第十二章 PCI驱动

虽然第9章介绍了硬件控制的最低级别，但这一章提供了更高级别的总线架构的概述。总线由电气接口和编程接口组成。在这一章，我们主要讨论编程接口。

这一章涵盖了许多总线架构。然而，主要的焦点是内核函数如何访问外围组件互连（PCI）设备，因为现在PCI总线是桌面和更大型计算机上最常用的外围总线。这种总线得到了内核的最佳支持。ISA对于电子爱好者来说仍然很常见，虽然它基本上是一种裸机类型的总线，除了第9章和第10章已经涵盖的内容，没有太多可以说的。

12.1. PCI 接口

尽管许多计算机用户认为PCI是一种布置电线的方式，但实际上它是一套完整的规范，定义了计算机的不同部分应如何互动。

PCI规范涵盖了与计算机接口相关的大部分问题。我们不会在这里全部介绍；在这一部分，我们主要关注PCI驱动程序如何找到其硬件并获得对其的访问权限。在第2章的“模块参数”和第10章的“自动检测IRQ号码”中讨论的探测技术可以用于PCI设备，但规范提供了一个比探测更好的替代方案。

PCI架构设计为替代ISA标准，有三个主要目标：在计算机和其外围设备之间传输数据时获得更好的性能，尽可能地独立于平台，并简化向系统添加和移除外围设备。

PCI总线通过使用比ISA更高的时钟频率来实现更好的性能；其时钟运行在25或33 MHz（其实际频率是系统时钟的一个因数），并且最近也部署了66 MHz甚至133 MHz的实现。此外，它配备了一个32位数据总线，规范中还包含了一个64位扩展。平台独立性通常是计算机总线设计的一个目标，对于PCI来说尤其重要，因为PC世界一直被处理器特定的接口标准所主导。PCI目前广泛用于IA-32、Alpha、PowerPC、SPARC64和IA-64系统，以及一些其他平台。

然而，对驱动程序编写者来说，最相关的是PCI对接口板自动检测的支持。PCI设备是无跳线的（不像大多数旧的外围设备）并在启动时自动配置。然后，设备驱动程序必须能够访问设备中的配置信息，以完成初始化。这无需进行任何探测。

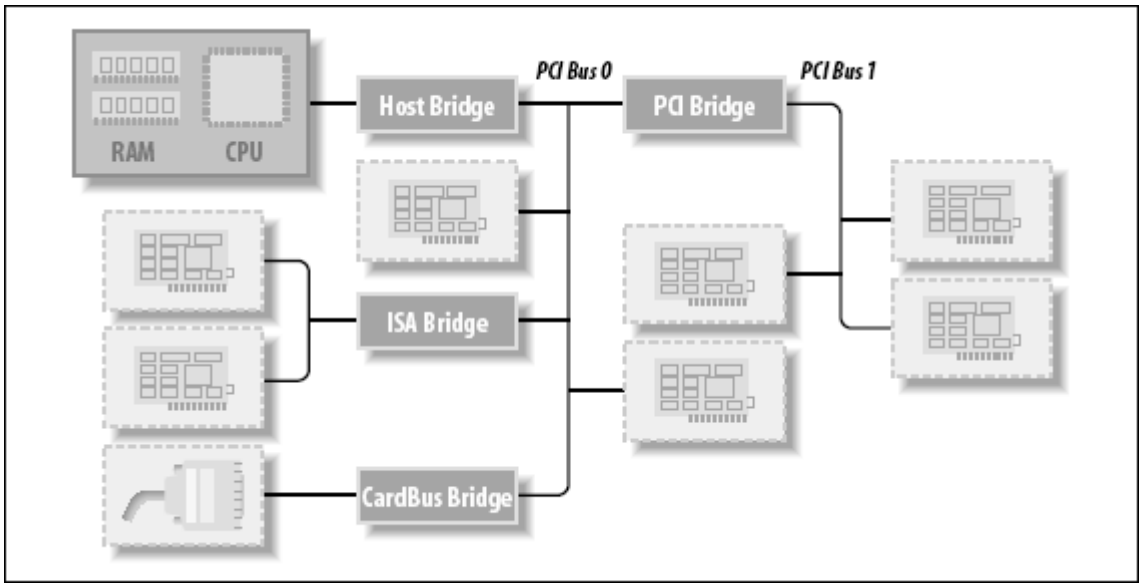
12.1.1. PCI 寻址

每个PCI外设都由一个总线号、一个设备号和一个功能号来标识。PCI规范允许一个系统最多承载256个总线，但由于256个总线对许多大型系统来说不够，Linux现在支持PCI

域。每个PCI域可以承载最多256个总线。每个总线可以承载最多32个设备，每个设备可以是一个多功能板（如带有CD-ROM驱动的音频设备），最多有八个功能。因此，每个功能可以在硬件级别通过一个16位地址或密钥来标识。然而，为Linux编写的设备驱动程序不需要处理这些二进制地址，因为它们使用一个特定的数据结构，称为pci_dev，来操作设备。

大多数最近的工作站至少有两个PCI总线。在一个系统中插入多个总线是通过桥来实现的，桥是特殊的PCI外设，其任务是连接两个总线。PCI系统的整体布局是一棵树，每个总线都连接到上层的总线，直到树根的总线0。CardBus PC卡系统也通过桥连接到PCI系统。一个典型的PCI系统在图12-1中表示，其中突出显示了各种桥。

与PCI外设相关的16位硬件地址，虽然大部分隐藏在struct pci_dev对象中，但偶尔还是可以看到，特别是在使用设备列表时。这样的情况之一是lspci的输出（是pciutils包的一部分，大多数发行版都有）和/proc/pci和/proc/bus/pci中信息的布局。sysfs表示的PCI设备也显示了这种寻址方案，还增加了PCI域信息。当显示硬件地址时，它可以显示为两个值（一个8位的总线号和一个8位的设备和功能号），三个值（总线、设备和功能），或四个值（域、总线、设备和功能）；所有的值通常都以十六进制显示。



