第十二章

PCI 驱动程序

虽然第 9 章介绍了最低级别的硬件控制,但本章提供了更高级别总线体系结构的概述。总线由电气接口和编程接口组成。在本章中,我们处理编程接口。

本章涵盖了许多总线架构。然而,主要关注的是访问外围组件互连 (PCI) 外围设备的内核功能,因为如今 PCI 总线是台式机和大型计算机上最常用的外围总线。总线是内核支持最好的总线。 ISA 对于电子爱好者来说仍然很常见,稍后会进行介绍,尽管它几乎是一种裸机类型的总线,除了第 9 章和第 10 章介绍的内容之外没有什么可说的。

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,但它实际上是一套完整的规范,定义了计算机的不同部分应如何交互。

PCI 规范涵盖了与计算机接口相关的大多数问题。我们不会在这里涵盖所有内容。在本节中,我们主要关注 PCI 驱动程序如何找到其硬件并访问它。第 2 章的“模块参数”和第 10 章的“自动检测 IRQ 号”部分讨论的探测技术可用于 PCI 设备,但规范提供了一种优于探测的替代方法。

PCI 架构被设计为 ISA 标准的替代品,具有三个主要目标:在计算机与其外围设备之间传输数据时获得更好的性能,尽可能独立于平台,以及简化添加和删除外围设备到系统。

PCI 总线通过使用比 ISA 更高的时钟频率来实现更好的性能;它的时钟运行在 25 或 33 MHz（其实际速率是系统时钟的一个因素），并且最近也部署了 66-MHz 甚至 133-MHz 的实现。

此外,它配备了 32 位数据总线,并在规范中包含了 64 位扩展。平台独立性通常是计算机总线设计的一个目标,它是 PCI 的一个特别重要的特性,因为 PC 世界一直由特定于处理器的接口标准主导。 PCI 目前广泛用于 IA-32、Alpha、PowerPC、SPARC64 和 IA-64 系统以及其他一些平台。

然而,与驱动程序编写者最相关的是 PCI 对接口板自动检测的支持。 PCI 设备是无跳线的（与大多数较旧的外围设备不同)并且在引导时自动配置。然后,设备驱动程序必须能够访问设备中的配置信息才能完成初始化。

这无需执行任何探测即可发生。

PCI 寻址每个 PCI 外

围设备由总线号、设备号和功能号标识。 PCI 规范允许单个系统承载多达 256 条总线,但由于 256 条总线对于许多大型系统来说是不够的,Linux 现在支持 PCI域。每个 PCI 域最多可承载 256 条总线。每条总线最多可承载 32 台设备,每台设备可以是一个多功能板（如附带 CD-ROM 驱动器的音频设备），最多具有 8 个功能。因此,每个功能都可以通过 16 位地址或密钥在硬件级别进行标识。但是,为 Linux 编写的设备驱动程序不需要处理这些二进制地址,因为它们使用称为 pci_dev的特定数据结构来作用于设备。

最新的工作站具有至少两条 PCI 总线。在一个系统中插入多条总线是通过桥接来完成的,桥接是专用的 PCI 外围设备,其任务是连接两条总线。 PCI 系统的整体布局是一棵树,其中每条总线都连接到上层总线,直到树根处的总线 0。 CardBus PC 卡系统也通过网桥连接到 PCI 系统。典型的 PCI 系统如图 12-1 所示,其中突出显示了各种桥接器。

与 PCI 外围设备相关的 16 位硬件地址,虽然大部分隐藏在struct pci_dev对象中,但偶尔仍然可见,尤其是在使用设备列表时。一种这样的情况是lspci的输出（ pciutils包的一部分,可用于大多数发行版)和/proc/pci和/proc/bus/pci 中的信息布局。 PCI 设备的 sysfs 表示也显示了这种寻址方案,并添加了 PCI 域信息。*当显示硬件地址时,它可以显示为两个值（一个 8 位总线号和一个 8 位

