第 8 章 硬件管理

尽管通过玩玩 *scull* 和类似的一些玩具程序来熟悉 Linux 设备驱动程序的软件接口显得很轻松，但是测试*真正的*设备还是要涉及硬件。驱动程序是软件概念和硬件设备间的一个抽象层；因此，我们两者都要谈谈。到目前为止，我们已经详细讨论了软件上的一些概念；本章将接着介绍驱动程序是如何在保证可移植的前提下访问 I/O 端口和 I/O 地址空间的。

和前面一样，我的示例代码也不针对特定的设备。但是我们不能再用象 *scull* 这样的基于内存的设备。本章的例子使用并口设备来讲解 I/O 指令，使用字符模式 VGA 卡的显示缓冲区来讲解基于内存映射的 I/O。

这里我选择并口的原因是它能提供位信息的输入输出。写向设备的数据位出现在输出引脚上，而输入引脚的电压值可以由处理器控制。实际上，你可以将 LED 连到并口上来观察 I/O 操作的结果。并口比串口更容易编程，而且几乎每台计算机(甚至 Alpha)都和 PC 机一样提供了并口。

至于基于内存映射的 I/O，字符模式的 VGA 是标准的内存映射设备，而每台计算机都有 VGA 兼容的字符模式。遗憾的是，不是每台 Alpha 都有 VGA 显示卡，Sparc 就肯定没有，所以我们的与 VGA 有关的代码就不象并口的例子那样可以移植了。并且，为了运行示例程序，你要切换到字符模式下，但这并不是个太大的限制。用 VGA 显示卡上上内存做实验可能造成的最大问题是示例驱动程序不可避免地会破坏前台的虚拟控制台。

使用 I/O 端口

I/O端口有点类似内存位置：可以用和访问内存芯片相同的电信号对它进行读写。但这两者实际上并不一样；断口操作是直接对外设进行的，和内存相比更不灵活。而且，有 8 位的端口，也有 16 位的端口和 32 位的端口，不能相互混淆[*]。

因此，C 语言程序必须调用不同的函数来访问大小不同的端口。Linux 内核头文件中(就在与体系结构相关的头文件**<asm/io.h>**中)定义了如下一些内联函数。

注意 从现在开始，如果我只使用 unsigned 而不进一步指定类型信息的话，那我是在谈及一个与体系结构相关的定义，此时就不必关心它的准确特性。这些函数基本是可移植的，因为编译器在赋值时会自动进行强制类型转换(cast)－类型被强制转换成 unsigned 类型防止了编译时出现的警告信息。只要程序员赋值时注意避免溢出，这种强制类型转换就不会丢失信息。在本章剩余部分我会一直保持这种“不完整的类型定义”的方式。

---

[*] 实际上，有时I/O端口*是*和内存一样对待的，(例如)你可以将 2 个 8 位的操作合并成一个 16 位的操作。例如，PC的显示卡就可以，但一般来说不能认为一定具有这种特性。

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

按字节(8 位宽度)读写端口。**port** 参数在一些平台上定义为 **unsigned long**，而在另一些平台上定义为 **unsigned short**。不同平台上 *inb* 返回值的类型也不相同。

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);

这些函数用于访问 16 位端口(“字宽度”)；Linux 的 M68k 版本不提供，因为该处理器只支持字节宽度的 I/O 操作，不支持字宽度或更大宽度的操作。

unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

这些函数用于访问 32 位端口。**doubleword** 参数根据不同平台定义成 **unsigned long** 类型或 **unsigned int** 类型。

除了每次只能传输一个数据单位的 in 和 out 操作，绝大多数处理器还提供了能传输多个字节，字或 long 类型数据的特殊指令。这些指令就是所谓的“串指令”，将在本章稍后处的“串操作”一节中介绍。

注意这里没有定义 64 位的 I/O 操作。即使在 64 位的体系结构上，I/O 端口也只使用 32 位的数据通路。

上面这些函数主要是提供给设备驱动程序使用的，但它们也可以在用户空间使用(预处理定义和内联声明没有用**#ifdef __KERNEL__**保护)。但是如果要在用户空间代码中使用 inb 及其相关函数，必须满足下面这些条件：

