LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 21 章　PCI 设备驱动

PCI（及 cPCI）总线在一般的小型手持设备中不太可能用到，但是在工控和通信设备及其 PC 中却引领着潮流。在 Linux 系统中，PCI 设备驱动和 USB 设备驱动有共性，那就是其驱动都由总线相关部分和自身设备类型驱动两部分组成。

21.1 节讲解了 PCI 总线及其配置空间，给出了 PCI 总线在 Linux 内核中的数据结构。

PCI 设备驱动的 PCI 相关部分围绕着 pci_driver 结构体的成员函数展开，21.2 节讲解了 pci_driver 结构体及其成员函数的含义，并分析了 PCI 设备驱动的框架结构。

21.3 节以 Intel 810 主板集成声卡为实例进一步讲解 PCI 设备驱动的编写方法。

21.1 PCI 总线与配置空间

21.1.1 PCI 总线的 Linux 描述

从本书第 2 章的图 2.16 可以看出，PCI 总线体系结构是一种层次式的体系结构。在这种层次式体系结构中，PCI 桥设备占据着重要的地位，它将父总线与子总线连接在一起，从而使整个系统看起来像一颗倒置的树型结构。树的顶端是系统的 CPU，它通过一个较为特殊的 PCI 桥设备——Host/PCI 桥设备与根 PCI 总线连接起来。

作为一种特殊的 PCI 设备，PCI 桥包括以下几种。

- Host/PCI 桥：用于连接 CPU 与 PCI 根总线，第 1 个根总线的编号为 0。在 PC 中，内存控制器也通常被集成到 Host/PCI 桥设备芯片中，因此，Host/PCI 桥通常也被称为“北桥芯片组（North Bridge Chipset）”。
- PCI/ISA 桥：用于连接旧的 ISA 总线。通常，PCI 中的类似 i8359A 中断控制器这样的设备也会被集成到 PCI/ISA 桥设备中，因此，PCI/ISA 桥通常也被称为“南桥芯片组（South Bridge Chipset）”。
- PCI-to-PCI 桥：用于连接 PCI 主总线（primary bus）与次总线（secondary bus）。PCI 桥所处的 PCI 总线称为“主总线”（即次总线的父总线），桥设备所连接的 PCI 总线称为“次总线”（即主总线的子总线）。

在 Linux 系统中，PCI 总线用 pci_bus 来描述，这个结构体记录了本 PCI 总线的信息以及本 PCI 总线的父总线、子总线、桥设备信息，这个结构体的定义如代码清单 21.1 所示。

代码清单 21.1 pci_bus 结构体

```
1  struct pci_bus
2  {
3    struct list_head node; /* 链表元素 node */
4    struct pci_bus *parent; /*指向该 PCI 总线的父总线，即 PCI 桥所在的总线 */
5    struct list_head children; /* 描述了这条 PCI 总线的子总线链表的表头 */
6    struct list_head devices; /* 描述了这条 PCI 总线的逻辑设备链表的表头 */
7    struct pci_dev *self; /* 指向引出这条 PCI 总线的桥设备的 pci_dev 结构 */
8    struct resource *resource[PCI_BUS_NUM_RESOURCES];
9        /* 指向应路由到这条 PCI 总线的地址空间资源 */
10
11   struct pci_ops *ops; /* 这条 PCI 总线所使用的配置空间访问函数 */
12   void *sysdata; /* 指向系统特定的扩展数据 */
13   struct proc_dir_entry *procdir;/*该 PCI 总线在/proc/bus/pci 中对应的
目录项*/
```

```
14
15   unsigned char number; /* 这条 PCI 总线的总线编号 */
16   unsigned char primary; /* 桥设备的主总线 */
17   unsigned char secondary; /* PCI 总线的桥设备的次总线号 */
18   unsigned char subordinate;/*PCI 总线的下属 PCI 总线的总线编号最大值*/
19
20   char name[48];
21
22   unsigned short bridge_ctl;
23   unsigned short pad2;
24   struct device *bridge;
25   struct class_device class_dev;
26   struct bin_attribute *legacy_io;
27   struct bin_attribute *legacy_mem;
28 };
```

假定一个如图 21.1 所示的 PCI 总线系统，根总线 0 上有一个 PCI 桥，它引出子总线 Bus 1，Bus 1 上又有一个 PCI 桥引出 Bus 2。

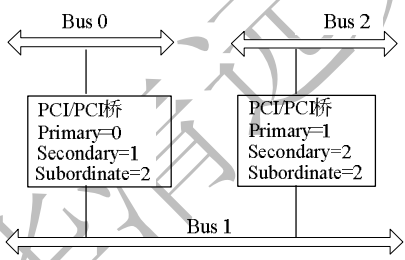

图 21.1　示例 PCI 总线系统

在上图中，Bus 0 总线的 pci_bus 结构体中的 number、primary、secondary 都应该为 0，因为它是通过 Host/PCI 桥引出的根总线；Bus 1 总线的 pci_bus 结构体中的 number 和 secondary 都为 1，但是它的 primary 应该为 0；Bus 2 总线的 pci_bus 结构体中的 number 和 secondary 都应该为 2，而其 primary 则应该等于 1。这 3 条总线的 subordinate 值都应该等于 2。

