYPE_DISK:
        case TYPE_MOD:
        case TYPE_ROM:
        case TYPE_WORM:
        case TYPE_TAPE: break;
        default:
            printk("Detected scsi generic sg%c at ",
            'a'+sg_template.dev_noticed,
            SDp->host->host_no, SDp->channel,
            SDp->id, SDp->lun);
    }
    sg_template.dev_noticed++;
    return 1;
}


```

该例程可以用来检测所发现的设备的类型，如果不能确认所发现的设备的类型，将会为该设备显示一条消息。

程序清单 11-22 Linux 下的 sg\_init 例程

该例程可以为所有可以使用的设备创建特殊文件。如果有必要的话，还会为每个可以使用的设备分配传输缓冲区。要为设备表格分配足够的空间，还要外加额外的一对设备的空间。

---

### 程序清单 11-23 Linux 下的 sg\_attach 例程

---

```
static int sg_attach(Scsi_Device * SDp)
{
    struct scsi_generic * gpnt;
    int i;

    if(sg_template.nr_dev >= sg_template.dev_max)
    {
        SDp->attached--;
        return 1;
    }

    for(gpnt = scsi_generics, i=0; i<sg_template.dev_max;
        i++, gpnt++)
        if(!gpnt->device) break;

    if(i >= sg_template.dev_max)
        panic ("scsi_devices corrupt (sg)");
    scsi_generics[i].device=SDp;
    scsi_generics[i].users=0;
    scsi_generics[i].generic_wait=NULL;
    scsi_generics[i].read_wait=NULL;
    scsi_generics[i].write_wait=NULL;
    scsi_generics[i].buff=NULL;
    scsi_generics[i].exclude=0;
    scsi_generics[i].pending=0;
    scsi_generics[i].timeout=SG_DEFAULT_TIMEOUT;
    sg_template.nr_dev++;
    return 0;
};
```

对可以得到的 SCSI 设备的数组进行初始化。

---

**程序清单 11-24 Linux 下的 sg\_detach 例程**

---

```
static void sg_detach(Scsi_Device * SDp)
{
    struct scsi_generic * gpnt;
    int i;