- 编译该程序时必须带-O 选项来强制内联函数的展开。
- 必须用 *ioperm* 或 *iopl* 来获取对端口进行 I/O 操作的权限。*ioperm* 用来获取对指定端口的操作权限，而 *iopl* 用来获取对整个 I/O 空间的操作权限。这两个函数都是 Intel 平台提供的。
- 必须以 root 身份运行该程序才能调用 *ioperm*。或者，该程序的某个祖先已经以 root 身份获取了对端口操作的权限。

示例程序 *misc-progs/inp.c* 和 *misc-progs/outp.c* 是在用户空间通过命令行读写 8 位端口的一个小工具。我已经在我的 PC 上成功运行过。但由于缺少 *ioperm* 函数的原因，它们不能在其它平台上运行。如果你想冒险，可以将它们设置上 SUID 位，那么不用显式地获取特权就可以使用硬件了。

平台相关性

如果你考虑移植问题，你会发现 I/O 指令是所有计算机指令中与处理器最密切相关的部分。因此，大部分与 I/O 端口有关的源代码都与平台相关。

Linux 系统，尽管是可移植的，但处理器的特性不是完全透明的。大部分硬件驱动程序在平台间是不可移植的，而且在同一模块中驱动程序所涵盖的平台一般不超过两到三种。

回头看看前面的函数列表，你可以看到一处不兼容的地方，数据类型，参数类型根据各平台体系结构上的不同要相应地使用不同的数据类型。例如，port 参数在 x86 平台(处理器只支持 64KB 字节的 I/O 空间)上定义为 **unsigned short**，但在 Alpha 平台上定义为 **unsigned long**。Alpha 平台上端口是和内存在同一地址空间内的一些特定区域，它在设计上就不存在 I/O 地址空间，它的端口是被当作为不能被高速缓冲的内存区域。

I/O 数据类型是核心中一个仍需要整理的部分，尽管现在能正常工作。对这些不够明确的类型最好的解决方法是定义一个与体系结构有关的 **port_t** 数据类型而对数据项则使用 **u8**，**u16** 和 **u32** 这些数据类型(参见第 10 章“*合理使用数据类型*”的“分配确定的空间大小给数据项”一节)。但还没人真正注意过这个问题，因为这个问题主要是个书写技巧上的问题。

其他一些与平台相关的问题来源于处理器结构根本上的差异，因此也无法避免。因为本书我假定你不会在不了解底层硬件的情况下为特定的系统写驱动程序，所以我不会详细讨论这些差异。下面是可以支持的体系结构的总结：

X86

该体系结构支持本章提到的所有函数。

Alpha

支持前面所有函数，但不同的 Alpha 平台上端口 I/O 操作的实现也有不同。串操作是用 C 语言实现的，在文件 *arch/alpha/lib/io.c* 中定义。但遗憾的是，2.0 系列的核心在 2.0.29 版之前还只开放 word 和 long 类型数据的串操作；因此，模块中无法使用 *insb* 和 *outsb* 函数。在 2.0.30 和 2.1.3 版中这个问题已经改正过来了。

Sparc

Sparc 不提供特殊的 I/O 指令。I/O 空间是通过内存映射获得，在页表中设置了标志。在头文件中 inb 和其它函数都被定义为空函数来避免第一次把驱动程序移植到 Sparc 体系结构上时编译器为此报告错误。

M68k

只支持 *inb*，*outb* 和它们相应的暂停式版本(见下节)。68000 上没有定义串操作，也没定义 *readb*，*writeb* 和相关函数。

Mips

支持前面所有函数。但串操作是用汇编语言写的紧凑循环(tight loop)实现的，因为 Mips 处理器不提供机器一级的串 I/O 操作。

PowerPC

除了串 I/O 操作，其它函数都能支持。

感兴趣的读者可以从 *io.h* 文件获得更多信息，除了我在本章介绍的函数，一些与体系结构相关的函数有时也由该文件定义。

值得提及的是，Alpha 处理器并不为端口提供不同的地址空间，虽然 AXP 机器一般带有 ISA 和 PCI 插槽，而且这两种总线都为内存和 I/O 操作提供了不同的信号线。利用特别的接口芯片将指定的内存地址引用转换成对 I/O 端口的访问，基于 Alpha 的 PC 可以实现一个与 Intel 系列兼容的 I/O 抽象层。