系统中当前存在的所有根总线都通过其 pci_bus 结构体中的 node 成员链接成一条全局的根总线链表，其表头由 list 类型的全局变量 pci_root_buses 来描述。而根总线下面的所有下级总线则都通过其 pci_bus 结构体中的 node 成员链接到其父总线的 children 链表中。这样，通过这两种 PCI 总线链表，Linux 内核就将所有的 pci_bus 结构体以一种倒置树的方式组织起来。假定对于如图 21.2 所示的多根 PCI 总线体系结构，它所对应的总线链表结构将如图 21.3 所示。

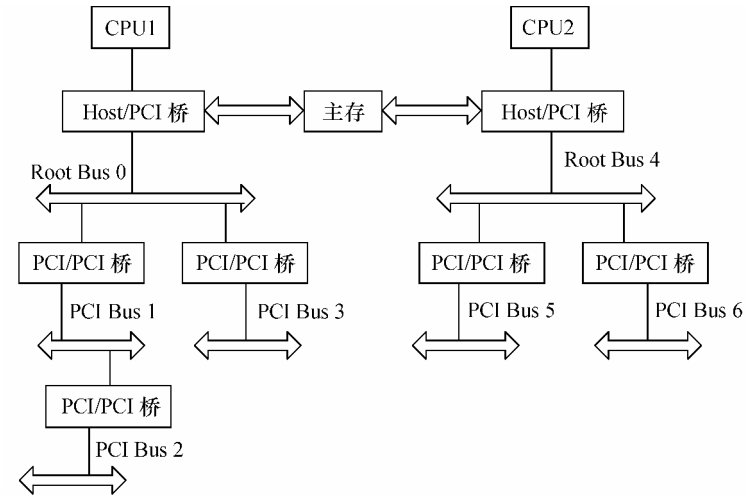

图 21.2　多根 PCI 总线体系结构

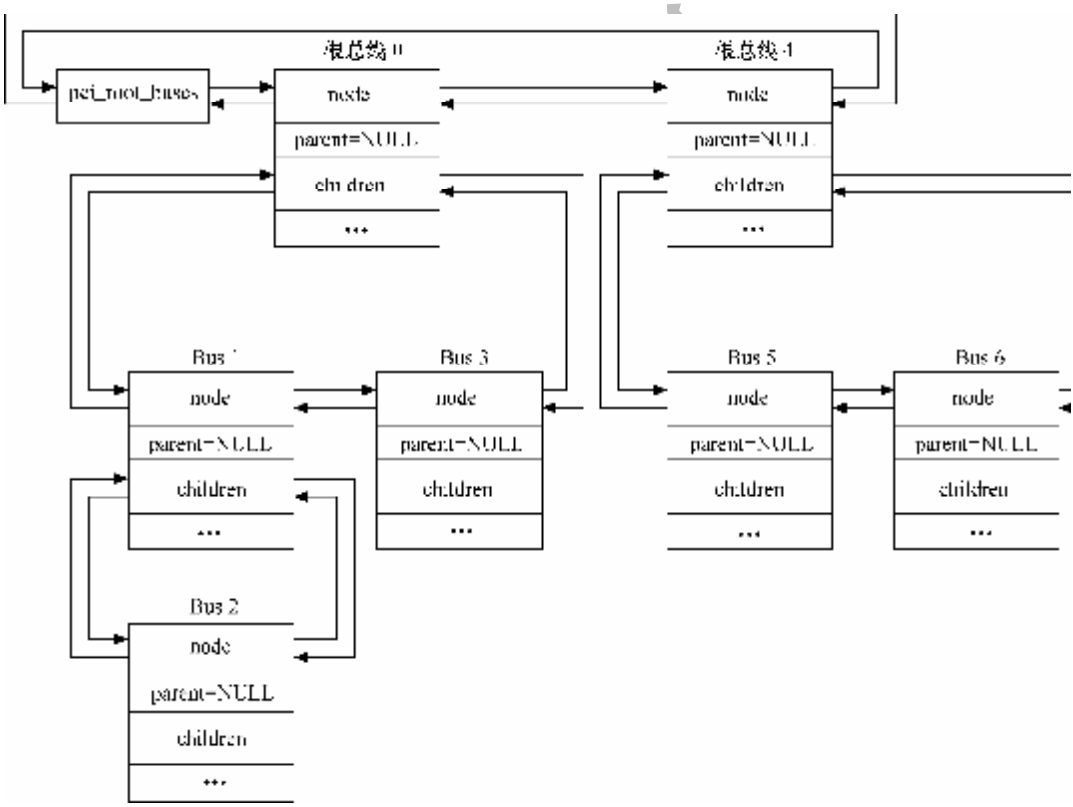

图 21.3　PCI 总线链表

21.1.2　PCI 设备的 Linux 描述

在 Linux 系统中，所有种类的 PCI 设备都可以用 pci_dev 结构体来描述，由于一个 PCI 接口卡上可能包含多个功能模块，每个功能被当作一个独立的逻辑设备，因此，每一个 PCI 功能，即 PCI 逻辑设备都唯一地对应一个 pci_dev 设备描述符，该结构体