    for(gpnt = scsi_generics, i=0; i<sg_template.dev_max;
        i++, gpnt++)
        if(gpnt->device == SDp) {
            gpnt->device = NULL;
            SDp->attached--;
            sg_template.nr_dev--;
            /*
             * 避免设备的关联 /dev/sg,
             * 每次插入或删除都会增加这
             * 种关联<dan@lectra.fr>
            */
            sg_template.dev_noticed--;
        }
    return;
}
```

释放所有的设备。

## ■ SCSI Pass\_Through 应用程序的例子

在 CD-ROM 中的手册中包含一个 SCSI Pass-Through 应用程序的例子，该例子是放在 Linux 的“SCSI Programming HOW TO ”的文档中。

## ■ 总结

我也知道，在本章中我所叙述的内容不足以让大家出去开始编写 UNIX SCSI 的驱动程序。尽管如此，我还是把一些重要的东西都给大家勾画了出来，大家从中获得编程所需要的必要的信息。在我为以上提到的系统中编写了驱动程序之后，我发现自己的亲自编写

一个简单的驱动程序是学习本章的最好的方法。阅读本章的内容可以使我们能够很好地理解决在开发应用程序的过程中所遇到的问题。

## 感谢

首先我要感谢 Linus Torwalds 和他的伙伴们为我们开发了如此令人愉快的操作系统。而且，特别是他们的程序和源代码都是免费的。

## 第十二章 常用问题及解决方法

到现在为止，你很可能已经看出 SCSI 是一种功能强大的技术，但是也许因为它的复杂性望而却步。如果你明白遇到的问题越多则收获越大的道理，那么在学习本章的时候也就会很轻松了。

编程者在工作的过程中遇到的错误可以分成有限的几种类型。一些问题经常出现，这样我们就对它非常熟悉，也就很容易对付它。一些解决问题的方法和小技巧可以帮助我们节省不少时间。

### 从硬件层开始

就我们各人力量而言，能够尽量减少 SCSI 问题的最重要的方法就是正确地对硬件进行设置。也许有人会认为，想要设置一个完全没有故障的 SCSI 系统几乎是不可能的事，因为其间必须经历众多的尝试和错误。然而实际情况并非如此，如果非要对此作一个清楚的解释的话，就是要保证电缆长度、终端负载以及其在物理层上的元器件设置正确。

如果想要找一本解决物理层问题的参考资料的话，我推荐可以看一下在“comp.periph.scsi”杂志上的月刊“SCSI Game Rules”。该文献主要是由 Gary Field 来负责的，他同时还负责同一专题的 SCSI FAQ 杂志。在该杂志中介绍了各种 SCSI 游戏的规则，并且还列举了如果不遵守规则将会造成的严重后果。

### SCSI 总线终端负载

在有关 SCSI 总线终端负载的叙述中，更多的是一些神话和误导信息。举一个最简单的例子，正常总线应该具有正好两个终端负载，既不能多，也不能少；而且，终端负载还必须在总线的两个端点。这也就意味着主机适配器是一个终端负载，但通常又不是这样。如果我们使用的设备中既有内部设备，又有外部设备，那么主机适配器很可能在链路的中间部分，而不是作为终端负载出现的。

很多时候，会有人认为应该为了确定终端负载的问题，把所有的连接设备终止。其实这种建议非常不可取，因为这样做太过于鲁莽，而且绝大多数的人是不会这样做的。

无源终端负载非常常见，但是有源终端负载更有利於排除故障。为了追求更高的数据传输速率，终端负载变得非常关键。这也正是 Fast SCSI 协议中为什么需要有源负载的重要原因。

## □ SCSI 终端负载电源

通常对于终端负载的电源所出现的问题，我们很难查找到其根源。主机适配器必须提供一个 TERMPWR 信号。但是这条规则又不是绝对的，有很多的例外情况。许多并行端口的 SCSI 适配器并不提供 TERMPWR 信号，相反它是靠与其连接的外部设备来提供该信号的。有一些设备可以提供该信号，但是通常的设备是不提供该信号的。例如艾美佳 (Iomega) 公司的 Zip 驱动器就不提供 TERMPWR 信号，所以它也就不能和大多数的并行端口适配器协调工作。

如果没有合适的终端负载的提供电源的话，很可能会出现非常严重的问题。如果不能确定终端负载的电源是否在允许的范围内，可以使用一个电压表来进行检验。对于单端总线来说，终端负载电源的范围应该在 4.25 伏特和 5.25 伏特之间。

## ■ 慎用电缆

SCSI 总线在电压供应方面要比其他的总线接口严格得多。这也正是我们在使用电缆时非常恼火的原因。真正好的 SCSI 电缆应该具有合适的屏蔽措施，还要有精确的电阻值，并且应该比我们所想像的价格要贵一些。在这里我们并不能有什么近路可走，如果使用一些勉强可用的短电缆，花费将会比使用长电缆更多，而且，还会断断续续地遇到几乎是不可避免的各种问题。

在选用电缆的时候，一定要保证电缆的长度。单端 SCSI 总线总的长度不应该超出 6 米。如果是在两个设备之间进行连接，电缆的长度越短越好。

另外还有一个很明显的问题，就是一定要保证所使用的电缆的两端都必须进行连接。如果电缆只连接了一端的话，通常会导致各种各样的电气事故。

## ■ 不要简单地对待说明书

当我们在使用 SCSI 设备的时候，通常会发现制造商的产品说明书跟超市的小报一样没有什么参考价值。通常这些说明书是为了说明产品的规格，并且在产品面市之前经历了多次修改才确定下来。

如果遇到下述情况，千万不要相信：就是在编程者的说明书中宣称通过引入错误条件来检验设备是如何反应的，看设备是否返回一个特定的错误代码。

对于说明书中含糊不清的地方一定要慎重处理。因为编程者通常比较乐观，他们在处理模棱两可的信息时，通常向着比较容易编写代码的方向来解释。当真的出现问题的时候，他们又会找出第二种说法，这时才会像一个悲观主义者。

## 要注意平台的依赖性

### SCSI 字节顺序

新入门的 SCSI 编程者经常会忽视 SCSI 命令中的字节顺序。其实规则很简单：在 SCSI 命令块中的数值是按照大数原则的顺序进行排列的，高位字节靠前，低位字节靠后。

基于 Intel 平台的编程者通常习惯于小数原则，也就是最低位的数值在最前面。需要注意的是，我们必须做一些移位操作或其他的可以使数值前后移动的操作。

### ASPI 字节顺序

另一方面，ASPI 在它的 SCSI 请求块中使用了基于 Intel 平台的字节顺序。因为 ASPI 是为 Intel 平台所创建，所以这样做也是很正常的。但是当我们使用大数原则的数据来构造 SCSI CDB，然后把它插入使用小数原则的数据组成的 ASPI CDB 中时，十分容易造成混淆。

稍微注意一下管理操作，我们对这个问题的看法将会更加清楚一些。我们会发现这样做对于为经常使用的变换定义宏命令是非常有帮助的。

### 结构队列

这里有一个小错误，但是却很难被发现。为了保持 SCSI 的 CDB（命令描述块），我们已经使用 C 代码定义了一种结构。我们可以给它赋予适当的数值，然后把它发送到一个设备来运行，结果只能是返回一个检测数据，来通知我们在 CDB 中有非法参数。到底是哪里出现了错误呢？

很有可能是我们的结构并没有进行适当的排列。SCSI CDB 必须按照字节的边界来进行排列。但是在许多编译程序中的默认情况是按照字或双字的边界来进行排列的。

所以我们必须首先检验我们的编译选项，检查是否选定了按照字节边界来进行结构排列。我们可以通过调试程序来检验结构是否按照字节边界来进行排列的。

### 缓冲调整

缓冲调整是存在的另一个问题。当我们使用 ASPI 工作的时候，通常需要检验主机适配器的性能。执行 SC\_HA\_INQUIRY 函数通常会返回一个缓冲调整掩码。通过这个掩码可以对我们为该主机而传递给 ASPI 控制器的缓冲器的指针进行排列。如果不按照这个规程来做，那到时候吃亏的将会是自己。

SCRIPTS 代码也为缓冲调整制定了严格的规则。在指向内存地址的寄存器中保留了

DWORD 的数值，所以所有的缓冲器必须按照 DWORD 边界来进行排列。

最早的符合缓冲调整规则的方法就是按照 DWORD 边界来排列所有的缓冲器。这样也能够满足不太严格按字或字节排列的缓冲器的需求。下面举一个例子，SCSI 的 Snooper 程序中采用这种方法定义了一个称为排列缓冲器（AlignedBuffer）的使用类。这个排列缓冲器（AlignedBuffer）类可以对内存进行分类，并且会返回一个排列在下一个双字节边界上的指针。

## 测试工具

对 SCSI 的代码进行测试通常是令人非常痛苦的，而且常常还会向一个设备也就是黑盒子发送 CDB 代码。顺利的话设备也许会工作良好，但是也许会出现你永远都不会想到的错误。

### 交互式命令的实用程序

如果有充足的时间的话，我们可以编写一个很有价值的小应用程序，通过这个应用程序我们可以编辑 CDB 代码，并把这些 CDB 代码发送给设备，从而来检查设备的输出。这样就可以使我们在完成任何生产代码之前使用并熟悉相应的设备。

无论是在商业领域还是在公共领域，都存在很多能够完成该功能的工具。SCSI 驱动器和控制器的制造商西部数据（Western Digital）公司开发了名为 ASPI Menu 的工具。这个工具可以完成系统中所安装的 ASPI 的驱动程序，这样我们就可以和 SCSI 设备进行交互式工作。我们可以从西部数据公司的站点“[www.wdc.com](http://www.wdc.com)”上免费下载 ASPI Menu 工具。

### 虚拟设备

如果想要从多方面广泛地了解一个特定的设备，那么采用软件仿真的方法通常是很帮助的。在测试的机器上使用不同的 SCSI ID 设置第二个主机适配器，然后把它与第一个主机适配器相连接。通过适当的软件，可以把第二个主机适配器当作一个 SCSI 外设来使用。