例如，/proc/bus/pci/devices使用一个单独的16位字段（以便于解析和排序），而/proc/bus/busnumber将地址分成三个字段。下面显示了这些地址如何出现，只显示输出行的开始部分。

```
$ lspci | cut -d: -f1-3
0000:00:00.0 Host bridge
0000:00:00.1 RAM memory
0000:00:00.2 RAM memory
0000:00:02.0 USB Controller
0000:00:04.0 Multimedia audio controller
0000:00:06.0 Bridge
0000:00:07.0 ISA bridge
0000:00:09.0 USB Controller
0000:00:09.1 USB Controller
0000:00:09.2 USB Controller
0000:00:0c.0 CardBus bridge
0000:00:0f.0 IDE interface
0000:00:10.0 Ethernet controller
0000:00:12.0 Network controller
0000:00:13.0 FireWire (IEEE 1394)
0000:00:14.0 VGA compatible controller
$ cat /proc/bus/pci/devices | cut -f1
0000
0001
0002
0010
0020
0030
0038
0048
0049
004a
0060
0078
0080
0090
0098
00a0
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:00.0 →
```

```
../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:00.1 →
../../../devices/pci0000:00/0000:00:00.1
|-- 0000:00:00.2 →
../../../devices/pci0000:00/0000:00:00.2
|-- 0000:00:02.0 →
../../../devices/pci0000:00/0000:00:02.0
|-- 0000:00:04.0 →
../../../devices/pci0000:00/0000:00:04.0
|-- 0000:00:06.0 →
../../../devices/pci0000:00/0000:00:06.0
|-- 0000:00:07.0 →
../../../devices/pci0000:00/0000:00:07.0
|-- 0000:00:09.0 →
../../../devices/pci0000:00/0000:00:09.0
|-- 0000:00:09.1 →
../../../devices/pci0000:00/0000:00:09.1
|-- 0000:00:09.2 →
../../../devices/pci0000:00/0000:00:09.2
|-- 0000:00:0c.0 →
../../../devices/pci0000:00/0000:00:0c.0
|-- 0000:00:0f.0 →
../../../devices/pci0000:00/0000:00:0f.0
|-- 0000:00:10.0 →
../../../devices/pci0000:00/0000:00:10.0
|-- 0000:00:12.0 →
../../../devices/pci0000:00/0000:00:12.0
|-- 0000:00:13.0 →
../../../devices/pci0000:00/0000:00:13.0
`-- 0000:00:14.0 →
../../../devices/pci0000:00/0000:00:14.0
```

所有三个设备列表都按相同的顺序排序，因为lspci使用/proc文件作为其信息来源。以VGA视频控制器为例，0x00a0在分成域（16位）、总线（8位）、设备（5位）和功能（3位）时，表示0000:00:14.0。

每个外设板的硬件电路回答关于三个地址空间的查询：内存位置、I/O端口和配置寄存器。前两个地址空间由同一PCI总线上的所有设备共享（即，当你访问一个内存位置

时，该PCI总线上的所有设备同时看到总线周期）。另一方面，配置空间利用地理寻址。配置查询一次只寻址一个插槽，所以它们永远不会冲突。

就驱动程序而言，通过inb、readb等方式以通常的方式访问内存和I/O区域。另一方面，通过调用特定的内核函数来访问配置寄存器，执行配置事务。关于中断，每个PCI插槽有四个中断引脚，每个设备功能可以使用其中一个，而不用关心这些引脚如何路由到CPU。这种路由是计算机平台的责任，并在PCI总线之外实现。由于PCI规范要求中断线路是可共享的，即使是中断线路数量有限的处理器，如x86，也可以承载许多PCI接口板（每个接口板有四个中断引脚）。

PCI总线中的I/O空间使用32位地址总线（导致4 GB的I/O端口），而内存空间可以使用32位或64位地址访问。64位地址在更近的平台上可用。地址应该是设备唯一的，但软件可能错误地将两个设备配置到同一个地址，使得无法访问任何一个设备。但这个问题除非驱动程序愿意玩弄它不应该触碰的寄存器，否则永远不会发生。好消息是，接口板提供的每个内存和I/O地址区域都可以通过配置事务进行重新映射。也就是说，固件在系统启动时初始化PCI硬件，将每个区域映射到不同的地址以避免冲突。这些区域当前映射到的地址可以从配置空间中读取，所以Linux驱动程序可以在不探测的情况下访问其设备。读取配置寄存器后，驱动程序可以安全地访问其硬件。

PCI配置空间由每个设备功能的256字节组成（除了PCI Express设备，每个功能有4 KB的配置空间），配置寄存器的布局是标准化的。配置空间的四个字节保存了一个唯一的功能ID，所以驱动程序可以通过查找该外设的特定ID来识别其设备。总的来说，每个设备板都是地理寻址以检索其配置寄存器；然后可以使用那些寄存器中的信息进行正常的I/O访问，而不需要进一步的地理寻址。

从这个描述中应该清楚，PCI接口标准相对于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驱动程序还需要能够访问配置空间，以免自己进行风险的探测任务。

在本章的其余部分，我们使用设备这个词来指代设备功能，因为在一个多功能板中的每个功能都作为一个独立的实体。当我们提到设备时，我们指的是“域号、总线号、设备号和功能号”的元组。

12.1.2. 启动时间

为了了解PCI如何工作，我们从系统启动开始，因为那是设备被配置的时候。

当电源应用到PCI设备时，硬件保持不活动状态。换句话说，设备只响应配置事务。在上电时，设备在计算机的地址空间中没有映射任何内存和I/O端口；每个其他设备特定的功能，如中断报告，也被禁用。幸运的是，每个PCI主板都配备了PCI感知的固件，根

据平台的不同，被称为BIOS、NVRAM或PROM。固件通过读写PCI控制器中的寄存器，提供对设备配置地址空间的访问。

在系统启动时，固件（或者如果配置了的话，Linux内核）与每个PCI外设进行配置事务，以便为它提供的每个地址区域分配一个安全的位置。当设备驱动程序访问设备时，其内存和I/O区域已经被映射到处理器的地址空间。驱动程序可以改变这个默认分配，但它永远不需要这样做。

如前所述，用户可以通过读取/proc/bus/pci/devices和`/proc/bus/pci/*/*`来查看PCI设备列表和设备的配置寄存器。前者是一个带有（十六进制）设备信息的文本文件，后者是二进制文件，报告每个设备的配置寄存器的快照，每个设备一个文件。sysfs树中的单个PCI设备目录可以在/sys/bus/pci/devices中找到。一个PCI设备目录包含许多不同的文件。

```
$ tree /sys/bus/pci/devices/0000:00:10.0
/sys/bus/pci/devices/0000:00:10.0
|-- class
|-- config
|-- detach_state
|-- device
|-- irq
|-- power
|   `-- state
|-- resource
|-- subsystem_device
|-- subsystem_vendor
`-- vendor
```

文件config是一个二进制文件，允许从设备读取原始的PCI配置信息（就像`/proc/bus/pci/*/*`提供的那样）。文件vendor、device、subsystem_device、subsystem_vendor和class都指的是这个PCI设备的特定值（所有PCI设备都提供这个信息）。文件irq显示当前分配给这个PCI设备的IRQ，文件resource显示这个设备当前分配的内存资源。