Alpha 平台上的 I/O 操作在“Alpha 参考手册”中详细介绍了，该手册可从 DEC 公司免费获得，手册详尽地阐述了 I/O 问题，介绍了 AXP 处理器是如何将虚拟地址空间划分为“类内存”和“不类内存”区域；后者用于内存映射的 I/O。

暂停式(pausing)I/O

一些平台－特别是在 i386 上－当处理器和总线间数据得传输太快是会带来问题。问题是源于相对 ISA 总线处理器的时钟频率太快了，当设备卡太慢时，这个问题就容易暴露出来；解决该问题的方法是，如果后面又跟着一条 I/O 指令，就在该条 I/O 指令后添加一小段延迟。如果你的设备会丢失数据，或者你担心它会丢失数据，你可以用暂停式的 I/O 操作取代通常的 I/O 操作。暂停式 I/O 函数很象前面列出的那些 I/O 函数，但它们的名字都以 *_p* 结尾；例如 *inb_p*，*outb_p* 等等。对所有支持的体系结构，如果定义了不暂停的 I/O 函数，那么也会定义相应的暂停式的 I/O 函数，虽然有些平台上它们会被扩展成相同的代码。

如果你想在驱动程序中显式地插入一小段延迟(小于用 *udelay* 可获得的延迟)，你可以显式地使用 **SLOW_DOWN_IO** 语句。由这个宏扩展成的一段指令只是用于延迟，不做任何其他的工作。你可以将这条语句插入到源代码中的一些关键位置上。实际上，**SLOW_DOWN_IO** 所执行的代码就是 *outb_p* 被扩展后相对于 *outb* 所增加的代码。

SLOW_DOWN_IO(和 **_p** 延迟)是否被定义取决于在包含**<asm/io.h>**文件前是否定义过 **SLOW_IO_BY_JUMPING** 和/或 **REALLLY_SLOW_IO**。好在新的硬件已经不要求程序处理这些问题了，所以我不会再讨论暂停问题了。感兴趣的读者可以读读头文件**<asm/io.h>**。但是在写驱动程序的时候，你必须记着 **SLOW_DOWN_IO** 在 Sparc 和 M68k 体系结构上是没有定义的(虽然也定义了象 *outb_p* 这样暂停式的函数调用，在本章前面的“平台相关性”一节中列出了限制条件)。

串操作

Linux 头文件中定义了进行串操作的函数，驱动程序可以使用它们来获得比 C 语言写的循环更好的性能。在 Linux 中，取决与相应的处理器或平台，串操作被实现为一条机器指令，或者是一个紧凑循环，但也可能不提供。

串操作函数的原型如下：

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
从内存地址 **addr** 开始连续读写 **count** 个字节。但只对一个端口 **port** 读写这些数据。

void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
对一个 16 位端口连续读写 16 位数据。

void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
对一个 32 位端口连续读写 32 位数据。

使用并口

我们用并口来测试我们的 I/O 代码，并口很简单，事实上，我很难想出一个比它更简单的接口适配卡。

尽管大部分读者都能取得并口规范，但为了方便读者，在你读下面要介绍的模块之前我先将它总结一下。

并口的基本知识

并口的最小配置(不涉及 ECP 和 EPP 模式)由一些 8 位端口组成。写到输出端口的数据表现为 25 脚插座的输出引脚上的电平信号，而从输入端口读到的数据则是输入引脚当前的逻辑电平值。

在并行通信中使用的电平信号是标准的 TTL 电平：0 伏和 5 伏，逻辑阈值大约为 1.2 伏；端口要求至少满足标准的 TTL LS 电流规格，而现代的大部分并口电流和电压都超过这个规格。

警告 并口插座没有和计算机的内部电路分开，这一点在你想把逻辑门直接连到端口时很有用。但要注意正确连线；否则在测试自己定制的电路时，并口很容易被烧毁。如果你担心会破坏你的主板的话，可以选用可插拔的并行接口。

位规范显示在图 8-1 中。你可以读写 12 个输出位和 5 个输入位，其中一些位在它们的信号通路上输入输出会有逻辑上的反转。唯一一个不与任何信号引脚有联系的位是 2 号端口的第 4 位(0x10)。我们将在第 9 章“*中断处理*”中使用到它。