* 一些架构还在/proc/pc和/proc/bus/pci文件中显示 PCI 域信息。

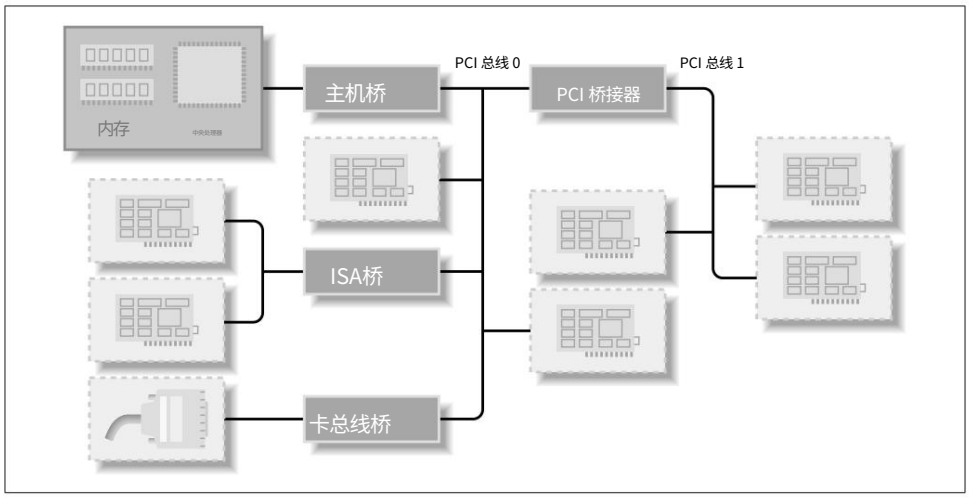

图 12-1。典型 PCI 系统的布局

设备和功能编号），作为三个值（总线、设备和功能），或作为四个值（域、总线、设备和功能）；所有值通常以十六进制显示。

例如， /proc/bus/pci/devices使用单个 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的显示方式，仅显示了输出行的开头： $ lspci | cut -d: -f1-3 0000:00:00.0 主机桥接器 0000:00:00.1 RAM 存储器 0000:00:00.2 RAM 存储器 0000:00:02.0 USB 控制器 0000:00:04.0 多媒体音频控制器 0000:00:06.0 桥接器 0000:00:07.0 ISA 桥 0000:00:09.0 USB 控制器 0000:00:09.1 USB 控制器 0000:00:09.2 USB 控制器 0000:00:0c.0 CardBus 桥 0000:00:0f.0 IDE 接口 0000:00: 10.0 以太网控制器 0000:00:12.0 网络控制器 0000:00:13.0 FireWire (IEEE 1394)

```
0000:00:14.0 VGA 兼容控制器 $ cat /proc/bus/pci/
devices |剪切-f1
0000
0001
0002
0010
0020
0030
```

```
0038
0048
0049
004a
0060
0078
0080
0090
0098
00a0
$树 /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 -> ../../../devic es/pci0000:00/0000:00:14.0
```

所有三个设备列表都以相同的顺序排序,因为lspci使用/proc文件作为其信息源。以VGA视频控制器为例， 0x00a0在划分为域（16位）、总线（8位）、设备（5位)和功能（3位)时表示0000:00:14.0 。

每个外围板的硬件电路回答与三个地址空间有关的查询: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,当您访问内存位置时,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。配置查询一次只处理一个时隙,因此它们永远不会发生冲突。

就驱动程序而言,内存和 I/O 区域是通过inb 、 readb等以通常的方式访问的。另一方面,配置事务是通过调用特定的内核函数来访问配置寄存器来执行的。关于中断,每个 PCI 插槽都有四个中断引脚,每个设备功能都可以使用其中一个,而不用关心这些引脚是如何路由到 CPU 的。这种路由是计算机平台的职责,并在 PCI 总线之外实现。由于 PCI 规范要求中断线是可共享的,即使是具有有限数量的 IRQ 线的处理器,例如 x86,也可以托管许多 PCI 接口板（每个都有四个中断引脚）。

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 位地址在更新的平台上可用。一个设备的地址应该是唯一的，但软件可能会错误地将两个设备配置为相同的地址，从而无法访问其中任何一个。但是这个问题永远不会发生，除非驱动程序愿意使用它不应该接触的寄存器。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，因此 Linux 驱动程序可以访问其设备无需试探。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 256 字节组成，用于每个设备功能（PCI Express 设备除外，每个功能都有 4 KB 的配置空间），并且配置寄存器的布局是标准化的。配置空间的四个字节保存一个唯一的功能 ID，因此驱动程序可以通过查找该外设的特定 ID 来识别其设备。这些寄存器中的信息随后可用于执行正常的 I/O 访问，而无需进一步的地理寻址。

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之外，PCI 驱动程序还需要能够访问配置空间，以使自己免于危险的探测任务。

在本章的其余部分，我们使用设备一词来指代设备功能，因为多功能板中的每个功能都充当独立的实体。
当我们提到设备时，我们指的是元组“域号、总线号、设备号和功能号”。

开机时间

要了解 PCI 的工作原理，我们从系统引导开始，因为那是配置设备的时候。

* 实际上，该配置不限于系统启动时间；例如，热插拔设备在引导时不可用，而是稍后出现。这里的要点是设备驱动程序不得更改 I/O 或内存区域的地址。 † 您可以在其自己的硬件手册中找到任何设备的 ID。一个列表包含在 pci.ids 文件、pciutils包的一部分和内核源代码中；它并不假装完整，而只是列出了最知名的供应商和设备。该文件的内核版本将不会包含在以后的内核系列中。

当 PCI 设备通电时,硬件保持非活动状态。换句话说,设备只响应配置事务。上电时,设备没有内存,也没有映射到计算机地址空间的 I/O 端口;所有其他特定于设备的功能,例如中断报告,也被禁用。

幸运的是,每个 PCI 主板都配备了支持 PCI 的固件,称为 BIOS、NVRAM 或 PROM,具体取决于平台。固件通过读取和写入 PCI 控制器中的寄存器来提供对设备配置地址空间的访问。

在系统启动时,固件（或 Linux 内核,如果已配置)与每个 PCI 外围设备执行配置事务,以便为其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,它的内存和 I/O 区域已经映射到处理器的地址空间。

驱动程序可以更改此默认分配,但它永远不需要这样做。

按照建议,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 树中的各个 PCI 设备目录可以在/sys/bus/pci/devices 中找到。 PCI 设备目录包含许多不同的文件：

```
$树 /sys/bus/pci/devices/0000:00:10.0 /sys/bus/
pci/devices/0000:00:10.0 |-- 类 |-- 配置 |-- 分离状
态 |-- 设备 |-- irq |-- 电源 | `-- 状态 |-- 资源 |-- 子系
统设备 |-- 子系统供应商 `-- 供应商
```

文件config是一个二进制文件,允许从设备读取原始 PCI 配置信息（就像/proc/bus/pci/*/*提供的一样。)文件vendor、 device、 subsystem_device、 subsystem_vendor和class all参考这个PCI设备的具体值（所有的PCI设备都提供这个信息）。文件irq显示了当前分配给这个PCI设备的IRQ,文件resource显示了这个设备当前分配的内存资源。