的定义如代码清单 21.2 所示。

代码清单 21.2　pci_dev 结构体

```
1  struct pci_dev
2  {
3    struct list_head global_list; /* 全局链表元素 */
4    struct list_head bus_list; /* 总线设备链表元素 */
5    struct pci_bus *bus; /* 这个 PCI 设备所在的 PCI 总线的 pci_bus 结构 */
6    struct pci_bus *subordinate; /* 指向这个 PCI 设备所桥接的下级总线 */
7
8    void *sysdata; /* 指向一片特定于系统的扩展数据 */
9    struct proc_dir_entry *procent; /* 该 PCI 设备在/proc/bus/pci 中对应
的目录项 */
10
11   unsigned int devfn; /* 这个 PCI 设备的设备功能号 */
12   unsigned short vendor; /* PCI 设备的厂商 ID*/
13   unsigned short device; /* PCI 设备的设备 ID */
14   unsigned short subsystem_vendor; /* PCI 设备的子系统厂商 ID */
15   unsigned short subsystem_device; /* PCI 设备的子系统设备 ID */
16   unsigned int class ; /* 32 位的无符号整数，表示该 PCI 设备的类别，
17            bit［7：0］为编程接口，bit［15：8］为子类别代码，bit［23：16］
18           为基类别代码，bit［31：24］无意义 */
19   u8 hdr_type; /* PCI 配置空间头部的类型 */
20   u8 rom_base_reg; /* 表示 PCI 配置空间中的 ROM 基地址寄存器在 PCI 配置空间
中的位置 */
21
22   struct pci_driver *driver; /* 指向这个 PCI 设备所对应的驱动 pci_driver
结构 */
23   u64 dma_mask; /* 该设备支持的总线地址位掩码，通常是 0xffffffff */
24
25   pci_power_t current_state; /* 当前的操作状态 */
26
27   struct device dev; /* 通用的设备接口 */
28
29   /* 定义这个 PCI 设备与哪些设备相兼容 */
30   unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
31   unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
32
33   int cfg_size; /* 配置空间大小 */
34
35   unsigned int irq;
36   struct resource resource[DEVICE_COUNT_RESOURCE];
37             /*表示该设备可能用到的资源，包括:
38             I/O 端口区域、设备内存地址区域以及扩展 ROM 地址区域 */
39
40   unsigned int transparent: 1; /* 透明 PCI 桥 */
41   unsigned int multifunction: 1; /* 多功能设备 */
42   /* keep track of device state */
43   unsigned int is_enabled: 1; /* pci_enable_device 已经被调用？ */
44   unsigned int is_busmaster: 1; /* 设备是主设备？ */
45   unsigned int no_msi: 1; /* 设备可不使用 msi？ */
46
47   u32 saved_config_space[16]; /* 挂起事保存的配置空间 */
48   struct bin_attribute *rom_attr; /* sysfs ROM 入口的属性描述 */
49   int rom_attr_enabled;
50    struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE];/*资源的
sysfs 文件*/
51 };
```

在 Linux 系统中，所有的 PCI 设备都通过其 pci_dev 结构体中的 global_list 成员链

接一条全局 PCI 设备链表 pci_devices。另外，同属一条 PCI 总线上的所有 PCI 设备也通过其 pci_dev 结构体中的 bus_list 成员链接成一个属于这条 PCI 总线的总线设备链表，表头则由该 PCI 总线的 pci_bus 结构中的 devices 成员所定义。图 21.4 所示为 PCI 设备链表的典型例子。

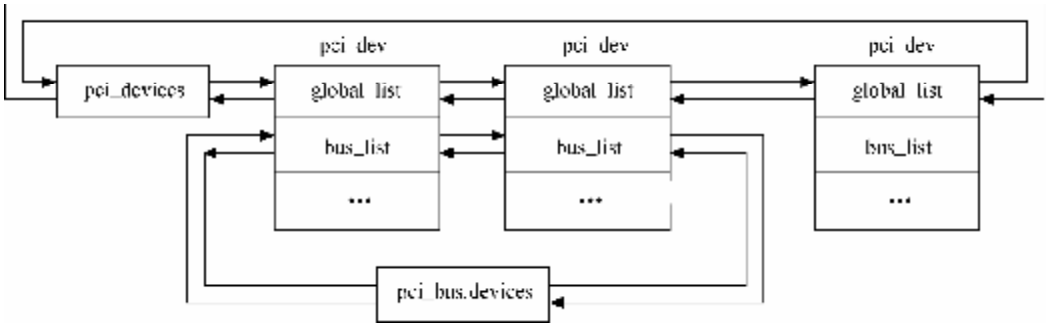

图 21.4　PCI 设备链表

21.1.3　PCI 配置空间访问

PCI 有 3 种地址空间：PCI I/O 空间、PCI 内存地址空间和 PCI 配置空间。CPU 可以访问所有的地址空间，其中 PCI I/O 空间和 PCI 内存地址空间由设备驱动程序使用，而 PCI 配置空间由 Linux 内核中的 PCI 初始化代码使用，这些代码用于配置 PCI 设备，比如中断号以及 I/O 或内存基地址。

