第十五章　外围总线概览

在第八章“硬件管理”中，我们介绍了最低级的硬件控制，本章提供一个较高级的总线体系结构的概览。总线由电气接口和编程接口组成。在这一章，我打算介绍编程接口。
本章覆盖了几种总线体系结构。不过，基本重点是访问 PCI 外围的核心功能，因为近来，PCI 总线是最常用的外围总线，也是核心支持最好的总线。

PCI 接口

尽管很多计算机用户认为 PCI（外围部件互连，Peripheral Component Interconnect）是布局电气线路的一种方法，但实际上，它是一组完全的规范，定义了计算机的不同部分是如何交互的。
PCI 规范覆盖了与计算机接口相关的绝大多数方面。我不打算在这里全部介绍，在本节中，我主要关心一个 PCI 驱动程序是如何找到它的硬件，并获得对它的访问的。在第二章“构造和运行模块”的“自动和手工配置”一节，及在第九章“中断处理”的“自动检测中断号”一节中讨论过的探测技术同样可以应用于 PCI 设备，但规范还提供了探测的另外办法。
PCI 结构被设计来替代 ISA 标准，由三个主要目标：在计算机和其外围之间传送数据时有更高的性能，尽可能地做到平台无关性，使在系统中增减外围设备得到简化。
PCI 通过使用比 ISA 高的时钟频率来获得更高的性能；它的时钟运行在 25 或 33MHZ（实际时钟是系统时钟的几分之一的整数倍），而且马上就会游 66MHZ 的扩展。另外，它被装配在 32 位的数据总线上，64 位的扩展正在规范中。平台无关性一直是计算机总线的一个设计目标，这是 PCI 的尤其重要的一个特征，因为 PC 世界一直以来总是被处理器特定的标准所主宰。
不过对驱动程序作者来说，最要紧的是对接口板自动检测的支持。PCI 设备是无跳线的（与大多数 ISA 外围不同），并且在引导时被自动配置。因此，设备驱动程序必须能访问设备上的配置信息来完成初始化。这些情形都不需要任何探测。

PCI 寻址

每个外围由一个总线号、一个设备号、和一个功能号确定。虽然 PCI 规范允许一个系统最多拥有 256 条总线，但 PC 只有一条。每条总线最多带 32 个设备，但每个设备可以是最多个功能的多功能板（如一个音频设备带一个 CD-ROM 驱动器）。每个功能可以由一个 16 位的键或两个 8 位的键确定。Linux 核心采用后一种方法。
每个外围板子的硬件电路回答与三个地址空间相关的询问：内存位置，I/O端口，和配置寄存器。前两个地址空间由PCI总线上的所有设备共享（也就是说，当你访问一个内存位置，所有的设备都将同时看到这个总线周期）。而配置空间则利用“地理寻址”，每个槽有一个配置事务的私用使能线，PCI控制器一次访问一个板子，不会有地址冲突。考虑到驱动程序，内存和I/O是以通常的*inb*,*memcpy*等来访问。而配置事务则通过调用特定的核心函数访问配置寄存器来完成。至于中断，每个PCI设备有 4 个中断管脚，它们到处理器中断线的路由是主板的任务；PCI中断可以设计为共享的，这样即使是一个有限中断线的处理器也能带很多PCI接口板。
PCI 总线的 I/O 空间使用 32 位的地址总线（这样就是 4GB 的 I/O 端口），而内存空间则可以用 32 位或 64 位地址访问。地址对每个设备来说应该是唯一的，但也有可能有两个设备错误

地映射到同一个地址，使得哪个都不能被访问。一个好消息是接口卡提供的每个内存和 I/O 地址区段都可以通过配置事务重映射。这就是设备可以在引导时被初始化从而避免地址冲突的机制这些区段当前映射到的地址可以从配置空间读出，因此 Linux 驱动程序可以不通过探测就访问其设备。一旦配置寄存器被读出，驱动程序就可以安全的访问它的硬件。

PCI配置空间由每个设备函数 256 个字节构成，配置寄存器的布局是标准化的。配置空间有四个字节含有一个唯一的函数ID，因此驱动程序可以通过在外围查找特定的ID来B确定它的设备*。总之，每个设备板子被地理寻址以取得它的配置寄存器；这个信息可以用来确定这个板子或采取进一步动作。

从前面的描述，应该清楚 PCI 接口标准比 ISA 的主要创新是配置地址空间。因此，除了通常的驱动程序代码外，PCI 驱动程序还需要访问配置空间的能力。

在本章的其余部分，我将使用单词“设备”来指一个设备功能，因为多功能板上的每个功能均是一个独立的实体。当我提到一个设备，我是指元组“总线号，设备号，功能号”。如前所述，每个元组在 Linux 中由两个 8 位数字表示。