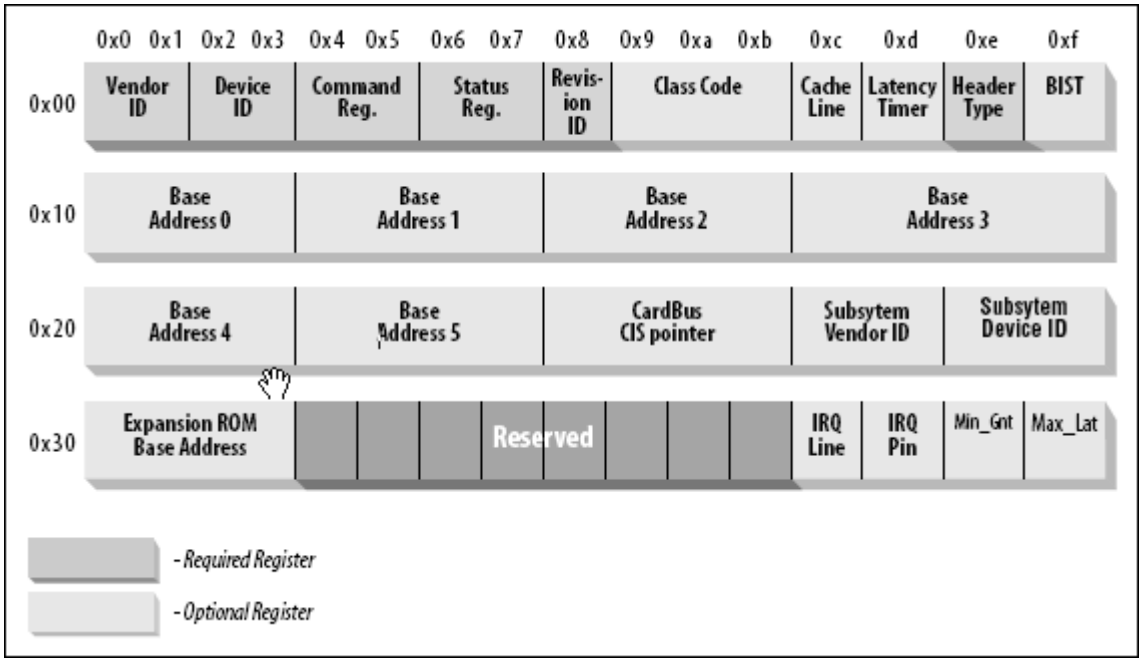
12.1.3. 配置寄存器和初始化

在这一部分，我们将查看PCI设备包含的配置寄存器。所有PCI设备都至少有一个256字节的地址空间。前64字节是标准化的，其余的则依赖于设备。图12-2显示了设备无关

配置空间的布局。

如图所示，一些PCI配置寄存器是必需的，一些是可选的。每个PCI设备必须在必需的寄存器中包含有意义的值，而可选寄存器的内容则取决于外设的实际能力。除非必需字段的内容表明它们是有效的，否则不使用可选字段。因此，必需字段断言了板的能力，包括其他字段是否可用。

值得注意的是，PCI寄存器总是小端。虽然这个标准是设计成与架构无关的，但PCI设计者有时会稍微偏向PC环境。驱动程序编写者在访问多字节配置寄存器时应该小心字节顺序；在PC上工作的代码可能在其他平台上不工作。Linux开发者已经解决了字节顺序问题（参见下一节，“访问配置空间”），但必须记住这个问题。如果你需要将数据从主机顺序转换为PCI顺序，或者反过来，你可以使用在<asm/byteorder.h>中定义的函数，知道PCI字节顺序是小端。



描述所有的配置项超出了本书的范围。通常，每个设备发布的技术文档会描述支持的寄存器。我们感兴趣的是驱动程序如何寻找其设备以及如何访问设备的配置空间。

有三个或五个PCI寄存器用于识别设备：vendorID、deviceID和class是始终使用的三个。每个PCI制造商都会为这些只读寄存器分配适当的值，驱动程序可以使用它们来查找设备。此外，字段subsystem vendorID和subsystem deviceID有时由供应商设置，以进一步区分类似的设备。

让我们更详细地看看这些寄存器：

- `vendorID` 这个16位寄存器识别硬件制造商。例如，每个Intel设备都用相同的供应商号码标记，即0x8086。有一个全球注册这些数字的注册表，由PCI特别兴趣小组维护，制造商必须申请一个唯一的数字分配给他们。

- `deviceID` 这是另一个16位寄存器，由制造商选择；设备ID不需要官方注册。这个ID通常与供应商ID配对，形成一个唯一的32位硬件设备标识符。我们用签名这个词来指代供应商和设备ID对。设备驱动程序通常依赖签名来识别其设备；你可以在目标设备的硬件手册中找到要查找的值。

- `class` 每个外围设备都属于一个类。类寄存器是一个24位的值，其顶部8位识别“基类”（或组）。例如，“以太网”和“令牌环”是属于“网络”组的两个类，而“串行”和“并行”类属于“通信”组。一些驱动程序可以支持几个类似的设备，每个设备都有一个不同的签名，但都属于同一个类；这些驱动程序可以依赖类寄存器来识别他们的外围设备，如后面所示。

- `subsystem vendorID subsystem deviceID` 这些字段可以用于进一步识别设备。如果芯片是一个通用的接口芯片到本地（板载）总线，它通常用于几个完全不同的角色，驱动程序必须识别它正在交谈的实际设备。子系统标识符被用于这个目的。

使用这些不同的标识符，PCI驱动程序可以告诉内核它支持哪种类型的设备。struct pci_device_id结构用于定义驱动程序支持的不同类型的PCI设备的列表。这个结构包含以下字段：

```c
__u32 vendor;

__u32 device;
```

这些指定设备的PCI供应商和设备ID。如果驱动程序可以处理任何供应商或设备ID，这些字段的值应该使用PCI_ANY_ID。

```c
__u32 subvendor;

__u32 subdevice;
```

这些指定设备的PCI子系统供应商和子系统设备ID。如果驱动程序可以处理任何类型的子系统ID，这些字段的值应该使用PCI_ANY_ID。

```c
__u32 class;

__u32 class_mask;
```

这两个值允许驱动程序指定它支持一种类型的PCI类设备。PCI设备的不同类别（VGA控制器是一个例子）在PCI规范中描述。如果驱动程序可以处理任何类型的类，这些字段的值应该使用0。

```c
kernel_ulong_t driver_data;
```

这个值不用于匹配设备，而是用于保存PCI驱动程序可以用来区分不同设备的信息，如果它想要的话。

有两个辅助宏应该用来初始化一个struct pci_device_id结构：

```c
PCI_DEVICE(vendor, device)
```

这创建一个只匹配特定供应商和设备ID的struct pci_device_id。宏将结构的subvendor和subdevice字段设置为PCI_ANY_ID。

```c
PCI_DEVICE_CLASS(device_class, device_class_mask)
```

这创建一个匹配特定PCI类的struct pci_device_id。

以下是使用这些宏来定义驱动程序支持的设备类型的示例，可以在以下内核文件中找到：

drivers/usb/host/ehci-hcd.c:

```c
static const struct pci_device_id pci_ids[ ] = { {

    /* handle any USB 2.0 EHCI controller */

    PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) |
0x20), ~0),

    .driver_data =  (unsigned long) &ehci_driver,

    },

    { /* end: all zeroes */ }

};
```

drivers/i2c/busses/i2c-i810:

```c
static struct pci_device_id i810_ids[ ] = {

    { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
PCI_DEVICE_ID_INTEL_82810_IG1) },

    { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
PCI_DEVICE_ID_INTEL_82810_IG3) },

    { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
PCI_DEVICE_ID_INTEL_82810E_IG) },

    { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
PCI_DEVICE_ID_INTEL_82815_CGC) },

    { PCI_DEVICE(PCI_VENDOR_ID_INTEL,
PCI_DEVICE_ID_INTEL_82845G_IG) },

    { 0, },

};
```