控制端口：基地址+2
状态端口：基地址+1
数据端口：基地址+0

关键字
输入信号线
输出信号线
位
引脚
反转 非反转
图 8-1：并口的插脚引线

驱动程序样例

我下面要介绍的驱动程序叫做 *short*(Simple Hardware Operations and Raw Tests，简单硬件的操作和原始测试)。它所做的就是读写并口(或其他 I/O 设备)的各种 8 位端口。每个设备节点(拥有唯一的次设备号)访问一个不同的端口。*short* 设备没有任何实际用途；将它独立出来只是为了能用一条指令来从外部对端口进行操作。如果你不太了解 I/O 端口，那么可以通过使用 *short* 来熟悉它；你可以测量它传输数据要消耗的时间或者进行其它的测试。

我建议你把一个 LED 焊到输出引脚上以便观察并口插座。每个 LED 都要串联一个 1KΩ的电阻到一个接地的引脚上。如果将输出端口接到输入端口上，你就可以产生自己的输入供输入端口读取。

如果你想将 LED 焊到 D 型插座上来观察并行数据，我建议你不要使用 9 号和 10 号引脚，因为在运行第 9 章的示例代码时我们要将它们相连。

至于 *short*，它是通过紧凑循环将用户数据拷贝到输出端口来实现写设备*/dev/short0* 的，每次一个字节：

```
while (count--)
    outb(*(ptr++), port);
```

你可以运行下面的命令来使你的 LED 发光：

```
echo -n any string > /dev/short0
```

每个 LED 监控输出端口的一个位。注意只有最后写的字符数据才会在输出引脚上稳定地保持下来，被你观察到。因此，我建议你将*-n* 选项传给 *echo* 程序来制止输出字符后的自动换行。

读端口也是使用类似的函数，只是用 *inb* 代替了 *outb*。为了从并口读取“有意义的”值，你需要将某个硬件连到并口插座的输入引脚上来产生信号。如果没有输入信号，你只会读到始终是相同字节的无穷的输出流。

覆盖所有可能的 I/O 函数，每个 *short* 设备都提供了三个变种：*/dev/short0* 执行的是上面给出的循环，*/dev/short0p* 使用 *outb_p* 和 *inb_p* 来替代前者使用的“较快的”函数，而 */dev/short0s* 使用了串指令。共有四个这样的设备，从 *short0* 到 *short3*，每个都读写一个端口。

这四个端口是连续的。

在 Alpha 上编译时，由于不提供 *insb* 和 *outb*，*short0s* 设备就和 *short0* 一样了。

虽然 *short* 不能进行“真正的”硬件控制，但它可以作为一个对不同的指令计时的有趣的测试平台，并且可以作为一个学习如何进行“真正的”硬件控制的开始。任何对写驱动程序有兴趣的人肯定拥有更多更有趣的设备，但老的傻瓜式的并口还是能作些有用的工作的－我自己就是在早上打开收音机后用它来为我准备咖啡的。

访问设备卡上的内存

上一章介绍了分配 RAM 内存的所有各种方法；现在我们要介绍计算机能提供的另一种内存：扩展卡上的内存。外设也有内存。显卡有帧缓冲区，视频捕捉卡用来存放捕捉到的数据，而网卡将接受到的数据包存放在内存区域中；此外，大多数设备卡上还有卡上 ROM，存放着系统启动时处理器要执行的代码。所有这些都是“内存”，因为处理器通过访存指令来对它们进行读写。这里我只讨论 ISA 和 PCI 设备，因为现在它们最常用。

在标准的 x86 机器上共有三种通用的设备内存：640KB 到 1MB 地址范围内的 ISA 内存，14MB 到 16MB 地址范围内的 ISA 内存，和在物理地址空间之上的 PCI 内存。上面这些用到的地址都是物理地址，是在计算机的地址总线上的跑的数值，与程序代码中所使用的虚拟地址没有任何关系(参见第 7 章“*获取内存*”的“vmalloc 和相关函数”一节)。I/O 内存使用这些物理位置主要是出于历史的原因，下面介绍这三种内存区域时会作出解释。

不幸的是(或者，幸运的是，如果你更倾向于好的体系结构设计而不是容易移植的话)。不是所有的 Linux 平台都支持 ISA 和 PCI；本节只限于讨论支持 ISA 和 PCI 的 Linux 平台。