引导时

让我们看一下 PCI 是如何工作的，从系统引导开始，因为那时设备被配置。

当 PCI 设备被加电时，硬件关闭。或者说，设备只对配置事务响应。加电时，设备没有映射到计算机地址空间的内存和 I/O 端口；所有其它的设备特定的特征，象中断线，也都被关闭。

幸运的是，每个 PCI 母板都装有懂得 PCI 的固件，根据平台的不同被称做 BIOS、NVRAM、或 PROM。固件提供对设备配置地址空间的访问，即使处理器的指令集不提供这样的能力。在系统引导时，固件对每个 PCI 外围执行配置事务，从而为它提供的任何地址区段分配一个安全的地方。到设备驱动程序访问设备时，它的内存和 I/O 区段已经被映射到处理器的地址空间。驱动程序可以改变这个缺省的分配，但它通常并不这样做，除非有一些设备相关的原因要求这样。

在 Linux 中，用户可以通过读*/proc/pci* 来查看 PCI 设备，这是个文本文件，系统中每个 PCI 板子有一项。下面是*/proc/pci* 中一项的例子：

（代码 344）

/proc/pci 中每一项是一个设备的设备无关特征的概述，如它的配置寄存器所描述的。例如，上面这一项告诉我们这个设备有板上内存，已被映射到地址 0xf1000000。一些古怪的细节的含义以后在我介绍过配置寄存器后将会清楚。

检测设备

如前面提到的，配置空间的布局是设备无关的。在这一节，我们将看看用来确定外围的配置寄存器。

PCI 设备有一个 256 字节的地址空间。前 64 个字节是标准化的，而其余的则是设备相关的。图 15-1 显示了设备无关配置空间的布局。

如图所示，有些 PCI 的配置寄存器是要求的，而有些则是可选的。每个 PCI 设备必须在必要寄存器中包含有意义的值，而可选寄存器的内容则以来与实际外围的能力。可选域并不使用，除非必要域的内容表明它们是有效的。这样，必要域断言了板子的能力，包括其它域可用与否。

有意思的是注意到 PCI 寄存器总是小印地安字节顺序的。尽管标准要设计为体系结构无关的，PCI 的设计者有时还是显示出对 PC 环境的偏见。驱动程序的作者应该留神字节顺序，

---

* 你可以在它自己的硬件手册中找到任何设备的ID。

特别是访问多字节的配置寄存器时；在 PC 上工作的代码可能在别的平台上就不行。Linux 的开发者已经注意了字节排序问题（见下一节“访问配置空间”），但这个问题还是要牢记在心。不幸的是，标准函数 *ntohs* 和 *ntohl* 都不能用，因为网络字节顺序与 PCI 顺序相反；在 Linux2.0 中没有标准函数将 PCI 字节顺序转换为主机字节顺序，每个用单个字节构成多字节值的驱动程序都应该特别小心地正确处理印地安字节序。核心版本 2.1.10 引入了几个函数来处理这些字节顺序问题，它们在第十七章“最近的发展”中“转换函数”一节介绍。

（图 15-1：标准化的 PCI 配置寄存器）

描述所有的配置项超出了本书的范围。通常，与设备一起发布的技术文档会描述它支持的寄存器。我们感兴趣的是驱动程序如何找到它的设备，以及它如何访问设备的配置空间。

三个 PCI 寄存器确定一个设备：销售商，设备 ID，和类。每个 PCI 外围把它自己的值放入这些只读寄存器，驱动程序可以用它们来查找设备。让我们更仔细地看看这些寄存器：

销售商

这个 16 位的寄存器确定硬件的生产商。例如，每个 Intel 的设备都会标上同样的销售商号，8086 hex（是个随即值？）。这样的号码有一个全球的注册，生产商必须申请一个唯一的号。

设备 ID

这是另一个 16 位寄存器，由生产商选择；不需要有官方的注册。这个 ID 通常与销售商 ID 成对出现，形成一个硬件设备的唯一的 32 位标志符。我将用单词“签名”来指销售商/设备 ID 对。一个设备驱动程序经常以来于签名来确定它的设备；驱动程序的作者从硬件文档中知道要寻找什么值。

类

每个外围设备都属于一个类。类寄存器是个 16 位的值，它的高八位确定 “基类”（或组）。例如，“以太网”和“令牌环”是属于“网络”组的两类，而“串行”和“并行”类属于“通信”组。有些驱动程序可以支持几种类似的设备，它们虽然有不同的签名，却属于同一类；这些驱动程序可以依赖于类寄存器来确定它们的外围，如以后所示。