PCI 规范定义了 3 种类型的 PCI 配置空间头部，其中 type 0 用于标准的 PCI 设备，type 1 用于 PCI 桥，type 2 用于 PCI CardBus 桥。如图 2.17 所示，不管是哪一种类型的配置空间头部，其前 16 个字节的格式都是相同的，/include/linux/pci_regs.h 文件中定义了 PCI 配置空间头部，如代码清单 21.3 所示。

代码清单 21.3　PCI 配置空间头部寄存器定义

```
1  #define PCI_VENDOR_ID        0x00    /* 16位厂商ID */
2  #define PCI_DEVICE_ID        0x02    /* 16位设备ID */
3
4  /* PCI命令寄存器 */
5  #define PCI_COMMAND      0x04        /* 16位 */
6  #define PCI_COMMAND_IO   0x1         /* 使能设备响应对 I/O 空间的访问 */
7  #define PCI_COMMAND_MEMORY   0x2         /* 使能设备响应对存储空间的访问 */
8  #define PCI_COMMAND_MASTER   0x4         /* 使能总线主模式 */
9  #define PCI_COMMAND_SPECIAL  0x8     /* 使能设备响应特殊周期 */
10 #define PCI_COMMAND_INVALIDATE 0x10 /*使用PCI内存写无效事务 */
11 #define PCI_COMMAND_VGA_PALETTE 0x20   /* 使能VGA调色板侦测 */
```

```
12 #define PCI_COMMAND_PARITY    0x40    /* 使能奇偶校验 */
13 #define PCI_COMMAND_WAIT     0x80  /* 使能地址/数据步进 */
14 #define PCI_COMMAND_SERR      0x100   /* 使能 SERR */
15 #define PCI_COMMAND_FAST_BACK  0x200  /* 使能背靠背写 */
16 #define PCI_COMMAND_INTX_DISABLE 0x400 /* 禁止中断竞争*/
17
18 /* PCI 状态寄存器 */
19 #define PCI_STATUS            0x06   /* 16 位 */
20 #define PCI_STATUS_CAP_LIST    0x10    /* 支持的能力列表 */
21 #define PCI_STATUS_66MHZ      0x20    /* 支持 PCI 2.1 66MHz */
22 #define PCI_STATUS_UDF        0x40    /* 支持用户定义的特征 */
23 #define PCI_STATUS_FAST_BACK   0x80    /* 快速背靠背操作 */
24 #define PCI_STATUS_PARITY     0x100   /* 侦测到奇偶校验错 */
25 #define PCI_STATUS_DEVSEL_MASK 0x600   /* DEVSEL 定时 */
26 #define PCI_STATUS_DEVSEL_FAST        0x000
27 #define PCI_STATUS_DEVSEL_MEDIUM      0x200
28 #define PCI_STATUS_DEVSEL_SLOW        0x400
29 #define PCI_STATUS_SIG_TARGET_ABORT    0x800 /* 目标设备异常 */
30 #define PCI_STATUS_REC_TARGET_ABORT    0x1000 /* 主设备确认 */
31 #define PCI_STATUS_REC_MASTER_ABORT 0x2000 /* 主设备异常 */
32 #define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000 /* 驱动了 SERR */
33 #define PCI_STATUS_DETECTED_PARITY  0x8000 /* 奇偶校验错 */
34
35 /* 类代码寄存器和修订版本寄存器 */
36 #define PCI_CLASS_REVISION   0x08  /* 高 24 位为类码，低 8 位为修订版本 */
37 #define PCI_REVISION_ID  0x08    /* 修订号 */
38 #define PCI_CLASS_PROG  0x09    /* 编程接口 */
39 #define PCI_CLASS_DEVICE       0x0a    /* 设备类 */
40 #define PCI_CACHE_LINE_SIZE    0x0c    /* 8 位 */
41 #define PCI_LATENCY_TIMER     0x0d    /* 8 位 */
42
43 /* PCI 头类型 */
44 #define PCI_HEADER_TYPE        0x0e    /* 8 位头类型 */
45 #define PCI_HEADER_TYPE_NORMAL        0
46 #define PCI_HEADER_TYPE_BRIDGE        1
47 #define PCI_HEADER_TYPE_CARDBUS       2
48
49 /* 表示配置空间头部中的 Built-In Self-Test 寄存器在配置空间中的字节地址索引
*/
50 #define PCI_BIST              0x0f    /* 8 位 */
```

```
51 #define PCI_BIST_CODE_MASK     0x0f    /* 完成代码 */
52 #define PCI_BIST_START         0x40    /* 用于启动 BIST*/
53 #define PCI_BIST_CAPABLE       0x80    /* 设备是否支持 BIST?  */
```