1M 地址空间之下的 ISA 内存

在第 2 章“*编写和运行模块*”的“ISA 内存”一节中我已经介绍过一种“容易的”(但有些毛病的)访问这种内存的方法，在那里我使用了存放物理地址的指针来正确地指向所申请的 I/O 内存。尽管这种技术在 x86 平台可行，但却不能移植到其它的 Linux 平台上。使用指针的方法对小的生命期较短的项目是较好的选择，但对作为产品的驱动程序并不推荐。

推荐的 I/O 内存接口是在 linux1.3 版的开发树中被引入内核的，更早的版本并不提供。与 *short* 范例模块一起发布的 *sysdep.h* 头文件为 1.2 版以来的各个内核版本实现了这种新的语义。

新的接口包括一系列宏和函数调用，取代了使用指针来访问 I/O 内存的方法。这些宏和函数是可移植的；这意味着相同的源码可以在不同的体系结构上编译和运行，只要它们拥有类型相同的设备总线。

当你在基于 Intel 处理器的平台上编译代码时，这些宏现在大部分会扩展对指针的操作，但它们的内部实现会在将来被适当地修改。例如，在 Linux 最初从 2.0 版转到 2.1 版时，这

样的修改就发生过，Linus 决定改变虚存的布局。在新的布局下，就不能再以第 2 章中介绍的旧的方式访问 ISA 内存了。

新的 I/O 内存接口由下面这些函数组成：

unsigned readb(address);
unsigned readw(address);
unsigned readl(address);

这些宏用于从 I/O 内存中取得 8 位，16 位和 32 位的数据值。1.2 版的 Linux 不提供。使用宏的优点是对参数的类型不作要求；参数 **address** 在使用前会被强制类型转换，因为这个值“不清楚是整数还是指针，但我们两者都能接受”(见 *asm-alpha/io.h*)。读和写函数都不会检查 address 参数的合法性，因为使用它就是为了能和使用指针一样快(我们已经知道有时它们实际上就是被扩展成指针操作)。

unsigned writeb(unsigned value, address);
unsigned writew(unsigned value, address);
unsigned writel(unsigned value, address);

和前面的函数类似的，这些函数(宏)用于写 8 位，16 位和 32 位的数据项。

memset_io(address,value,count);

当你要对 I/O 调用 *memset* 进行操作时，这个函数可以满足你的需要，并且也保留了 *memset* 原来的语义。

memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);

这些函数用于成块传输 I/O 内存的数据，和 *memcpy_tofs* 的功能有些相似。它们是和上面这些函数一起引入 Linux 中的，1.2 版的 Linux 不提供。与示例代码一起发布的 *sysdep.h* 头文件修正了函数的版本相关性问题，为 1.2 版以后的所有内核提供了这些函数的定义。

和 I/O 端口函数一样的，这些函数在能支持的体系结构间的移植性现在也很有限。一些平台根本不提供这些函数；一些平台上它们是被扩展为指针操作的宏，而在另一些平台上它们则是真正的函数。

象我这种习惯于旧 PC 机的平的(flat)内存模式的人，可能会不愿意为了只是读写“物理地址区域”而麻烦地使用新的一套接口。实际上，要习惯使用某套接口只需要练习练习使用这些函数。当然，没有比看看一个访问 I/O 内存的傻瓜式(silly)模块更好的获得自信的方法了。我下面要给你展示的模块就叫作 *silly*，是“Simple Tool for Unloading and Printing ISA Data，卸载和打印 ISA 数据的简单工具”的简称。

该模块包括四个用了不同的数据传输函数来完成相同任务的设备节点。*silly* 设备是作为 I/O 内存之上的一个窗口，与 */dev/mem* 有些类似。对该设备，你可以读写数据，*lseek* 到一个任意的 I/O 内存地址，也可以将 I/O 内存区域 *mmap* 到你的进程中来(参见第 13 章“*Mmap 和 DMA*”的“mmap 设备操作”一节)。

/dev/sillyb，次设备号为 0，调用 *readb* 和 *writeb* 函数来读写内存空间。下面的代码给出了 *read* 的实现，它将 0xA0000 到 0xFFFFF 的地址范围重映射到设备文件的偏移量从 0 到 0x5FFFF 的位置。*read* 函数将不同的访问模式组织进一个 **switch** 语句中；下面是 *sillyb* 设备的 **case** 分支：

```
case M_8:
    while (count){
        *ptr=readb(add);
        add++;count--;ptr++;
    }
    break;
```