对于编写目标模式的软件开发商来说，通常这也是唯一的一种可以在应用到固件中之前对代码进行测试的方法。性能拙劣的应用程序即使是在仿真的设备上也不能承受破坏，当然在实际的设备上更是如此。

我们很难对在 SCSI 磁盘驱动器上使用的软件进行安全性的测试。在本书中我们在很多例子中使用了 Iomega 的 Zip 驱动器，这种驱动器的存储介质很容易更换。其实更好的解决方案就是使用虚拟磁盘驱动器。在本书附带的 CD-ROM 中有一个 SCRIPTS 的代码的例子，它通过建立这样一个虚拟磁盘驱动器来展示了目标模式的 SCSI 编程。如果你使用的是兼容 Symbios Logic 的主机适配器，那么很有必要学习一下该应用程序的源代码。

## ■ SCSI 总线分析程序

最原始的 SCSI 调试工具就是 SCSI 总线分析程序。总线分析程序可以使总线上的错误变得显而易见。非法的相位传输和无用的信号线都暴露无遗。

尽管 SCSI 总线分析程序有各种各样的形式，但是它的基本目的都是为了能够在任何给定的时刻获得 SCSI 总线的使用状况信息。总线分析程序可以通过检验信号线来判断出总线所处的相位阶段。根据模型设定的情况，总线分析程序既可以显示原始数据，还可以显示格式化后的命令描述块（Command Descriptor Blocks）。在许多模型中需要测量信号的电压值和总线的时钟。另外有的模型还能够通过捕获和分析来让我们了解命令和数据传输的序列过程。

如果非要找出总线分析程序的缺点的话，那就是它的价格可能会令人望而生畏。即使那些对某些小工程无能为力的总线分析程序，大多数也都在几千美元以上。但是，如果真的要深入学习 SCSI 的话，这点投资还是蛮值得的。



## 记录

当我们自己使用 SCSI 的时候，很可能会产生一些自己的建议和想法。随着 SCSI 规范的引入，也就暴露出了编程者的一些潜在的错误。要想避免这些错误的发生，可以随时记录下来以备后用。如果你在整个工程期间没有任何记录可供参考的话，你将会惊奇地发现自己是多么的健忘。

## 第十三章 应用举例：SCSI Snooper

到现在为止，我们仅仅就一些例子中的代码片断来介绍了 SCSI 的概念。现在我们该使用这些信息来创建一个特殊的应用程序了。

大多数的编程者在学习 SCSI 的时候，都是从编写用来定位和鉴别总线连接情况的报表程序开始的。我们将会深入地探讨这个方法。

我们将要开发一个包括 SCSI 接口类和设备类的 C++ 类库，供 ASPI 调用。我们当然不会从头建立该类库，而是从基本类继承而衍生出特殊的设备类库。

然后我们将会使用建立的类库来创建 SCSI 检查工具来检测主机适配器和与之相连的外围设备。这是一个 32 位的 Windows 应用程序，可以运行于 Windows 95 和 NT 工作环境中。

如果要创建 Windows NT 平台上的应用程序，需要首先作一些准备工作。因为我们在这个应用程序中将要使用 ASPI32 服务程序，但这个应用程序并不属于 Windows NT 的默认安装组件。除非我们已经拥有安装了该应用程序的环境，否则必须自己动手对其进行安装。我们可以参考附录 C 中的说明，来对该应用程序进行安装。

因为该应用程序是用 Microsoft Visual C++ 语言进行编写的，所以安装该应用程序还必须首先得到 MSVCRT40.DLL 和 MFC40.DLL 两个文件。如果幸运的话，这两个文件可能已经存在于计算机中。如果计算机中还没有这两个文件的话，从本书附带的 CD-ROM 中拷贝到系统目录中即可。

### SCSI Snooper 的概观

首先让我们来看看下面例子中的应用程序是如何工作的，然后再检查它究竟是如何进行配置的。

在这个应用程序中显示了表示主机适配器和其外围设备的图标。通过单击这些图标，我们可以单击这些图标来获得来自 ASPI 控制器和 SCSI 请求命令的相关信息。

在主机适配器的信息屏幕上显示了来自于 ASPI 驱动器的各种相关信息。它可以显示主机适配器的编号，ASPI 控制器的名称和驱动器确认字符串。

如果在不同的机器上运行该应用程序，你也许感到非常惊奇。许多的设备驱动器制造商采用了 ASPI 模型。例如，Iomega 在它们的并行端口的 Zip 驱动器中就使用了 ASPI 兼容的设备驱动器。现在流行的 IDE 驱动器就在它们兼容 ASPI 驱动器的基础上，基于 SCSI 协议使用了 ATAPI 接口界面。我们在 IBM ThinkPad 笔记本电脑上曾对此进行了测试，该笔记本电脑上安装了 ATAPI CD-ROM 驱动器，结果出现在 ASPI 主机适配器列表上的内部接口标识为“ESDI\_506”。

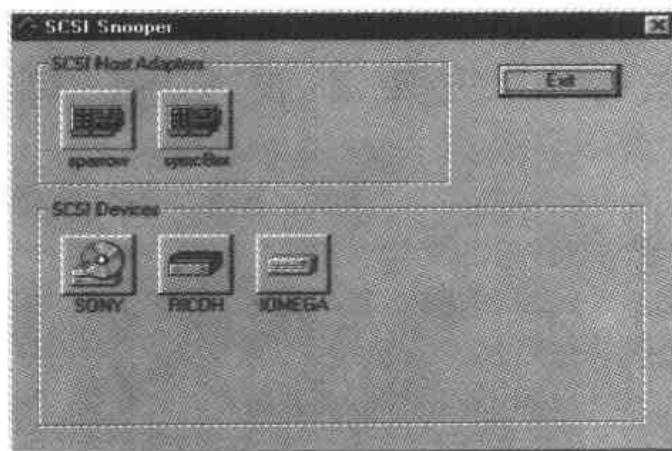


图 13-1 SCSI Snooper 开放界面

下面我们举一个使用 Symbios Logic SYM53C825 和 Adaptec1522 主机适配器的 Windows95 的例子。在它的显示屏幕上显示的信息表明，它是一种能够支持 16 个设备，使用编号为“7”的 SCSI ID 的 Wide SCSI 控制器。

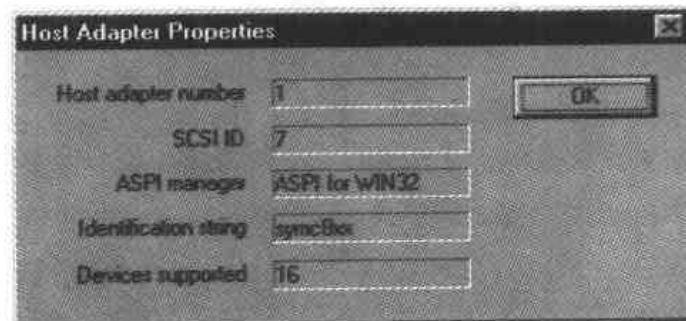


图 13-2 主机适配器显示屏幕

不管连接的是什么主机适配器，所有连接的外设都会被当作是一个设备组合来对待。当我们单击其中某个图标的时候，将会在信息屏幕上显示相应的设备地址、类型和标识。Iomega 公司的 Zip 驱动器所显示的信息如图 13-3 所示。

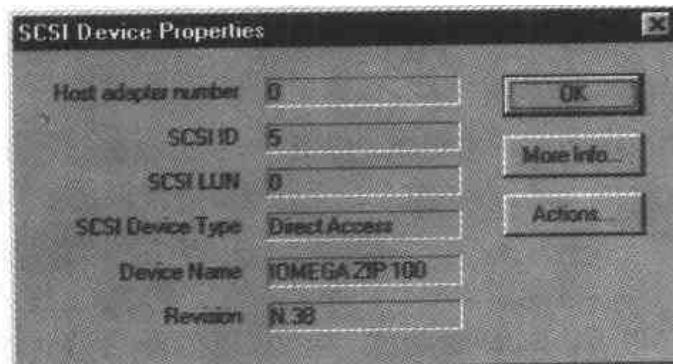


图 13-3 Iomega Zip 驱动器基本信息

如果单击其中的“More Info...”命令按钮，将会显示一些有关该设备的高级信息。我们可以得到该设备所支持的特性等，我们还可以通过 Inquiry 命令来获得该信息。

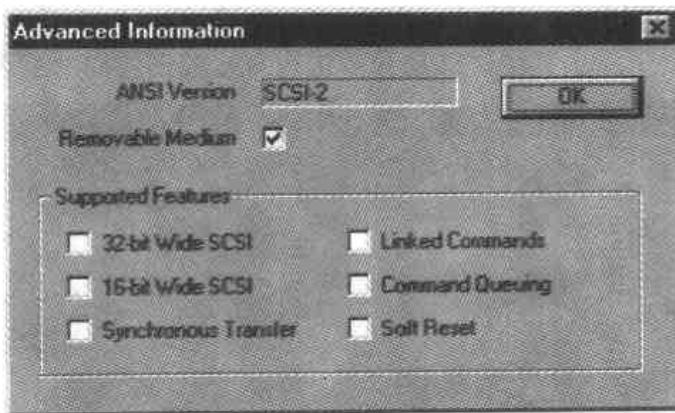


图 13-4 Iomega Zip 驱动器高级信息

下面是一个 SCSI 的 CD-ROM 驱动器的信息。从这些基本信息中我们可以知道该驱动器是 Sony 公司的 CD-ROM 驱动器，使用的 SCSI ID 编号为“2”。

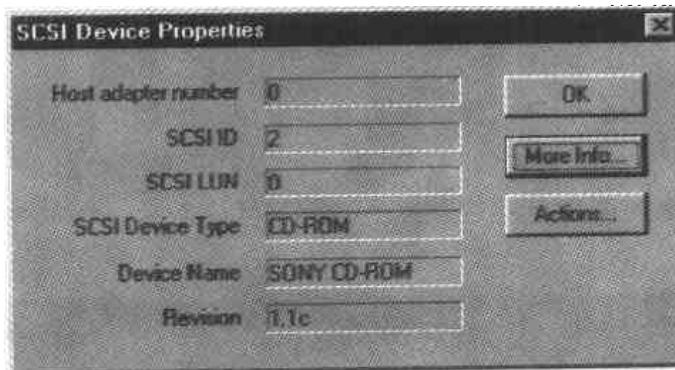


图 13-5 CD-ROM 驱动器基本信息

再一次单击“More Info...”命令按钮，将会弹出高级信息框来。从这个高级信息框中，我们可以知道该设备支持连接的命令，还可以进行同步数据传输。

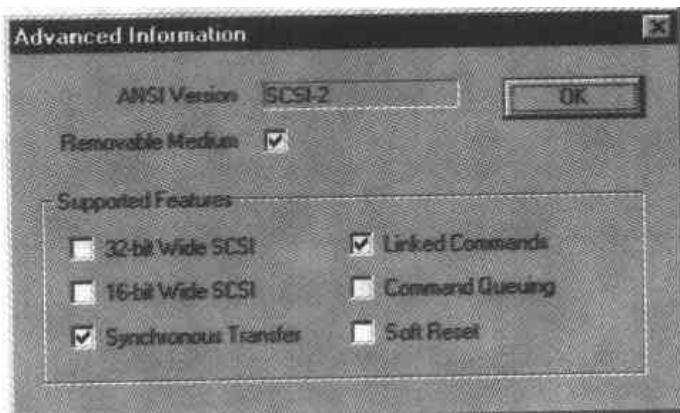


图 13-6 CD-ROM 驱动器高级信息框

单击设备屏幕上的“Actions...”命令按钮，将会弹出一个我们可以在特定设备上执行的命令活动。无论我们所使用的设备如何，都可以看到如下的相同的基本信息：“Test Unit Ready, Device Inquiry and Read Sense”。其他的活动根据外设是否支持而不同。例如，在 Zip 驱动器的显示中就列出了属于利用可移动介质进行直接存储访问的列表。

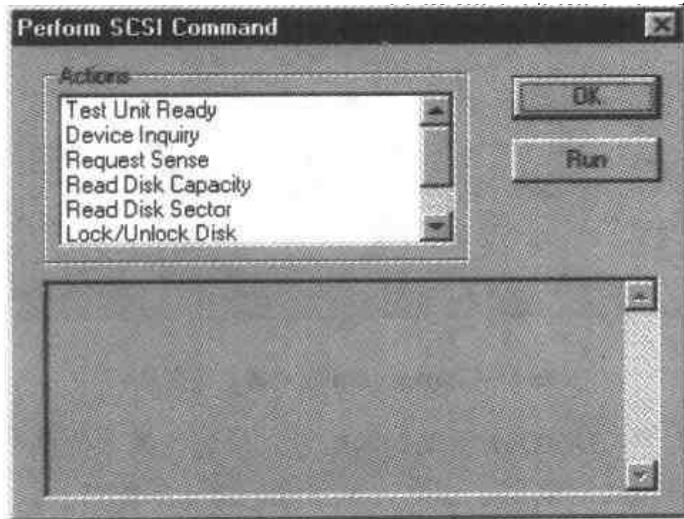


图 13-7 Zip 驱动器活动列表

其中进行直接存储的命令包括读取磁盘、读取扇区、锁定和解锁或卸载可移动介质等命令。首先在活动列表中选取“Read Disk Sector”，然后单击“Run”命令按钮，将会弹出一个对话框，让用户选择用来访问的数据所在的扇区。然后，该命令的执行结果将会出现在输出框中。

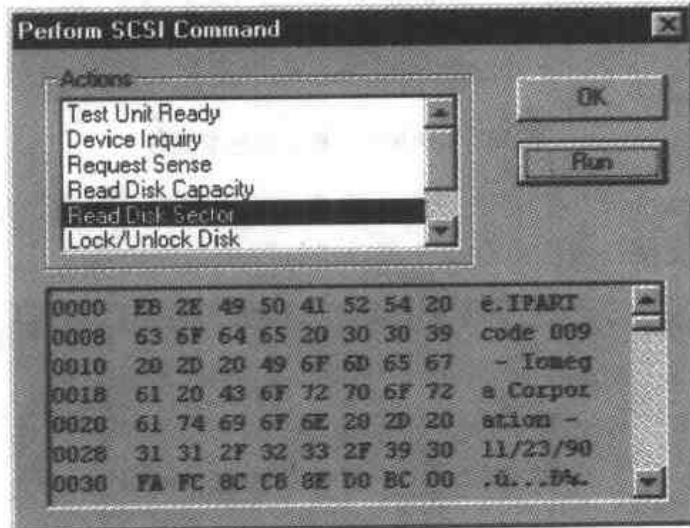


图 13-8 Zip 驱动器读取扇区结果

如果该命令没有成功执行，那么在输出框中将会显示错误信息，并显示所有的检测数据。

CD-ROM 的活动显示屏幕和直接存储设备的非常类似。

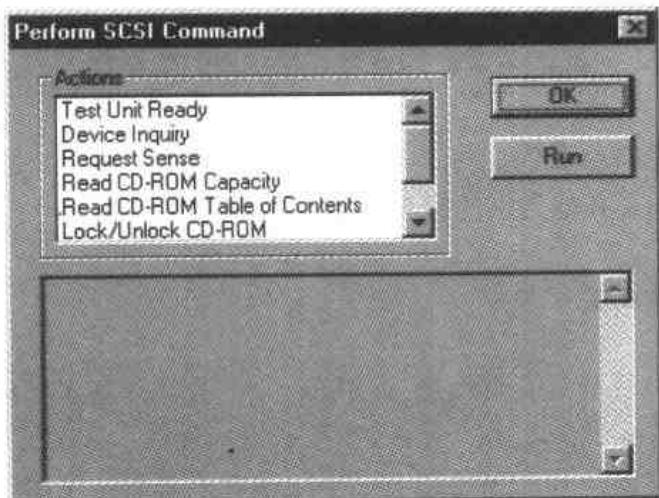


图 13-9 CD-ROM 驱动器活动

在 CD-ROM 的活动列表中，用户可以选择读取 CD-ROM 容量、锁定和开启 CD-ROM 以及卸载 CD-ROM 等活动。这里并没有读取一个扇区数据的命令，取而代之的是读取 CD-ROM 内容列表的命令。执行该命令将会在输出框中显示有关磁道的信息。

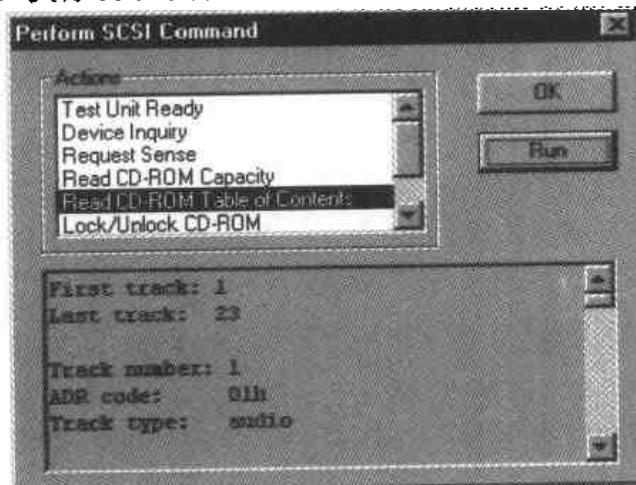


图 13-10 CD-ROM 的 Read CD-ROM Table of Contents 命令执行结果

在这个版本的 SCSI Snooper 中仅仅提供了直接存储设备和 CD-ROM 驱动器的扩展命令。而对于像扫描仪和磁带驱动器这样的 SCSI 外围设备，所提供的活动仅限于适合所有设备的命令。但是，我们可以很容易地增加其他类型设备的支持。下面我们将就应用程序的执行过程来向大家介绍如何增加这种支持。

## ASPI 类库

SCSI Snooper 应用程序的基础就是 ASPI 类库。该类库可以通过使用我们在第七章中讨

论的基本原理来创建设备的接口类。它在基本的 ASPI 功能的基础上进行了简单的包装，增加了 SCSI 请求分程序，并把该请求分程序传递给 ASPI 控制器，检验该程序是否执行完毕，并显示映射错误。

需要大家注意的是，当我们使用“类”这个词语时，通常指的是 C++ 的类，而不是我们所说的设备类，也不是其他类型的跟 Windows 编程相关的类。

## ■ ScsiInterface 类

我们所说的类库中最基本的类就是 `ScsiInterface` 类。该基本类提供了一个机制，通过这个机制可以实现软件和设备之间基于 ASPI 控制器的通信。在 `ScsiInterface` 类中包括一个包含了主机适配器信息的阵列，另外还有一个跟该适配器相连接的设备列表。通过这个列表，我们可以不必知道某个设备所连接的适配器，而直接与特定的设备进行数据交换。

下面是 `ScsiInterface` 类的定义。

程序清单 13-1 `ScsiInterface` 类定义

