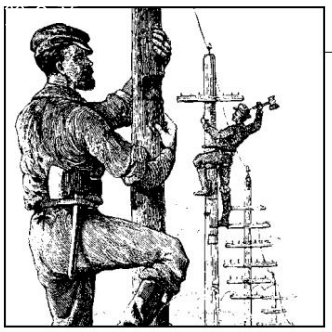

第 9 章

与硬件通信

虽然玩scull和类似玩具是对 Linux 设备驱动程序软件接口的一个很好的介绍,但实现一个真实的设备需要硬件。驱动程序是软件概念和硬件电路之间的抽象层;因此,它需要与他们双方交谈。到目前为止,我们已经研究了软件概念的内部;本章通过向您展示驱动程序如何访问 I/O 端口和 I/O 内存,同时可跨 Linux 平台移植来完成这幅图。

本章延续了尽可能独立于特定硬件的传统。然而,在需要具体示例的地方,我们使用简单的数字 I/O 端口（例如标准 PC 并行端口)来显示 I/O 指令如何工作,并使用普通帧缓冲视频内存来显示内存映射 I/O。

我们选择了简单的数字 I/O,因为它是最简单的输入/输出端口形式。
此外,并行端口实现原始 I/O,并且在大多数计算机中都可用:写入设备的数据位出现在输出引脚上,而输入引脚上的电压电平可由处理器直接访问。实际上,您必须将 LED 或打印机连接到端口才能真正看到数字 I/O 操作的结果,但底层硬件非常易于使用。

I/O 端口和 I/O 内存

每个外围设备都通过写入和读取其寄存器来控制。大多数时候,一个设备有几个寄存器,它们在连续的地址上被访问,要么在内存地址空间,要么在 I/O 地址空间。

在硬件级别,内存区域和 I/O 区域之间没有概念上的区别:它们都是通过在地址上断言电信号来访问的

总线和控制总线（即读取和写入信号）*以及通过读取或写入
到数据总线。

虽然一些 CPU 制造商在其芯片中实现了单一地址空间,但其他制造商认为外围设备不同于内存,因此值得

一个单独的地址空间。一些处理器（尤其是 x86 系列)具有用于 I/O 端口和特殊 CPU 指令的独立读写电线
访问端口。

因为外围设备是为适应外围总线而构建的,而最流行的 I/O
总线是在个人计算机上建模的,即使处理器没有单独的 I/O 端口地址空间,也必须假装读写 I/O 端口。

访问一些外围设备,通常是通过外部芯片组或 CPU 内核中的额外电路。后一种解决方案在微型处理器中很常见,这意味着
用于嵌入式。

出于同样的原因,Linux 在所有计算机上实现了 I/O 端口的概念
它运行的平台,即使在 CPU 实现单个地址的平台上
空间。端口访问的实现有时取决于具体的make
和主机型号（因为不同的型号使用不同的芯片组
将总线事务映射到内存地址空间）。

即使外围总线为 I/O 端口提供单独的地址空间,也不是所有设备
将它们的寄存器映射到 I/O 端口。虽然 I/O 端口的使用在 ISA 外设板上很常见,但大多数 PCI 设备将寄存器映射到内存地址区域。这个 I/O
通常首选内存方法,因为它不需要使用专用处理器指令； CPU 内核更有效地访问内存，

并且编译器在寄存器分配和寻址模式方面有更大的自由度
访问内存时的选择。

I/O 寄存器和常规内存

尽管硬件寄存器和内存之间非常相似,但访问 I/O 寄存器的程序员必须小心避免被 CPU（或编译器)优化所欺骗,这些优化可以修改预期的 I/O 行为。

I/O 寄存器和 RAM 的主要区别在于 I/O 操作有边
效果,而内存操作没有:内存写入的唯一效果是
将值存储到某个位置,然后内存读取返回最后写入的值。
由于内存访问速度对 CPU 性能至关重要,因此无副作用
case 已经在几个方面进行了优化:值被缓存并且读/写指令被重新排序。

* 并非所有计算机平台都使用读取和写入信号;有些有不同的方法来处理外部电路。但是,这种差异在软件级别上是无关紧要的,我们假设所有人都具有读写能力以简化讨论。

编译器可以将数据值缓存到 CPU 寄存器中,而无需将它们写入内存,即使它存储它们,写入和读取操作也可以在缓存内存上进行操作,而无需到达物理 RAM。重新排序也可以在编译器级别和硬件级别发生:如果以不同于程序文本中出现的顺序运行指令序列,通常可以更快地执行指令序列,例如,为了防止互锁RISC 流水线。

在 CISC 处理器上,需要大量时间的操作可以与其他更快的操作同时执行。

这些优化在应用于传统内存时是透明且良性的(至少在单处理器系统上),但它们对于正确的 I/O 操作可能是致命的,因为它们会干扰那些“副作用”,而这些“副作用”是驱动程序的主要原因访问 I/O 寄存器。处理器无法预测其他进程(运行在单独的处理器上,或发生在 I/O 控制器内部的某些事情)取决于内存访问顺序的情况。编译器或 CPU 可能只是试图智取你并重新排序你请求的操作;结果可能是非常难以调试的奇怪错误。因此,驱动程序必须确保在访问寄存器时不执行缓存并且不发生读取或写入重新排序。

硬件缓存的问题是最容易面对的:底层硬件已经配置(自动或通过 Linux 初始化代码)以在访问 I/O 区域(无论是内存区域还是端口区域)时禁用任何硬件缓存。

编译器优化和硬件重新排序的解决方案是在必须以特定顺序对硬件(或另一个处理器)可见的操作之间放置内存屏障。 Linux 提供了四个宏来满足所有可能的订购需求:

#include <linux/kernel.h> 无
效屏障(无效)
这个函数告诉编译器插入一个内存屏障,但对硬件没有影响。编译后的代码将所有当前修改并驻留在 CPU 寄存器中的值存储到内存中,并在以后需要时重新读取它们。对屏障的调用会阻止编译器跨屏障进行优化,但让硬件可以自由地进行自己的重新排序。

#include <asm/system.h>
void rmb(void);无效
read_barrier_depends(无效);无
效 wmb(无效);无效mb(无效);
这些函数在编译指令流中插入硬件内存
屏障;它们的实际实例化取决于平台。 rmb (读取内存屏障)保证在屏障之前出现的任何读取都已完成

在执行任何后续读取之前。 wmb保证写操作的顺序,而mb指令保证两者。这些功能中的每一个都是屏障的超集。 read_barrier_depends是一种特殊的、较弱的读屏障形式。 rmb阻止所有读取跨屏障的重新排序,而read_barrier_depends仅阻止依赖于来自其他读取的数据的读取的重新排序。这种区别是微妙的,它并不存在于所有架构上。除非您确切了解发生了什么,并且您有理由相信完整的读取障碍会造成过高的性能成本,否则您可能应该坚持使用rmb。

无效 smp_rmb(无效);
无效 smp_read_barrier_depends（无效）;无效 smp_wmb(无效);无效 smp_mb（无效）;这些版本的屏障宏仅在为 SMP 系统编译内核时插入硬件屏障;否则,它们都会扩展为一个简单的屏障调用。

设备驱动程序中内存屏障的典型用法可能具有以下形式：

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size); writel(dev-
>registers.operation, DEV_READ); wmb(); writel(dev-
>registers.control, DEV_GO);
```

在这种情况下,重要的是要确保控制特定操作的所有设备寄存器在开始之前都已正确设置。 memory bar rier 以必要的顺序强制完成写入。

因为内存屏障会影响性能,所以它们应该只在真正需要的地方使用。不同类型的屏障也可能具有不同的性能特征,因此值得使用尽可能具体的类型。例如,在 x86 架构上， wmb()目前什么都不做,因为处理器外部的写入不会重新排序。然而,读取被重新排序,因此mb()比wmb( ) 慢。

值得注意的是,大多数其他处理同步的内核原语,例如自旋锁和atomic_t操作,也起到内存屏障的作用。
另外值得注意的是,一些外围总线（例如 PCI 总线)有自己的缓存问题;当我们在后面的章节中讨论它们时,我们会讨论它们。

一些架构允许分配和内存屏障的有效组合。内核提供了一些执行这种组合的宏;在默认情况下,它们的定义如下：

```
#define set_mb(var, value) 做 {var = value; mb( );} while 0 #define
set_wmb(var, value) do {var = value; wmb( );} while 0 #define
set_rmb(var, value) do {var = value; rmb( );} 而 0
```

在适当的情况下， <asm/system.h>定义这些宏以使用更快速地完成任务的体系结构特定指令。请注意， set_rmb仅由少数架构定义。 （使用do...while结构是一个标准 C 习惯用法，它使扩展宏在所有上下文中都作为正常的 C 语句工作。）

使用 I/O 端口

I/O 端口是驱动程序与许多设备通信的方式，至少在部分时间是这样。本节介绍了可用于使用 I/O 端口的各种功能；我们还谈到了一些可移植性问题。

I/O 端口分配

正如您所料，您不应该在没有首先确保您拥有对这些端口的独占访问权的情况下开始攻击 I/O 端口。内核提供了一个注册接口，允许您的驱动程序声明它需要的端口。该接口中的核心函数是request_region：

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
```

这个函数告诉内核你想使用n 个端口，从first 开始。 name参数应该是您设备的名称。如果分配成功，则返回值为非 NULL 。如果您从request_region 返回 NULL，您将无法使用所需的端口。

所有端口分配都显示在/proc/ioports 中。如果您无法分配所需的一组端口，则可以查看谁先到达那里。

当您完成一组 I/O 端口时（可能在模块卸载时），它们应该返回给系统：

```
void release_region(unsigned long start, unsigned long n);
```

还有一个功能可以让您的驱动程序检查给定的一组
I/O 端口可用：

```
int check_region(unsigned long first, unsigned long n);
```

在这里，如果给定端口不可用，则返回值为负错误代码。
该函数已被弃用，因为它的返回值不能保证分配是否成功；检查和稍后分配不是原子操作。我们在这里列出它是因为几个驱动程序仍在使用它，但您应该始终使用request_region，它执行所需的锁定以确保以安全、原子的方式完成分配。

操作 I/O 端口在驱动程序请求它

需要在其活动中使用的 I/O 端口范围之后,它必须读取和/或写入这些端口。为此,大多数硬件区分 8 位、16 位和 32 位端口。通常你不能像往常一样混合使用它们来访问系统内存。*因此,AC 程序必须调用不同的函数来访问不同大小的端口。如上一节所述,仅支持内存映射 I/O 的计算机体系结构通过将端口地址重新映射到内存地址来注册假端口 I/O,并且内核对驱动程序隐藏细节以简化可移植

性。 Linux 内核头文件（特别是与体系结构相关的头文件<asm/io.h>)定义了以下内联函数来访问 I/O 端口： unsigned inb(unsigned port); void outb（无符号字符字节,无符号端口）;读取或写入字节端口（八位宽）。对于某些平台,端口参数定义为unsigned long ,而对于其他平台,则定义为unsigned short 。 inb的返回类型也因架构而异。

无符号 inw（无符号端口）;
void outw（无符号短字,无符号端口）;这些函数访问
16 位端口（一个字宽）;在为仅支持字节 I/O 的 S390 平台编译时,它们不可用。

无符号 inl（无符号端口）;无效
outl（无符号长字,无符号端口）;这些函数访问 32 位
端口。根据平台， longword被声明为unsigned long或unsigned int 。与单词 I/O 一样，“长”
I/O 在 S390 上不可用。