接下来的两种设备是*/dev/sillyw*(次设备号为 1)和*/dev/sillyl*(次设备号为 2)。它们和 */dev/sillyb* 设备差不多，只是分别使用了 16 位和 32 位的函数。下面是 *sillyl* 设备的 *write* 函数的实现，也是 **switch** 语句的一部分。

```
case M_32:
    while (count>=4 ){
        writel(*(u32*)ptr,add);
        add+=4;count-=4;ptr+=4;
    }
    break;
```

最后一种设备是*/dev/sillycp*(次设备号为 3)，该设备用 *memcpy_*io* 函数来执行相同的任务。下面是它的 *read* 函数实现的核心：

```
case M_memcpy:
    memcpy_fromio(ptr, add, count);
    break;
```

1M 地址空间之上的 ISA 内存

有些 ISA 设备卡带有的卡上内存会映射到物理地址空间的 14MB 和 16MB 范围内。这些设备正在逐渐消失，但仍值得介绍一下如何访问它们的内存区域。但本讨论仅适用于 x86 体系结构；我没有这种 ISA 卡在 Alpha 或其它体系结构上相应行为方面的资料。

在使用 80286 处理器的旧时代，物理地址空间的宽度是 20 个位(16MB)，所有的地址线都在 ISA 总线上，几乎没有计算机带的 RAM 会超过 1 兆或 2 兆。那为什么扩展卡不能“偷”点高端的内存地址用做它的缓冲区呢？这种想法并不新鲜；相同的概念已经出现在对 1M 地址空间之下的 ISA 内存的使用上了，后面又会再次用来实现对高端 PCI 内存的使用上。选作 ISA 设备卡内存的地址范围是最顶端的 2M，尽管大多数卡只使用了最顶端的 1M。

只到今天，在一些主板仍能使用旧式的设备卡，即使物理内存超过了 14M。要正确地处理这段内存区域要求你对这段地址范围作些处理，避免将内存地址和总线地址重叠起来。

如果你有一块带有高端内存的 ISA 设备，又不幸地 RAM 小于 16MB，那管理内存就很容易。你的软件在处理时就好象拥有着高端 PCI 缓冲区(参见下一节)，只是更慢些，因为 ISA 内存比较慢。

而如果你拥有的 ISA 设备带了高端内存而你的 RAM 要多于 16MB，那么就有麻烦了。

一种可能就是你的主板不能正确地支持“ISA 空洞”。在这种情况下，除非你拔掉一些内存，否则无法访问卡上内存。另一种可能是，主板能处理 ISA 空洞，你仍要告诉 Linux 内核这种内存的存在，作一些处理以便能访问 RAM 的其它部分(超过 16M 的地址范围)。

你需要作些“黑客”的工作来正确保留高端的 ISA 内存，同时又仍能对其余的 RAM 进行正常访问，要修改的部分是对计算机的物理内存进行映射的部分。这个映射部分是在源文件 *arch/i386/mm/init.c* 的函数 *mem_init* 中实现的。数组 **mem_map** 中存放着与内存页有关的信息；如果某页的 **PG_reserved** 位被设置了，内核就不会对该页进行正常的页面处理(也即，该页被“保留”了，不要碰它)。但驱动程序仍可以使用保留页；640KB 到 1MB 间的地址范围被标记为“保留的”，但仍可以被用作为设备内存。

下面的代码，插在 *mem_init* 函数中，正确地保留了 15MB 和 16MB 间的地址空间：

```
while(start_mem<high_memory){
    if (start_mem>=0xf00000 && start_mem<0x1000000){
        /* keep it reserved, and prevent couting as data */
        reservedpages++; datapages--;
    }
    else
        clear_bit(PG_reserved, &mem_map[MAP_NR(start_mem)].falgs);
    start_mem += PAGE_SIZE;
}
```

最初所有的内存区域都被标记为“保留的”，上面给出的代码行保证了不会将对高端的 I/O 内存去除“保留”标记；原来的代码只有上面给出的循环的 **else** 分支。因为在内核代码之后的每个保留页都算作内核数据，要修改两个计数器 reservedpages 和 datapages 以避免启动时给出不匹配的信息。我的机器，有 32MB 内存，用前面的代码来访问 ISA 空洞，启动是的报告如下：