```
Class ScsiInterface {
    Public:
        int AspiIsOpen;
        Unsigned NumAdapters;
        Unsigned NumDevices;

        AdapterInfo *AdapterList;

        ItemList ScsiDevList;

        ScsiInterface();
        ScsiInterface(int BuildDeviceList, int type=-1, Int scan_luns=0);
        ~ScsiInterface();

        ScbError OpenAspiLayer();
        Unsigned GetNumAdapters();

        Unsigned GetNumDevices();
        ScsiDevice *GetDevice(unsigned i);

        Int BuildDeviceList(int type=-1,int Scan_luns=0);
        Void ClearDeviceList();
```

```
ScbError RescanBus(int type=-1,int scan_luns=0);

ScbError AttachDevice(unsigned adapter,unsigned unit,
    Unsigned lun,int type=-1);
Void RemoveDevice(unsigned adapter,unsigned unit,
    Unsigned lun);

ScsiDevice *FindDevice(char *name);
ScsiDevice *FindDevice(unsigned adapter,unsigned unit,
    Unsigned lun);

};
```

在 ScsiInterface 类中有一个很重要的元素就是 ScsiDeviceList 成员。该成员是一个用来描述所连接的 SCSI 设备的对象列表。另外还有类成员函数可以帮助我们管理该列表，并可以按照名字和 SCSI 地址进行排列。

我们又单独为该列表中的对象定义了一个类。

## ■ ScsiDevice 类

ScsiDevice 类用来描述 SCSI 外围设备的属性和特征，另外还定义了用来访问该设备的函数。该类要比 ScsiInterface 类复杂一点。下面我们给出该类的定义。

---

程序清单 13-2 ScsiDevice 类定义

---

```
class ScsiDevice
{
public:

    int Adapter;
    int Unit;
    int Lun;
    int Type;
    char *RealName;
    char *Name;
    char *Revision;
    unsigned int AnsiVersion;
    unsigned int bRemovable;
    unsigned int bWide32;
    unsigned int bWide16;
    unsigned int bSync;
```

```

unsigned int bLinked;
unsigned int bQueue;
unsigned int bSoftReset;
long RetryOnScsiBusy;                                //等待几毫秒
long RetryOnScsiError;                               //等待几毫秒
long RetryOnScsiError;                               //等待几毫秒
long RetryOnUnitAttention;                          //等待几毫秒
long RetryOnTargetBusy;                            //等待几毫秒
long RetryOnTargetNotReady;                         //等待几毫秒
long RetryOnTargetBecomingReady;                   //等待几毫秒

ScsiDevice();
~ScsiDevice();

char *GetName() { return Name; }
char *GetRealName() { return RealName; }
char *GetRevision() { return Revision; }
int GetAdapter() { return Adapter; }
int GetUnit() { return Adapter; }
int GetLun() { return Adapter; }
int GetType() { return tYpe; }

unsigned int GetAnsiVersion() { return AnsiVersion; }
unsigned int IsRemovable() { return bRemovable; }
unsigned int IsWide32() { return bWide32; }
unsigned int IsWide16() { return bWide6; }
unsigned int IsSync() { return bSync; }
unsigned int IsLinked() { return bLinked; }
unsigned int IsQueue() { return bQueue; }

unsigned int IsSoftReset() { return bSoftReset; }
int SetName(char *name);

ScbError Init(unsigned adapter,unsigned unit,unsigned lun );
ScbError ExecuteScb( ScsiCmdBlock &scb,log timeout );
};


```

在 ScsiDevice 类中定义了用来存储设备名字、SCSI 地址、主机适配器和所支持的特征的数据成员。该类的某些成员函数为了简化而定义成内联的形式，并且可以返回上述参数值。

ScsiDevice 类中最常用的功能函数就是 ExecuteScb 成员函数。该成员函数通过 ScsiCmdBlock 类来处理 SCSI 命令的执行。

## ■ ScsiCmdBlock 类

ScsiCmdBlock 类用来创建 ASPI SCSI 请求分程序，并把这些分程序传递给 ASPI 控制器来执行。

### 程序清单 13-3 ScsiCmdBlock 类的定义

```
class ScsiCmdBlock
{
public:
    ScsiRequestBlock srb;
    ScbError LastError;

    ScsiCmdBlock();
    ~ScsiCmdBlock();

    void Init( unsigned cmd,unsigned adapter=0,
               unsigned target=0,unsigned lun=0 );

    //下面的例程假定为 SC_EXEC_SCSI_CMD 类型的命令
    void SetCdb(void *cdb,unsigned nbytes);
    void GetSense(void *sense,unsigned maxbytes);
    void SetDataBuffer(void *bufp,unsigned buflen);
    ScbError Execute(long timeout = -1L);

};


```

Execute 函数将一个 SCSI 请求分程序传递给带有下划线的 ASPI 控制器。其他的类成员函数为 I/O 操作设置数据缓冲区，并负责取回检测数据。

## ■ ScsiInterface 类的初始化

下面让我们来看一下这些类是如何工作的。我们首先介绍 ScsiInterface 构造函数的第二种形式。

```
ScsiInterface (int BuildDeviceList, int type=-1, int scan_luns=0);
```

通过执行该构造函数，可以对 SCSI 总线进行扫描，并且建立一个设备列表。而且我们

还可以首先制定设备的类型然后进行查找，还可以选择是否检查每个地址的逻辑设备号。系统的默认设置是对所有类型的设备进行检查，但是仅仅扫描 LUN 编号为“0”的区块。该构造函数除了调用的是 BuildDeviceList 成员函数之外，与默认的格式非常类似。BuildDeviceList 扫描设备所连接的主机适配器，并建立一个与该主机适配器相连接的设备的列表。

---

#### 程序清单 13-4 ScsiInterface::BuildDeviceList 成员函数

---

```

int ScsiInterface::BuildDeviceList(int type,int scan_luns)
{
    unsigned adapter,unit,lun;
    ScbError err;

    If (!AspiIsOpen)
    {
        err = OpenAspiLayer();
        if (err)
            return 0;
    }
    // 痴心妄想
    assert(NumAdapters <= MAX_HOST_ADAPTERS);

    for (adapter=0;adapter<NumAdapters;adapter++)
    {
        ScsiCmdBlock scb;
        scb.Init(SC_HA_INQUIRY,adapter,0,0);
        err = scb.Execute(3000);
        if(!err)
        {
            unsigned host_unit = scb.srb.hai.HA_SCSI_ID;
            unsigned max_units = scb.srb.hai.HA_Unique[3];
            if (max_units == 0)      // 旧的 ASPI 管理器?
                Max_units = 8;

            //保存主机适配器信息
            if (AdapterList)
            {
                char *s;
                int i;

                AdapterList[adapter].AdapterNum = adapter;
                AdapterList[adapter].ScsiId = host_unit;
                AdapterList[adapter].MaxUnits = max_units;
            }
        }
    }
}

```

---