这些示例创建了一个struct pci_device_id结构的列表，列表中的最后一个值是一个所有字段都设置为零的空结构。这个ID数组用于struct pci_driver（下面描述），也用于告诉用户空间这个特定驱动程序支持哪些设备。

12.1.4. MODULE_DEVICE_TABLE 宏

这个pci_device_id结构需要导出到用户空间，以便让热插拔和模块加载系统知道哪个模块与哪个硬件设备配合工作。宏MODULE_DEVICE_TABLE实现了这一点。一个例子是：

```c
MODULE_DEVICE_TABLE(pci, i810_ids);
```

这个语句创建了一个名为**mod_pci_device_table的局部变量，它指向struct pci_device_id的列表。稍后在内核构建过程中，depmod程序搜索所有模块的符号**mod_pci_device_table。如果找到了那个符号，它就从模块中提取数据，并将其添加到文件/lib/modules/KERNEL_VERSION/modules.pcimap中。在depmod完成后，所有由内核中的模块支持的PCI设备都会列在那个文件中，同时还会列出它们的模块名称。当内核告诉热插拔系统找到了一个新的PCI设备时，热插拔系统使用modules.pcimap文件来找到适当的驱动程序进行加载。

12.1.5. 注册一个 PCI 驱动

所有PCI驱动程序都必须创建的主要结构，以便正确地在内核中注册，是struct pci_driver结构。这个结构由一些函数回调和描述PCI驱动程序的变量组成。以下是PCI驱动程序需要注意的这个结构中的字段：

```c
const char *name;
```

驱动程序的名称。它在所有内核中的PCI驱动程序中必须是唯一的，通常设置为与驱动程序的模块名称相同。当驱动程序在内核中时，它会出现在/sys/bus/pci/drivers/下的sysfs中。

```c
const struct pci_device_id *id_table;
```

指向前面在本章中描述的struct pci_device_id表的指针。

```c
int (*probe) (struct pci_dev *dev, const struct
pci_device_id *id);
```

指向PCI驱动程序中的probe函数的指针。当PCI核心有一个它认为这个驱动程序想要控制的struct pci_dev时，它会调用这个函数。也会将PCI核心用来做出这个决定的struct pci_device_id的指针传递给这个函数。如果PCI驱动程序声明了传递给它的struct pci_dev，它应该正确地初始化设备并返回0。如果驱动程序不想声明设备，或者发生错误，它应该返回一个负的错误值。关于这个函数的更多细节将在本章后面介绍。

```c
void (*remove) (struct pci_dev *dev);
```

当struct pci_dev从系统中移除，或者当PCI驱动程序从内核中卸载时，PCI核心调用的函数的指针。关于这个函数的更多细节将在本章后面介绍。

```c
int (*suspend) (struct pci_dev *dev, u32 state);
```

当struct pci_dev被挂起时，PCI核心调用的函数的指针。挂起状态在state变量中传递。这个函数是可选的；驱动程序不必提供它。

```c
int (*resume) (struct pci_dev *dev);
```

当struct pci_dev被恢复时，PCI核心调用的函数的指针。它总是在suspend被调用后调用。这个函数是可选的；驱动程序不必提供它。

总的来说，要创建一个正确的struct pci_driver结构，只需要初始化四个字段：

```c
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove,
};
```

要将struct pci_driver注册到PCI核心，需要调用pci_register_driver，并传入struct pci_driver的指针。这通常在PCI驱动程序的模块初始化代码中完成：

```c
static int __init pci_skel_init(void)

{

    return pci_register_driver(&pci_driver);

}
```

注意，pci_register_driver函数要么返回一个负的错误号，要么如果一切都注册成功则返回0。它不返回绑定到驱动程序的设备的数量，也不返回如果没有设备绑定到驱动程序的错误号。这是从2.6版本以前的内核的改变，原因是以下情况：

- 在支持PCI热插拔的系统，或者CardBus系统中，PCI设备可以在任何时候出现或消失。如果驱动程序可以在设备出现之前加载，这将有助于减少初始化设备所需的时间。
- 2.6内核允许在驱动程序加载后动态分配新的PCI ID。这是通过在sysfs中所有PCI驱动程序目录中创建的new_id文件来完成的。如果正在使用内核还不知道的新设备，这非常有用。用户可以将PCI ID值写入new_id文件，然后驱动程序绑定到新设备。如果驱动程序在系统中存在设备之前不允许加载，这个接口将无法工作。

当要卸载PCI驱动程序时，需要从内核中注销struct pci_driver。这是通过调用pci_unregister_driver来完成的。当这个调用发生时，任何当前绑定到这个驱动程序的PCI设备都会被移除，并且在pci_unregister_driver函数返回之前，会调用这个PCI驱动程序的remove函数。

```c
static void __exit pci_skel_exit(void)

{

    pci_unregister_driver(&pci_driver);

}
```

12.1.6. 老式 PCI 探测

在旧的内核版本中，函数pci_register_driver并不总是被PCI驱动程序使用。相反，它们要么手动遍历系统中的PCI设备列表，要么调用一个可以搜索特定PCI设备的函数。在驱动程序中遍历系统中的PCI设备列表的能力已经从2.6内核中移除，以防止驱动程序在设备被同时移除时修改PCI设备列表，从而导致内核崩溃。

如果真的需要找到特定的PCI设备，以下函数是可用的：

```c
struct pci_dev *pci_get_device(unsigned int vendor,
unsigned int device,

                               struct pci_dev *from);
```

此函数扫描当前在系统中存在的PCI设备列表，如果输入参数匹配指定的供应商和设备ID，它会增加找到的struct pci_dev变量的引用计数，并返回给调用者。这防止了结构在没有任何通知的情况下消失，并确保内核不会出错。在驱动程序完成对由函数返回的struct pci_dev的使用后，它必须调用函数pci_dev_put来正确地将使用计数减少，以允许内核在设备被移除时清理设备。

from参数用于获取具有相同签名的多个设备；该参数应指向最后找到的设备，以便搜索可以继续，而不是从列表头开始重新开始。要找到第一个设备，from被指定为NULL。如果没有找到（更多）设备，返回NULL。

如何正确使用此函数的示例是：

```c
struct pci_dev *dev;

dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO,
NULL);

if (dev) {

    /* Use the PCI device */

    ...

    pci_dev_put(dev);

}
```

此函数不能在中断上下文中调用。如果是，将向系统日志打印警告。

```c
struct pci_dev *pci_get_subsys(unsigned int vendor,
unsigned int device,

  unsigned int ss_vendor, unsigned int ss_device, struct
pci_dev *from);
```