从现在开始,当我们在没有进一步类型规范的情况下使用unsigned时,我们指的是一个依赖于体系结构的定义,其确切性质是不相关的。这些函数几乎总是可移植的,因为编译器会在赋值期间自动转换值 它们是无符号的,有助于防止编译时警告。只要程序员分配合理的值以避免溢出,此类转换就不会丢失任何信息。在本章中,我们坚持“不完整输入”的约定。

请注意,没有定义 64 位端口 I/O 操作。即使在 64 位架构上,端口地址空间也使用 32 位（最大)数据路径。

* 有时 I/O 端口像内存一样排列,您可以（例如)将两个 8 位写入绑定到一个 16 位操作中。例如,这适用于 PC 视频板。但一般来说,你不能指望这个功能。

从用户空间访问 I/O 端口刚才描述的功能主要是

供设备驱动程序使用的,但它们也可以从用户空间使用,至少在 PC 级计算机上。 GNU C 库在<sys/io.h>中定义了它们。为了在用户空间代码中使用inb和朋友,应满足以下条件:

- 必须使用-O选项编译程序以强制展开内联
  功能。
- 必须使用ioperm或iopl系统调用来获得对端口执行 I/O 操作的权限。 ioperm获得单个端口的权限,而iopl获得整个 I/O 空间的权限。这两个函数都是特定于 x86 的。
- 程序必须以 root 身份运行才能调用ioperm或iopl。*或者,它的一个

  祖先必须获得以 root 身份运行的端口访问权限。

如果宿主平台没有ioperm和iopl系统调用,用户空间仍然可以使用/dev/port设备文件访问 I/O 端口。但是请注意,该文件的含义是非常特定于平台的,除了 PC 之外可能对其他任何东西都没有用。

示例源misc-progs/inp.c和misc-progs/outp.c是在用户空间中从命令行读取和写入端口的最小工具。他们希望以多个名称安装(例如, inb 、 inw和 inl,并根据用户调用的名称来操作字节、字或长端口)。他们在 x86 下使用ioperm或iopl ,在其他平台上使用/dev/port 。

如果您想危险地生活并在不获得明确权限的情况下使用您的硬件,则可以将程序设置为 setuid root。但是,请不要在生产系统上安装它们 set uid;它们是设计的安全漏洞。

字符串操作除了单次输入和

输出操作之外,一些处理器还执行特殊指令以在单个 I/O 端口或相同大小的输入/输出端口之间传输字节、字或长整数序列。这些就是所谓的字符串指令,它们执行任务的速度比 C 语言循环更快。以下宏通过使用单个机器指令或在目标处理器没有执行字符串 I/O 的指令时执行紧密循环来实现字符串 I/O 的概念。为 S390 平台编译时根本没有定义宏。

这应该不是可移植性问题,因为这个平台通常不与其他平台共享设备驱动程序,因为它的外围总线是不同的。

* 从技术上讲,它必须具有CAP_SYS_RAWIO功能,但这与在大多数 cur 上以 root 身份运行相同
出租系统。

字符串函数的原型是：

void insb(unsigned port, void *addr, unsigned long count); void outsb(unsigned port, void *addr, unsigned long count);从内存地址addr开始读取或写入count个字节。从单端口端口读取或写入数据。

void insw(unsigned port, void *addr, unsigned long count);无效 outsw（无符号端口,无效 *addr,无符号长计数）;读取或写入 16 位值到单个 16 位端口。

void insl(unsigned port, void *addr, unsigned long count); void outsl(unsigned port, void *addr, unsigned long count);读取或写入 32 位值到单个 32 位端口。

使用字符串函数时要记住一件事：它们将直接字节流移入或移出端口。当端口和主机系统具有不同的字节排序规则时，结果可能会令人惊讶。如果需要，使用inw读取端口会交换字节，以使读取的值与主机顺序匹配。相反，字符串函数不执行这种交换。

暂停 I/O某些平

台（尤其是 i386)在处理器试图将数据传输到总线或从总线传输得太快时可能会出现问题。当处理器相对于外围总线超频时可能会出现问题（此处考虑 ISA），并且可能会在设备板太慢时出现。解决方案是在每条 I/O 指令之后插入一个小的延迟，如果后面有另一个这样的指令。在 x86 上，通过对端口 0x80 执行out b指令（通常但不总是未使用)或忙等待来实现暂停。有关详细信息，请参阅平台的asm子目录下的io.h文件。

如果您的设备丢失了一些数据，或者您担心它可能会丢失一些数据，您可以使用暂停功能代替正常功能。暂停函数与前面列出的完全一样，但它们的名称以_p 结尾；它们被称为inb_p、 outb_p 等。这些函数是为大多数受支持的架构定义的，尽管它们通常会扩展为与非暂停 I/O 相同的代码，因为如果架构使用相当现代的外围总线运行，则不需要额外的暂停。

平台相关性 I/O 指令本质上是高

度依赖于处理器的。因为它们处理处理器如何处理移入和移出数据的细节，所以很难隐藏系统之间的差异。因此，与端口 I/O 相关的大部分源代码都依赖于平台。

通过回顾函数列表，您可以看到其中不兼容的地方之一，即数据类型，其中参数的类型根据架构差异而不同

平台之间。例如,端口在 x86 上是unsigned short （处理器支持 64 KB I/O 空间）,但在其他平台上是unsigned long ,其端口只是与内存相同地址空间中的特殊位置。

其他平台依赖性源于处理器的基本结构差异,因此是不可避免的。我们不会详细介绍这些差异,因为我们假设您不会在不了解底层硬件的情况下为特定系统编写设备驱动程序。相反,这里是内核支持的架构功能的概述:

IA-32 (x86)
x86_64该架构

支持本章描述的所有功能。端口号是unsigned short 类型。

IA-64（安腾）

支持所有功能;端口是无符号长的（和内存映射的）。
字符串函数在 C 中实现。

Alpha

支持所有功能,并且端口是内存映射的。不同的Alpha平台,端口I/O的实现是不同的,根据他们使用的芯片组。字符串函数在 C 中实现并在arch/alpha/lib/io.c 中定义。端口是无符号长的。

ARM

Ports是内存映射的,支持所有功能;字符串函数在 C 中实现。端口的类型为unsigned int。

Cris

这种架构即使在仿真模式下也不支持 I/O 端口抽象;各种端口操作被定义为什么都不做。

M68k
M68k-名词

端口是内存映射的。支持字符串函数,端口类型为unsigned char *。

MIPS
MIPS64

MIPS 端口支持所有功能。字符串操作是通过紧密的汇编循环实现的,因为处理器缺少机器级字符串 I/O。
端口是内存映射的;它们是无符号长的。

PA-RISC支

持所有功能;端口在基于 PCI 的系统上是int ,在 EISA 系统上是unsigned short ,但字符串操作除外,它使用unsigned long端口号。

电源PC
电源PC64

支持所有功能;端口在 32 位系统上具有unsigned char *类型,在 64 位系统上具有unsigned long类型。

S390

与 M68k类似,此平台的标头仅支持字节宽端口 I/O,不支持字符串操作。端口是字符指针并且是内存映射的。

Super-H端

口是无符号整数（内存映射）,并且支持所有功能。

SPARC
SPARC64再

一次,I/O 空间是内存映射的。端口函数的版本被定义为使用无符号长端口。

好奇的读者可以从io.h文件中提取更多信息,除了我们在本章中描述的功能之外,这些文件有时还定义了一些特定于体系结构的功能。但是请注意,其中一些文件很难阅读。

有趣的是,x86 系列以外的处理器都没有为端口提供不同的地址空间,尽管一些受支持的系列附带 ISA 和/或 PCI 插槽（并且两种总线都实现了单独的 I/O 和内存地址空间）。

此外,一些处理器（尤其是早期的 Alpha 处理器)缺少一次移动一个或两个字节的指令。*因此,它们的外围芯片组通过将它们映射到内存中的特殊地址范围来模拟 8 位和 16 位 I/O 访问地址空间。因此,作用于同一端口的inb和inw指令由两个在不同地址上操作的 32 位存储器读取实现。幸运的是,所有这些都被本节中描述的 mac ros 的内部结构隐藏在设备驱动程序编写者之外,但我们认为这是一个有趣的特性。如果您想进一步探索,请在include/asm-alpha/core_lca.h 中查找示例。

每个平台上如何执行 I/O 操作在每个平台的程序员手册中都有很好的描述;这些手册通常可以在 Web 上以 PDF 格式下载。

* 单字节 I/O 并不像人们想象的那么重要,因为它是一种罕见的操作。要将单个字节读/写到任何地址空间,您需要实现一个数据路径,将寄存器集数据总线的低位连接到外部数据总线中的任何字节位置。这些数据路径需要额外的逻辑门,这些逻辑门会妨碍每次数据传输。删除字节范围的加载和存储可以提高整体系统性能。

I/O 端口示例

我们用来显示设备驱动程序中的端口 I/O 的示例代码作用于通用数字 I/O 端口;在大多数计算机系统中都可以找到此类端口。

最常见的数字 I/O 端口是一个字节宽的 I/O 位置,可以是内存映射的,也可以是端口映射的。当您将值写入输出位置时,在输出引脚上看到的电信号会根据正在写入的各个位而发生变化。当您从输入位置读取值时,在输入引脚上看到的当前逻辑电平将作为单个位值返回。

此类 I/O 端口的实际实现和软件接口因系统而异。大多数时候,I/O 引脚由两个 I/O 位置控制:一个允许选择哪些引脚用作输入以及哪些引脚用作输出,另一个是您可以实际读取或写入逻辑电平。然而,有时事情甚至更简单,这些位被硬连线为输入或输出(但在这种情况下,它们不再被称为“通用 I/O”);所有个人计算机上的并行端口就是这样一种不太通用的 I/O 端口。无论哪种方式,我们很快介绍的示例代码都可以使用 I/O 引脚。

并行端口概述

因为我们希望大多数读者使用称为“个人计算机”形式的 x86 平台,所以我们觉得有必要解释一下 PC 并行端口的设计方式。
并行端口是在个人计算机上运行数字 I/O 示例代码的首选外围接口。尽管大多数读者可能都有可用的并行端口规范,但为了您的方便,我们在此对其进行了总结。

并行接口在其最小配置中(我们忽略了 ECP 和 EPP 模式)由三个 8 位端口组成。 PC 标准在0x378处启动第一个并行接口的 I/O 端口,在0x278 处启动第二个并行接口的 I/O 端口。第一个端口是双向数据寄存器;它直接连接到物理连接器上的引脚 2-9。第二个端口是只读状态寄存器;当并行端口用于打印机时,该寄存器报告打印机状态的几个方面,例如在线、缺纸或忙碌。第三个端口是一个仅输出控制寄存器,除其他外,它控制是否启用中断。

并行通信中使用的信号电平是标准晶体管-晶体管逻辑 (TTL) 电平:0 和 5 伏,逻辑阈值约为 1.2 伏。您可以指望这些端口至少满足标准 TTL LS 电流额定值,尽管大多数现代并行端口在电流和电压额定值方面都做得更好。

并行连接器不与计算机的内部电路隔离,如果您想将逻辑门直接连接到端口,这很有用。但是您必须小心正确接线;当您使用自己的自定义电路时,并行端口电路很容易损坏,除非您在电路中添加光隔离器。如果您担心会损坏主板,您可以选择使用插入式并行端口。