下面的头文件，宏，以及函数都将被 PCI 驱动程序用来寻找它的硬件设备：

#include <linux/config.h>

驱动程序需要知道是否 PCI 函数在核心是可用的。通过包含这个头文件，驱动程序获得了对 CONFIG_宏的访问，包括 CONFIG_PCI(将在下面介绍)。从 1.3.73 以来，这个头文件包含在<linux/fs.h>中；如果想向后兼容，你必须把它显式地包含。

CONFIG_PCI

如果核心包括对 PCI BIOS 调用的支持，那么这个宏被定义。并不是每个计算机都有 PCI 总线，所以核心的开发者应该把 PCI 的支持做成编译时选项，从而在无 PCI 的计算机上运行 Linux 时节省内存。如果 CONFIG_PCI 没有定义，那么这个列表中其它的函数都不可用，驱动程序应使用预编译的条件语句将 PCI 支持全都排除在外，以避免加载时的“未定义符号”错。

#include <linux/bios32.h>

这个头文件声明了本节介绍的所有的原型，因此一定要被包含。这个头文件还定义了函数返回的错误代码的符号值。它在 1.2 和 2.0 之间没有改变，因此没有可移植性问题。

int pcibios_present(void)

由于 PCI 相关的函数在无 PCI 的计算机上毫无意义，*pcibios_present* 函数就是告诉驱动程序计算机是否支持 PCI；如果 BIOS 懂得 PCI，它返回一个为真布尔值。即使 CONFIG_PCI 被定义了，PCI 功能仍是一个运行时选项。因此，你在调用下面介绍的函

数之前要检查一下 *pcibios_present*，保证计算机支持 PCI。

```
#include <linux/pci.h>
```

这个头文件定义了下面函数使用的所有数值的符号名。并不是所有的设备 ID 都在这个文件中列出了，但你在为你的 ID，销售商，类定义宏之前，最好还是看看这个文件。注意这个文件一直在变大，因为不断有新设备的符号定义被加入。

```
int pcibios_find_device(unsigned short vendor, unsigned short id, unsigned short index,
                        unsigned char *bus, unsigned char *function);
```

如果 CONFIG_PCI 被定义了，并且 *pcibios_present* 也是真，这个函数被用来从 BIOS 请求关于设备的信息。销售商/ID 对确定设备。index 用来支持具有同样的销售商/ID 对的几个设备，下面将会解释。对这个函数的调用返回设备在总线上的位置以及函数指针。返回代码为 0 表示成功，非 0 表示失败。

```
int pcibios_find_class(unsigned int class_code, unsigned short index,
                       unsigned char *bus, unsigned char *function);
```

这个函数和上一个类似，但它寻找属于特定类的设备。参数 class_code 传递的形式为：16 位的类寄存器左移八位，这与 BIOS 接口使用类寄存器的方式有关。这次还是，返回代码为 0 表示成功，非 0 表示有错。

```
char *pcibios_strerror(int error);
```

这个函数用来翻译一个 PCI 错误代码（象 *pcibios_find_device* 返回的）为一个字符串。你也许在查找函数返回的即不是 PCIBIOS_SUCCESSFUL(0)，也不是 PCIBIOS_DEVICE_NOT_FOUND 时（这是当所有的设备都被找过以后所期望返回的错误代码），希望打印一条错误信息。

下面的代码是驱动程序在加载时检测设备所使用的典型代码。如上面所提到的，这个查找可以基于签名或者设备类。不管是哪种情况，驱动程序不许存储 bus 和 function 值，它们在后面确定设备时要用到。function 的前五位确定设备，后三位确定函数。

下面的代码中，每个设备特定的符号加前缀 jail_（另一个指令列表），大写或小写依赖于符号的种类。

如果驱动程序可以依赖于唯一的销售商/ID 对，下面的循环可以用来初始化驱动程序：

（代码 347）

（代码 348）

如果这个代码段只处理由 JAIL_VENDOR 和 JAIL_ID 确定的一类 PCI 设备，那么它是正确的。

不过，很多驱动程序非常灵活，能够同时处理 PCI 和 ISA 板子。在这种情况下，驱动程序仅在没有检测到 PCI 板子或 CONFIG_PCIBIOS 没有定义时才探测 ISA 设备。

使用 *pcibios_find_class* 要求 *jail_init_dev* 完成比例子中要多的工作。只要它找到了一个属于指定类的设备，这个函数就成功返回，但驱动程序还要确认其签名也是被支持的。这个任务通过一系列的条件语句完成，结果是抛弃很多不期望的设备。

有些 PCI 外围包含通用目的的 PCI 接口芯片和设备特定的电路。所有使用同样接口芯片的外围板子都有同样的签名，驱动程序必须进行额外的探测以保证它在处理正确的外围设备。因此，有时象 *jail_init_dev* 之类的函数必须准备好做一些设备特定的额外的检测，以抛弃那些可能有正确签名的设备。