配置寄存器和初始化

在本节中,我们将了解 PCI 设备包含的配置寄存器。全部
PCI 设备具有至少 256 字节的地址空间。前 64 个字节是标准化的,其余的则取决于设备。图
12-2 显示了器件独立配置空间的布局。

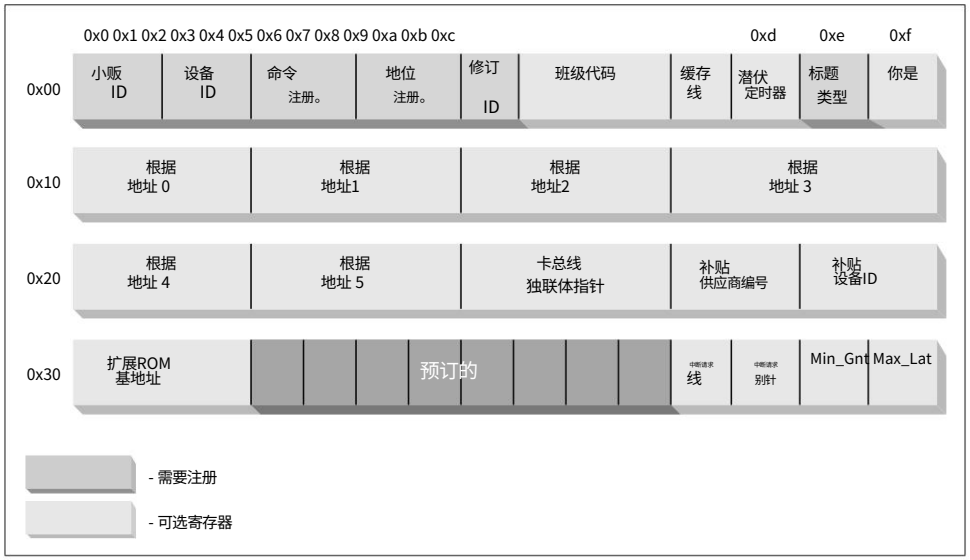

图 12-2。标准化的 PCI 配置寄存器

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 设备都必须在所需寄存器中包含有意义的值,而可选寄存器的内容取决于实际功能
的外围设备。除非必填项的内容,否则不使用可选字段
字段表明它们是有效的。因此,必填字段声明了板的功能,包括其他字段是否可用。

有趣的是,PCI 寄存器始终是 little-endian。虽然
标准被设计为独立于架构,PCI 设计者有时
显示出对 PC 环境的轻微偏见。驱动作者要小心
关于访问多字节配置寄存器时的字节顺序;代码
在 PC 上运行可能无法在其他平台上运行。 Linux 开发人员有
处理了字节顺序问题(请参阅下一节,“访问配置空间”),但必须牢记这个问题。如果您需要转换数据
从主机订单到 PCI 订单,反之亦然,您可以使用中定义的函数
<asm/byteorder.h>,在第 11 章中介绍,知道 PCI 字节顺序是小端。

描述所有配置项超出了本书的范围。通常,随每个设备发布的技术文档描述了支持的寄存器。

我们感兴趣的是驱动程序如何查找其设备以及如何访问设备的配置空间。

三个或五个 PCI 寄存器标识一个设备： 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,有时供应商会设置字段子系统供应商 ID和子系统设备 ID ,以进一步区分类似设备。

让我们更详细地看一下这些寄存器：

vendorID

这个 16 位寄存器标识硬件制造商。例如,每个英特尔设备都标有相同的供应商编号0x8086。 PCI 特别兴趣小组维护了此类编号的全球注册尝试,制造商必须申请为其分配一个唯一编号。

deviceID

这是另一个 16 位寄存器,由制造商选择;设备 ID 无需正式注册。此 ID 通常与供应商 ID 配对,为硬件设备创建一个唯一的 32 位标识符。我们使用签名一词来指代供应商和设备 ID 对。设备驱动程序通常依靠签名来识别其设备。您可以在目标设备的硬件手册中找到要查找的值。

类每

个外围设备都属于一个类。类寄存器是一个 16 位值,其前 8 位标识“基类”（或组）。例如,“ethernet”和“token ring”是属于“network”组的两个类,而“serial”和“parallel”类属于“communication”组。一些驱动程序可以支持几个类似的设备,每个设备都有不同的签名,但都属于同一类;这些驱动程序可以依靠类寄存器来识别它们的外围设备,如下所示。

子系统供应商ID 子系
统设备ID这些字段可

用于进一步识别设备。如果芯片是本地（板载）总线的通用接口芯片,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与之通信的实际设备。为此使用子系统标识符。

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设备。 struct pci_device_id结构用于定义不同的列表

驱动程序支持的 PCI 设备类型。此结构包含以下字段：

__u32 供应商；
__u32 设备;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,则值PCI_ANY_ID应用于这些字段。 __u32 子供应商； __u32 子设备;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,则值PCI_ANY_ID应该用于这些字段。

__u32 类；
__u32 类掩码;这两个
值允许驱动程序指定它支持一种 PCI 类设备。 PCI 规范中描述了不同类别的 PCI 设备（例如 VGA 控制器）。如果驱动程序可以处理任何类型的子系统 ID,则值PCI_ANY_ID应该用于这些字段。

kernel_ulong_t driver_data；
此值不用于匹配设备,但用于保存 PCI 驱动程序可用于区分不同设备的信息（如果需要）。

应该使用两个辅助宏来初始化struct pci_device_id
结构体：

PCI_DEVICE（供应商,设备）
这将创建一个仅匹配特定供应商和设备 ID 的结构 pci_device_id 。该宏将结构的子供应商和子设备字段设置为PCI_ANY_ID 。