图 9-1 概述了位规范。您可以访问 12 个输出位和 5 个输入位,其中一些在其信号路径过程中被逻辑反转。
唯一没有关联信号引脚的位是端口 2 的位 4 (0x10),它启用来自并行端口的中断。我们在第 10 章中使用该位作为中断处理程序实现的一部分。

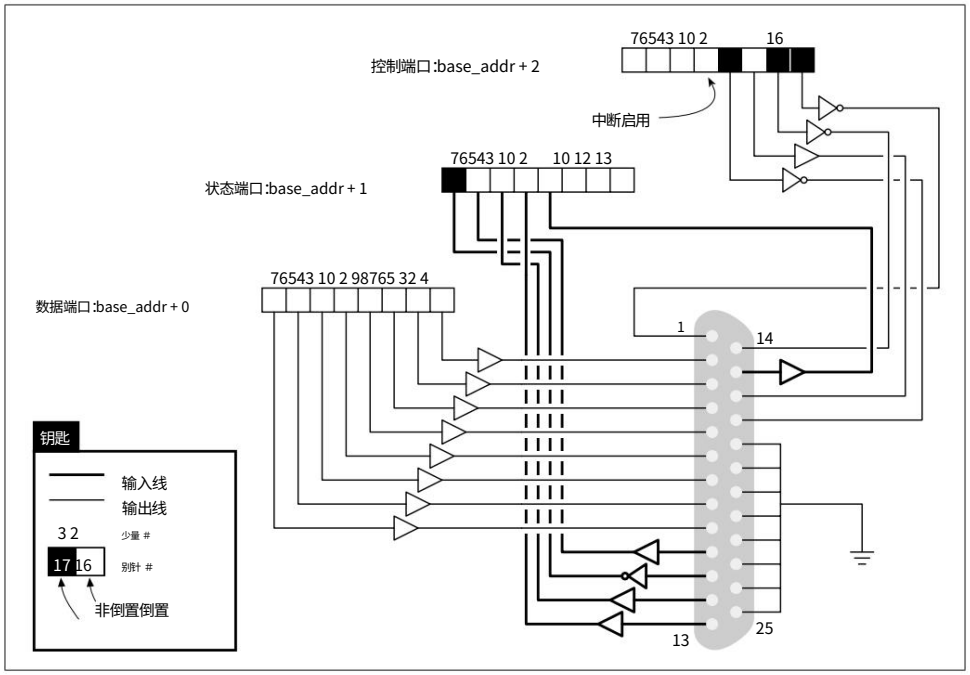

图 9-1。并口的引脚排列

一个示例驱动程序我

们介绍的驱动程序称为short （简单硬件操作和原始测试）。它所做的只是读取和写入几个 8 位端口,从您在加载时选择的端口开始。默认情况下,它使用分配给 PC 并行接口的端口范围。每个设备节点（具有唯一的次要编号)访问不同的端口。短驱动程序没有做任何有用的事情。它只是将外部使用隔离为作用于端口的单个指令。如果不习惯port I/O,可以使用short来获取

熟悉它;您可以测量通过端口传输数据或玩其他游戏所需的时间。

简而言之,要在您的系统上运行,它必须可以自由访问底层硬件设备（默认情况下,并行接口）;因此,可能没有其他驱动程序分配它。大多数现代发行版将并行端口驱动程序设置为仅在需要时才加载的模块,因此争用 I/O 地址通常不是问题。但是,如果您从简短（在控制台或系统日志文件中)收到“无法获取 I/O 地址”错误,则其他驱动程序可能已经占用了该端口。

快速查看/proc/ioports通常会告诉您哪个驱动程序阻碍了您。如果您不使用并行接口,则同样的警告适用于其他 I/O 设备。

从现在开始,我们只提到“并行接口”来简化讨论。
但是,您可以在加载时设置基本模块参数以将短路重定向到其他 I/O 设备。此功能允许示例代码在您可以访问可通过outb和inb访问的数字 I/O 接口的任何 Linux 平台上运行（即使实际硬件在除 x86 之外的所有平台上都是内存映射的）。

稍后,在“使用 I/O 内存”一节中,我们将展示如何将short用于通用内存映射数字 I/O。

要查看并行连接器上发生的情况,并且如果您有一点使用硬件的倾向,您可以将几个 LED 焊接到输出引脚上。每个 LED 应串联到一个 1-KΩ 电阻器,该电阻器通向接地引脚（当然,除非您的 LED 具有内置电阻器）。如果将输出引脚连接到输入引脚,您将生成自己的输入以从输入端口读取。

请注意,您不能只将打印机连接到并行端口并查看发送到短路的数据。此驱动程序实现对 I/O 端口的简单访问,并且不执行打印机操作数据所需的握手。在下一章中,我们将展示一个示例驱动程序（称为shortprint）,它能够驱动并行打印机;然而,该驱动程序使用中断,所以我们还不能完全理解它。

如果您打算通过将 LED 焊接到 D 型连接器来查看并行数据,我们建议您不要使用引脚 9 和 10,因为我们稍后将它们连接在一起以运行第 10 章中显示的示例代码。

就short而言， /dev/short0写入和读取位于 I/O 地址基址的 8 位端口（0x378,除非在加载时更改）。 /dev/short1写入位于base + 1 的 8 位端口,依此类推,直到base + 7。

/dev/short0执行的实际输出操作基于使用outb 的紧密循环。内存屏障指令用于确保输出操作实际发生并且没有被优化掉:

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

您可以运行以下命令来点亮 LED：

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

short0每个 LED 监控输出端口的一位。请记住，只有最后写入的字符在输出引脚上保持稳定的时间足以被您的眼睛感知。因此，我们建议您通过将-n选项传递给echo 来防止自动插入尾随换行符。

