第九章 与硬件通信

虽然摆弄scull 和类似的玩具是对 Linux 设备驱动程序的软件接口的一个很好的介绍，但实现一个真正的设备需要硬件。驱动程序是软件概念和硬件电路之间的抽象层；因此，它需要与它们两者进行交谈。到目前为止，我们已经检查了软件概念的内部；本章通过向你展示驱动程序如何在跨 Linux 平台的同时访问 I/O 端口和 I/O 内存，来完成这个画面。

本章继续保持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（如标准的 PC 并行端口）来展示 I/O 指令是如何工作的，以及使用正常的帧缓冲视频内存来展示内存映射的 I/O。

我们选择简单的数字 I/O，因为它是输入/输出端口的最简单形式。此外，并行端口实现了原始的 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，输入引脚上的电压级别可以直接由处理器访问。在实践中，你必须将 LED 或打印机连接到端口上，才能真正看到数字 I/O 操作的结果，但底层硬件非常容易使用。

9.1. I/O 端口和 I/O 内存

每个外设设备都是通过写入和读取其寄存器来控制的。大多数时候，设备有多个寄存器，它们在内存地址空间或 I/O 地址空间的连续地址上被访问。

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总线和控制总线（即读取和写入信号）上施加电信号，并通过从数据总线读取或写入来访问的。

虽然一些 CPU 制造商在他们的芯片中实现了单一的地址空间，但其他人决定外设设备与内存不同，因此，应该有一个单独的地址空间。一些处理器（尤其是 x86 系列）有单独的读取和写入电线用于 I/O 端口，并且有特殊的 CPU 指令来访问端口。

因为外设设备是为了适应外设总线而制造的，而最流行的 I/O 总线是根据个人计算机进行建模的，所以即使那些没有为 I/O 端口设置单独地址空间的处理器在访问某些外设设备时也必须伪造读取和写入 I/O 端口，通常是通过外部芯片组或 CPU 核心中的额外电路来实现。后一种解决方案在用于嵌入式使用的微小处理器中很常见。

出于同样的原因，Linux 在所有它运行的计算机平台上都实现了 I/O 端口的概念，即使在 CPU 实现单一地址空间的平台上也是如此。端口访问的实现有时依赖于主机计算机

的特定型号和型号（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）。

即使外设总线有一个单独的地址空间用于 I/O 端口，也并非所有设备都将其寄存器映射到 I/O 端口。虽然使用 I/O 端口对于 ISA 外设板很常见，但大多数 PCI 设备将寄存器映射到内存地址区域。这种 I/O 内存方法通常是首选，因为它不需要使用特殊目的的处理器指令；CPU 核心访问内存的效率更高，而且在访问内存时，编译器在寄存器分配和寻址模式选择方面有更多的自由。

9.1.1. I/O 寄存器和常规内存

尽管硬件寄存器和内存之间有很强的相似性，但访问 I/O 寄存器的程序员必须小心，以避免被 CPU（或编译器）优化所欺骗，这些优化可能会修改预期的 I/O 行为。

I/O 寄存器和 RAM 的主要区别在于，I/O 操作有副作用，而内存操作没有：内存写入的唯一效果是将值存储到一个位置，内存读取返回最后写入那里的值。因为内存访问速度对 CPU 性能至关重要，所以没有副作用的情况已经通过多种方式进行了优化：值被缓存，读/写指令被重新排序。

- 并非所有的计算机平台都使用读取和写入信号；有些平台有不同的方式来寻址外部电路。然而，在软件层面，这种差异是无关紧要的，为了简化讨论，我们将假设所有的平台都有读取和写入功能。

编译器可以将数据值缓存到 CPU 寄存器中，而不将它们写入内存，即使它存储它们，读写操作也可以在缓存内存上操作，而不需要到达物理 RAM。在编译器级别和硬件级别，都可以进行重新排序：通常，如果按照与程序文本中出现的顺序不同的顺序执行指令序列，可以更快地执行。例如，为了防止在 RISC 管道中的互锁。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并行执行。

当应用于常规内存时（至少在单处理器系统上），这些优化是透明且良性的，但它们可能对正确的 I/O 操作是致命的，因为它们干扰了驱动程序访问 I/O 寄存器的主要原因——那些“副作用”。处理器不能预见到某个其他进程（在单独的处理器上运行，或者在 I/O 控制器内部发生的事情）依赖于内存访问的顺序的情况。编译器或 CPU 可能只是试图比你更聪明，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进行任何缓存，并且不进行任何读写重新排序。

硬件缓存的问题最容易解决：底层硬件已经配置（自动或通过 Linux 初始化代码）在访问 I/O 区域时禁用任何硬件缓存（无论它们是内存还是端口区域）。

解决编译器优化和硬件重新排序的方法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一个内存屏障。Linux 提供了四个宏来覆盖所有可能的排序需求：

```c
#include <linux/kernel.h>

void barrier(void)
```

此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，并在稍后需要时重新读取它们。调用 barrier 阻止编译器在屏障之间进行优化，但让硬件自由进行自己的重新排序。

```c
#include <asm/system.h>

void rmb(void);

void read_barrier_depends(void);

void wmb(void);

void mb(void);
```