Memory: 30120/32768k available (512k kernel code, 1408k reserved, 728k data)

我是在自己的安装了 2.0.29 版内核的 Intel 机器(主板支持 ISA 空洞)上测试这段代码的。如果你运行的内核版本并不相同，你可能需要修改一下代码－与内存管理有关的内部数据结构在 2.1 版的内核中有一点改动，而在 1.2 版的内核中则很不一样。要支持旧式的(有时是不良设计的)硬件设备不可避免地必须“黑客”内核代码。

高端 PCI 内存

访问高端 PCI 内存比访问高端 ISA 内存要更容易。PCI 卡上的高端内存*真的*很高－高于任何合理的物理 RAM 地址(至少对未来若干年而言)。

正如第 7 章的“vmalloc 和相关函数”一节讨论到的，访问该内存只要调用 *vremap*(2.1 版的内核是 *ioremap*)。但是如果你希望代码能在不同平台上移植的话，你应该只通过 *readb* 和其它类似函数来访问重映射的内存区域。由于不是所有平台都能将 PCI 缓冲区直接映射到处理器的地址空间，所以必须加上这个限制。

访问字符模式的视频缓冲区

silly 模块解释了如何访问内存 640KB 到 1MB 地址范围内的视频缓冲区，而下面要介绍的更“直观的”演示程序则能帮助你熟悉 *readb* 和 *writeb* 函数。*silly* 模块拥有两个额外的设备节点：*/dev/sillytxt*(次设备号为 4)和*/dev/sillitest*(次设备号为 5)。

警告 这样的设备只能在运行在字符模式下的 VGA 兼容的显示卡上使用；在没有 VGA 显示卡的系统上使用这样的设备，和任何对硬件资源的不受控制的读写一样，具有潜在的破坏性。

第一个设备，*sillytxt*，只是 VGA 字符缓冲区上的一个窗口。与其它的 *silly* 节点不同的是，它可以作为输出重定向的目标和用于覆盖控制台上的显示。这让我们想起*/dev/vcs*，但 *silly* 的实现是不可移植的而且也没有象 *vcs* 那样集成进内核。

最后一个设备只是和你开个玩笑：它将字母从字符屏幕上移走。每向该设备写进一个字符都会导致屏幕上的一个字符落到屏幕的底部。提供这个设备只是为了演示在 I/O 内存如何进行更复杂的操作－可以用同样的代码来操作 VGA 缓冲区或其它内存，比如网络数据包或者帧捕捉卡上的视频数据。

要注意对字符屏幕的任何修改都是易丧失的，并且会干扰内核自己的字符处理。如果你真的需要在应用程序中访问字符缓冲区，那么有更好的完成这个任务的方法：通过 *ncurses* 库或者通过*/dev/vcs* 设备。*vcs* 设备是“Virtual Console Screen，虚拟控制台屏幕”，可以用它来获得每个虚拟控制台的字符缓冲区当前的快照或者进行修改。*vcs* 设备的文档就在它自己的源码中：内核源码树中的 *drivers/char/vc_screen.c* 文件。或者，你可以在最新的 *man-pages* 帮助页发布中找到关于该设备的描述。

快速参考

本章引入下面一些与操纵硬件有关的符号：

#include <asm/io.h>

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

这些函数用于读写 I/O 端口。如果能正确获取访问端口的权限，它们也可以被用户空间的程序调用。不是所有平台都支持所有这些函数，它们与底层的硬件设计有关。

SLOW_DOWN_IO；
unsigned inb_p(unsigned port);
…

有时需要用 **SLOW_DOWN_IO** 语句来处理 x86 平台上低速的 ISA 卡。如果在每个 I/O 操作之后需要一小段延迟，你可以用与上面引入的 6 个函数相应的暂停式版本，它们得在相应的函数名后要加个**_p**。

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);

“串操作”用于优化从输入端口到内存区域(或者反过来)的数据传输。这种传输是通过对同一个端口读写 **count** 次完成的。

unsigned readb(address);
unsigned readw(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy(dest, source, nbytes);

所有这些函数都用于访问 I/O 内存区域－低端的 ISA 内存或者高端的 PCI 缓冲区(在调用 *vremap* 后)。