读取由类似的函数执行，围绕inb而不是outb 构建。为了从并行端口读取“有意义的”值，您需要将一些硬件连接到连接器的输入引脚以生成信号。如果没有信号，您将读取无穷无尽的相同字节流。如果您选择从输出端口读取，您很可能会取回写入端口的最后一个值（这适用于并行接口和大多数其他常用的数字 I/O 电路）。因此，那些不愿拿出烙铁的人可以通过运行以下命令读取端口 0x378 上的当前输出值：

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

为了演示所有 I/O 指令的使用，每个短设备都有三种变体： /dev/short0执行刚刚显示的循环， /dev/short0p使用outb_p和inb_p代替“快速”函数，和/ dev/short0s使用字符串指令。

有 8 个这样的设备，从short0到short7。虽然 PC 并行接口只有三个端口，但如果使用不同的 I/O 设备来运行测试，您可能需要更多端口。

短驱动程序执行绝对最少的硬件控制，但足以显示如何使用 I/O 端口指令。感兴趣的读者可能想查看parport和parport_pc模块的源代码，以了解该设备在现实生活中的复杂程度，以便在并行端口上支持一系列设备（打印机、磁带备份、网络接口）。

使用 I/O 内存

尽管 I/O 端口在 x86 世界中很流行，但用于与设备通信的主要机制是通过内存映射寄存器和设备内存。两者都称为I/O 内存，因为寄存器和内存之间的区别对软件是透明的。

I/O 内存只是设备通过总线向处理器提供的类似 RAM 的区域。该内存可用于多种用途，例如保存视频数据或以太网数据包，以及实现与 I/O 端口类似的设备寄存器（即，它们具有与读取和写入相关的副作用）。

访问 I/O 内存的方式取决于所使用的计算机体系结构、总线和设备，尽管原则在任何地方都是相同的。讨论

本章主要涉及 ISA 和 PCI 内存,同时也试图传达一般信息。虽然这里介绍了对 PCI 内存的访问,但对 PCI 的全面讨论推迟到第 12 章。

根据所使用的计算机平台和总线,I/O 内存可能会或可能不会通过页表访问。当访问通过页表时,内核必须首先安排物理地址对您的驱动程序可见,这通常意味着您必须在执行任何 I/O 之前调用 ioremap 。如果不需要页表,I/O 内存位置看起来很像 I/O 端口,您可以使用适当的包装函数读取和写入它们。

无论是否需要ioremap来访问 I/O 内存,不鼓励直接使用指向 I/O 内存的指针。尽管(如“I/O 端口和 I/O 内存”一节中介绍的)I/O 内存在硬件级别像普通 RAM 一样被寻址,但“I/O 寄存器和常规内存”一节中概述的额外注意建议避免普通指针。用于访问 I/O 内存的包装函数在所有平台上都是安全的,并且只要直接指针取消引用可以执行操作,就会被优化掉。

因此,即使在 x86 上解除引用指针(目前)有效,但未能使用正确的宏会阻碍驱动程序的可移植性和可读性。

I/O 内存分配和映射I/O 内存区域必须在使用前分

配。分配内存区域的接口(定义在<linux/ioport.h> 中)是:

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char
                                    *name);
```

此函数从 start 开始分配len个字节的内存区域。如果一切顺利,则返回一个非 NULL指针;否则返回值为NULL。所有 I/O 内存分配都列在/proc/iomem 中。

不再需要时应释放内存区域:

```
void release_mem_region(unsigned long start, unsigned long len);
```

还有一个用于检查 I/O 内存区域可用性的旧函数: int check_mem_region(unsigned

long start, unsigned long len);但是,与check_region 一样,此功能是不安

全的,应避免使用。

在可以访问该内存之前,分配 I/O 内存并不是唯一需要的步骤。您还必须确保内核可以访问此 I/O 内存。获取 I/O 内存不仅仅是取消引用指针的问题;在许多系统上,根本无法以这种方式直接访问 I/O 内存。所以必须先建立一个映射。这就是ioremap函数的作用,在小节中介绍

第 1 章中的“vmalloc 和朋友”。该函数专门设计用于为 I/O 内存区域分配虚拟地址。

一旦配备了ioremap （和iounmap），设备驱动程序就可以访问任何 I/O 内存地址，无论它是否直接映射到虚拟地址空间。
但请记住，从ioremap返回的地址不应直接取消引用；相反，应该使用内核提供的访问器函数。

在我们进入这些函数之前，我们最好回顾一下ioremap原型并介绍一些我们在上一章中忽略的细节。

根据以下定义调用函数：

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size); void
*ioremap_nocache(unsigned long phys_addr, unsigned long size); void iounmap(void * addr);
```

首先，您会注意到新函数ioremap_nocache。我们在第 8 章中没有涉及它，因为它的含义肯定与硬件有关。引用其中一个内核头文件：“如果某些控制寄存器位于这样的区域中，这很有用，并且不希望写入组合或读取缓存。”实际上，该函数的实现与大多数计算机平台上的ioremap相同：在所有 I/O 内存已经通过不可缓存地址可见的情况下，没有理由实现单独的、非缓存版本的ioremap。

访问 I/O 内存在某些平台上，您

可能不使用ioremap的返回值作为指针。这种使用是不可移植的，并且内核开发人员越来越多地致力于消除任何此类使用。获取 I/O 内存的正确方法是通过为此目的提供的一组函数（通过 <asm/io.h> 定义）。

要从 I/O 内存读取，请使用以下方法之一：

```
无符号整数 ioread8(void *addr);无符号整数
ioread16(void *addr);无符号整数 ioread32(void
*addr);
```

这里， addr应该是从ioremap获得的地址（可能带有一个整数偏移）；返回值是从给定 I/O 内存中读取的值。

有一组类似的函数用于写入 I/O 内存：

```
void iowrite8(u8 值, void *addr);无效 iowrite16（u16
值,无效 *addr）；无效 iowrite32（u32 值,无效 *addr）；
```