这些函数在编译的指令流中插入硬件内存屏障；它们的实际实例化是平台依赖的。rmb（读内存屏障）保证在屏障之前出现的任何读取在执行任何后续读取之前都已完成。wmb 保证写操作的顺序，mb 指令保证两者。每个这些函数都是 barrier 的超集。

read_barrier_depends 是读屏障的一种特殊、较弱的形式。而 rmb 阻止所有读取在屏障之间的重新排序，read_barrier_depends 只阻止依赖于其他读取的数据的读取的重新排序。这种区别是微妙的，并且并非在所有架构上都存在。除非你完全理解正在发生的事情，并且有理由相信全读屏障正在施加过多的性能成本，否则你可能应该坚持使用 rmb。

```c
void smp_rmb(void);

void smp_read_barrier_depends(void);

void smp_wmb(void);

void smp_mb(void);
```

这些屏障宏的版本只在内核为 SMP 系统编译时插入硬件屏障；否则，它们都扩展为一个简单的屏障调用。

设备驱动程序中的内存屏障的典型用法可能有这种形式：

```c
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，重要的是确保在告诉设备开始操作之前，所有控制特定操作的设备寄存器都已经正确设置。内存屏障强制以必要的顺序完成写入。

因为内存屏障会影响性能，所以只有在真正需要的地方才应该使用它们。不同类型的屏障也可能有不同的性能特性，所以使用最具体的类型是值得的。例如，在 x86 架构上，wmb() 目前什么也不做，因为处理器外部的写入不会被重新排序。然而，读取是被重新排序的，所以 mb() 比 wmb() 慢。

值得注意的是，大多数其他处理同步的内核原语，如 spinlock 和 atomic_t 操作，也作为内存屏障功能。同样值得注意的是，一些外围总线（如 PCI 总线）有自己的缓存问题；我们在后面的章节中讨论这些问题。

一些架构允许有效地组合赋值和内存屏障。内核提供了一些执行此组合的宏；在默认情况下，它们被定义如下：

```c
#define set_mb(var, value)  do {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( );}
while 0
```

在适当的地方，<asm/system.h> 定义这些宏以使用更快完成任务的架构特定指令。注意，只有少数架构定义了 set_rmb。（do...while 构造是一个标准的 C 习语，使得扩展的宏在所有上下文中都像正常的 C 语句一样工作。）

9.2. 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，至少在部分时间内是这样。这一部分将介绍用于使用 I/O 端口的各种函数；我们也会涉及一些可移植性问题。

在 x86 架构上，I/O 端口是一个单独的地址空间，与内存地址空间分开。许多其他架构（包括所有现代的 RISC 架构）没有这样的设备，它们通过特殊的内存地址来访问 I/O 设备。为了在所有架构上都能工作，Linux 提供了一组函数，用于访问 I/O 端口或者它们的内存映射等效物。

9.2.1. I/O 端口分配

正如你可能预期的那样，在开始对 I/O 端口进行操作之前，你应该首先确保你对这些端口有独占访问权。内核提供了一个注册接口，允许你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：

```c
#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，你希望使用从 first 开始的 n 个端口。name 参数应该是你的设备的名称。如果分配成功，返回值是非 NULL。如果从 request_region 返回 NULL，你将无法使用所需的端口。

所有的端口分配都会出现在 /proc/ioports 中。如果你无法分配需要的一组端口，那么这是查看谁先到达的地方。

当你完成一组 I/O 端口的使用（可能在卸载模块时），它们应该用以下方式返回给系统：

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

还有一个函数允许你的驱动程序检查给定的一组 I/O 端口是否可用：

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

在这里，如果给定的端口不可用，返回值是一个负的错误代码。这个函数已经被弃用，因为它的返回值不能保证分配是否会成功；检查和后来的分配不是一个原子操作。我们在这里列出它，因为还有一些驱动程序在使用它，但你应该始终使用 request_region，它执行必要的锁定，以确保分配以安全的、原子的方式进行。

9.2.2. 操作 I/O 端口

在驱动程序请求了它在活动中需要使用的 I/O 端口范围后，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。通常你不能像通常在系统内存访问中那样混合使用它们。

因此，C 程序必须调用不同的函数来访问不同大小的端口。如前一节所示，只支持内存映射 I/O 寄存器的计算机架构通过将端口地址重新映射到内存地址来伪造端口 I/O，内核隐藏了这些细节，以便于移植。Linux 内核头文件（具体来说，是依赖于架构的头文件 <asm/io.h>）定义了以下内联函数来访问 I/O 端口：

```c
unsigned inb(unsigned port);

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

读取或写入字节端口（宽度为八位）。对于一些平台，port 参数被定义为 unsigned long，对于其他平台，被定义为 unsigned short。inb 的返回类型在不同的架构中也是不同的。

```c
unsigned inw(unsigned port);

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

这些函数访问 16 位端口（宽度为一个字）；在为 S390 平台编译时，它们不可用，因为 S390 只支持字节 I/O。

```c
unsigned inl(unsigned port);

void outl(unsigned longword, unsigned port);
```

这些函数访问 32 位端口。longword 被声明为 unsigned long 或 unsigned int，取决于平台。像字 I/O 一样，S390 不支持“长” I/O。