此函数的工作方式就像pci_get_device，但是它允许在查找设备时指定子系统供应商和子系统设备ID。

此函数不能在中断上下文中调用。如果是，将向系统日志打印警告。

```c
struct pci_dev *pci_get_slot(struct pci_bus *bus,
unsigned int devfn);
```

此函数在指定的struct pci_bus上搜索系统中的PCI设备列表，以查找PCI设备的指定设备和函数号。如果找到了匹配的设备，它的引用计数会增加，并返回一个指向它的指针。当调用者完成对struct pci_dev的访问时，它必须调用pci_dev_put。

所有这些函数都不能在中断上下文中调用。如果是，将向系统日志打印警告。

12.1.7. 使能 PCI 设备

在PCI驱动程序的probe函数中，驱动程序在访问PCI设备的任何设备资源（I/O区域或中断）之前，必须调用pci_enable_device函数：

```c
int pci_enable_device(struct pci_dev *dev);
```

此函数实际上启用了设备。它唤醒设备，并在某些情况下也分配其中断线和I/O区域。例如，这发生在CardBus设备上（在驱动程序级别已完全等同于PCI）。

12.1.8. 存取配置空间

在驱动程序检测到设备后，通常需要读取或写入三个地址空间：内存、端口和配置。特别是，访问配置空间对驱动程序至关重要，因为这是它唯一可以找出设备在内存和I/O空间中映射位置的方式。

由于微处理器无法直接访问配置空间，计算机供应商必须提供一种方法来实现。要访问配置空间，CPU必须在PCI控制器中写入和读取寄存器，但具体实现取决于供应商，对于这个讨论并不相关，因为Linux提供了一个标准接口来访问配置空间。

就驱动程序而言，可以通过8位、16位或32位数据传输来访问配置空间。相关函数在<linux/pci.h>中原型化：

```c
int pci_read_config_byte(struct pci_dev *dev, int where,
u8 *val);

int pci_read_config_word(struct pci_dev *dev, int where,
u16 *val);

int pci_read_config_dword(struct pci_dev *dev, int where,
u32 *val);
```

从由dev标识的设备的配置空间读取一个、两个或四个字节。where参数是从配置空间开始的字节偏移。从配置空间获取的值通过val指针返回，函数的返回值是一个错误代码。word和dword函数将刚读取的值从小端转换为处理器的本机字节顺序，因此你不需要处理字节顺序。

```c
int pci_write_config_byte(struct pci_dev *dev, int where,
u8 val);

int pci_write_config_word(struct pci_dev *dev, int where,
u16 val);

int pci_write_config_dword(struct pci_dev *dev, int
where, u32 val);
```

写一个、两个或四个字节到配置空间。设备如常由dev标识，正在写入的值作为val传递。word和dword函数在写入外设之前将值转换为小端。

所有前面的函数都实现为内联函数，实际上调用以下函数。如果驱动程序在任何特定时间点都无法访问struct pci_dev，可以随时使用这些函数代替上述函数：

```c
int pci_bus_read_config_byte (struct pci_bus *bus,
unsigned int devfn, int

  where, u8 *val);

int pci_bus_read_config_word (struct pci_bus *bus,
unsigned int devfn, int

  where, u16 *val);

int pci_bus_read_config_dword (struct pci_bus *bus,
unsigned int devfn, int

  where, u32 *val);
```

就像pci*read*函数一样，但是需要`struct pci_bus *`和`devfn`变量，而不是`struct pci_dev *`。

```c
int pci_bus_write_config_byte (struct pci_bus *bus,
unsigned int devfn, int

  where, u8 val);

int pci_bus_write_config_word (struct pci_bus *bus,
unsigned int devfn, int

  where, u16 val);

int pci_bus_write_config_dword (struct pci_bus *bus,
unsigned int devfn, int

  where, u32 val);
```

就像pci*write*函数一样，但是需要`struct pci_bus *`和`devfn`变量，而不是`struct pci_dev *`。

使用pci*read*函数来访问配置变量的最好方式是通过在<linux/pci.h>中定义的符号名。例如，以下小函数通过将符号名传递给where来获取设备的修订ID：

```c
static unsigned char skel_get_revision(struct pci_dev
*dev)

{

    u8 revision;

    pci_read_config_byte(dev, PCI_REVISION_ID,
&revision);

    return revision;

}
```

12.1.9. 存取 I/O 和内存空间

PCI设备实现了多达六个I/O地址区域。每个区域由内存或I/O位置组成。大多数设备在内存区域中实现其I/O寄存器，因为这通常是更理智的方法（如第9章的“I/O端口和I/O内存”部分所解释的）。然而，与普通内存不同，I/O寄存器不应由CPU缓存，因为每次访问都可能有副作用。实现I/O寄存器为内存区域的PCI设备通过在其配置寄存器中设置“内存可预取”位来标记差异。如果内存区域被标记为可预取，CPU可以缓存其内容并对其进行各种优化；另一方面，非预取内存访问不能被优化，因为每次访问都可能有副作用，就像I/O端口一样。将其控制寄存器映射到内存地址范围的外设将该范围声明为非预取，而像PCI板上的视频内存这样的东西是可预取的。在本节中，我们使用单词区域来指代通用的I/O地址空间，无论是内存映射还是端口映射。

接口板使用配置寄存器报告其区域的大小和当前位置——如图12-2所示的六个32位寄存器，其符号名为PCI_BASE_ADDRESS_0到PCI_BASE_ADDRESS_5。由于PCI定义的I/O空间是一个32位地址空间，因此对于内存和I/O使用相同的配置接口是有意义的。如果设备使用64位地址总线，它可以通过为每个区域使用两个连续的

PCI_BASE_ADDRESS寄存器来在64位内存空间中声明区域，先低位。一个设备可能同时提供32位区域和64位区域。

在内核中，PCI设备的I/O区域已经被集成到通用资源管理中。因此，你不需要访问配置变量就可以知道你的设备在内存或I/O空间中的映射位置。获取区域信息的首选接口包括以下函数：

```c
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
```

该函数返回与六个PCI I/O区域之一相关联的第一个地址（内存地址或I/O端口号）。区域由整数bar（基地址寄存器）选择，范围从0-5（包含）。

```c
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
```