PCI_DEVICE_CLASS（设备类,设备类掩码）
这将创建一个与特定 PCI 类匹配的struct pci_device_id 。

可以在以下内核文件中找到使用这些宏定义驱动程序支持的设备类型的示例：

驱动程序/USB/主机/ehci-hcd.c：

```
static const struct pci_device_id pci_ids[ ] = { /* 处理任何 USB 2.0 EHCI 控
        制器 */ PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) |
        0x20), ~0), .driver_data = (unsigned long) &ehci_driver, }, { /* 结束:全零 */ }


};
```

驱动程序/i2c/busses/i2c-i810.c：

```
静态结构 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结构列表,其中一个空结构设置为全零作为列表中的最后一个值。这个 ID 数组用于struct pci_driver （如下所述）,它还用于告诉用户空间该特定驱动程序支持哪些设备。

MODULE_DEVICE_TABLE这个

pci_device_id结构需要导出到用户空间,以允许热插拔和模块加载系统知道什么模块与什么硬件设备一起工作。
宏MODULE_DEVICE_TABLE完成了这项工作。一个例子是：

```
MODULE_DEVICE_TABLE(pci,
```

i810_ids);此语句创建一个名为__mod_pci_device_table的局部变量,该变量指向struct pci_device_id列表。稍后在内核构建过程中， depmod程序在所有模块中搜索符号__mod_pci_device_table。如果找到该符号,它将从模块中提取数据并将其添加到文件/lib/modules/KERNEL_VERSION/modules.pcimap中。 depmod完成后,内核中的模块支持的所有 PCI 设备及其模块名称都会在该文件中列出。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文件来找到要加载的正确驱动程序。

注册 PCI 驱动程序 为了正确注册到内

核,所有 PCI 驱动程序必须创建的主要结构是struct pci_driver结构。此结构由许多函数回调和变量组成，这些函数向 PCI 核心描述 PCI 驱动程序。

以下是 PCI 驱动程序需要注意的结构中的字段：

常量字符*名称;司机姓

名。它在内核中的所有 PCI 驱动程序中必须是唯一的,并且通常设置为与驱动程序的模块名称相同的名称。当驱动程序在内核中时,它会显示在/sys/bus/pci/drivers/下的 sysfs 中。

const struct pci_device_id *id_table;指向本

章前面描述的struct pci_device_id表的指针。

int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
指向 PCI 驱动程序中的探测函数的指针。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,这个函数会被 PCI 核心调用。指向PCI 核心用来做出此决定的struct pci_device_id的指针也传递给此函数。如果 PCI 驱动程序声明了传递给它的struct pci_dev ,它应该正确初始化设备并返回0。如果驱动程序不想声明该设备,或者发生错误,它应该返回一个负错误值。有关此功能的更多详细信息将在本章后面介绍。 void (*remove) (struct pci_dev *dev);

指向当struct pci_dev从系统中删除或 PCI 驱动程序从内核中卸载时 PCI 内核调用的函数的指针。有关此功能的更多详细信息将在本章后面介绍。 int (*suspend) (struct pci_dev *dev, u32 state);当struct pci_dev被挂起时,指向 PCI 核心调用的函数的指针。挂起状态在状态变量中传递。该功能是可选的;司机不必提供。 int (*resume) (struct pci_dev *dev);

指向当struct pci_dev被恢复时 PCI 核心调用的函数的指针。它总是在挂起被调用之后被调用。该功能是可选的;司机不必提供。

总之,要创建一个合适的struct pci_driver结构,只需要初始化四个字段:

```
静态结构 pci_driver pci_driver = { .name =
        pci_skel    , .id_table = ids, .probe = 探
    针, .remove = 删除,

};
```

要将struct pci_driver注册到 PCI 内核,使用指向 struct pci_driver 的指针调用 pci_register_driver 。这通常在 PCI 驱动程序的模块初始化代码中完成:

```
静态 int __init pci_skel_init(void) {

    返回 pci_register_driver(&pci_driver);
}
```

请注意， pci_register_driver函数要么返回负错误号,要么返回0 ,如果一切都成功注册。如果没有设备绑定到驱动程序,则它不返回绑定到驱动程序的设备数或错误号

司机。这是对 2.6 版本之前的内核的更改,是因为以下情况而完成的:

- 在支持 PCI 热插拔的系统或 CardBus 系统上,PCI 设备可以随时出现或消失。如果可以在设备出现之前加载驱动程序,这有助于减少初始化设备所需的时间。
- 2.6 内核允许在加载驱动程序后将新的 PCI ID 动态分配给驱动程序。这是通过在 sysfs 的所有 PCI 驱动程序目录中创建的文件new_id完成的。如果正在使用内核尚不知道的新设备,这将非常有用。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。如果在系统中存在设备之前不允许加载驱动程序,则此接口将无法工作。

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。当此调用发生时,当前绑定到此驱动程序的所有 PCI 设备都将被删除,并且此 PCI 驱动程序的删除函数在pci_unregister_driver函数之前调用

化回报。