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，你可以（例如）将两个 8 位写入绑定到一个单独的 16 位操作。例如，这适用于 PC 视频板。但通常，你不能依赖这个特性。

- 从现在开始，当我们在没有进一步类型规定的情况下使用 unsigned 时，我们指的是一个与架构相关的定义，其确切性质并不重要。这些函数几乎总是可移植的，因为编译器在赋值时自动转换值——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，这样的转换就不会丢失信息。我们在本章中坚持这种“不完全类型”的约定。

9.2.3. 从用户空间的 I/O 存取

刚刚描述的函数主要是供设备驱动程序使用的，但它们也可以在用户空间中使用，至少在 PC 类计算机上可以。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 和朋友们，应该满足以下条件：

- 程序必须使用 `-O` 选项编译，以强制扩展内联函数。
- 必须使用 `ioperm` 或 `iopl` 系统调用来获取对端口进行 I/O 操作的权限。`ioperm` 获取对单个端口的权限，而 iopl 获取对整个 I/O 空间的权限。这两个函数都是 x86 特有的。
- 程序必须以 root 身份运行才能调用 ioperm 或 iopl。* 或者，它的一个祖先必须以 root 身份运行以获得端口访问权限。

如果主机平台没有 `ioperm` 和 `iopl` 系统调用，用户空间仍然可以通过使用 /dev/port 设备文件来访问 I/O 端口。然而，请注意，文件的含义非常依赖于平台，除了 PC 之外，可能对其他任何事情都没有用。

示例源代码 `misc-progs/inp.c` 和 `misc-progs/outp.c` 是一个最小的工具，用于从用户空间的命令行读取和写入端口。它们期望被安装在多个名称下（例如，inb，inw，和 inl，并根据用户调用的名称操作字节，字，或长端口）。它们在 x86 下使用 ioperm 或 iopl，在其他平台上使用 /dev/port。

如果你想冒险并在没有获取明确权限的情况下玩硬件，可以将程序设置为 setuid root。然而，请不要在生产系统上安装它们为 setuid；它们在设计上就是一个安全漏洞。

9.2.4. 字串操作

除了单次的 in 和 out 操作外，一些处理器实现了特殊的指令，用于将一系列字节、字或长字从同一大小的 I/O 端口传输到该端口，并从该端口传输出去。这些是所谓的字符串指令，它们比 C 语言循环更快地执行任务。以下宏实现了字符串 I/O 的概念，要么通过使用单个机器指令，要么通过执行紧密的循环，如果目标处理器没有执行字符串 I/O 的指令。当为 S390 平台编译时，这些宏根本没有定义。这不应该是一个可移植性问题，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。

字符串函数的原型是：

```c
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 字节。数据从单个端口 port 读取或写入。

```c
void insw(unsigned port, void *addr, unsigned long
count);

void outsw(unsigned port, void *addr, unsigned long
count);
```

读取或写入单个 16 位端口的 16 位值。

```c
void insl(unsigned port, void *addr, unsigned long
count);

void outsl(unsigned port, void *addr, unsigned long
count);
```

读取或写入单个 32 位端口的 32 位值。

在使用字符串函数时，有一件事需要记住：它们将一个直接的字节流移动到端口或从端口移动出来。当端口和主机系统有不同的字节排序规则时，结果可能会令人惊讶。使用 inw 读取端口时，如果需要，会交换字节，使读取的值与主机排序匹配。相反，字符串函数不执行这种交换。

9.2.5. 暂停 I/O

一些平台——最值得注意的是 i386——在处理器试图过快地向总线传输数据或从总线传输数据时可能会出现问题。当处理器相对于外围总线（在这里想想 ISA）超频时，问题可能会出现，当设备板太慢时，问题也可能会出现。解决方案是在每个 I/O 指令后插入一个小的延迟，如果后面还有这样的指令。在 x86 上，通过对端口 0x80（通常但不总是未使用）执行 outb 指令，或者通过忙等待来实现暂停。有关详细信息，请查看平台的 asm 子目录下的 io.h 文件。

如果你的设备丢失了一些数据，或者你担心它可能会丢失一些数据，你可以使用暂停函数来代替正常的函数。暂停函数与前面列出的函数完全相同，但它们的名称以 `_p` 结尾；它们被称为 `inb_p, outb_p` 等。这些函数被定义为大多数支持的架构，尽管它们通常扩展为与非暂停 I/O 相同的代码，因为如果架构运行的是一个相当现代的外围总线，就没有必要进行额外的暂停。