```
AdapterList[adapter].Residual = Scb.srb.hai.HA_Unique[2];
AdapterList[adapter].Align =
    ((WORD) scb.srb.hai.HA_Unique[0] |
     (WORD) scb.srb.hai.HA_Unique[1] << 16);

//保存适配器管理器 ID
s = AdapterList[adapter].ManagerId;
for (i=0; i<sizeof(scb.srb.hai.HA_ManagerId); i++)
    *s++ = scb.srb.hai.HA_ManagerId[i];
*s = '\0';

// 修整后面的空间
while (--s > AdapterList[adapter].ManagerId)
{
    if (isascii(*s) && isspace(*s))
        *s = '\0';
    else
        break;
}

// 保存适配器标识符
s = AdapterList[adapter].Identifier;

for (i=0; i<sizeof(scb.srb.hai.HA_Identifier); i++)
    *s++ = scb.srb.hai.HA_Identifier[i];
*s = '\0';

// 修整后面的空间
while (--s > AdapterList[adapter].Identifier)
{
    if (isascii(*s) && isspace(*s))
        *s = '\0';
    else
        break;
}

for (unit=0;unit<max_units;unit++)
{
    if (unit != host_unit)
    {
        if (scan_luns)
        {
            for (lun=0; lun<8; lun++)
            {
```

```

        err = AttachDevice(adapter,unit,lun,type);
        if (err)
            break;
    }
}
else
{
    lun = 0;
    AttachDevice(adapter,unit,lun,type);
}
}
}
}

return NumDevices;
}

```

该成员函数并没有什么神秘的地方。执行该成员函数首先检验 ASPI 层是否处于打开状态，如果有必要的话还会调用初始化函数。不过执行该初始化函数的一个副作用就是它将会记录当前主机适配器的编号。

该成员函数的另一个功能就是通过主机适配器循环，并搜集和存储有关主机适配器和相连接的设备的有关信息。而对于每个主机适配器来说，它将会通过调用 AttachDevice 例程来检验所有可能正在使用的外围设备的 SCSI 地址。AttachDevice 函数用来执行实际的查询功能。程序清单 13-5 中列出了该函数的具体形式。

---

#### 程序清单 13-5 ScsiInterface::AttachDevice 成员函数

---

```

ScbError ScsiInterface::AttachDevice(unsigned adapter,
    Unsigned unit,unsigned lun, int type)
{
    ScbError err;
    ScsiCmdBlock scb;
    ScsiDevice *dev;

    If (!AspiIsOpen)
    {
        err = OpenAspiLayer();
        if (err)
            return err;
    }

    //必须确保我们尚未连接它

```

```
if ( (dev=FindDevice(adapter,unit,lun)) !=NULL )
{
    if ( (type == 1) || (type == dev->GetType()) )
        return Err_None;
    else
        return Err_NoDevice; // 错误类型
}

//通过获取设备类型检查设备是否已经存在，这应该在初始化时进行
scb.Init(SC_GET_DEV_TYPE,adapter,unit,lun);
err = scb.Execute(1000L);
if (err)
    return err;

if ( (type != -1) && (type != scb.srb.gdt.SRB_DeviceType) )
{
    // 错误类型
    return Err_NoDevice;
}

dev = new ScsiDevice;
if (!dev)
    return Err_OutOfMemory;

err = dev->Init(adapter,unit,lun);
if (err)
{
    delete dev;
    return err;
}

dev->SetName(dev->GetRealName());
if (!ScsiDevList.AddItem(dev))
{
    delete dev;
    return Err_OutOfMemory;
}

NumdDevices++;
return Err_None;
}
```

AttachDevice 成员函数首先执行 ASPI 中的 SC\_GET\_DEV\_TYPE 函数来确定与特定 SCSI ID 和 LUN 相连接的设备类型。如果 ASPI 调用返回的是合法的设备类型的话，AttachDevice 接着将会调用 ScsiDevice Init 成员函数。该 Init 函数首先执行一个 SCSI Inquiry 调用，然后保存设备的标志字符串和性能指标以备后用。下面我们给出它的例子。

---

### 程序清单 13-6 ScsiDevice::Init 成员函数

---

```
ScbError ScsiDevice::Init(unsigned adapter,unsigned unit,unsigned lun )
{
    ScbError err;
    ScsiCmdBlock scb;
    SCSI_Cdb_Inquiry_t cdb;
    SCSI_InquiryData_t inq;

    Adapter = adapter;
    Unit = unit;
    Lun = lun;

    Scb.Init(SC_EXEC_SCSI_CMD,Adapter,Unit,Lun);

    memset(&cdb,0,sizeof(cdb));
    cdb.CommandCode = SCSI_Cmd_Inquiry;
    cdb.Lun = Lun;
    cdb.Evpd = 0;
    cdb.PageCode = 0;
    cdb.AllocationLength = sizeof(inq);
    scb.SetCdb(&cdb,6);

    memset(&inq,0,sizeof(inq));
    scb.SetDataBuffer(&inq,sizeof(inq));

    {
        long tmp = RetryOnScsiError;
        err = ExecuteScb(scb,3000L);
        RetryOnScsiError = tmp;
    }

    if (!err)
    {
```

```
Type = inq.DeviceType;
if (RealName)
    free(RealName);
RealName = (char *) malloc(sizeof(inq.VendorId) +
    sizeof(inq.ProductId)+2);
if (RealName)
{
    char *s = RealName;
    int i;
    for (i=0; i<sizeof(inq.VendorId); i++)
        *s++ = inq.VendorId[i];
    //修整后面的空间
    while (--s > RealName)
    {
        if (!isascii(*s) && isspace(*s))
            break;
    }
    s++;
    *s++ = ' ';
    for (i=0;i<sizeof(inq.ProductId); i++)
        *s++ = inq.ProductId[i];
    *s = '\0';
    //修整后面的空间
    while (--s > RealName)
    {
        if (isascii(*s) && isspace(*s))
            *s = '\0';
        else
            break;
    }
    if (Name == NULL)
        SetName(RealName);
}

// 保存修正的字符串
if (Revision)
    free(Revision);
Revision = (char *)
    Malloc(sizeof(inq.ProductRevisionLevel+1));
```

```

if (Revision)
{
    char *s = Revision;
    int i;
    for (i=0; i<sizeof(inq.ProductRevisionLevel); i++)
        *s++ = inq.ProductRevisionLevel[i];
    *s = '\0';

    //修整后面的空间
    while (--s > Revision)
    {
        if (isascii(*s) && isspace(*s))
            *s = '\0';
        else
            break;
    }
}

// 保存其他属性
AnsiVersion = inq.AnsiVersion;
bRemovable = inq.RemovableMedia;
bWide32 = inq.WideBus32Support;
bWide16 = inq.WideBus16Support;
bSync = inq.SynchronousTransferSupport;
bLinked = inq.LinkedCommandSupport;
bQueue = inq.CommandQueueSupport;
bSoftReset = inq.SoftResetSupport;
}

else
{
    Type = 0x1F;
}

return err;
}

```

我们首先仔细阅读一下这些代码。因为这些代码描述了如何建立 SCSI CDB 为 ScsiCmdBlock 制定 CDB 代码、设置数据缓冲区的地址、传递 ScsiCmdBlock 并执行的方法。ScsiDevice 中所有的 SCSI 调用都符合该模型。

## 口 执行 ScsiCmdBlock

我们终于该讨论实际的 ASPI 函数的调用了。ASPI 函数的调用发生在 ScsiCmdBlock 类上。其中，Execute 成员函数设置 SCSI Request Block 的一些标志位，并将其传递给 DoAspiCommand 成员函数。DoAspiCommand 是一个静态函数，而且它并不属于 ScsiCmdBlock 类中的成员。

### 程序清单 13-7 DoAspiCommand 函数

```
Static int DoAspiCommand(ScsiRequestBlock *p, long timeout)
{
    HANDLE hEvent;
    long wait;

    //获得事件处理器
    hEvent = p->io.SRB_PostProc;

    //映射超时设定
    wait = (timeout == -1L) ? INFINITE : timeout;

    ResetEvent(hEvent);

    aspi_SendCommand(p);

    if (p->io.SRB_Status == SS_pending)
    {
        if (WaitForSingleObject(hEvent, wait) == WAIT_OBJECT_0)
        //事件完成
        {
            ResetEvent(hEvent);
            Retrn 1;
        }

        time_t elapsed_time;
        time_t starttime = time(NULL);

        while (p->io.SRB_Status == SS_PENDING)
        {
            elapsed_time = time(NULL) - starttime;
```