如果您必须读取或写入一系列值到给定的 I/O 内存地址，您可以使用函数的重复版本：

```
void ioread8_rep(void *addr, void *buf, unsigned long count); void ioread16_rep(void
*addr, void *buf, unsigned long count);
```

```
void ioread32_rep(void *addr, void *buf, unsigned long count); void
iowrite8_rep(void *addr, const void *buf, unsigned long count); void
iowrite16_rep(void *addr, const void *buf, unsigned long count); void
iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

这些函数从给定的buf读取或写入计数值到给定的addr。请注意,计数以正在写入的数据的大小表示； ioread32_rep读取从buf开始的count 32 位值。

上述函数对给定地址执行所有 I/O。相反,如果您需要对一块 I/O 内存进行操作,则可以使用以下之一： void memset_io(void *addr, u8 value, unsigned int count); void

```
memcpy_fromio(void *dest, void *source, unsigned int count); void
memcpy_toio(void *dest, void *source, unsigned int count);
```

这些函数的行为类似于它们的 C 库类似物。

如果您通读内核源代码,您会看到在使用 I/O 内存时许多对旧函数集的调用。这些函数仍然有效,但不鼓励在新代码中使用它们。除其他外,它们不太安全,因为它们不执行相同类型的类型检查。尽管如此,我们在这里对其进行描述： unsigned readb(address);无符号读取（地址）;无符号读取（地址）;这些宏用于从 I/O 检索 8 位、16 位和 32 位数据值

记忆。

void writeb（无符号值,地址）； void
writew（无符号值,地址）； void writel
（无符号值,地址）；

与前面的函数一样,这些函数（宏)用于写入 8 位、16 位和 32 位数据项。

一些 64 位平台还提供readq和writeq,用于 PCI 总线上的四字（8 字节)内存操作。四字命名法是所有真实处理器都有 16 位字时的历史遗留问题。实际上,用于 32 位值的L命名也变得不正确,但是重命名所有内容会更加混乱。

端口作为 I/O 内存一些硬件

有一个有趣的特性:一些版本使用 I/O 端口,而其他版本使用 I/O 内存。在这两种情况下,导出到处理器的寄存器都是相同的,但访问方法不同。作为一种让处理这种硬件的驱动程序的工作更轻松的方法,并且作为一种最小化 I/O 端口和内存访问之间明显差异的方法,2.6 内核提供了一个名为 ioport_map 的函数：

```
void *ioport_map（无符号长端口,无符号整数）；
```

此函数重新映射计数I/O 端口并使它们看起来是 I/O 内存。
从那时起,驱动程序可能会在返回的地址上使用ioread8和朋友,而根本忘记它正在使用 I/O 端口。

当不再需要此映射时,应撤消它:

```
void ioport_unmap(void *addr);
```

这些函数使 I/O 端口看起来像内存。但是请注意,I/O 端口仍然必须使用request_region分配,然后才能在此重新映射
方法。

重用 I/O 内存的short 前面介绍的用于访

问I/O 端口的short示例模块也可用于访问I/O 内存。为此,您必须告诉它在加载时使用 I/O 内存;此外,您需要更改基地址以使其指向您的 I/O 区域。

例如,这就是我们如何使用短路来点亮 MIPS 开发板上的调试 LED:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

I/O 内存的缩写与 I/O 端口的使用相同。

以下片段显示了short在写入内存位置时使用的循环:

```
while (count--)
      { iowrite8(*ptr++, 地址); wmb();

}
```

注意这里使用了写内存屏障。因为iowrite8可能在许多架构上变成直接赋值,所以需要内存屏障来确保写入按预期顺序发生。 short使用inb和outb来展示它是如何完成的。然而,对于读者来说,将其更改为使用 ioport_map 重新映射 I/O 端口将是一个简单的练习,并大大简化了其余代码。

ISA 内存低于 1 MB最著名的 I/O 内

存区域之一是个人计算机上的 ISA 范围。这是 640 KB (0xA0000)和 1 MB (0x100000)之间的内存范围。因此,它出现在常规系统 RAM 的中间。这个定位可能看起来有点奇怪;这是 1980 年代初做出的决定的产物,当时 640 KB 的内存似乎比任何人都多

使用。

此内存范围属于非直接映射的内存类别。*您可以使用前面解释的short模块在该内存范围中读取/写入几个字节,即通过在加载时设置use_mem 。

尽管 ISA I/O 内存只存在于 x86 级计算机中,但我们认为值得花上几句话和一个示例驱动程序在上面。

本章不讨论 PCI 内存,因为它是最干净的 I/O 内存:一旦知道物理地址,就可以简单地重新映射和访问它。 PCI I/O 内存的“问题”是它不适合本章的工作示例,因为我们无法提前知道您的 PCI 内存映射到的物理地址,或者它是否安全访问这些范围中的任何一个。我们选择描述 ISA 内存范围,因为它既不干净又更适合运行示例代码。

为了演示对 ISA 内存的访问,我们使用了另一个愚蠢的小模块（示例源的一部分）。事实上,这个被称为silly,是 Simple Tool for Unloading and Printing ISA Data 的首字母缩写,或类似的东西。

该模块通过访问整个 384 KB 内存空间并显示所有不同的 I/O 功能来补充short的功能。它具有四个设备节点,它们使用不同的数据传输功能执行相同的任务。愚蠢的设备充当 I/O 内存的窗口,其方式类似于/dev/mem。您可以读取和写入数据,并查找任意 I/O 内存地址。

因为silly提供对 ISA 内存的访问,所以它必须首先将物理 ISA 地址映射到内核虚拟地址。在 Linux 内核的早期,人们可以简单地分配一个指向感兴趣的 ISA 地址的指针,然后直接取消引用它。然而,在现代世界中,我们必须首先使用虚拟内存系统并重新映射内存范围。这个映射是用 ioremap 完成的,如前所述:

```
#define ISA_BASE        0xA0000
#define ISA_MAX         0x100000 /* 用于一般内存访问 */

    /* 这行出现在 silly_init */ io_base =
    ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap返回一个指针值,该指针值可用于ioread8和“访问 I/O 内存”一节中介绍的其他函数。

让我们回顾一下我们的示例模块,看看如何使用这些函数。 /dev/sillyb,具有次要编号0,使用ioread8和iowrite8访问 I/O 内存。
以下代码显示了读取的实现,它使地址

* 实际上,这并不完全正确。内存范围如此之小且使用如此频繁,以至于内核在启动时构建页表来访问这些地址。但是,用于访问它们的虚拟地址与物理地址不同,因此无论如何都需要ioremap 。

范围0xA0000-0xFFFFF可用作 0-0x5FFFF 范围内的虚拟文件。 read函数被构造为不同访问模式的switch语句;这是愚蠢的情况：

```
case M_8:
  while (count)
      { *ptr = ioread8(add);添
      加++;数数 - ;指针++;


  } 休息;
```

接下来的两个设备是/dev/sillyw （次要编号 1)和/dev/sillyl （次要编号 2）。它们的行为类似于/dev/sillyb,不同之处在于它们使用 16 位和 32 位函数。这是silyl的写实现,也是开关的一部分：

```
case M_32:
  while (count >= 4)
      { iowrite8(*(u32 *)ptr, add);添加
      += 4;计数-= 4;指针 += 4;


  } 休息;
```

最后一个设备是/dev/sillycp （次要编号 3）,它使用memcpy_*io函数来执行相同的任务。这是其读取实现的核心：

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

因为ioremap用于提供对 ISA 内存区域的访问,所以silly必须在模块卸载时调用iounmap：

```
iounmap(io_base);
```

isa_readb 和朋友看一下内核源

代码会发现另一组例程,其名称如isa_readb。事实上,刚才描述的每个函数都有一个isa_等价物。这些函数提供对 ISA 内存的访问,而无需单独的ioremap步骤。然而,来自内核开发人员的说法是,这些函数旨在作为临时的驱动程序移植辅助工具,并且它们将来可能会消失。因此,您应该避免使用它们。

快速参考

本章介绍了以下与硬件管理相关的符号：

#include <linux/kernel.h> 无

效屏障（无效）

这种“软件”内存屏障要求编译器考虑该指令中的所有内存易失性。

#include <asm/system.h>
void rmb(void);无效
read_barrier_depends（无效）；无
效 wmb（无效）；无效mb（无效）；
硬件内存屏障。它们要求 CPU（和编译
器)检查该指令中所有内存读取、写入或两者的点。

#include <asm/io.h>

unsigned inb(unsigned port);
void outb（无符号字符字节，无符号端口）；无符号
inw（无符号端口）； void outw（无符号短字，无符
号端口）；无符号 inl（无符号端口）； void outl（无符
号双字，无符号端口）；用于读取和写入 I/O 端口的函数。
用户空间程序也可以调用它们，前提是它们具有访问端口的
权限。

无符号 inb_p（无符号端口）；
...

如果 I/O 操作后需要一点延迟，您可以使用上一篇介绍的函数的六个暂停对应项；这些暂停函数的名称以_p 结尾。 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);无效 outsw（无符号端口，无效 *addr，无符号长计数）； void insl(unsigned port, void *addr, unsigned long count); void outsl(unsigned port, void *addr, unsigned long count); “字符串函数”经过优化，可以将数据从输入端口传输到内存区域，或者相反。通过读取或写入相同的端口计数时间来执行此类传输。

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len, char
  *姓名）；
void release_region(unsigned long start, unsigned long len); int
check_region(unsigned long start, unsigned long len); I/O 端口的资
    源分配器。 （已弃用)检查函数返回0表示成功,小于0表示错误。

struct resource *request_mem_region(unsigned long start, unsigned long len,
  字符*名称）；
void release_mem_region(unsigned long start, unsigned long len); int
check_mem_region(unsigned long start, unsigned long len);处理内存区域资
    源分配的函数。
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr); ioremap将物理地址范围重新映射到处理器的
    虚拟地址空间,使其可供内核使用。 iounmap在不再需要时释放映射。


#include <asm/io.h>
unsigned int ioread8(void *addr);无符号
整数 ioread16(void *addr);无符号整数
ioread32(void *addr); void iowrite8(u8
值, void *addr);无效 iowrite16（u16 值,无效
*addr）;无效 iowrite32（u32 值,无效 *addr）；
用于处理 I/O 内存的访问器函数。 void
    ioread8_rep(void *addr, void *buf, unsigned long count); void
ioread16_rep(void *addr, void *buf, unsigned long count); void
ioread32_rep(void *addr, void *buf, unsigned long count); void
iowrite8_rep(void *addr, const void *buf, unsigned long count); void
iowrite16_rep(void *addr, const void *buf, unsigned long count); void
iowrite32_rep(void *addr, const void *buf, unsigned long count); I/O 内存原语的“重
复”版本。
```

无符号读取（地址）;无符号
读取（地址）;无符号读取
（地址）； void writeb（无
符号值,地址）； void writew（无符号值，
地址）； void writel（无符号值,地址）；
memset_io（地址,值,计数）；
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);用于
访问 I/O 内存的旧的、类型不安全的函数。

void *ioport_map（无符号长端口,无符号整数）； void ioport_unmap(void *addr);想要将 I/O 端口视为 I/O 内存的驱动程序作者可以将这些端口传递给ioport_map。当不再需要时,应该进行映射（使用ioport_unmap）。