```c
unsigned char inb_p(unsigned port);

void outb_p(unsigned char value, unsigned port);
```

这些函数从给定的端口读取或写入一个字节，并在操作后插入一个小的延迟。

9.2.6. 平台依赖性

I/O 指令由于其性质，高度依赖处理器。因为它们处理处理器如何移动数据进出的细节，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是平台依赖的。

你可以通过回顾函数列表，看到其中的一个不兼容性，数据类型，其中的参数根据平台之间的架构差异进行了不同的类型化。例如，端口在 x86 上是 unsigned short（其中处理器支持 64 KB 的 I/O 空间），但在其他平台上是 unsigned long，其端口只是与内存相同的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。我们不会详细讨论这些差异，因为我们假设你不会为特定的系统编写设备驱动程序，而不了解底层硬件。相反，这里是内核支持的架构的能力概述：

IA-32（x86） x86_64 架构支持本章描述的所有函数。端口号是 unsigned short 类型。

IA-64（Itanium） 支持所有函数；端口是 unsigned long（并且是内存映射的）。字符串函数在 C 中实现。

Alpha 支持所有函数，端口是内存映射的。在不同的 Alpha 平台中，端口 I/O 的实现是不同的，这取决于它们使用的芯片组。字符串函数在 C 中实现，并在 arch/alpha/lib/io.c 中定义。端口是 unsigned long。

ARM 端口是内存映射的，支持所有函数；字符串函数在 C 中实现。端口是 unsigned int 类型。

Cris 这种架构甚至在模拟模式下都不支持 I/O 端口抽象；各种端口操作被定义为完全不做任何事情。

M68k M68k-nommu 端口是内存映射的。支持字符串函数，端口类型是 unsigned char *。

MIPS MIPS64 MIPS 端口支持所有函数。字符串操作是用紧密的汇编循环实现的，因为处理器缺乏机器级的字符串 I/O。端口是内存映射的；它们是 unsigned long。

PA-RISC 支持所有函数；在基于 PCI 的系统上，端口是 int，在 EISA 系统上是 unsigned short，除了字符串操作，它们使用 unsigned long 端口号。

PowerPC PowerPC64 支持所有函数；在 32 位系统上，端口类型为 `unsigned char *`，在 64 位系统上为 unsigned long。

S390 与 M68k 类似，这个平台的头文件只支持字节宽的端口 I/O，没有字符串操作。端口是 char 指针，并且是内存映射的。

Super-H 端口是 unsigned int（内存映射的），支持所有函数。

SPARC SPARC64 再次，I/O 空间是内存映射的。定义了端口函数的版本，用于处理 unsigned long 端口。

好奇的读者可以从 io.h 文件中提取更多信息，这些文件有时除了我们在本章中描述的那些之外，还定义了一些架构特定的函数。但是要警告的是，这些文件中的一些阅读起来相当困难。

值得注意的是，除了 x86 系列之外，没有其他处理器具有不同的端口地址空间，尽管支持的几个系列都配备了 ISA 和/或 PCI 插槽（这两个总线都实现了独立的 I/O 和内存地址空间）。

此外，一些处理器（尤其是早期的 Alpha）缺乏一次移动一或两个字节的指令。* 因此，它们的外围芯片组通过将它们映射到内存地址空间中的特殊地址范围来模拟 8 位和 16 位的 I/O 访问。因此，对同一端口执行的 inb 和 inw 指令是通过对不同地址进行两次 32 位内存读取来实现的。幸运的是，所有这些都被隐藏在本节描述的宏的内部，但我们认为这是一个值得注意的有趣特性。如果你想进一步探索，可以在 include/asm-alpha/core_lca.h 中寻找示例。

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网上下载为 PDF 格式。

9.3. 一个 I/O 端口例子

我们用来展示设备驱动程序中的端口 I/O 的示例代码操作的是通用数字 I/O 端口；这样的端口在大多数计算机系统中都可以找到。

数字 I/O 端口，在其最常见的形式中，是一个字节宽的 I/O 位置，可以是内存映射的或端口映射的。当你向输出位置写入一个值时，输出引脚上看到的电信号会根据被写入的各个位而改变。当你从输入位置读取一个值时，输入引脚上看到的当前逻辑级别将作为各个位值返回。

这种 I/O 端口的实际实现和软件接口从系统到系统是不同的。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，哪些引脚用作输出，另一个可以实际读取或写入逻辑级别。然而，有时候，事情更简单，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”）；在所有个人计算机上找到的并行端口就是这样一个不太通用的 I/O 端口。无论哪种方式，I/O 引脚都可以被我们即将介绍的示例代码使用。

9.3.1. 并口纵览

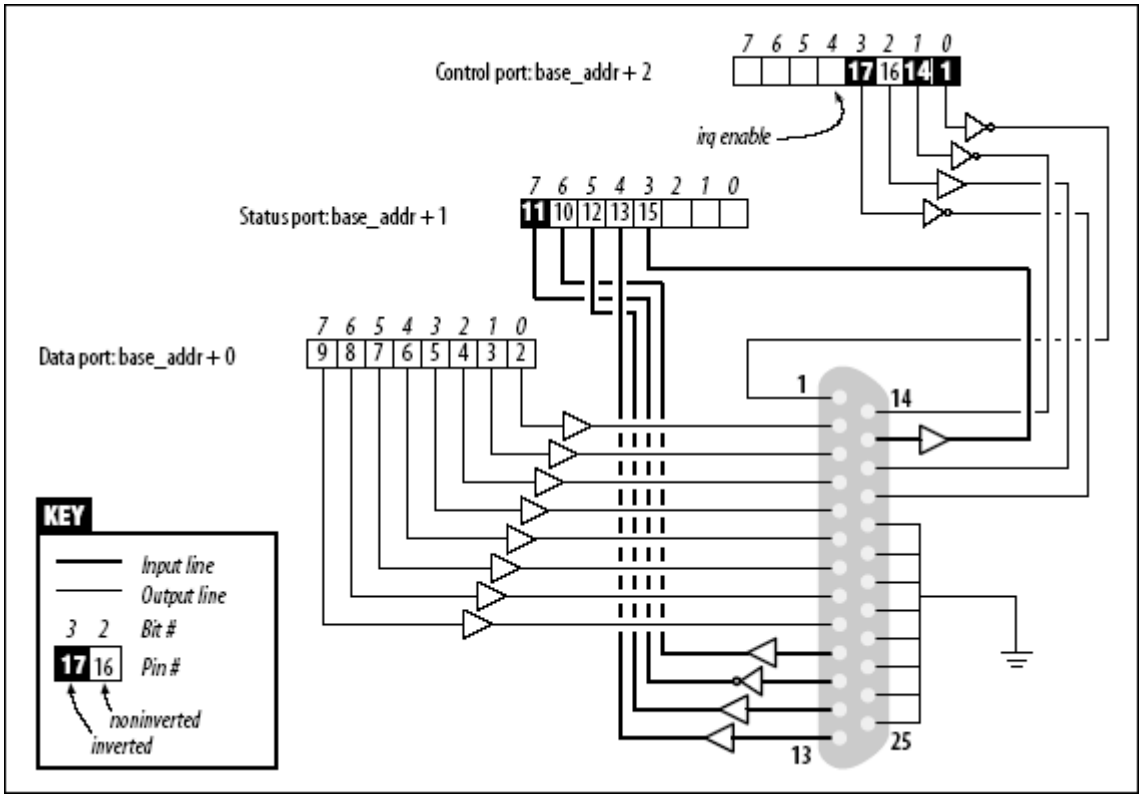
因为我们预计大多数读者使用的是被称为“个人计算机”的 x86 平台，我们认为有必要解释 PC 并行端口的设计。并行端口是在个人计算机上运行数字 I/O 示例代码的首选外设