```
静态无效__exit pci_skel_exit（无效){

    pci_unregister_driver(&pci_driver);
}
```

旧式 PCI 探测在旧内核版本中,

函数pci_register_driver并不总是被 PCI 驱动程序使用。相反,他们要么手动遍历系统中的 PCI 设备列表,要么调用可以搜索特定 PCI 设备的函数。从 2.6 内核中删除了在驱动程序中遍历系统中 PCI 设备列表的功能,以防止驱动程序在同时删除设备时修改 PCI 设备列表而导致内核崩溃时间。

如果确实需要查找特定 PCI 设备的能力,可以使用以下功能:

struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device,
struct pci_dev *from);此函数扫描系统中当前存在的 PCI 设备列表,如果输入参数与指定的供应商和设备ID 匹配,它会增加找到的struct pci_dev变量的引用计数,并将其返回给调用者。这可以防止结构在没有任何通知的情况下消失,并确保内核不会出现 oops。驱动程序处理完函数返回的struct pci_dev后,它必须调用函数pci_dev_put来减少

将使用计数正确地返回以允许内核在设备被删除时对其进行清理。

from参数用于获取具有相同签名的多个设备;该参数应指向已找到的最后一个设备,以便可以继续搜索,而不是从列表的头部重新开始。要查找第一个设备,将 from指定为NULL。如果没有找到（更多)设备,则返回NULL 。

如何正确使用此功能的示例是：

```
结构 pci_dev *dev; dev
= pci_get_device（PCI_VENDOR_FOO,PCI_DEVICE_FOO,NULL）;如果
（开发){
    /* 使用 PCI 设备 */
    ...
    pci_dev_put(dev);
}
```

不能从中断上下文调用此函数。如果是,则会在系统日志中打印一条警告。 struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);

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

不能从中断上下文调用此函数。如果是,则会在系统日志中打印一条警告。 struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);

该函数在指定的struct pci_bus上搜索系统中的PCI 设备列表,查找该PCI 设备的指定设备和功能号。如果找到匹配的设备,则增加其引用计数并返回指向它的指针。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从中断上下文中调用。如果是,则会将警告打印到系统日志中。

启用 PCI 设备

在PCI 驱动程序的探测函数中,在驱动程序可以访问任何设备资源之前（I/O 区域或中断)PCI 设备,驱动程序必须调用pci_enable_device函数：

int pci_enable_device(struct pci_dev *dev);此功能实际上启用了设备。它唤醒设备,在某些情况下还分配其中断线和 I/O 区域。例如,CardBus 设备（在驱动程序级别已完全等同于 PCI)会发生这种情况。

访问配置空间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,因为这是它可以找出设备在内存和 I/O 空间中的映射位置的唯一方法。

由于微处理器无法直接访问配置空间,因此计算机供应商必须提供一种方法来完成它。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,与本次讨论无关,因为 Linux 提供了访问配置空间的标准接口。

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

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dev标识的设备的配置空间中读取一个、两个或四个字节。 where参数是从配置空间开始的字节偏移量。通过val指针返回从配置空间取出的值,函数的返回值为错误码。

word和dword函数将刚刚从 little-endian 读取的值转换为处理器的本机字节顺序,因此您无需处理字节顺序。

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);
将一个、两个或四个字节写入配置空间。设备像往常一样由dev标识,写入的值作为val 传递。 word和dword函数在写入外围设备之前将值转换为 little-endian 。

前面的所有函数都实现为真正调用以下函数的内联函数。如果驱动程序在任何特定时刻都无法访问struct pci_dev ,请随意使用这些函数而不是上述函数:

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,但需要struct pci_bus *和devfn变量而不是struct pci_dev *。

int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn, int
其中,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,但需要struct pci_bus *和devfn变量而不是struct pci_dev *。

使用pci_read_函数寻址配置变量的最佳方法是使用<linux/pci.h>中定义的符号名称。例如,以下小函数通过将where的符号名称传递给pci_read_config_byte 来检索设备的修订 ID:

```
静态无符号字符 skel_get_revision(struct pci_dev *dev) {

    u8 修订版;

    pci_read_config_byte(dev, PCI_REVISION_ID, &revision);返回修
    订;
}
```

访问 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它们的 I/O 寄存器,因为这通常是一种更明智的方法(如第 9 章“I/O 端口和 I/O 内存”一节所述)。然而,与普通内存不同的是,I/O 寄存器不应该被 CPU 缓存,因为每次访问都会产生副作用。将 I/O 寄存器实现为内存区域的 PCI 设备通过在其配置寄存器中设置“memory-is-prefetchable”位来标记差异。*如果内存区域被标记为可预取,CPU 可以缓存其内容并执行各种优化;另一方面,不可预取的内存访问无法优化,因为每次访问都会产生副作用,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围设备声明该范围是不可预取的,而 PCI 板上的视频内存之类的东西是可预取的。在本节中,我们使用区域一词来指代内存映射或端口映射的通用 I/O 地址空间。

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,其符号名称是PCI_BASE_ADDRESS_0到PCI_BASE_ADDRESS_5。由于 PCI 定义的 I/O 空间是 32 位地址空间,所以使用相同的配置接口是有意义的

* 信息位于基地址 PCI 寄存器的低位之一中。这些位在<linux/pci.h> 中定义。

用于内存和 I/O。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寄存器来声明 64 位内存空间中的区域,低位在前。一台设备可以同时提供 32 位区域和 64 位区域。

在内核中,PCI 设备的 I/O 区域已集成到通用资源管理中。因此,您无需访问配置变量即可了解您的设备在内存或 I/O 空间中的映射位置。

获取区域信息的首选接口包含以下函数:

unsigned long pci_resource_start(struct pci_dev *dev, int bar);
该函数返回与六个 PCI I/O 区域之一关联的第一个地址(内存地址或 I/O 端口号)。该区域由整数条(基地址寄存器)选择,范围为 0-5(含)。

unsigned long pci_resource_end(struct pci_dev *dev, int bar);该函数返回作为 I/O 区域编号栏一部分的最后一个地址。
请注意,这是最后一个可用地址,而不是区域之后的第一个地址。

无符号长 pci_resource_flags(struct pci_dev *dev, int bar);此函数返回与此资源关联的标志。

资源标志用于定义单个资源的某些特性。对于与 PCI I/O 区域相关的 PCI 资源,信息是从基地址寄存器中提取的,但对于与 PCI 设备不相关的资源,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如果关联的 I/O 区域存在,则设置这些标志中的一个且只有一个。

IORESOURCE_PREFETCH
IORESOURCE_READONLY
这些标志告诉内存区域是否可预取和/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 PCI 寄存器,因为系统已经使用它们来构造资源信息。

PCI 中断就中断而

言,PCI 很容易处理。到 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即可。中断号存储在一个字节宽的配置寄存器 60 (PCI_INTERRUPT_LINE)中。这允许作为

多达 256 条中断线,但实际限制取决于所使用的 CPU。
驱动程序不需要检查中断号,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,则寄存器 61 (PCI_INTERRUPT_PIN)为0;否则,它是非零的。然而,由于驱动程序知道它的设备是否是中断驱动的,它通常不需要读取PCI_INTERRUPT_PIN。

因此,处理中断的 PCI 专用代码只需要读取配置字节即可获得保存在局部变量中的中断号,如下面的代码所示。除此之外,第 10 章中的信息也适用。