```
if (timeout != -1)
{
    if (elapsed_time > (timeout/1000 + 1) )
    {
        if (p->io.SRB_Cmd != SC_ABORT_SRB)
        {
            if ( p->io.SRB_Cmd != SC_ABORT_SRB)
            {
                // 异常中断
                SRB_Abort a;
                Memset(&a,0,sizeof(a));
                a.SRB_Cmd = SC_ABORT_SRB;
                a.SRB_HaId = p->io.SRB_HaId;
                a.SRB_ToAbort = p;
                aspi_SendCommand(&a);
                starttime = time(NULL);
                while (a.SRB_Status == SS_PENDING)
                {
                    if ( time (NULL) > (starttime + 4) )
                    {
                        //在某些地方已经发生严重错误，甚至我们不能取消
                        //命令的执行，忽略该命令，假装原来执行的命令超时
                        break;
                    }
                    Sleep(10L);
                }
            }
            // 中断取消并返回相应的代码
            return 0;
        }
    }
    if ( elapsed_time > 2)      //判断是否为长命令，如果是则分配
        Sleep(1000L);          //给操作分流更多的时间,
    Else
        Sleep(10L);           //如果不是，就少给一点时间.
    }
}
return 1;
```

我们看到上述代码时一定不会陌生，因为在第七章中已经介绍过一个类似的例子。我们使用事件标志来检测命令是否执行完毕，并增加了一些超时和其他类型的检验。

## 使用 ASPI 类库

我们已经介绍了 ASPI 底层的工作原理，下面接着介绍如何将其应用到实际应用中。

### 口 衍生 SCSI 设备类型

我们在使用 SCSI 外围设备时一个很有用的抽象就是为设备定义一个通用的设备类型，并从该设备类型衍生相应的设备。通用的设备能够支持常用的 SCSI 功能函数，例如 Test Unit Ready、Read Sense、Inquiry 以及其他适用于所有外围设备类型的功能函数。从这些基本类，我们可以衍生出针对直接存储设备、CD-ROM 驱动器和其他设备的类。

这也正是我们在 SCSI Snooper 中所使用的方法。这里我们首先定义一个称为 ScsiBaseDevice 的基类。该基类可以处理普通的函数和错误映射处理。在程序清单 13-8 中给出了该基类的定义。

---

#### 程序清单 13-8 ScsiBaseDevice 类定义

---

```
//衍生 SCSI 设备类的基本类
class ScsiBaseDevice
{
public:

    ScsiDevice *Device;

    int IsOpen;
    int LastError
    int SystemError;
    int LastScsiError;

    unsigned Adapter;
    unsigned Unit;
    unsigned Lun;

    ScsiCmdBlock Scb;      //所有的命令都使用这个 scsiCmdBlock
```

```
MutexSemaphore ScbMutex;

Union
{
    SCSI_SenseData_t Sense;
    Unsigned char SenseBuffer[SENSE_LEN];
};

ScsiDeviceAttributes_t Attributes;
SCSI_InquiryData_t      InquiryData;

ScsiBaseDevice();
~ScsiBaseDevice();

ScsiDevice *GetScsiDevice();

ScsiError_t Open(ScsiDevice *dev,
                 ScsiDeviceAttributes_t *attr=0);
ScsiError_t Close(void);
ScsiError_t DoCommand( void *cdb, unsigned cdblen,
                      void *dbuf,unsigned long dbuflen, int dir,
                      long timeout );

int ValidResidualCount();
long GetResidualCount();
unsigned MapAscAsq();
ScsiError_t MapScsiError();

ScsiError_t WaitTilReady(long timeout = -1);

ScsiError_t TestUnitReady();
ScsiError_t RequestSense(void *bufp,unsigned maxbytes);
ScsiError_t Inquiry( void *bufp, unsigned maxbytes,
                     int evpd=0,int page_code=0 );
ScsiError_t ModeSelect( ovid *bufp,unsigned nbytes,
                       int pf=0, int sp=0 );
```

```
ScsiError_t ModeSense( void *bufp,unsigned maxbytes,
    int page_code=0, int pc=0, int dbd=0 );

void QueryErrorString(Scsierror_t errcode,char *bufp,
    unsigned maxbytes);
char *QueryMajorErrorString(ScsiError_t errcode);

};
```

Device 成员变量是一个指向类库中 ScsiDevice 对象的指针。在使用之前该对象必须已经存在，并且要首先通过 Open 函数传递给 ScsiBaseDevice 对象。

稍高层次的函数都会调用 DoCommand，该函数可以使指针指向一个 CDB 和参数数据缓冲区。最简单的例子就是 Inquiry 函数，该函数执行一个 SCSI 的 Inquiry 命令。

---

#### 程序清单 13-9 ScsiBaseDevice::Inquiry 成员函数

---

```
ScsiError_t ScsiBaseDevice::Inquiry( void *bufp,
    unsigned maxbytes, int evpd, int page_code )
{
    SCSI_Cdb_Inquiry_t cdb;
    Char buf[260];
    memset(bufp,0,maxbytes);
    if (maxbytes > 255)
        maxbytes = 255;
    memset(&cdb,0,sizeof(cdb));
    cdb.CommandCode = SCSI_Cmd_Inquiry;
    cdb.Lun = Lun;
    cdb.Evpd = evpd;
    cdb.PageCode = page_code;
    cdb.AllocationLength = 0xFF;

    LastError = DoCommand(&cdb,6,buf,maxbytes,Scsi_Dir_In,
        Attributes.ShortTimeout);
    memcpy(bufp,buf,maxbytes);

    return Lasterror;
}
```

我们在这里使用的方法很简单：首先创建一个 SCSI CDB，然后把它和数据缓冲区的信息传递到 DocCommand 函数。这样，我们就可以很容易地为其他 SCSI 命令增加函数功能。

### 程序清单 13-10 ScsiDiskDevice 类定义

```
//SCSI 直接访问设备类
class ScsiDiskDevice : public ScsiBaseDevice
{
public:
    ScsiDiskDevice();
    ~ScsiDiskDevice();

    ScsiError_t ReadCapacity(DWORD *blklast, DWORD *blksize);
    ScsiError_t ReadSector(DWORD sectnum, void *bufp,
                           DWORD maxbytes, DWORD *bytesread = NULL);
    ScsiError_t LockUnlock(int fLock);
    ScsiError_t Eject();

};