访问配置空间

在驱动程序检测到设备后，它通常要对三个地址空间读或写：内存、端口和配置。特别地，

访问配置空间对驱动程序来说极为重要，以呢这是它发现设备被映射到内存和 I/O 空间什么地方的唯一的办法。
由于微处理器无法直接访问配置空间，计算机销售商必须提供一个办法来完成它。准确的实现因此是销售商相关的，与我们这里的讨论无关。幸运的是，这个事务的软件接口（下面描述）是标准化的，驱动程序或 Linux 核心都不需要知道它的细节。
至于驱动程序，配置空间可以通过 8 位、16 位、32 位的数据传送来访问。相关函数的原型在<linux/bios32.h>：

```
int pcibios_read_config_byte(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_word(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_dword(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
```

从由 bus 和 function 确定的设备的配置空间读取 1，2，4 个字节。参数 where 是从配置空间开始处的字节偏移。 从配置空间取出的值通过 ptr 返回，这些函数的返回值是错误代码。字和双字函数将刚从小印地安字节序读出的值转换为处理器本身的字节序，因此你并不需要处理字节序。

```
int pcibios_write_config_byte(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char val);
int pcibios_write_config_word(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned short val);
int pcibios_write_config_dword(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned int val);
```

向配置空间里写 1，2，4 个字节。设备仍由 bus 和 function 确定，要写的值由 val 传递。字和双字函数在向外围设备写之前将数值转换为小印地安字节序。
访问配置变量的最好办法是使用在<linux/pci.h>中定义的符号名。例如，下面的两行程序通过给 *pcibios_read_config_byte* 的 where 传递符号名来获取一个设备的修正 ID。

```
Unsigned char jail_get_revision(unsigned char bus, unsigned char fn)
{
    unsigned char *revision;

    pcibios_read_config_byte(bus,fn, PCI_REVISION_ID,&revision);
    return revision;
}
```

当访问多字节值时，程序远一定要记住字节序的问题。

看看一个配置快照

如果你向浏览你系统上 PCI 设备的配置空间，你可以编译并加载模块 *pci/pcidata.c*，它在 O’Reilly FTP 站点上提供的源文件中。
这个模块生成一个动态的*/proc/pcidata* 文件，包含有你的 PCI 设备配置空间的二进制快照。这个快照在文件每次被读时更新。*/proc/pcidata* 的大小被限制为 PAGE_SIZE 字节(这是动态 */proc* 文件的限制，在第四章“调试技术”中“使用*/proc* 文件系统”一节介绍过)。这样，它只列出前 PAGESIZE/256 个设备的配置内存，意味着 16 或 32 个设备（也许对你的系统已经够了）。我选择把*/proc/pcidata* 作成二进制文件，而不是象其它*/proc* 文件那样是文本的，就