```
结果 = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq); if
(result) { /* 处理错误 */

}
```

本节的其余部分为好奇的读者提供了额外的信息,但在编写驱动程序时不需要。

一个 PCI 连接器有四个中断引脚,外围板可以使用其中任何一个或全部。每个引脚都单独路由到主板的中断控制器,因此可以共享中断而不会出现任何电气问题。然后中断控制器负责将中断线(引脚)映射到处理器的硬件;这种依赖于平台的操作留给控制器,以实现总线本身的平台独立性。

位于PCI_INTERRUPT_PIN的只读配置寄存器用于告诉计算机实际使用了哪个引脚。值得记住的是,每个设备板最多可以托管 8 个设备;每个设备使用单个中断引脚并在其自己的配置寄存器中报告它。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相同的中断引脚。

另一方面,PCI_INTERRUPT_LINE 寄存器是读/写的。当计算机启动时,固件会扫描其 PCI 设备并根据其 PCI 插槽的中断引脚路由方式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。但是,对于设备驱动程序,PCI_INTERRUPT_LINE寄存器是只读的。有趣的是,最新版本的 Linux 内核在某些情况下可以分配中断线,而无需借助 BIOS。

硬件抽象

我们通过快速浏览系统如何处理市场上可用的过多 PCI 控制器来完成对 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延伸到最低级别。

用于实现硬件抽象的机制是包含方法的常用结构。这是一种强大的技术,它只增加了将指针解引用到函数调用的正常开销的最小开销。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 PCI 世界中的其他所有操作都是通过直接读取和写入 I/O 和内存地址空间来完成的,而那些是在 CPU 的直接控制下。

因此,配置寄存器访问的相关结构仅包括两个字段:

```
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。

作用于 PCI 配置空间的两个函数比解引用指针有更多的开销;由于代码的高度面向对象性,它们使用级联指针,但是在很少执行且从不在速度关键路径中执行的操作中,开销不是问题。例如 pci_read_config_byte(dev, where, val)的实际实现扩展为:

```
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struct pci_bus项目被创建并与它们的特性相关联,包括ops字段。

通过“硬件操作”数据结构实现硬件抽象是 Linux 内核的典型做法。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,负责处理可能在不同的基于 Alpha 的计算机上发生变化的所有内容。

回顾:ISA

ISA 总线在设计上相当陈旧,性能极差,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您想支持旧主板,则 ISA 实现比 PCI 更可取。

这个旧标准的另一个优点是,如果您是电子爱好者,您可以轻松构建自己的 ISA 设备,而 PCI 绝对不可能做到这一点。

另一方面,ISA 的一大缺点是它与 PC 架构紧密绑定。接口总线具有 80286 处理器的所有限制,给系统程序员带来无尽的痛苦。 ISA 设计（继承自最初的 IBM PC)的另一个大问题是缺乏地理寻址,这导致了许多问题和冗长的拔出-重新跳线-插入测试周期来添加新设备。有趣的是,即使是最古老的 Apple II 计算机也已经在利用地理寻址,并且它们具有无跳线扩展板。

尽管有很大的缺点,ISA 仍然在几个意想不到的地方使用。例如,几个掌上电脑中使用的 VR41xx 系列 MIPS 处理器具有与 ISA 兼容的扩展总线,这看起来很奇怪。 ISA 的这些意外使用背后的原因是某些传统硬件的成本极低,例如基于 8390 的以太网卡,因此具有 ISA 电信号的 CPU 可以轻松利用糟糕但便宜的 PC 设备。

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 64 KB 的 I/O 端口内存（即,处理器断言 16 条地址线）,一些旧的 PC 硬件只解码最低的 10 条地址线。这将可用地址空间限制为 1024 个端口,因为任何仅解码低地址行的设备都会将 1 KB 到 64 KB 范围内的任何地址误认为是低地址。一些外围设备通过仅将一个端口映射到低千字节并使用高地址线在不同的设备寄存器之间进行选择来规避这一限制。例如,映射到0x340的设备可以安全地使用端口0x740、0xB40等。

如果 I/O 端口的可用性是有限的,内存访问仍然更糟。 ISA 设备只能将 640 KB 和 1 MB 之间以及 15 MB 和 16 MB 之间的内存范围用于 I/O 寄存器和设备控制。 640-KB 到 1-MB 的范围被 PC BIOS、VGA 兼容的视频板和各种其他设备使用,为新设备留下的可用空间很小。另一方面，Linux 不直接支持 15 MB 的内存,现在破解内核来支持它是浪费编程时间。

ISA 设备板可用的第三个资源是中断线。有限数量的中断线被路由到 ISA 总线,它们由所有接口板共享。因此,如果设备配置不正确,它们会发现自己使用相同的中断线。

尽管最初的 ISA 规范不允许跨平台共享中断
设备,大多数设备板都允许。*软件级别的中断共享是
在第 10 章的“中断共享”一节中进行了描述。

ISA编程

就编程而言,内核或 BIOS 中没有特定的帮助
以简化对 ISA 设备的访问（例如,对于 PCI）。唯一的设施
您可以使用 I/O 端口和 IRQ 线的注册表,在 章节中描述
第 10 章中的“安装中断处理程序”。

贯穿本书第一部分的编程技术适用于
ISA 设备;驱动程序可以探测 I/O 端口,并且必须使用“自动检测 IRQ”部分中显示的技术之一自动检测中断线
数”第 10 章。

辅助函数isa_readb和它的朋友在第 9 章的“使用 I/O 内存”一节中已经简单介绍过,关于它们就不再赘述了。

即插即用规范

一些新的 ISA 设备板遵循特殊的设计规则,需要特殊的初始化序列,以简化附加接口板的安装和配置。这些板的设计规范称为即插即用

(PnP) 并且包含用于构建和配置无跳线的繁琐规则集
ISA 设备。 PnP 设备实现可重定位的 I/O 区域； PC 的 BIOS 负责重新定位 让人想起 PCI。

简而言之,即插即用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
定义一组独立于设备的配置寄存器和一种对接口板进行地理寻址的方法,即使物理总线不承载每块板（地理)布线 每条 ISA 信号线都连接到每个可用插槽。

地理寻址通过分配一个小整数来工作,称为卡选择
编号(CSN),连接到计算机中的每个 PnP 外围设备。每个 PnP 设备都有一个
唯一的串行标识符,64 位宽,硬连线到外围板中。
CSN 分配使用唯一的序列号来识别 PnP 设备。但是
只有在引导时才能安全地分配 CSN,这需要 BIOS 为 PnP

* 中断共享的问题是电气工程问题:如果设备驱动信号线处于非活动状态（通过应用低阻抗电压电平）,则无法共享中断。另一方面,如果
设备使用一个上拉电阻到非活动逻辑电平,共享是可能的。这是当今的常态。然而,仍然存在丢失中断事件的潜在风险,因为 ISA 中断是边沿触发的,而不是
水平触发。边缘触发的中断更容易在硬件中实现,但不适合
安全分享。

知道的。出于这个原因,即使设备支持 PnP,旧计算机也需要用户获取并插入特定的配置软盘。

遵循 PnP 规范的接口板在硬件层面很复杂。它们比 PCI 板复杂得多,并且需要复杂的软件。安装这些设备时遇到困难并不罕见,即使安装顺利,您仍然面临性能限制和 ISA 总线有限的 I/O 空间。最好尽可能安装 PCI 设备并享受新技术。

如果你对PnP配置软件感兴趣,可以浏览drivers/net/3c509.c,它的探测功能是针对PnP设备的。 2.6 内核在 PnP 设备支持方面做了很多工作,因此与之前的内核版本相比,清理了很多不灵活的接口。

PC/104 和 PC/104+

目前在工业界,两种总线架构相当流行:PC/104和PC/104+。两者都是 PC 级单板计算机的标准配置。

这两个标准都涉及印刷电路板的特定外形尺寸,以及电路板互连的电气/机械规格。这些总线的实际优势是它们允许使用设备一侧的插头和插座类型的连接器垂直堆叠电路板。

两条总线的电气和逻辑布局与 ISA (PC/104) 和 PCI (PC/104+) 相同,因此软件不会注意到通常的桌面总线与这两者之间的任何区别。

其他 PC 总线

PCI 和 ISA 是 PC 世界中最常用的外围接口,但它们并不是唯一的。以下是 PC 市场中其他总线的功能摘要。

马华

微通道架构 (MCA) 是用于 PS/2 计算机和一些膝上型计算机的 IBM 标准。在硬件层面,Micro Channel 比 ISA 有更多的功能。它支持多主机 DMA、32 位地址和数据线、共享中断线和地理寻址,以访问每板配置寄存器。此类寄存器称为可编程选项选择(POS),但它们不具备 PCI 寄存器的所有功能。 Linux 对 Micro Channel 的支持包括导出到模块的函数。

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。如果符号是预处理器宏,则还定义宏MCA_bus__is_a_ 宏。如果MCA_bus__is_a_macro未定义,则MCA_bus是导出到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。

环境安全评估

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 EISA 连接器。额外的电线在 ISA 触点下方布线。