该函数返回是I/O区域编号bar的一部分的最后一个地址。注意，这是最后可用的地址，而不是区域后的第一个地址。

```c
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

此函数返回与此资源相关联的标志。

资源标志用于定义单个资源的一些特性。对于与PCI I/O区域相关联的PCI资源，信息是从基地址寄存器中提取的，但对于与PCI设备无关的资源，信息可以来自其他地方。

所有资源标志都在<linux/ioport.h>中定义；最重要的是：

`IORESOURCE_IO` `IORESOURCE_MEM` 如果相关的I/O区域存在，这些标志中的一个且只有一个会被设置。

`IORESOURCE_PREFETCH` `IORESOURCE_READONLY` 这些标志告诉是否可以预取内存区域和/或写保护。后一个标志永远不会为PCI资源设置。

通过使用`pci_resource_`函数，设备驱动程序可以完全忽略底层的PCI寄存器，因为系统已经使用它们来构造资源信息。

12.1.10. PCI 中断

就中断而言，PCI很容易处理。当Linux启动时，计算机的固件已经为设备分配了一个唯一的中断号，驱动程序只需要使用它。中断号存储在配置寄存器60（PCI_INTERRUPT_LINE）中，宽度为一个字节。这允许多达256条中断线，但实际限制取决于使用的CPU。

驱动程序不需要去检查中断号，因为在PCI_INTERRUPT_LINE中找到的值保证是正确的。

如果设备不支持中断，寄存器61（PCI_INTERRUPT_PIN）为0；否则，它是非零。然而，由于驱动程序知道其设备是否由中断驱动，所以它通常不需要读取PCI_INTERRUPT_PIN。

因此，处理中断的PCI特定代码只需要读取配置字节以获取保存在本地变量中的中断号，如下面的代码所示。除此之外，第10章的信息适用。

```c
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE,
&myirq);

if (result) {

    /* deal with error */

}
```

本节的其余部分为好奇的读者提供了额外的信息，但写驱动程序时并不需要。

PCI连接器有四个中断引脚，外设板可以使用任何一个或全部。每个引脚都单独路由到主板的中断控制器，所以中断可以在没有任何电气问题的情况下共享。然后，中断控制

器负责将中断线（引脚）映射到处理器的硬件；这个依赖于平台的操作留给控制器，以便在总线本身实现平台独立。

位于PCI_INTERRUPT_PIN的只读配置寄存器用于告诉计算机实际使用的单个引脚。值得记住的是，每个设备板可以承载多达八个设备；每个设备使用一个单独的中断引脚，并在其自己的配置寄存器中报告。同一设备板上的不同设备可以使用不同的中断引脚或共享同一个。

另一方面，PCI_INTERRUPT_LINE寄存器是读/写的。当计算机启动时，固件扫描其PCI设备，并根据其PCI插槽的中断引脚路由情况为每个设备设置寄存器。该值由固件分配，因为只有固件知道主板如何将不同的中断引脚路由到处理器。然而，对于设备驱动程序来说，PCI_INTERRUPT_LINE寄存器是只读的。有趣的是，最近版本的Linux内核在某些情况下可以分配中断线，而不需要依赖BIOS。

12.1.11. 硬件抽象

通过快速查看系统如何处理市场上众多的PCI控制器，我们完成了对PCI的讨论。这只是一个信息部分，旨在向好奇的读者展示内核的面向对象布局如何扩展到最低级别。

实现硬件抽象的机制是使用包含方法的常规结构。这是一种强大的技术，只增加了对指针解引用的最小开销到函数调用的正常开销。在PCI管理的情况下，唯一依赖硬件的操作是读取和写入配置寄存器的操作，因为在PCI世界中的其他所有事情都是通过直接读取和写入I/O和内存地址空间来完成的，而这些都直接受CPU的控制。

因此，用于配置寄存器访问的相关结构只包括两个字段：

```c
struct pci_ops {

    int (*read)(struct pci_bus *bus, unsigned int devfn,
int where, int size,

                u32 *val);

    int (*write)(struct pci_bus *bus, unsigned int devfn,
int where, int size,

                 u32 val);

};
```

该结构在<linux/pci.h>中定义，并由drivers/pci/pci.c使用，其中定义了实际的公共函数。

对PCI配置空间进行操作的两个函数比解引用一个指针有更多的开销；由于代码的高度面向对象性，它们使用级联指针，但在很少执行的操作中，以及在速度关键路径中，这种开销不是问题。例如，pci_read_config_byte(dev, where, val)的实际实现扩展为：