是因为这个大小限制。
pcidata 的另一个限制是它只扫描系统的第一条 PCI 总线。如果你的系统有到其它 PCI 总线的桥，*pcidata* 将忽略它们。
在*/proc/pcidata* 中设备出现的顺序与*/proc/pci* 中相反。这是因为*/proc/pci* 读的是一个从头部生长的链表，而*/proc/pcidata* 则是一个简单的查找循环，它按照取到的顺序将所有的东西输出。
例如，我的抓图器在*/proc/pcidata* 的第二个出现，（目前）有下面的配置寄存器：

```
morgana% dd bs=256 skip=1 count=1 if=/proc/pcidata | od –Ax –t x1
1+0 records in
1+0 records out
000000 86 80 23 12 06 00 00 02 00 00 00 04 00 20 00 00
000010 00 00 00 f1 00 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 00 00 0a 01 00 00
000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

如果你将上面的输出和图 15-1 比较，你就可以理解这些数字。或者，你可以使用 *pcidump* 程序，在可以从 FTP 站点上找到，它将输出列表格式化并标号。
pcidump 的代码并不值得在这儿列出，因为这个简单程序只是一个长表，外加十行扫描这个表的代码。相反，让我们看看一些选择的输出行：
（代码 351）
pcidata 和 *pcidump*，与 *grep* 配合使用，对调试驱动程序的初始化代码非常有用。不过注意，*pcidata.c* 模块是 GPL 的，因为我是从核心源码中取的 PCI 扫描循环。这不应该对你作为一个驱动程序的作者有什么影响，因为我只是以一个支持工具的形式将这个模块包含在源文件中，而不是新驱动程序的可重用模版。

访问 I/O 和内存空间

一个PCI外围实现六个地址区段。每个区段由内存或I/O位置组成，或者压根不存在。大多数设备用一个内存区段代替它们的I/O端口，因为有些处理器（象Alpha）没有本身的I/O空间，还因为PC上的I/O空间都相当拥挤。内存和I/O空间的结构化的不同通过实现一个“内存可预取”位*来表达。将其控制寄存器映射到内存地址范围的外围将这个范围声明为不可预取的，而PCI板子上的有些东西如视频内存是可预取的。在本节中，只要讨论适用于内存或I/O，我就用单词“区段”来指一个PCI地址范围。
一个接口板子用配置寄存器（在图 15-1 中所示的 6 个 32 位寄存器，它们的符号名从 PCI_BASE_ADDRESS_0 到 PCI_BASE_ADDRESS_5）报告它的区段的大小和当前位置。由于 PCI 定义的 I/O 空间是一个 32 位的地址空间，因此用对内存和 I/O 适用同样的配置接口是可行的。如果设备使用 64 位的地址总线，它可以为每个区段用两个连续的 PCI_BASE_ADDRESS 寄存器在 64 位的内存空间来声明区段。因此有可能一个设备同时提供 32 位和 64 位的区段。
我不想在这儿讨论太多的细节，因为如果你打算写一个 PCI 驱动程序，你总会这个设备的硬件手册的。特别地，我不打算使用寄存器的预取位或两个“类型”位，并且我将讨论限制在 32 位外围上。不过，了解一下一般情况下是如何实现的，以及 Linux 驱动程序是如何处理 PCI 内存是很有趣的。
PCI 规范要求每个被实现的区段百升微被映射到一个可配置地址上。这意味着设备必须位它

* 这个信息居于基地址PCI寄存器的某个低序位中。

实现的每个区段装备一个可编程 32 位解码器，并且利用 64 位 PCI 扩展的板子必须有一个 4 位可编程解码器。尽管在 PC 上没有 64 位 PCI 总线，一些 Alpha 工作站则有。

由于通常一个区段的字节数是 2 的幂，如 32、64、4KB 或 2MB，所以可编程解码器的实际实现和使用都被简化了。而且，将一个区段映射到一个未对齐的地址上意义也不大；1MB 的区段自然在 1M 整数倍的地址处对齐，32 字节的区段则在 32 的整数倍处。PCI 规范利用了这个对齐;它要求地址解码器需要且只需查看地址总线的高位,并且只有高位是可编程的。这个约定也意味着任何区段的大小都必须是 2 的幂。

这样，重映射一个 PCI 区段可以通过在配置寄存器的高位设置一个合适的值来完成。例如，一个 1M 的区段，有 20 位的地址空间，可以通过设置寄存器的高 12 位进行重映射；向寄存器写 0x008xxxxx 告诉板子对 8MB-9MB 的地址区间响应。实际上，只有非常高的地址被用来映射 PCI 区段。

这种“部分解码”有几个额外的好处就是软件可以通过检查配置寄存器中非可编程位的数目来确定 PCI 区段的大小。为了这个目的，PCI 标准要求未使用的位必须总是读作 0。通过强制 I/O 区段的最小大小为 8 字节，内存区段为 16 字节，标准可以把一些额外的信息放入同一个 PCI 寄存器中：“空间”位，表明区段是内存的还是 I/O 的；两个“类型”位；一个“预取”位，只是位内存定义的。类型位在 32 位区段、64 位区段、以及“必须映射在 1M 一下的 32 位区段”进行选择。最后这个值用于那些仍然运行于一些 PC 上的过时软件。

检测一个 PCI 区段的大小可以通过使用几个定义在<linux/pci.h>中的位掩码来简化：是个内存区段时 PCI_BASE_ADDRESS_SPACE 被置位；PCI_BASE_ADDRESS_MEM_MASK 为内存区段掩去配置位；PCI_BASE_ADDRESS_TO_MASK 位 I/O 区段掩去这些位。规范还要求地址区段必须按序分配，从 PCI_BASE_ADDRESS_0 到 PCI_BASE_ADDRESS_5；这样一旦一个基地址未用（也就是被置未 0），你就可以知道所有的后续地址都未用。

报告 PCI 区段当前位置和大小的典型代码如下：

（代码 353）

（代码 354 #1）

这个代码是 *pciregion* 模块的一部分，与 *pcidata* 在同一个目录下发布；这个模块生成一个 */pci/pciregions* 文件，用上面给出的代码产生数据。当配置寄存器被修改时，中断报告被关闭，以防止驱动程序访问被映射到错误位置的区段。使用 *cli* 而不是 *save_flags* 是因为这个函数只在 *read* 系统调用时被执行，我们知道在系统调用的时候中断是打开的。

例如，这里是我的抓图器的*/proc/pciregion* 的报告：

（代码 #2）

计算机的固件在引导时用一个类似于前面给出的循环来正确地映射区段。由于固件防止了任何地址赋值时的冲突，Linux 驱动程序通常并不改变 PCI 区间的映射。

有趣的是注意到上面的程序报告的内存大小有可能被夸大。例如，*/proc/pciregion* 报告说我的视频板子是一个 16MB 的设备。但这并不真实（尽管我有可能扩展我的视频 RAM）。但由于这个大小信息只是被固件用来分配地址区间,夸大区段大小对驱动程序的作者来说并不是一个问题，他设备的内部并能正确地处理由固件分配的地址区间。

PCI 中断

至于中断，PCI 很容易处理。计算机的固件已经给设备分配了一个唯一的中断号，驱动程序只需要去用它即可。中断号存在配置寄存器 60 中（PCI_INTERRUPT_LINE），它是一个字节宽。这允许最多 256 条中断线，但实际限制依赖于使用的 CPU。驱动程序不必麻烦去检查中断号，因为在 PCI_INTERRUPT_LINE 中找到的一定是正确的。

如果设备不支持中断，寄存器 61（PCI_INTERRUPT_PIN）为 0；不然为非 0。不过由于驱

动程序知道它的设备是否是中断驱动的，因此并不常需要去读 PCI_INTERRUPT_PIN。
这样，处理中断的 PCI 特定的代码只需要这个配置字节以取得中断号，如下面所示的代码。不然，应用第九章的信息。