紧接着前 16 个字节的寄存器为基地址寄存器 0～基地址寄存器 5，其中，PCI_BASE_ADDRESS_ 2～5 仅对标准 PCI 设备的 0 类型配置空间头部有意义，而 PCI_BASE_ADDRESS_0～1 则适用于 0 类型和 1 类型配置空间头部。

基地址寄存器中的 bit［0］的值决定了这个基地址寄存器所指定的地址范围是在 I/O 空间还是在内存映射空间内进行译码。当基地址寄存器所指定的地址范围位于内存映射空间中时，其 bit［2：1］表示内存地址的类型，bit［3］表示内存范围是否为可预取（Prefetchable）的内存。

1 类型配置空间头部适用于 PCI-PCI 桥设备，其基地址寄存器 0 与基地址寄存器 1 可以用来指定桥设备本身可能要用到的地址范围，而后 40 个字节（0x18～0x39）则被用来配置桥设备的主、次编号以及地址过滤窗口等信息。

pci_bus 结构体中的 pci_ops 类型成员指针 ops 指向该 PCI 总线所使用的配置空间访问操作的具体实现，pci_ops 结构体的定义如代码清单 21.4 所示。

代码清单 21.4　pci_ops 结构体

```
1 struct pci_ops
2 {
3   int(*read)(struct pci_bus *bus, unsigned int devfn, int where, int
size, u32
4     *val);//读配置空间
5   int(*write)(struct pci_bus *bus, unsigned int devfn, int where, int
size, u32
6     val); //写配置空间
7 };
```

read()和 write()成员函数中的 size 表示访问的是字节、2 字节还是 4 字节，对于 write()而言，val 是要写入的值；对于 read()而言，val 是要返回的读取到的值的指针。通过 bus 参数的成员以及 devfn 可以定位相应 PCI 总线上相应 PCI 逻辑设备的配置空间。在 Linux 设备驱动中，可用如下一组函数来访问配置空间：

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

上述函数只是对如下函数进行调用：

```
int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn,
int where, u8 *val); //读字节
```

```
    int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn,
int where, u16 *val); //读字
    int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn,
int where, u32 *val); //读双字
    int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn,
int where, u8 val); //写字节
    int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn,
int where, u16 val); //写字
    int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn,
int where, u32 val); //写双字
```

最后，我们来看一下 PCI 总线、设备与驱动在/proc 和/sysfs 文件系统中的描述。首先，通过查看/proc/bus/pci 中的文件，可以获得系统连接的 PCI 设备的基本信息描述。在 VmWare 虚拟机 Linux 上的/proc/bus/pci 目录下的树型结构如下：

```
    |-- 00
    |   |-- 00.0
    |   |-- 01.0
    |   |-- 07.0
    |   |-- 07.1
    |   |-- 07.2
    |   |-- 07.3
    |   |-- 0f.0
    |   |-- 10.0
    |   |-- 11.0
    |   '-- 12.0
    '-- devices
```

sysfs 文件系统/sys/bus/pci 目录中也给出了系统中总线上挂接的设备及驱动信息，该目录下的树型结构如下：

```
    |-- devices
    |   |-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
    |   |-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
    |   |-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
    |   |-- 0000:00:07.1 -> ../../../devices/pci0000:00/0000:00:07.1
    |   |-- 0000:00:07.2 -> ../../../devices/pci0000:00/0000:00:07.2
    |   |-- 0000:00:07.3 -> ../../../devices/pci0000:00/0000:00:07.3
    |   |-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
    |   |-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
    |   |-- 0000:00:11.0 -> ../../../devices/pci0000:00/0000:00:11.0
    |   '-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
    '-- drivers
       |   ...
       |-- PIIX_IDE
       |                                    |--            0000:00:07.1
```

```
-> ../../../../devices/pci0000:00/0000:00:07.1
       |   |-- bind
       |   |-- new_id
       |   '-- unbind
       |   ...
       '-- pcnet32
           |--                                                0000:00:11.0
-> ../../../../devices/pci0000:00/0000:00:11.0
           |-- bind
           |-- new_id
           '-- unbind
```