接口。虽然大多数读者可能有并行端口规格可用，但我们在这里为您总结一下，以方便您。

并行接口，在其最小配置（我们忽略 ECP 和 EPP 模式）是由三个 8 位端口组成。PC 标准将第一个并行接口的 I/O 端口开始于 0x378，第二个开始于 0x278。第一个端口是一个双向数据寄存器；它直接连接到物理连接器的 2-9 引脚。第二个端口是一个只读状态寄存器；当并行端口被用于打印机时，这个寄存器报告打印机状态的几个方面，如在线、缺纸或忙。第三个端口是一个只输出的控制寄存器，它控制是否启用中断。

并行通信使用的信号级别是标准的晶体管-晶体管逻辑（TTL）级别：0 和 5 伏，逻辑阈值约为 1.2 伏。你可以依赖端口至少满足标准 TTL LS 电流等级，尽管大多数现代并行端口在电流和电压等级上都做得更好。

位规格在图 9-1 中概述。你可以访问 12 个输出位和 5 个输入位，其中一些在信号路径中逻辑上被反转。唯一没有关联信号引脚的位是端口 2 的位 4（0x10），它启用并行端口的中断。我们在第 10 章中使用这个位作为我们实现中断处理程序的一部分。



- 并行连接器并未与计算机的内部电路隔离，如果你想直接将逻辑门连接到端口，这是有用的。但你必须小心正确地进行接线；当你使用自己的定制电路时，除非你在电路中添加光耦，否则并行端口电路很容易被损坏。如果你担心会损坏你的主板，你可以选择使用插入式并行端口。

9.3.2. 一个例子驱动

我们介绍的驱动程序叫做 short（Simple Hardware Operations and Raw Tests）。它所做的只是读取和写入一些 8 位端口，从你在加载时选择的那个开始。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（具有唯一的次要号）访问不同的端口。short 驱动程序不做任何有用的事情；它只是将对端口的单个指令隔离为外部使用。如果你不习惯于端口 I/O，你可以使用 short 来熟悉它；你可以测量通过端口传输数据所需的时间或玩其他游戏。

为了让 short 在你的系统上工作，它必须能够自由访问底层硬件设备（默认情况下，是并行接口）；因此，没有其他驱动程序可能已经分配了它。大多数现代发行版将并行端口驱动程序设置为只在需要时加载的模块，所以对 I/O 地址的争用通常不是问题。然而，如果你从 short 得到一个“无法获取 I/O 地址”的错误（在控制台或系统日志文件中），可能有其他驱动程序已经占用了端口。快速查看 /proc/ioports 通常可以告诉你哪个驱动程序在阻碍。如果你不使用并行接口，同样的警告也适用于其他 I/O 设备。

从现在开始，我们只是简单地提到“并行接口”以简化讨论。然而，你可以在加载时设置基础模块参数，将 short 重定向到其他 I/O 设备。这个特性允许示例代码在任何你可以访问到通过 outb 和 inb 可访问的数字 I/O 接口的 Linux 平台上运行（即使实际的硬件在所有平台上都是内存映射的，除了 x86）。稍后，在“使用 I/O 内存”一节中，我们展示了如何使用通用的内存映射数字 I/O 也可以使用 short。

要观察并行连接器上发生的情况，如果你有一点硬件工作的倾向，你可以将一些 LED 焊接到输出引脚上。每个 LED 应该串联连接到一个 1-KΩ 的电阻，然后连接到地引脚（除非，当然，你的 LED 内置了电阻）。如果你将一个输出引脚连接到一个输入引脚，你将生成你自己的输入，可以从输入端口读取。

注意，你不能只是将打印机连接到并行端口，然后看到发送到 short 的数据。这个驱动程序实现了对 I/O 端口的简单访问，并没有执行打印机需要操作数据的握手。在下一章中，我们展示了一个能够驱动并行打印机的示例驱动程序（称为 shortprint）；然而，那个驱动程序使用中断，所以我们现在还不能使用它。

如果你打算通过将 LED 焊接到 D 类连接器上来查看并行数据，我们建议你不要使用引脚 9 和 10，因为我们稍后会将它们连接在一起，以运行在第 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（除非在加载时更改，否则为 0x378）的 8 位端口。/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。使用内存屏障指令来确保输出操作实际发生，而不是被优化掉：

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

你可以运行以下命令来点亮你的 LED：

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

每个 LED 监视输出端口的单个位。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够长，才能被你的眼睛感知。因此，我们建议你通过向 echo 传递 -n 选项来防止自动插入尾随的换行符。

读取是由一个类似的函数执行的，它围绕 inb 而不是 outb 构建。为了从并行端口读取“有意义”的值，你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，你会读取到一连串相同的字节。如果你选择从输出端口读取，你最有可能得到的是最后写入到端口的值（这适用于并行接口和大多数常用的其他数字 I/O 电路）。因此，那些不愿意拿出焊接铁的人可以通过运行如下命令来读取端口 0x378 上的当前输出值：