};
```

我们在此仅仅增加了有限的几个针对特定设备类型的函数。通过 C++ 的内在机制可以为 ScsiDiskDevice 对象在 ScsiBaseDevice 类中构造功能函数。

很幸运的是，层次越高的抽象其代码就越简单。

## ■ SCSI Snooper 应用程序的框架

SCSI Snooper 使用 Microsoft 的基本类库（MFC）来构造应用程序的基本框架和用户接口的基本元素。在结构上，Snooper 程序是一个对话框序列，在对话框上包含了能够激活所选命令的控件。

不过应用程序中的大部分代码都是通过 Microsoft Development Studio 创建的，因为其容量太大，所以就不在这里赘述了。不过所有有关的内容都包括在了本书附带的光盘中。我们可以按照自己的要求自由地学习，还可以进行修改以适应自己的要求。

我们可以使用所包含的 Make 程序的描述文件来编写应用程序或创建 Visual C++ 工程文件。ASPI 类库文件保存在一个单独的子目录下。一定要保证该子目录在搜索目录中，否则的话编译器就不能定位类库的头文件。

## ■ SCSI Snooper 应用程序结构

在应用程序类 CSnooperApp 中包括一个执行 ScsiInterface 对象的指针变量。而主对话框类就是使用该指针来定位主机适配器的，并且还为主机适配器创建图标按钮。在 CSnooperDlg 初始化例程中，将会读取设备列表并映射到一个专门的数组中。该数组可以通过索引号来创建和跟踪设备的图标按钮。

主机适配器所使用的信息对话框类就是 CAdapterDlg。CSnooperDlg 对象将会向应用程序中的 ScsiInterface 对象发送一个指针，通过这个指针该对象可以用来显示所选的主机适配器的相关信息。

设备信息对话框使用的是 CDeviceDlg。它将会从调用它的 CSnooperDlg 对象那里接收一个指针给 ScsiDevice。通过这个指针，它可以取回设备的名称和标志字符串并显示出来。

CMoreinfoDlg 用来负责扩展的信息对话框。它将从其 CDeviceDlg 对象处接收一个指针给 ScsiDevice 对象。所有的 SCSI 的特征将会以对话框的形式显示出来，如果设备支持这种方式，将会对其进行检测。

CActionDlg 类要比其他的类复杂得多。通过从其 CDeviceDlg 对象处接收的 ScsiDevice 对象的指针，它可以确定 SCSI 外围设备的类型，并在一个列表框中显示支持的合适的活动。一旦该活动执行完毕，将会显示出错信息，或直接输出在一个文本域中。

通过 SCSI Snooper 检验设备的时候，很可能会遇到这样的情况，就是当我们对设备进行检验的时候，有些设备根本没有响应。举个例子说，在执行 Read Sense 命令的时候，当我们循环打开一个设备时，就会出现这种情况，然后返回 Unit Attention condition。但是，有些设备可能会忽略这种情况。

我们可以自己把这个应用程序进行扩展，把它作为一个学习工具。希望大家不要害怕试验和失败，要努力钻研和进取。

## 附录 A 缩写词汇表

在有关 SCSI 的讨论中，我们经常会遇到各种各样奇怪的术语，也会碰到千奇百怪且容易混淆的缩写词汇。下表中列出了比较常用的缩写词汇以及它们所代表的确切含义。

### 通用词汇

ANSI	美国国家标准协会 (American National Standards Institute)。该协会负责制定并维护工业的通用标准。
ASC	附加检测码 (additional Sense Code)。附加检测码用来确定产生特定错误的根源。
ASCD	附加检测码限定 (Additional Sense Code Qualifier)。附加检测码限定用来确定特定错误的详细信息。
CDB	命令描述块 (Command Descriptor Block)。命令描述块用来向 SCSI 设备传递命令和参数。
CRC	循环冗余校验 (Cyclic Redundancy Check)。循环冗余校验用来对传输的数据进行错误校验。
CCS	通用命令组 (Common Command Set)。通用命令组为直接存储设备的标准命令组。
DLL	动态链接库 (Dynamic Link Library)。在 Windows 应用程序中，动态链接库用来保存共享代码或数据。
LUN	逻辑单元号 (Logical Unit Number)。逻辑单元号用来确定目标设备上的子单元。
LVD	低电压差动线圈 (Low Voltage Differential)。低电压差动线圈是专门为高传输速率设计的交变线圈。
SCAM	SCSI 自动配置 (SCSI Configured AutoMagnically)。该协议是为即插即用 SCSI 配置所定义的协议。

### SCSI-2 定义

#### SCSI-2 协议

PH1	SSA 的物理层 1。该层用来定义串行存储结构的物理层。
S2P	SSA 的 SCSI-2 协议。该协议定义了在串行存储结构上的传输规范。
TL1	SSA 传输层 1。该层定义了串行存储结构物理层之上的传输协议。

## ■ SCSI-3 定义

### ■ SCSI-3 结构

- SAM      SCSI-3 结构模型  
SAM2     SCSI-3 第二代的结构模型

### ■ SCSI-3 命令组

- MMC     多媒体指令 (Multi-Media Command)。专门为多媒体设备如 CD-ROM 所定义的指令。  
SBC     SCSI-3 块指令 (SCSI-3 Block Command)。为块寻址的直接存储设备所定义的指令。  
SCC     SCSI-3 控制器指令 (SCSI-3 Controller Command)。专门为 RAID 设备所定义的指令。  
SES     SCSI-3 附件服务 (SCSI-3 Enclosure Services)。为附件的使用所定义的指令。  
SMC     SCSI-3 介质转换器指令 (SCSI-3 Medium Changer Command)。专门为介质转换器如投币式自动电唱机所定义的指令。  
SPC     SCSI-3 基本指令 (SCSI-3 Primary Command)，适合于所有的 SCSI-3 设备的基本指令。  
SSC     SCSI-3 流指令 (SCSI-3 Stream Command)。专门为流定位的顺序存储设备所定义的指令。

### ■ SCSI-3 协议

- FPC     光纤通道管路协议 (Fibre Channel Protocol)。该协议定义了通过光纤通道接口的 SCSI 传输。  
PH2     SSA 物理层 2 (SSA Physical Level 2)。该物理层定义了串行存储结构的物理层的结构。  
S3P     SSA SCSI-3 协议 (SSA SCSI-3 Protocol)。该协议定义了在串行存储结构上的 SCSI-3 传输规范。  
SBP     SCSI-3 串行总线协议 (SCSI-3 Serial Bus Protocol)。该协议定义了 IEEE1394 接口上的 SCSI-3 传输规范。  
SBP-2    串行总线协议第二代 (Serial Bus Protocol second generation)。该协议定义了 IEEE1394 接口上的通用的传输规范。  
SIP     SCSI-3 互锁协议 (SCSI-3 Interlocked Protocol)。

---

SPI	SCSI-3 物理接口 (SCSI-3 Physical Interface)。
SPI-2	SCSI-3 物理互连-2 (SCSI-3Physical Interconnect-2)。描述的是 SCSI-3 与 SPI、Fast-20 和 SIP 之间的连接。
SSA	串行存储结构 (Serial Storage Architecture)。
STS	能过 SBP-2 的 SCSI 传输 (SCSI Transport via SBP-2)。定义了通过 SBP-2 的 SCSI-3 传输规范。
TL2	SSA 传输层 2 (SSA Transport Level 2)，该层定义了在串行存储结构的物理层上的传输协议。

## SCSI 软件接口

ASPI	高级 SCSI 编程接口 (Advanced SCSI Programming Interface)。
CAM	SCSI-2 通用存储方法 (SCSI-2 Common Access Method)。
CAM-3	SCSI-3 通用存储方法 (SCSI-3 Common Access Method)。
SRB	SCSI 请求块 (SCSI Request Block)。在 ASPI 编程中所使用的命令结构。

## 附录 B SCSI 资源

我们可以获得很多有关 SCSI 的资料，寻找这些资料也有一定的技巧。在此列出了一些电子或书籍形式的资源列表。以这些资料作为学习 SCSI 的起始点，逐步学习其他更多的有关知识。令我们欣慰的是，随着 SCSI 的飞速发展，有关 SCSI 的资源也在不断地推陈出新。

### 书籍

有关 SCSI 的书籍并不多，而且这些为数不多的书籍大多还是关于 SCSI 硬件的，这样一来，描述 SCSI 编程的书就少得可怜了。下面所列出的书籍中有的已经不再出版，但是我们还可以从一些固定的书商或网上书店购买到。

*ANSI SCSI-2 Standard*

Global Engineering Documents  
15 Inverness Way East  
Englewood, CO 80112  
(800) 854-7179

*The SCSI Bus and IDE Interface: Protocols, Applications and Programming*  
*Friedhelm Schmidt*

Addison Wesley Longman  
ISBN 0-201-42284-0

*The Indispensable PC Hardware Book: Your Hardware Questions*

*Answered, Second Edition*

Hans-Peter Messmer  
Addison Wesley Longman  
ISBN 0-201-87697-3

*The Book of SCSI*

Peter M. Ridge  
No Starch Press  
ISBN 1-886411-02-6

*The SCSI Encyclopedia*

ENDL Publishing  
14426 Black Walnut Ct.  
Saratoga, CA 95090  
(408) 867-6642

***The SCSI Bench Reference***

ENDL Publishing  
14426 Black Wahut Ct.  
Saratoga, CA 95090  
(408) 867-6642

***What Is SCSI? Understanding the Small Computer Systems Interface***

Prentice-Hall  
ISBN 0-13-796855-8

***In-Depth Exploration of SCSI***

Solution Technology, SCSI Publications  
P.O. Box 104  
Boulder Creek, CA 95006  
(408) 338-4285

 **杂志和日报****“The Advanced SCSI Programming Interface”**

Brian Sawert  
*Dr. Dobb's Journal*, March 1994, pages 154–158

**“The SCSI Bus, Part 1”**

L. Brett Glass  
*Byte Magazine*, February 1990, pages 267–274

**“The SCSI Bus, Part 2”**

L. Brett Glass  
*Byte Magazine*, March 1990, pages 291–298

**“More Than Just Fast”**

Rick Grehn  
*Byte Magazine*, December 1990, pages 361–369

 **在线信息**

我们可以在Internet上获得当前有关SCSI标准的最好的信息资源。Web站点为我们提供了交互式的文档的浏览，FTP站点则提供了当前流行的规范草案。下面我们就列出一些对于SCSI编程者很有用的资源站点。

## □ Web 站点

### **ANSI X3T10公司主页 <http://www.symbios.com/x3t10/>**

该站点是X3T10工作委员会的站点。我们可以从这个站点找到有关SCSI-3的一些最新消息。

### **ANSI X3T10委员会草案 <http://www.symbios.com/x3t10/drafts.htm>**

该链接来自于X3T10公司的主页，我们可以从这里得到有关SCSI的最新的标准草案和文档。

### **SCSI-2规范说明书（Draft X3T9.2 Rev 10L） <http://scitexdv.com/SCSI2/>**

该站点上有关于SCSI-2规范说明书草案的超文本文件。