```
result = pcibios_read_config_byte(bus,fnct,PCI_INTERRUPT_LINE, &my_irq);
if(result){/*deal with result*/}
```

本节的其余部分为感兴趣的读者提供一些额外的信息，但对写驱动程序并不需要。
一个 PCI 连接器有四个中断脚，外围板子可以任意使用。每个管脚都是独立地路由到主板的中断控制器，因此中断可以共享，而没有任何电气问题。中断控制器负责将中断线（脚）映射到处理器的硬件；将这个平台相关的操作留给控制器是为了获得总线本身的平台无关性。
位于 PCI_INTERRUPUT_PIN 的只读配置寄存器用来告诉计算机哪一个管脚被使用了。值得记住的是每个设备板子最多可带 8 个设备；每个设备使用一个中断脚并在它自己的配置寄存器中报告它。同一个设备板子的不同设备可以使用不同的中断脚，也可以共享同一个。
另一方面，PCI_INTERRUPT_LINE 寄存器是读/写的。在计算机引导时，固件扫描它的 PCI 设备，并按照中断脚是如何路由到它的 PCI 槽的为每个设备设置这个寄存器。这个值由固件来赋，因为只有固件知道母板是如何将不同的中断脚路由到处理器的。然而，对设备驱动程序来说，PCI_INTERRUPT_LINE 寄存器是只读的。

回顾：ISA

ISA 总线在设计上相当老了，而且在性能方面也名声扫地，但它依然占据着扩展设备的很大一块市场。如果速度不是很重要，并且你想支持旧的主板，那么 ISA 实现要比 PCI 更令人喜欢。这个旧标准的一个额外的优势是，如果你是个电子爱好者，你可以很容易地构造你自己的设备。
令一方面，ISA 的一个巨大的缺点是它紧密地绑定在 PC 体系结构上；接口总线具有 80286 处理器的所有限制，导致系统程序员无穷的痛苦。ISA 设计的令一个巨大的问题（从原先的 IBM PC 继承下来的）是缺乏地理寻址，这导致了无穷的问题和为加一个新设备时漫长的“拔下--重跳线—插上—测试”循环。有趣的是注意到即使是最老的 Apple II 计算机都已经利用了地理寻址，它们的特征是无跳线的扩展板。

硬件资源

一个 ISA 设备可以装配 I/O 端口，内存区域，和中断线。
即使 x86 处理器支持 64KB 的 I/O 端口内存（也就是说，处理器申明 16 根地址线），有些老的 PC 硬件也只能对最低的 10 根地址线解码。这将可用的地址空间限制为 1024 个端口，因为在 1KB-64KB 区间的任何地址会被任何只能解码低地址线的设备错误地看成低地址。一些外围通过只映射一个端口到低 KB，并使用高地址线在不同的设备寄存器中选择的办法绕过了这个限制。例如，一个映射到 0x340 的设备可以安全地使用端口 0x740，0xB40，等等。如果说 I/O 端口的可用性受到了限制，那么内存访问就更糟了。一个 ISA 设备只能使用 640KB-1MB 和 15MB-16MB 之间的内存区间。640KB-1MB 区间被 PC BIOS、VGA 兼容的视频板、以及各种其它设备使用，留给新设备很少的可用空间。另一方面，15M 处的内存，Linux 并不直接支持；这个问题在第八章的“访问设备板子上的内存”讨论过。
ISA 设备板子上第三个可用的资源是中断线。有限的中断线被路由到 ISA 总线，它们被所有的接口板共享。造成的结果是，如果设备没有被正确地配置，它们可以用同样的中断线找到

它们自己。
尽管原先的ISA规范不允许跨设备的中断共享，多数设备板子还是允许的*。软件级的中断共享在第九章的“中断共享”中描述过。

ISA 程序设计

至于程序设计，除了 Linux 核心通过维护 I/O 和 IRQ 寄存器提供有限的帮助外（在第二章中的“使用资源”和第九章的“安装中断处理程序”中描述过），核心和 BIOS 中没有任何东西使得使用 ISA 设备更容易一些。
本书整个第一部分给出的编程技巧同样适用于 ISA 设备；驱动程序可以探测 I/O 端口，中断线必须用在第九章“自动检测 IRQ 号”介绍过的技术之一来自动检测。

“即插即用”规范

有些 ISA 设备板子遵循特殊的设计准则，要求特别的初始化序列，以简化增加接口板的安装和配置。这类板子的设计规范被称做“即插即用（PnP）”，它由一组构造和配置无跳线 ISA 设备的繁杂的规则集组成。PnP 设备实现了可重定位的 I/O 区段；PC 的 BIOS 负责这个重定位——PCI 的风格。
简单地说，PnP 的目的就是 PCI 设备具有的同样的灵活性，而不改变底层的电气接口（ISA 总线）。为了这个目的，规范定义了一组设备无关的配置寄存器和地理寻址接口板的方法，即使物理总线并不携带每个板子（地理的）的走线----每个 ISA 信号线于每个可用槽相连。地理寻址工作的方式是：给计算机的每个 ISA 外围分配一个小整数，称做“卡选择号（CSN）”。每个 PnP 设备有一个唯一的序列标志符，64 位宽，被硬写入外围板子。CSN 的分配用这个唯一的序列号来确定 PnP 设备。但 CSN 只能在引导时被安全地分配，这要求 BIOS 理解 PnP。由于这个原因，如果没有一个配置盘，老计算机不能支持 PnP。
符合 PnP 规范的接口板在硬件级十分复杂。它们比 PCI 板要精细的多，同时要求符杂的软件。安装这类设备遇到困难并不罕见；即使安装没有问题，你仍然面对性能限制和 ISA 总线有限的 I/O 空间。按我的观点，只要可能，最好是安装 PCI 设备并享受其新技术。