```c
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 short 设备有三种变体：/dev/short0 执行刚刚显示的循环，/dev/short0p 使用 outb_p 和 inb_p 替代“快速”函数，/dev/short0s 使用字符串指令。有八种这样的设备，从 short0 到 short7。虽然 PC 并行接口只有三个端口，但如果你使用不同的 I/O 设备来运行你的测试，你可能需要更多的端口。

short 驱动程序执行的硬件控制是绝对的最小限度，但足以显示如何使用 I/O 端口指令。感兴趣的读者可能想看看 parport 和 parport_pc 模块的源代码，看看为了支持一

系列设备（打印机、磁带备份、网络接口）在并行端口上，这个设备在现实生活中可能有多复杂。

9.4. 使用 I/O 内存

尽管在 x86 世界中 I/O 端口非常流行，但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。这两者都被称为 I/O 内存，因为寄存器和内存之间的区别对软件来说是透明的。

I/O 内存只是设备通过总线向处理器提供的一系列类似 RAM 的位置。这个内存可以用于多种目的，如保存视频数据或以太网数据包，以及实现像 I/O 端口一样的设备寄存器（即，它们与读写它们相关的副作用）。

访问 I/O 内存的方式取决于使用的计算机架构、总线和设备，尽管原则在所有地方都是相同的。本章的讨论主要涉及 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 的详细讨论将推迟到第 12 章。

根据使用的计算机平台和总线，I/O 内存可能通过页表访问，也可能不通过。当访问通过页表时，内核必须首先安排物理地址对你的驱动程序可见，这通常意味着你必须在进行任何 I/O 操作之前调用 ioremap。如果不需要页表，I/O 内存位置看起来很像 I/O 端口，你可以使用适当的包装函数来读写它们。

无论是否需要 ioremap 来访问 I/O 内存，都不鼓励直接使用指向 I/O 内存的指针。尽管（如在“ I/O 端口和 I/O 内存”一节中介绍的那样）在硬件级别，I/O 内存像普通 RAM 一样被寻址，但在“ I/O 寄存器和常规内存”一节中概述的额外注意事项建议避免使用普通指针。用于访问 I/O 内存的包装函数在所有平台上都是安全的，并且在直接指针解引用可以执行操作时，它们会被优化掉。

因此，尽管在 x86 上解引用指针是有效的（目前），但是不使用适当的宏会妨碍驱动程序的可移植性和可读性。

9.4.1. I/O 内存分配和映射

在使用之前，必须先分配 I/O 内存区域。内存区域分配的接口（在 <linux/ioport.h> 中定义）是：

```c
struct resource *request_mem_region(unsigned long start,
unsigned long len,

                                    char *name);
```

此函数从 start 开始，分配 len 字节的内存区域。如果一切顺利，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。

不再需要的内存区域应该被释放：

```c
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：

```c
int check_mem_region(unsigned long start, unsigned long
len);
```

但是，与 check_region 一样，这个函数是不安全的，应该避免使用。

分配 I/O 内存不是访问该内存所需的唯一步骤。你还必须确保这个 I/O 内存已经对内核可访问。获取 I/O 内存不仅仅是解引用一个指针的问题；在许多系统上，I/O 内存根本不能以这种方式直接访问。所以必须首先设置一个映射。这就是 ioremap 函数的作用，它在第 8 章的“vmalloc 和 Friends”一节中介绍过。这个函数专门设计用来为 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。但是，请记住，从 ioremap 返回的地址不应直接解引用；相反，应使用内核提供的访问器函数。

在我们深入这些函数之前，我们最好先回顾一下 ioremap 的原型，并介绍一些我们在上一章中略过的细节。

函数按照以下定义调用：