### **SCSI常见问题解答（FAQ）**

### **<http://www.cis.ohio-state.edu/hypertext/faq/usenet/sisi-faq/top.htm>**

该站点（由Gary Field维护）包括了来自comp.periph.scsi新闻组的常见问题解答的文档信息。

### **Adaptec公司主页 <http://www.adaptec.com>**

这是Adaptec公司的主页，Adaptec公司是SCSI控制器和设备的生产厂商。Adaptec公司也涉足Trantor和未来领域，所以我们也可以从该网站获得相关支持。

### **Adaptec开发商信息 <http://www.adaptec.com/support/dev.html>**

该站点发布了有关SCSI开发商的信息，另外还公布了一些有关ASPI的信息和文档。

### **Symbios Logic公司主页 <http://www.symbios.com>**

这是Symbios Logic公司（SCSI设备和控制器的生产厂商）的主页。Symbios公司是由NCR微电子公司（SCSI工业的先驱）发展而来的。Symbios公司负责维护X3T10委员会的主页。

### **Symbios Logic公司章程 <http://www.symbios.com/articles/articles.htm>**

我们可以把该主页作为SCSI信息的起始点来进行查询。

### **Ancot公司主页 <http://www.ancot.com/>**

Ancot公司是一个SCSI设备和测试仪器的制造商，该主页上有关于SCSI技术的讨论并有到其他的SCSI的资源信息站点的链接。此外该主页还提供了一个小册子，我们可以在线按照顺序学习有关SCSI的基本知识。

### 西部数据 (Western Digital) 公司主页 <http://www.wdc.com/>

西部数据公司是一个生产SCSI设备和主机适配器的制造商。在该公司的主页上有一些有用的标准检查程序、ASPI实用工具和一些测试工具。

### Linux并行端口主页 <http://www.torque.net/linux-pp.html>

如果需要了解Linux并行端口的信息，可以到该主页看一看。另外该主页还包括了与并行端口SCSI适配器和Iomega并行端口Zip驱动器的信息的链接。

### Linux文献项目主页 <http://sunsite.unc.edu/mdw/linux.html>

该站点是一个有关Linux文献项目的主页，我们可以从该站点获得很多有关支持SCSI的Linux设备驱动器的资源和常用问题解答。

## □ Usenet 新闻组

新闻组是一个讨论SCSI相关主题的大论坛。在此，用户可以直接提问，也可以回答别人的问题，通常一个问题会引起很热烈的讨论。

**Comp.periph.scsi**——一个有关SCSI信息的权威的新闻组

**Comp.sys.ibm.pc.hardware.chips**——一个偏重于讨论SCSI控制器的新闻组

**Comp.os.ms-windows.programmer.nt.kernel-mode**——Windows NT环境下有关SCSI支持的问题与解答。

**Comp.os.linux.hardware**——Linux环境下有关SCSI支持的问题与解答。

## □ ftp 站点

### I/O标准委员会Ftp服务器 <ftp://ftp.symbios.com/pub/standards/io/>

该站点是由Symbios公司维护，内容包括SCSI标准的草案、应用工具、详细规范说明书和新闻组的科研报告集。我们还可以发现与SCSI-2、SCSI-3、即插即用SCSI和其他有关SCSI主题的技术文档。

### Tulane大学SCSI档案 <ftp://ftp.cs.tulane.edu/pub/scsi/>

该站点是SCSI BBS的一个镜像系统。其中包括SCSI的标准草案、工作论文和其他的一些文档。

## Linux Ftp站点

- <ftp://sunsite.unc.edu/pub/linux/>
- <ftp://tsx-11.mit.edu/pub/linux>
- <ftp://ftp.redhat.com/pub/>

上述站点是有关Linux分配和文献的常用站点。

## ¤ BBS (Bulletin Board Systems)

### SCSI BBS (719) 574-0424

该BBS系统是一个与SCSI相关的权威机构，是由X3T10工作委员会的成员负责维护。

## ■ 制造商联系信息

下表中列出了本文中提到的制造商的联系信息。

### 全球工程技术文献 (Global Engineering Documents)

15 Inverness Way East

Englewood, CO 80112

(800) -7179

全球工程技术负责出版和经销ANSI SCSI-2规范说明书。文献中记录了400页的详细内容，但是对于重要的开发工作来说它们并不是必不可少的。

### Adaptec公司

691 South Milpitas Boulevard

Milpitas, California 95035

Adaptec公司专门生产SCSI控制器、主机适配器和其他的SCSI设备。可以通过与他们相关的开发部门联系，购买ASPI的软件开发包。

### Symbios Logic公司

Western Sales Division

1731 Technology Drive, Suite 610

(408) 441-1080

Symbios公司专门生产SCSI控制芯片、主机适配器和其他的SCSI设备。公司还专门为他们的产品的初级编程配发了SCRIPTS编译器。

### Iomega公司

West Iomega Way

Roy, Utah 84067

(800) 778-1000

Iomega公司生产通用的Zip驱动器。Zip驱动器分为SCSI和并行端口两种版本，对于我们开发和测试SCSI代码非常方便实用。并行端口模式的Zip驱动器是作为一种ASPI兼容设备驱动器而产生的。

## 附录 C 在 Windows NT 环境下安装 ASPI32 服务

Windows NT 的缺省安装模式并不包括对 ASPI32 的支持。在 Windows NT 环境中对 SCSI 的支持与 Windows 95 不同，它并不像 Windows 95 一样需要一个 ASPI 层。除非有一个现成的 ASPI 服务的安装程序，否则我们必须自己亲自来完成。

如果我们购买了 Adaptec 公司的 EZ-SCSI 软件，那么可以通过安装该软件来安装支持 ASPI32 的必要的库文件和驱动程序。该软件中还包括 ASPI 开发程序包。

如果我们没有购买 EZ-SCSI 软件，那么也可以轻松地通过手工配置来完成对 ASPI32 的支持。其中所需要的文件可以在 Adaptec 公司的主页 [www.adaptec.com](http://www.adaptec.com) 上下载。

要想从 Adaptec 公司主页下载 Windows 32 ASPI 驱动程序和 DLL（动态链接库），可能要费一番力气。在 Windows NT 环境下安装 ASPI 服务共需要四个文件。

### ASPI32 支持文件

<b>WNASPI32.DLL</b>	32 位 ASPI 管理器
<b>ASPI32.SYS</b>	ASPI 内核模式驱动程序
<b>WINASPI.DLL</b>	16 位 ASPI 管理器
<b>WOWPOST.EXE</b>	在 16 位的应用中对回收信号的支持文件

将 WNASPI32.DLL 拷贝到“\WINNT\SYSTEM32”目录中；将 ASPI32.SYS 拷贝到“\WINNT\SYSTEM32\DIVERS”目录中。其他两个文件拷贝到“\WINNT\SYSTEM”目录中来支持 16 位的应用程序。

通过编辑注册表来安装 ASPI32.SYS 驱动程序。运行注册表编辑器，在“\HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services”分支下新建一个名为 ASPI32 的子键。在这个新的子键下增加如下的三个键值。需要注意的是，这三个键值都是 REGDWORD 类型的数据。

### ASPI32 注册表入口点

ErrorControl	(REGDWORD) 1
Start	(REGDWORD) 2
Type	(REGDWORD) 1

退出注册表编辑器并重新启动系统。

系统重新启动之后，在控制面板中单击“设备”，可以看到其中已经列出了 ASPI32 图标，系统重起的时候已经对它自动进行了配置。

就是它了！现在我们的 Windows NT 系统可以运行 ASPI32 的应用程序了。

## 附录 D 本书附带光盘的内容

在本书的附带光盘中包括一些例程代码、工具和一些对深入了解 SCSI 很有帮助的文献资料。需要注意的是，SCSI 是一个发展很快的技术，所以当你看到本书时候，本书附带光盘中的某些内容可能已经过时，尽管如此，这些内容对于我们解决一些小问题还是非常有用的。

### 例程代码

在附带光盘的“SampCode”目录中有关于 SCSI Snooper 应用程序的例程代码和相关的源代码。我们不仅可以看到编译后的程序，还可以找到这些程序的源代码。

我们还可以在“SampCode”目录中找到 TSPI 目标模式 API 库文件的源代码。

#### SCSRIPTS 例程代码

SCSRIPTS 例程代码中包括在 SCRIPTS 一章中提到的例程。它还包括一个关于 SCSI 编目程序的描述文件。该描述文件是与 Borland 公司的编译工具相适应的，所以一般都是随着他们的 C++ 编译器而销售的。该描述文件假设用户使用的是 Borland C++ 和 Turbo 汇编程序，如果用户想要应用到其他类型的编译器上，要对源代码进行一些相应的改动。

#### SCSI Snooper 应用程序

SCSI Snooper 应用程序的代码包括一个在微软的 Visual C++ 开发环境中产生的程序描述文件。我们可以直接使用该描述文件，也可以通过该描述文件产生一个项目文件。编译后的应用程序和所需要的库文件都保存在“Program”子目录中。

#### TSPI 目标模式 SCSI 编程接口

TSPI 目标模式的编程接口的源代码在“TSPI”子目录中。另外在该子目录中还可以找到在第九章中的例程代码。

## SCSI 规范说明书

在“SCSISpec”目录中有 SCSI-1、SCSI-2 和 SCSI-3 规范草案。比较古老的规范说明书是纯文本或 Adobe Acrobat 的 PDF 模式的文件；而 SCSI-3 的规范草案是以 PDF 模式和 PostScript 语言模式出现的。我们可以通过 Adobe 公司的 Acrobat Reader 来阅读 PDF 模式的文件。其“Acrobat”目录中有该阅读器的程序文件。

需要注意的是，SCSI-3 的规范说明书的很多地方仍然在不断地进行修正。我们可以从 T10 委员会的站点“<http://www.symbios.com/x3T10>”中获得 SCSI-3 规范的最新版本。

另外，这里还有一个超文本版本的 SCSI-2 的规范说明书。其实也就是 Gary Bartlett 对 SCSI-2 标准草案的 HTML 的改写本，并发布在“<http://scitexdv.com/SCSI2>”站点上。我们可以通过浏览器来直接查阅本书附带光盘中的“SCSISpec\HTML\index.html”内容。如果用户的计算机中安装了 Adobe Acrobat Reader 插件，还可以通过浏览器直接浏览有关 SCSI-3 的 PDF 格式的模范说明书。

## SCSI 常见问题解答

任何一个有关 SCSI 的书籍都会参考 Gary Fiary 的 SCSI 常见问题解答（FAQ）。在本书附带光盘的“SCSIFaq”目录中可以找到这些内容。comp.periph.scsi 的 usenet 新闻组每月定时更新 FAQ。

## Symbios SCRIPTS 支持

在“Symbios”目录中包括使用 SCPIPTS 语言的一些工具和例程代码。在“Tools”子目录中有一个 ANSM 编译器，还有一个 NVPCI 调试程序。

在“8xxdev”目录中包括演示如何使用 SCRIPTS 语言编写 SCSI 启动器代码的例程源代码。这个目录中的其他源文件是一些用户自己开发软件时可能会用到的辅助工具。

在“8xxtarg”目录中包括了演示如何使用 SCRIPTS 语言编写 SCSI 目标器应用程序的例程源代码。这些例程可以创建一个虚拟的 SCSI 磁盘驱动程序，通过这个虚拟的驱动器程序可以测试其他的应用程序。

## Linux SCSI 文献

在本书附带光盘的“Linux”目录中，可以找到一些有关 SCSI 支持和 SCSI 编程的 Linux

HOWTO 文档。有关 SCSI 编辑的 HOWTO 文档包括一个应用于 Linux SCSI pass-through 特性的应用程序。

另外还有一个有关 Linux2.0.30 的内核源代码的压缩文件。我们可以使用 Gnu Zip 来解压该文件，也可以使用 Windows 中的解压工具例如 Winzip 来解压缩。解压缩后在“Linux/drivers/scsi”目录中有各种各样的 SCSI 主机适配器的源代码，另外该目录中还包括 SCSI pass-through 驱动器的源代码（sg.c）。在解压缩后的“Linux/include/scsi”目录中是其他的 SCSI 支持文件。