如果你对 PnP 的配置软件有兴趣，你可以浏览 *drivers/net/3c509.c*，它的探测函数处理了 PnP 设备。Linux2.1.33 也为 PnP 增加了一些初始支持，见目录 *drivers/pnp*。

其它 PC 总线

PCI 和 ISA 是 PC 世界最常用的外围接口，但它们并不为仅有。下面是 PC 市场上找得到的其它总线特征的概述。

MCA

“微通道体系结构（MCA）”是用在 PS/2 计算机和一些笔记本上的一个 IBM 标准。微通道的主要问题是缺乏文档，这导致了 Linux 上对 MCA 支持的缺乏。不过，在 2.1.15，已经飘荡多时的 MCA 补丁被加入了正式的核心；因此，新的核心可以在 PS/2 计算机上运行。
在硬件级，微通道具有比 ISA 多的特征。它支持多主 DMA，32 位地址和数据线，共享中断线，以及访问每个板子配置寄存器的地理寻址。这种寄存器被称做“可编程选项选择（POS）”，但它们并不具有 PCI 寄存器的所有特征。Linux 对 MCA 的支持包括一些引出到模块的函数。

---

* 中断共享的问题是属于电气工程的；如果一个设备驱动程序驱动信号线为不活动----通过应用一个低阻抗电压级----中断便不能共享。另一方面，如果设备对不活动逻辑级使用一个上拉电阻，那么共享就是可能的。多数ISA接口板使用上拉的方法。

设备驱动程序可以通过读取整数值 MCA_bus 来确定它是否运行在一个微通道计算机上。如果核心运行在一个 MCA 单元中，那么 MCA_bus 非 0。如果这个符号是个预处理器宏，那么宏 MCA_bus__is_a_macro 也要被定义。如果 MCA_bus__is_a_macro 未定义，那么 MCA_bus 是引出到模块化代码的一个整数变量。事实上，MCA_bus 对除 PC 外的所有平台仍然是个硬写为 0 的宏----Linux X86 的移植在 2.1.15 将宏改为变量。MCA_bus 和 MCA_bus__is_a_macro 都在<asm/processor.h>中定义。

EISA

扩展的 ISA（EISA）是 ISA 的 32 位扩展，带一个兼容的接口连接器；ISA 的设备板子可以插入一个 EISA 连接器。额外的线路是在 ISA 连接下路由。

象 PCI 和 MCA，EISA 总线被设计来带无跳线设备，它与 MCA 有同样的特征：32 位地址和数据线，多主 DMA，以及共享中断线。EISA 设备由软件配置，但它们不需要任何特别的操作系统支持。EISA 驱动程序已经为以太网设备和 SCSI 控制器存在与 Linux 核心中。

EISA 驱动程序检查 EISA_bus 确定是否主机带有 EISA 总线。类似于 MCA_bus，EISA_bus 要么是宏，要么是变量，这依赖于 EISA_bus__is_a_macro 是否被定义了。这两个符号定义在<asm/processor.h>。

至于驱动程序，核心中没有对 EISA 的特别支持，程序员必须自己处理 ISA 的扩展。驱动程序用标准的 EISA I/O 操作访问 EISA 寄存器。核心中已有的驱动程序可以作为示例代码。

VLB

ISA 的另一个扩展是“VESA Local Bus”接口总线，它通过增加一个长度方向的槽扩展 ISA 连接器。这个额外的槽可以被 VLB 设备“单独”使用；由于它从 ISA 连接器复制了所有重要的信号，设备可以被构造成只插入 VLB 插槽，而不用 ISA 插槽。单独的 VLB 外围很少见，因为多数设备需要到达背板，这样它们的外部连接器是可用的。

Sbus

虽然多数 Alpha 计算机装备有 PCI 或 ISA 接口总线，多数基于 Sparc 的工作站使用 Sbus 连接它们的外围。

Sbus 是相当先进的一种设计，尽管它已经存在了很长时间了。它是想成为处理器无关的，并专为 I/O 外围板子进行了优化。换句话说，你不能在 Sbus 的槽中插入额外的 RAM。这个优化的目的是简化硬件设备和系统软件的设计，这是以母板上的额外复杂性为代价的。

总线的这种 I/O 偏向导致了一类外围，它们用虚地址传送数据，这样绕过了分配连续缓冲区的需要。母板负责解码虚地址，并映射到物理地址上。这要求在 Sbus 上附带一些 MMU（内存管理单元）的能力，这部分负责的电路被称为“IOMMU”。这种总线的另一个特征是：设备板子是地理寻址的，因此不需要在每个外围上实现地址解码器，也不需要处理地址冲突。Sbus 外围在 PROM 中使用 Forth 语言来初始化它们。选择 Forth 是因为这个解释器是轻量级的，可以容易地在任何计算机系统的固件里实现。另外，Sbus 规范描述了引导过程，因此符合条件的 I/O 设备可以简单地接入系统，并在系统引导时被识别出来。

至于 Linux，直到核心 2.0 也没有对 Sbus 设备的特别支持被引出到模块中。版本 2.1.8 增加了对 Sbus 的特定支持，我鼓励感兴趣的读者去看看最近的核心。

快速参考

本节象往常一样，概述在本章中介绍的符号。

#include <linux/config.h>

CONFIG_PCI

这个宏用来条件编译 PCI 相关的代码。当一个 PCI 模块被加载入一个非 PCI 核心时，*insmod* 会抱怨说几个符号不能解析。

#include <linux/pci.h>

这个头文件包含 PCI 寄存器和几个销售商及设备 ID 的符号名。

#include <linux/bios32.h>

所有下面列出的 pcibios_函数在这个头文件中定义原型。