```c
#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，你会注意到新函数 ioremap_nocache。我们在第 8 章没有涉及它，因为它的含义绝对与硬件相关。引用内核头文件中的一句话："如果某些控制寄存器在这样的区域中，而且不希望进行写组合或读缓存，那么它就很有用。"实际上，这个函数的实现在大多数计算机平台上与 ioremap 相同：在所有 I/O 内存都已经通过非缓存地址可见的情况下，没有理由实现一个单独的，非缓存版本的 ioremap。

9.4.2. 存取 I/O 内存

在某些平台上，你可能可以将 ioremap 的返回值用作指针。但这种用法并不可移植，而且，内核开发人员越来越多地在努力消除这种用法。访问 I/O 内存的正确方式是通过一组为此目的提供的函数（通过 <asm/io.h> 定义）。

要从 I/O 内存中读取，使用以下之一：

```c
unsigned int ioread8(void *addr);

unsigned int ioread16(void *addr);

unsigned int ioread32(void *addr);
```

这里，addr 应该是从 ioremap 获得的地址（可能有一个整数偏移）；返回值是从给定的 I/O 内存读取的内容。

有一组类似的函数用于写入 I/O 内存：

```c
void iowrite8(u8 value, void *addr);

void iowrite16(u16 value, void *addr);

void iowrite32(u32 value, void *addr);
```

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

```c
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);
```

ioread 函数从给定的 addr 读取 count 个值到给定的 buf，而 iowrite 函数从给定的 buf 写入 count 个值到给定的 addr。请注意，count 是以正在写入的数据的大小表示的；ioread32_rep 从 buf 开始读取 count 个 32 位值。

上述函数都执行给定 addr 的 I/O。如果你需要操作一块 I/O 内存，你可以使用以下之一：

```c
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 库对应项。

如果你阅读内核源码，你会看到在使用 I/O 内存时有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。其中一个原因是，它们不如上述函数安全，因为它们不执行同样的类型检查。尽管如此，我们在这里描述它们：

```c
unsigned readb(address);

unsigned readw(address);

unsigned readl(address);
```

这些宏用于从 I/O 内存中检索 8 位、16 位和 32 位的数据值。

```c
void writeb(unsigned value, address);

void writew(unsigned value, address);

void writel(unsigned value, address);
```

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

一些 64 位平台还提供 readq 和 writeq，用于在 PCI 总线上进行四字（8 字节）内存操作。四字的名称是从所有真正的处理器都有 16 位字的时代遗留下来的。实际上，用

于 32 位值的 L 命名也已经不正确了，但是重命名所有东西会使事情更加混乱。

9.4.3. 作为 I/O 内存的端口

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。导出到处理器的寄存器在任何情况下都是相同的，但访问方法是不同的。为了让处理这种硬件的驱动程序的生活更轻松，以及最小化 I/O 端口和内存访问之间的明显差异，2.6 内核提供了一个名为 ioport_map 的函数：

```c
void *ioport_map(unsigned long port, unsigned int count);
```

此函数重新映射 count 个 I/O 端口，并使它们看起来像 I/O 内存。从那时起，驱动程序可以在返回的地址上使用 ioread8 和朋友们，并忘记它正在使用 I/O 端口。

当不再需要此映射时，应撤销此映射：

```c
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，然后才能以这种方式重新映射。

9.4.4. 重用 short 为 I/O 内存

之前介绍的用于访问 I/O 端口的简短示例模块，也可以用于访问 I/O 内存。为此，你必须在加载时告诉它使用 I/O 内存；此外，你需要更改基地址，使其指向你的 I/O 区域。例如，我们是这样使用 short 来点亮 MIPS 开发板上的调试 LED 的：

```c
mips.root# ./short_load use_mem=1 base=0xb7ffffc0

mips.root# echo -n 7 > /dev/short0
```

使用 short 进行 I/O 内存的操作与其用于 I/O 端口的操作相同。

以下片段显示了 short 在写入内存位置时使用的循环：

```c
while (count--) {
    iowrite8(*ptr++, address);
    wmb( );
}
```

注意这里使用了写内存屏障。因为 iowrite8 在许多架构上可能变成直接赋值，所以需要内存屏障来确保写入按预期的顺序发生。

short 使用 inb 和 outb 来展示如何完成这个操作。然而，对于读者来说，将 short 更改为使用 ioport_map 重新映射 I/O 端口，并大大简化其余的代码，将是一个直接的练习。

9.4.5. 在 1 MB 之下的 ISA 内存

最知名的 I/O 内存区域之一是个人计算机上找到的 ISA 范围。这是 640 KB (0xA0000) 和 1 MB (0×100000) 之间的内存范围。因此，它就出现在常规系统 RAM 的中间。这种定位可能看起来有点奇怪；这是 1980 年代初做出的一个决定的产物，当时 640 KB 的内存看起来比任何人都能使用的更多。