与 PCI 和 MCA 一样,EISA 总线设计用于托管无跳线设备,它具有与 MCA 相同的特性:32 位地址和数据线、多主机 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。 EISA 驱动程序已经存在于 Linux 内核中,用于以太网设备和 SCSI 控制器。

EISA 驱动程序检查值EISA_bus以确定主机是否承载 EISA 总线。与MCA_bus 一样， EISA_bus既可以是宏也可以是变量,这取决于是否定义了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具有完整的 EISA 支持。它位于驱动程序/eisa目录中。

VLB

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。设备只需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 插槽复制来自 ISA 连接器的所有重要信号。这种不使用 ISA 插槽的“独立”VLB 外围设备很少见,因为大多数设备需要接触到后面板,以便它们的外部连接器可用。

VESA 总线的功能比 EISA、MCA 和 PCI 总线更受限制,并且正在从市场上消失。 VLB 没有特殊的内核支持。
但是,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其设备的 VLB 版本。

系统总线

虽然现在大多数计算机都配备了 PCI 或 ISA 接口总线,但大多数老式的基于 SPARC 的工作站使用 SBus 连接其外围设备。

SBus 是一种相当先进的设计,尽管它已经存在了很长时间。这是
意味着独立于处理器（即使只有 SPARC 计算机使用它)和
针对 I/O 外围板进行了优化。换句话说,你不能插入额外的
RAM 插入 SBus 插槽（RAM 扩展板早已被遗忘,即使在
ISA 世界,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,代价是
主板中的一些额外的复杂性。

总线的这种 I/O 偏差导致外设使用虚拟地址来传输数据，
从而绕过了分配连续 DMA 缓冲区的需要。主板是
负责解码虚拟地址并将它们映射到物理地址
地址。这需要在总线上附加一个 MMU（内存管理单元）；
负责该任务的芯片组称为 IOMMU。尽管在某种程度上比在接口总线上使用物理地址更复杂,但
这种设计大大简化了,因为 SPARC 处理器的设计始终保持

MMU 内核与 CPU 内核分开（物理上或至少在概念上）。
实际上,这种设计选择由其他智能处理器设计共享,并且总体上是有益的。该总线的另一个特点
是设备板利用海量地理寻址,因此无需在每个

外围或处理地址冲突。

SBus 外设在它们的 PROM 中使用 Forth 语言来初始化它们自己。
选择 Forth 是因为解释器是轻量级的,因此可以很容易地
在任何计算机系统的固件中实现。此外,系统总线规范概述了启动过程,因此兼容的 I/O 设备可
以轻松地融入系统并在系统启动时被识别。这是支持多平台设备的重要一步;这与我们以 PC 为
中心的 ISA 完全不同

习惯了。然而,由于各种商业原因,它没有成功。

尽管当前的内核版本为 SBus 设备提供了相当全功能的支持，
现在公共汽车很少使用,不值得在这里详细介绍。有兴趣的读者可以查看arch/sparc/kernel和
arch/sparc/mm中的源文件。

努布斯

另一个有趣但几乎被遗忘的接口总线是 NuBus。它是在较旧的
Mac 计算机（具有 M68k 系列 CPU 的计算机）。

所有总线都是内存映射的（就像 M68k 的所有东西一样）,并且设备
仅在地理上寻址。这是苹果的好和典型,因为

较旧的 Apple II 已经有类似的总线布局。糟糕的是,几乎不可能在 NuBus 上找到文档,因为 Apple 一直遵循其 Mac 计算机的一切政策（与以前的 Apple II 不同,其源代码和原理图可以以很少的成本获得）。

文件drivers/nubus/nubus.c包含了我们所知道的关于这个总线的几乎所有内容,读起来很有趣;它显示了逆向工程开发人员必须做的事情。

外部总线

接口总线领域的最新条目之一是整个外部总线类。这包括 USB、FireWire 和 IEEE1284（基于并行端口的外部总线）。这些接口有点类似于较旧的非外部技术,例如 PCMCIA/CardBus 甚至 SCSI。

从概念上讲,这些总线既不是功能齐全的接口总线（如 PCI）,也不是哑通信通道（如串行端口）。很难对利用其功能所需的软件进行分类,因为它通常分为两个级别:硬件控制器的驱动程序（如“PCI 接口”部分中介绍的 PCI SCSI 适配器或 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,所谓的 PCI 驱动程序处理插入总线的卡）。

快速参考

本节总结了本章介绍的符号:

#include <linux/pci.h>
包含 PCI 寄存器的符号名称以及多个供应商和设备 ID 值的标头。

结构 pci_dev;表
示内核中 PCI 设备的结构。

结构 pci_driver;表
示 PCI 驱动程序的结构。所有 PCI 驱动程序都必须定义这一点。

结构 pci_device_id;描
述此驱动程序支持的 PCI 设备类型的结构。 int pci_register_driver(struct pci_driver *drv); int pci_module_init(struct pci_driver *drv);无效 pci_unregister_driver(struct pci_driver *drv);

从内核注册或注销 PCI 驱动程序的函数。

struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device, struct
pci_dev *from); struct pci_dev
*pci_find_device_reverse（无符号整数供应商,无符号整数
设备, const struct pci_dev *from);
struct pci_dev *pci_find_subsys（无符号整数供应商,无符号整数设备，
无符号整数 ss_vendor,无符号整数 ss_device,const struct pci_dev *from);
struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);

在设备列表中搜索具有特定签名或属于特定类的设备的功能。如果没有找到,则返回值为NULL 。 from 用于继续搜索;第一次调用任一函数时它必须为NULL ,如果要搜索更多设备,它必须指向刚刚找到的设备。

不建议使用这些函数,请改用pci_get_变体。

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);

在设备列表中搜索具有特定签名或属于特定类的设备的功能。如果没有找到,则返回值为NULL 。 from 用于继续搜索;第一次调用任一函数时它必须为NULL ,如果要搜索更多设备,它必须指向刚刚找到的设备。

返回的结构的引用计数增加,在调用者完成后,必须调用函数pci_dev_put 。 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);

读取或写入 PCI 配置寄存器的函数。尽管 Linux 内核负责字节顺序,但程序员在从单个字节组装多字节值时必须注意字节顺序。 PCI 总线是小端的。

int pci_enable_device(struct pci_dev *dev);

启用 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);无符号长
pci_resource_flags(struct pci_dev *dev, int bar);

处理 PCI 设备资源的函数。