此外，pciutils（PCI 工具）中的 lspci 工具会分析/proc/bus/pci 中的文件，从而可被用户用于查看系统中 PCI 设备的描述信息，例如笔者在 VmWare 虚拟机 Linux 上运行 lspci 的结果为：

```
    00:00.0 Host bridge: Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX Host bridge
(rev 01)
    00:01.0 PCI bridge: Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge
(rev 01)
    00:07.0 ISA bridge: Intel Corp. 82371AB/EB/MB PIIX4 ISA (rev 08)
    00:07.1 IDE interface: Intel Corp. 82371AB/EB/MB PIIX4 IDE (rev 01)
    00:07.2 USB Controller: Intel Corp. 82371AB/EB/MB PIIX4 USB
    00:07.3 Bridge: Intel Corp. 82371AB/EB/MB PIIX4 ACPI (rev 08)
    00:0f.0 VGA compatible controller: VMWare Inc: Unknown device 0405
    00:10.0 SCSI storage controller: BusLogic BT-946C (BA80C30)
[MultiMaster 10] (rev 01)
    00:11.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32
LANCE] (rev 10)
    00:12.0 Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev
02)
```

21.2 PCI 设备驱动结构

21.2.1 pci_driver 结构体

从本质上讲 PCI 只是一种总线，具体的 PCI 设备可以是字符设备、网络设备、USB 主机控制器等，因此，一个通过 PCI 总线与系统连接的设备的驱动至少包含以下两部

分内容。

- PCI 设备驱动。
- 设备本身的驱动。

PCI 驱动只是为了辅助设备本身的驱动，它不是目的，只是手段，PCI 设备本身含有双重以上的身份。例如对于通过 PCI 总线与系统连接的字符设备，则驱动中除了要实现 PCI 驱动部分外，其主体仍然是设备作为字符设备本身的驱动，即实现 file_operations 成员函数并注册 cdev。分析 Linux 内核可知，在/drivers/block/、/drivers/atm/、/drivers/char/、/drivers/i2c/、/drivers/ieee1394/、/drivers/media/、/drivers/mtd/、/drivers/net/、/drivers/serial/、/drivers/video/、/sound/等目录中均广泛分布着 PCI 设备驱动。

在 Linux 内核中，用 pci_driver 结构体来定义 PCI 驱动，该结构体中包含了 PCI 设备的探测/移除、挂起/恢复等函数，其定义如代码清单 21.5 所示。

代码清单 21.5　pci_driver 结构体

```
1  struct pci_driver
2  {
3    struct list_head node;
4    char *name;
5    struct module *owner;
6    const struct pci_device_id *id_table; /*不能为 NULL，以便 probe 函数调用*/
7    /* 新设备添加 */
8    int(*probe)(struct pci_dev *dev, const struct pci_device_id *id);
9    void(*remove)(struct pci_dev *dev); /* 设备移出 */
10   int(*suspend)(struct pci_dev *dev, pm_message_t state); /* 设备挂起 */
11   int(*resume)(struct pci_dev *dev); /* 设备唤醒 */
12   /* 使能唤醒事件 */
13    int(*enable_wake)(struct pci_dev *dev, pci_power_t state, int enable);
14   void(*shutdown)(struct pci_dev *dev);
15
16   struct device_driver driver;
17   struct pci_dynids dynids;
18 };
```

对 pci_driver 的注册和注销通过如下函数来实现：

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

有一个对应于 pci_register_driver()的已经过时的宏定义：

```
#define pci_module_init pci_register_driver
```

在 PCI 设备驱动中其他常用的函数（或宏）如下所示。

- 获取 I/O 或内存资源。

```
#define pci_resource_start(dev,bar)   ((dev)->resource[(bar)].start)
#define pci_resource_end(dev,bar)     ((dev)->resource[(bar)].end)
#define pci_resource_flags(dev,bar)   ((dev)->resource[(bar)].flags)
#define pci_resource_len(dev,bar) \
        ((pci_resource_start((dev),(bar)) == 0 &&  \
         pci_resource_end((dev),(bar)) ==          \
          pci_resource_start((dev),(bar))) ? 0 :   \
                                                   \
         (pci_resource_end((dev),(bar)) -          \
          pci_resource_start((dev),(bar)) + 1))
```

- 申请/释放 I/O 或内存资源。

```
int pci_request_regions(struct pci_dev *pdev, const char *res_name);
void pci_release_regions(struct pci_dev *pdev);
```

- 获取/设置驱动私有数据。

```
void *pci_get_drvdata (struct pci_dev *pdev);
void pci_set_drvdata (struct pci_dev *pdev, void *data);
```

- 使能/禁止 PCI 设备。

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

- 设置为总线主 DMA。

```
void pci_set_master(struct pci_dev *dev);
```

- 寻找指定总线指定槽位的 PCI 设备。

```
struct pci_dev *pci_find_slot (unsigned int bus, unsigned int devfn);
```

- 设置 PCI 能量管理状态（0=D0 ... 3=D3）。

```
int pci_set_power_state(struct pci_dev *dev, pci_power_t state);
```

- 在设备的能力表中找出指定的能力。

```
int pci_find_capability (struct pci_dev *dev, int cap);
```