这个内存范围属于非直接映射类别的内存。*你可以使用之前解释的 `short` 模块在该内存范围中读/写几个字节，也就是说，通过在加载时设置 use_mem。

尽管 ISA I/O 内存只存在于 x86 类计算机中，但我们认为花几句话和一个示例驱动程序来描述它是值得的。

我们不打算在这一章讨论 PCI 内存，因为它是最干净的 I/O 内存：一旦你知道物理地址，你就可以简单地重新映射和访问它。PCI I/O 内存的“问题”在于，它不适合作为这一章的工作示例，因为我们无法预先知道你的 PCI 内存映射到的物理地址，或者访问这些范围中的任何一个是否安全。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（部分示例源码）。事实上，这个被称为 silly，作为 Simple Tool for Unloading and Printing ISA Data 的首字母缩写，或者类似的东西。

该模块补充了 short 的功能，通过提供对整个 384-KB 内存空间的访问，并显示所有不同的 I/O 函数。它具有四个设备节点，使用不同的数据传输函数执行相同的任务。silly 设备像 /dev/mem 一样，作为 I/O 内存的窗口。你可以读取和写入数据，并 lseek 到任意的 I/O 内存地址。

由于 silly 提供了对 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，人们可以简单地将一个指针赋值给感兴趣的 ISA 地址，然后直接解引用它。然而，在现代世界，我们必须与虚拟内存系统一起工作，并首先重新映射内存范围。这个映射是通过 ioremap 完成的，就像之前为 short 解释的那样：

```c
#define ISA_BASE    0xA0000

#define ISA_MAX     0x100000  /* for general memory
access */

    /* this line appears in silly_init */

    io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap 返回一个可以与 ioread8 和其他在“访问 I/O 内存”部分解释的函数一起使用的指针值。

让我们回顾一下我们的示例模块，看看这些函数可能如何使用。/dev/sillyb，具有次要号 0，使用 ioread8 和 iowrite8 访问 I/O 内存。以下代码显示了 read 的实现，它使地址范围 0xA0000-0xFFFFF 作为虚拟文件在范围 0-0x5FFFF 中可用。read 函数是作为一个 switch 语句结构化的，针对不同的访问模式；这是 sillyb 的情况：

```c
case M_8:
  while (count) {
      *ptr = ioread8(add);
      add++;
      count--;
      ptr++;
  }
  break;
```

接下来的两个设备是 /dev/sillyw（次要号 1）和 /dev/sillyl（次要号 2）。它们像 /dev/sillyb 一样，只是它们使用 16 位和 32 位函数。这是 sillyl 的写实现，再次作为一个 switch 的一部分：

```c
case M_32:
  while (count ⩾ 4) {
      iowrite32(*(u32 *)ptr, add);
      add += 4;
      count -= 4;
      ptr += 4;
  }
  break;
```

最后一个设备是 /dev/sillycp（次要号 3），它使用 `memcpy_*io` 函数执行相同的任务。这是其 read 实现的核心：

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

因为 ioremap 被用来访问 ISA 内存区域，所以当模块被卸载时，silly 必须调用 iounmap：

```c
iounmap(io_base);
```

- 实际上，这并不完全正确。内存范围如此小，使用频率如此高，以至于内核在启动时构建页面表来访问这些地址。然而，用于访问它们的虚拟地址并不是物理地址，

因此无论如何都需要 ioremap。

9.4.6. isa_readb 和其友

查看内核源码会发现另一组名为 isa*readb 等的例程。实际上，刚刚描述的每个函数都有一个 isa* 等效项。这些函数提供了对 ISA 内存的访问，无需单独的 ioremap 步骤。然而，内核开发者的说法是，这些函数旨在作为临时的驱动程序移植辅助工具，未来可能会消失。因此，你应该避免使用它们。

9.5. 快速参考

本章介绍下列与硬件管理相关的符号:

```
#include <linux/kernel.h>
void barrier(void)
```

这个"软件"内存屏蔽要求编译器对待所有内存是跨这个指令而非易失的.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

硬件内存屏障. 它们请求 CPU(和编译器)来检查所有的跨这个指令的内存读, 写, 或都有.

```
#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);
```

用来读和写 I/O 端口的函数. 它们还可以被用户空间程序调用, 如果它们有正当的权限来存取端口.

```
unsigned inb_p(unsigned port);
```

如果在一次 I/O 操作后需要一个小延时, 你可以使用在前一项中介绍的这些函数的 6 个暂停对应部分; 这些暂停函数有以 _p 结尾的名子.

```
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);
```

这些"字串函数"被优化为传送数据从一个输入端口到一个内存区, 或者其他的方式. 这些传送通过读或写到同一端口 count 次来完成.

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start,
unsigned long len, char *name);
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. 这个检查函数成功返回 0 并且在错误时小于 0.

```
struct resource *request_mem_region(unsigned long start,
unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long
len);
int check_mem_region(unsigned long start, unsigned long
len);
```

为内存区处理资源分配的函数

```
#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 重映射一个物理地址范围到处理器的虚拟地址空间, 使它对内核可用. iounmap 释放映射当不再需要它时.

```
#include <asm/io.h>
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

用来使用 I/O 内存的存取者函数.

```
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.

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(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_toio(dest, source, nbytes);
```

旧的, 类型不安全的存取 I/O 内存的函数.

```
void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);
```

一个想对待 I/O 端口如同它们是 I/O 内存的驱动作者, 可以传递它们的端口给 ioport_map. 这个映射应当在不需要的时候恢复(使用 ioport_unmap)