```
int pcibios_present(void);
```

这个函数返回一个布尔值表明我们运行的计算机是否具有 PCI 能力。

```
int pcibios_find_device(unsigned short vendor, unsigned short id, unsigned short index,
                        unsigned char *bus, unsigned char *function);
int pcibios_find_class(unsigned int class_code, unsigned short index,
                       unsigned char *bus, unsigned char *function);
```

这些函数询问 PCI 固件关于设备是否有某个特定的签名，或属于某个特定的类。返回值是一个出错说明；成功时，bus 和 function 用来存储设备的位置。index 第一次必须被传给 0，以后没查找一个新设备，就增加 1。

PCIBIOS_SUCCESSFUL

PCIBIOS_DEVICE_NOT_FOUND

```
char *pcibios_strerror(int error);
```

这些宏，还有其它几个表示 pcibios 函数返回的整数值。DEVICE_NOT_FOUND 一般被认为是个成功值，因为成功地发现没有设备。pcibios_strerror 函数用来将每个整数返回值转换为一个字符串。

```
int pcibios_read_config_byte(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_word(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_dword(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char *ptr);
int pcibios_write_config_byte(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char val);
int pcibios_write_config_word(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned short val);
int pcibios_write_config_dword(unsigned char bus, unsigned char function,
                               unsigned char where, unsigned int val);
```

这些函数用来读写 PCI 配置寄存器。尽管 Linux 核心负责字节序，程序员在从单个字节组装多字节值时必须特别注意字节序。PCI 总线是小印地安字节序。