- 启用设备内存写无效事务。

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

- 禁用设备内存写无效事务。

```
void pci_clear_mwi(struct pci_dev *dev);
```

pci_driver 的 probe()函数要完成 PCI 设备的初始化及其设备本身身份（字符、TTY、网络等）驱动的注册。当 Linux 内核启动并完成对所有 PCI 设备进行扫描、登录和分配资源等初始化操作的同时，会建立起系统中所有 PCI 设备的拓扑结构，probe()函数将负责硬件的探测工作并保存配置信息。

drivers/net/pci-skeleton.c 给出了一个 PCI 接口网络设备驱动程序的“骨架”，其 pci_driver 中 probe()成员函数 netdrv_init_one()及其调用的 netdrv_init_board()完成了

PCI 设备初始化及对应的网络设备注册工作，代码清单 21.6 所示为这两个函数的实现。

代码清单 21.6　PCI 设备驱动的 probe()函数

```
1  static int __devinit netdrv_init_one (struct pci_dev *pdev,
2                            const struct pci_device_id *ent)
3  {
4  struct net_device *dev = NULL;
5  struct netdrv_private *tp;
6  int i, addr_len, option;
7  void *ioaddr = NULL;
8  static int board_idx = -1;
9
10   ...
11
12 board_idx++;
13 i = netdrv_init_board (pdev, &dev, &ioaddr);
14 if (i < 0) {
15     return i;
16 }
17
18 ...
19 /* 赋值 net_device 的成员函数 */
20 dev->open = netdrv_open;
21 dev->hard_start_xmit = netdrv_start_xmit;
22 dev->stop = netdrv_close;
23 dev->get_stats = netdrv_get_stats;
24 dev->set_multicast_list = netdrv_set_rx_mode;
25 dev->do_ioctl = netdrv_ioctl;
26 dev->tx_timeout = netdrv_tx_timeout;
27 dev->watchdog_timeo = TX_TIMEOUT;
28
29 dev->irq = pdev->irq;
30 dev->base_addr = (unsigned long) ioaddr;
31 ...
32 pci_set_drvdata(pdev, dev);//设置 PCI 私有数据
33 ...
34
35 return 0;
36 }
37
38 static int __devinit netdrv_init_board(struct pci_dev *pdev, struct
```

```
net_device
    39    **dev_out, void **ioaddr_out)
    40  {
    41    void *ioaddr = NULL;
    42    struct net_device *dev;
    43    struct netdrv_private *tp;
    44    int rc, i;
    45    u32 pio_start, pio_end, pio_flags, pio_len;
    46    unsigned long mmio_start, mmio_end, mmio_flags, mmio_len;
    47    u32 tmp;
    48
    49    *ioaddr_out = NULL;
    50    *dev_out = NULL;
    51
    52    /* 分配 ethernet 设备 */
    53    dev = alloc_etherdev(sizeof(*tp));
    54    if (dev == NULL)
    55    {
    56      printk(KERN_ERR PFX "unable to alloc new ethernet\n");
    57      DPRINTK("EXIT, returning -ENOMEM\n");
    58      return  - ENOMEM;
    59    }
    60    SET_MODULE_OWNER(dev);
    61    SET_NETDEV_DEV(dev, &pdev->dev);
    62    tp = dev->priv;
    63
    64    /* 使能设备 */
    65    rc = pci_enable_device(pdev);
    66    if (rc)
    67      goto err_out;
    68
    69    /* 获取 I/O 和内存基地址 */
    70    pio_start = pci_resource_start(pdev, 0);
    71    pio_end = pci_resource_end(pdev, 0);
    72    pio_flags = pci_resource_flags(pdev, 0);
    73    pio_len = pci_resource_len(pdev, 0);
    74
    75    mmio_start = pci_resource_start(pdev, 1);
```

```
    76    mmio_end = pci_resource_end(pdev, 1);
    77    mmio_flags = pci_resource_flags(pdev, 1);
    78    mmio_len = pci_resource_len(pdev, 1);
    79
    80    DPRINTK("PIO region size == 0x%02X\n", pio_len);
    81    DPRINTK("MMIO region size == 0x%02lX\n", mmio_len);
    82
    83    /* 确保 PCI 基地址寄存器 0 是 PIO */
    84    if (!(pio_flags &IORESOURCE_IO))
    85    {
    86      printk(KERN_ERR PFX "region #0 not a PIO resource, aborting\n");
    87      rc =  - ENODEV;
    88      goto err_out;
    89    }
    90
    91    /* 确保 PCI 基地址寄存器 1 是 MMIO */
    92    if (!(mmio_flags &IORESOURCE_MEM))
    93    {
    94          printk(KERN_ERR  PFX  "region  #1  not  an  MMIO  resource,
aborting\n");
    95      rc =  - ENODEV;
    96      goto err_out;
    97    }
    98
    99    /* 检查 weird/broken PCI 区域报告 */
    100       if   ((pio_len   <   NETDRV_MIN_IO_SIZE)   ||   (mmio_len   <
NETDRV_MIN_IO_SIZE))
    101   {
    102     printk(KERN_ERR PFX "Invalid PCI region size(s), aborting\n");
    103     rc =  - ENODEV;
    104     goto err_out;
    105   }
    106
    107   rc = pci_request_regions(pdev, "pci-skeleton");
    108   if (rc)
    109     goto err_out;
    110
    111   pci_set_master(pdev);
    112
    113   #ifdef USE_IO_OPS
```

```
    ioaddr = (void*)pio_start;
  #else
    /* ioremap MMIO 区域 */
    ioaddr = ioremap(mmio_start, mmio_len);
    if (ioaddr == NULL)
    {
      printk(KERN_ERR PFX "cannot remap MMIO, aborting\n");
      rc =  - EIO;
      goto err_out_free_res;
    }
  #endif /* USE_IO_OPS */

  /* 软重启芯片 */
     NETDRV_W8(ChipCmd,  (NETDRV_R8(ChipCmd)  &ChipCmdClear)  |
CmdReset);

  /* 检查芯片已完成复位，让芯片进入低能耗模式 */
  /* <在此插入设备特定的代码> */
  ...

  /* 注册网络设备 */
  i = register_netdev(dev);
  if (i)
    goto err_out_unmap;

  DPRINTK("EXIT, returning 0\n");
  *ioaddr_out = ioaddr;
  *dev_out = dev;
  return 0;

  err_out_unmap:
  ...
}
```