```c
dev->bus->ops->read(bus, devfn, where, 8, val);
```

系统中的各种PCI总线在系统启动时被检测到，这时创建struct pci_bus项，并与其特性关联，包括ops字段。

通过“硬件操作”数据结构实现硬件抽象在Linux内核中是典型的。一个重要的例子是struct alpha_machine_vector数据结构。它在<asm-alpha/machvec.h>中定义，并处理可能在不同的基于Alpha的计算机之间变化的所有事情。

12.2. 回顾: ISA

ISA总线的设计相当老旧，性能表现糟糕，但它仍然占据了扩展设备市场的很大一部分。如果速度不重要，并且你想支持旧的主板，那么ISA实现比PCI更可取。这个老标准

的另一个优点是，如果你是一个电子爱好者，你可以轻松地构建自己的ISA设备，这是PCI绝对无法做到的。

另一方面，ISA的一个很大的缺点是它与PC架构紧密绑定；接口总线具有80286处理器的所有限制，并给系统程序员带来无尽的痛苦。ISA设计的另一个大问题（从原始的IBM PC继承）是缺乏地理寻址，这导致了许多问题和长时间的拔插-重接-插入-测试周期以添加新设备。有趣的是，即使是最早的Apple II电脑也已经在利用地理寻址，并且它们具有无跳线的扩展板。

尽管ISA有很大的缺点，但它仍然在一些意想不到的地方被使用。例如，用于几种掌上电脑的MIPS处理器VR41xx系列具有一个ISA兼容的扩展总线，尽管这看起来很奇怪。这些意想不到的ISA使用背后的原因是一些遗留硬件的极低成本，如基于8390的以太网卡，所以一个具有ISA电气信号的CPU可以轻松地利用那些糟糕但便宜的PC设备。

12.2.1. 硬件资源

ISA设备可以配备I/O端口、内存区域和中断线。尽管x86处理器支持64KB的I/O端口内存（即，处理器断言16个地址线），但一些旧的PC硬件只解码最低的10个地址线。这限制了可用的地址空间到1024个端口，因为在1KB到64KB的范围内的任何地址都会被任何只解码低地址线的设备误认为是低地址。一些外设通过只将一个端口映射到低千字节，并使用高地址线来选择不同的设备寄存器来规避这个限制。例如，映射在0x340的设备可以安全地使用端口0x740、0xB40等。

如果I/O端口的可用性有限，内存访问还会更糟。ISA设备只能使用640KB到1MB和15MB到16MB之间的内存范围用于I/O寄存器和设备控制。640KB到1MB的范围被PC BIOS、VGA兼容的视频板和各种其他设备使用，留给新设备的空间很少。另一方面，15MB的内存并未直接得到Linux的支持，现在修改内核以支持它是浪费编程时间。

ISA设备板可用的第三种资源是中断线。有限数量的中断线路由到ISA总线，它们被所有的接口板共享。结果是，如果设备没有正确配置，它们可能会发现自己使用相同的中断线。

尽管原始的ISA规范不允许设备之间共享中断，但大多数设备板允许它。在软件级别的中断共享在第10章的“中断共享”部分中描述。

12.2.2. ISA 编程

就编程而言，内核或BIOS中没有特定的辅助功能来简化对ISA设备的访问（例如，对于PCI就有）。你可以使用的唯一设施是I/O端口和IRQ线的注册表，在第10章的“安装中断处理程序”部分中有描述。

本书第一部分展示的编程技术适用于ISA设备；驱动程序可以探测I/O端口，中断线必须使用第10章的“自动检测IRQ号”部分中展示的技术之一进行自动检测。

在第9章的“使用I/O内存”部分中，已经简要介绍了辅助函数isa_readb及其朋友们，关于它们没有更多要说的。

12.2.3. 即插即用规范Plug-and-Play

一些新的ISA设备板遵循特殊的设计规则，并需要一个特殊的初始化序列，旨在简化附加接口板的安装和配置。这些板的设计规范被称为即插即用（PnP），它包括一套繁琐的规则集，用于构建和配置无跳线的ISA设备。PnP设备实现了可重定位的I/O区域；PC的BIOS负责重定位——这让人想起了PCI。

简而言之，PnP的目标是在不改变底层电气接口（ISA总线）的情况下，获得与PCI设备相同的灵活性。为此，规格定义了一组设备无关的配置寄存器和一种地理寻址接口板的方式，尽管物理总线并不携带每个板（地理）布线——每个ISA信号线连接到每个可用的插槽。

地理寻址通过为计算机中的每个PnP外设分配一个小整数，称为卡选择号（CSN），来工作。每个PnP设备都有一个唯一的序列标识符，宽64位，硬连线到外设板上。CSN分配使用唯一的序列号来识别PnP设备。但是，CSN只能在启动时安全地分配，这需要BIOS具有PnP感知。因此，旧电脑需要用户获取并插入特定的配置磁盘，即使设备具有PnP能力。

遵循PnP规格的接口板在硬件级别上很复杂。它们比PCI板更复杂，需要复杂的软件。安装这些设备时遇到困难是很常见的，即使安装顺利，你仍然面临ISA总线的性能限制和有限的I/O空间。尽可能安装PCI设备，并享受新技术，这样会好得多。

如果你对PnP配置软件感兴趣，你可以浏览drivers/net/3c509.c，其探测功能处理PnP设备。2.6内核在PnP设备支持区域做了很多工作，所以与之前的内核版本相比，许多不灵活的接口已经被清理掉了。

12.3. PC/104 和 PC/104+

在当前的工业世界中，有两种总线架构非常流行：PC/104和PC/104+。它们都是PC级单板计算机的标准。

这两个标准都指的是印刷电路板的特定形状因素，以及板间连接的电气/机械规格。这些总线的实际优点是，它们允许电路板使用设备一侧的插头和插座类型的连接器垂直堆叠。

这两个总线的电气和逻辑布局与ISA（PC/104）和PCI（PC/104+）相同，所以软件不会注意到通常的桌面总线和这两个总线之间的任何区别。

12.4. 其他的 PC 总线

在PC世界中，PCI和ISA是最常用的外设接口，但它们并不是唯一的。以下是在PC市场中发现的其他总线的特性概述。

1. **EISA**：扩展的ISA（EISA）是一种16位总线，它在物理上与ISA兼容，但在逻辑上提供了更多的功能。EISA总线的主要优点是它支持总线主控制，这使得多个设备可以同时在总线上进行操作。然而，由于其高昂的成本，EISA总线主要用于高端服务器和工作站。
2. **MCA**：微通道架构（MCA）是IBM为其PS/2系列计算机开发的一种总线。MCA提供了许多先进的特性，包括总线主控制和即插即用功能。然而，由于IBM对MCA的严格许可政策，这种总线没有在PC市场上获得广泛的接受。
3. **VESA Local Bus**：VESA本地总线（VLB）是一种32位总线，主要用于图形卡。VLB提供了与处理器的直接连接，从而提供了高带宽的数据传输。然而，由于VLB的物理设计限制了其在高速处理器上的使用，这种总线已经被PCI所取代。
4. **AGP**：加速图形端口（AGP）是一种专为图形卡设计的总线。AGP提供了直接访问系统内存的能力，从而允许图形卡快速处理大量的图形数据。然而，随着PCI Express的出现，AGP已经逐渐被淘汰。
5. **PCI Express**：PCI Express（PCIe）是一种高速串行计算机扩展总线标准，设计用于替代旧的PCI、PCI-X和AGP总线标准。PCIe提供了更高的系统总线带宽和更好的可扩展性，已经成为现代PC和服务器的主要总线标准。
6. **USB**：通用串行总线（USB）是一种连接计算机和外部设备的标准。USB提供了即插即用功能，允许设备在不重新启动计算机的情况下连接和断开。USB已经成为连接键盘、鼠标、打印机、摄像头等设备的主要方式。
7. **Thunderbolt**：Thunderbolt是一种高速硬件接口，由Intel和Apple共同开发。Thunderbolt支持数据、视频、音频和电源在单一连接中的传输，提供了高达40Gbps的带宽。

12.4.1. MCA总线

微通道架构（MCA）是IBM在PS/2计算机和一些笔记本电脑中使用的标准。在硬件级别，微通道比ISA具有更多的功能。它支持多主DMA，32位地址和数据线，共享中断线，以及地理寻址来访问每个板的配置寄存器。这样的寄存器被称为可编程选项选择（POS），但它们并没有所有的PCI寄存器的功能。Linux对微通道的支持包括导出给模块的函数。

设备驱动程序可以读取整数值MCA_bus，以查看它是否在微通道计算机上运行。如果该符号是预处理器宏，那么宏MCA_bus**is_a_macro也被定义。如果MCA_bus**is_a_macro未定义，那么MCA_bus是导出给模块化代码的整数变量。MCA_BUS和MCA_bus__is_a_macro都在<asm/processor.h>中定义。

12.4.2. EISA 总线

扩展ISA（EISA）总线是ISA的32位扩展，具有兼容的接口连接器；ISA设备板可以插入到EISA连接器中。额外的线路在ISA接触点下面布线。

像PCI和MCA一样，EISA总线设计为承载无跳线设备，并且具有与MCA相同的功能：32位地址和数据线，多主DMA和共享中断线。EISA设备由软件配置，但它们不需要任何特定的操作系统支持。Linux内核中已经存在用于以太网设备和SCSI控制器的EISA驱动程序。

EISA驱动程序检查EISA_bus的值，以确定主机计算机是否携带EISA总线。像MCA_bus一样，EISA_bus是宏或变量，取决于是否定义了EISA_bus__is_a_macro。这两个符号都在<asm/processor.h>中定义。

内核对具有sysfs和资源管理功能的设备提供了完全的EISA支持。这位于drivers/eisa目录中。

12.4.3. VLB 总线

ISA的另一个扩展是VESA本地总线（VLB）接口总线，它通过添加第三个纵向插槽来扩展ISA连接器。设备可以直接插入这个额外的连接器（无需插入两个相关的ISA连接器），因为VLB插槽复制了ISA连接器的所有重要信号。这种不使用ISA插槽的“独立”VLB外设很少见，因为大多数设备需要到达后面板，以便它们的外部连接器可用。

VESA总线的能力比EISA，MCA和PCI总线要有限得多，并且正在从市场上消失。VLB没有特殊的内核支持。然而，Linux 2.0中的Lance以太网驱动程序和IDE磁盘驱动程序都可以处理他们设备的VLB版本。

12.5. SBus

虽然现在的大多数计算机都配备了PCI或ISA接口总线，但大多数较旧的基于SPARC的工作站使用SBus来连接它们的外设。

SBus是一个相当先进的设计，尽管它已经存在了很长时间。它旨在独立于处理器（尽管只有SPARC计算机使用它），并且针对I/O外设板进行了优化。换句话说，你不能将

额外的RAM插入SBus插槽（RAM扩展板在ISA世界中已经被遗忘很久了，PCI也不支持它们）。这种优化旨在简化硬件设备和系统软件的设计，但需要在主板上增加一些复杂性。

总线的这种I/O偏向导致外设使用虚拟地址传输数据，从而绕过分配连续DMA缓冲区的需要。主板负责解码虚拟地址并将它们映射到物理地址。这需要将MMU（内存管理单元）连接到总线；负责这项任务的芯片组被称为IOMMU。尽管在接口总线上使用物理地址比这种设计更复杂，但由于SPARC处理器始终通过将MMU核心与CPU核心分开（物理上或至少在概念上）来设计，这种设计大大简化了。

实际上，这种设计选择也被其他智能处理器设计所共享，并且总体上是有益的。这个总线的另一个特性是设备板利用大量的地理寻址，所以没有必要在每个外设中实现地址解码器或处理地址冲突。

SBus外设在它们的PROM中使用Forth语言来初始化自己。选择Forth是因为解释器轻量级，因此可以轻松地在任何计算机系统的固件中实现。此外，SBus规范概述了引导过程，以便符合规范的I/O设备可以轻松地适应系统，并在系统引导时被识别。这是支持多平台设备的一个重要步骤；这与我们习惯的以PC为中心的ISA东西完全是两个世界。然而，由于各种商业原因，它并没有成功。

尽管当前的内核版本为SBus设备提供了相当全面的支持，但现在这种总线的使用非常少，因此在这里详细介绍并不值得。感兴趣的读者可以查看arch/sparc/kernel和arch/sparc/mm中的源文件。

12.6. NuBus 总线

另一个有趣但几乎被遗忘的接口总线是NuBus。它可以在较旧的Mac电脑上找到（那些使用M68k CPU系列的电脑）。

总线的所有部分都是内存映射的（像M68k的所有东西一样），设备只进行地理寻址。这很好，也是Apple的典型特点，因为更早的Apple II已经有了类似的总线布局。不好的是，由于Apple一直对其Mac电脑采取的封闭一切的政策，几乎不可能找到关于NuBus的文档（与之前的Apple II不同，其源代码和电路图的获取成本很低）。

文件drivers/nubus/nubus.c包含了我们对这个总线的几乎所有了解，这是一篇有趣的阅读；它展示了开发人员必须进行多少艰难的逆向工程。

12.7. 外部总线

在接口总线领域，最近的一项是整个类别的外部总线。这包括USB，FireWire和IEEE1284（基于并行端口的外部总线）。这些接口在某种程度上类似于较旧且不那么外部的技术，如PCMCIA/CardBus甚至SCSI。

从概念上讲，这些总线既不是全功能的接口总线（如PCI），也不是愚蠢的通信通道（如串行端口）。很难分类需要利用它们的特性的软件，因为它通常分为两个级别：硬件控制器的驱动程序（如PCI SCSI适配器的驱动程序或在“PCI接口”部分介绍的PCI控制器的驱动程序）和特定“客户端”设备的驱动程序（如sd.c处理通用SCSI磁盘和所谓的PCI驱动程序处理插入总线的卡）。

12.8. 快速参考

本节总结在本章中介绍的符号:

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

包含 PCI 寄存器的符号名和几个供应商和设备 ID 值的头文件.

```
struct pci_dev;
```

表示内核中一个 PCI 设备的结构.

```
struct pci_driver;
```

代表一个 PCI 驱动的结构. 所有的 PCI 驱动必须定义这个.

```
struct pci_device_id;
```

描述这个驱动支持的 PCI 设备类型的结构.

```
int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
void pci_unregister_driver(struct pci_driver *drv);
```

从内核注册或注销一个 PCI 驱动的函数.

```
struct pci_dev *pci_find_device(unsigned int vendor,
unsigned int device, struct pci_dev *from);
struct pci_dev *pci_find_device_reverse(unsigned int
vendor, unsigned int device, const struct pci_dev *from);
struct pci_dev *pci_find_subsys (unsigned int vendor,
unsigned int device, unsigned int ss_vendor, unsigned int
ss_device, const struct pci_dev *from);
struct pci_dev *pci_find_class(unsigned int class, struct
pci_dev *from);
```

在设备列表中搜寻带有一个特定签名的设备, 或者属于一个特定类的. 返回值是 NULL 如果没找到. from 用来继续一个搜索; 在你第一次调用任一个函数时它必须是 NULL, 并且它必须指向刚刚找到的设备如果你寻找更多的设备. 这些函数不推荐使用, 用 pci*get* 变体来代替.

```
struct pci_dev *pci_get_device(unsigned int vendor,
unsigned int device, struct pci_dev *from);
struct pci_dev *pci_get_subsys(unsigned int vendor,
unsigned int device, unsigned int ss_vendor, unsigned int
ss_device, struct pci_dev *from);
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned
int devfn);
```

在设备列表中搜索一个特定签名的设备, 或者属于一个特定类. 返回值是 NULL 如果没找到. from 用来继续一个搜索; 在你第一次调用任一个函数时它必须是 NULL, 并且它必须指向刚刚找到的设备如果你寻找更多的设备. 返回的结构使它的引用计数递增, 并且在调用者完成它, 函数 pci_dev_put 必须被调用.

```
int pci_read_config_byte(struct pci_dev *dev, int where,
u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where,
u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where,
u32 *val);
int pci_write_config_byte (struct pci_dev *dev, int where,
u8 *val);
int pci_write_config_word (struct pci_dev *dev, int where,
u16 *val);
int pci_write_config_dword (struct pci_dev *dev, int
where, u32 *val);
```

读或写 PCI 配置寄存器的函数. 尽管 Linux 内核负责字节序, 程序员必须小心字节序当从单个字节组合多字节值时. PCI 总线是小端.

```
int pci_enable_device(struct pci_dev *dev);
```

使能一个 PCI 设备.

```
unsigned long pci_resource_start(struct pci_dev *dev, int
bar);
unsigned long pci_resource_end(struct pci_dev *dev, int
bar);
unsigned long pci_resource_flags(struct pci_dev *dev, int
bar);
```

处理 PCI 设备资源的函数.