从上述代码可以看出，probe()函数中的主体分为两个部分，即 PCI 配置部分和网络设备初始化和注册部分。pci_driver 的 remove()成员函数完成相反的工作，即释放 PCI 设备和网络设备，如代码清单 21.7 所示。

代码清单 21.7　PCI 设备驱动的 remove()函数

```
1  static void __devexit netdrv_remove_one (struct pci_dev *pdev)
2  {
3   struct net_device *dev = pci_get_drvdata (pdev);
4   struct netdrv_private *np;
5
6   np = dev->priv;
7
8   unregister_netdev (dev);//注销网络设备
9  #ifndef USE_IO_OPS
10  iounmap (np->mmio_addr);//iounmap
11 #endif /* !USE_IO_OPS */
12
13  pci_release_regions (pdev);//释放 pci 区域
14  free_netdev (dev);//释放 ethernet
15  pci_set_drvdata (pdev, NULL);
16  pci_disable_device (pdev);//禁止 PCI 设备
17 }
```

如同在 USB 设备驱动中定义 usb_device_id 结构体数组一样，在 PCI 设备驱动中，也需要定义一个 pci_device_id 结构体数组并导出到用户空间，使热插拔和模块装载系统知道驱动模块所针对的硬件设备。pci_device_id 结构体的定义如代码清单 21.8 所示。

代码清单 21.8　pci_device_id 结构体

```
1 struct pci_device_id {
2          __u32 vendor, device;         /* 厂商和设备 ID 或 PCI_ANY_ID*/
3          __u32 subvendor, subdevice;   /* 子系统 ID 或 PCI_ANY_ID */
4          __u32 class, class_mask;      /* (类、子类、prog-if) 三元组 */
5          kernel_ulong_t driver_data;   /* 驱动私有数据 */
6 };
```

其中的 PCI_ANY_ID 定义为~0，即对任意 ID 都适用。pci_device_id 结构体数组使用宏 MODULE_DEVICE_TABLE 导出到用户空间。/drivers/net/pci-skeleton.c 中定义的 pci_device_id 结构体数组及 MODULE_DEVICE_TABLE 导出代码如清单 21.9 所示。

代码清单 21.9　PCI 设备驱动的 pci_device_id 数组及 MODULE_DEVICE_TABLE

```
1  static struct pci_device_id netdrv_pci_tbl[] = {
2   {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
3   {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, NETDRV_CB },
4   {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, SMC1211TX },
5  /*   {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, MPX5030 },*/
6   {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, DELTA8139 },
7   {0x4033, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, ADDTRON8139 },
8   {0,}
```

```
    9 };
    10 MODULE_DEVICE_TABLE (pci, netdrv_pci_tbl);
```

21.2.2 PCI 设备驱动的组成

下面以一个 PCI 接口字符设备为例，给出 PCI 设备驱动的完整模板。其一部分代码实现 pci_driver 成员函数，一部分代码实现字符设备的 file_operations 成员函数，如代码清单 21.10 所示。

代码清单 21.10 PCI 设备驱动的程序模板

```
    1 /* 指明该驱动程序适用于哪一些 PCI 设备 */
    2 static struct pci_device_id xxx_pci_tbl [] __initdata = {
    3     {PCI_VENDOR_ID_DEMO, PCI_DEVICE_ID_DEMO,
    4      PCI_ANY_ID, PCI_ANY_ID, 0, 0, DEMO},
    5     {0,}
    6 };
    7 MODULE_DEVICE_TABLE(pci, xxx_pci_tbl);
    8
    9 /* 中断处理函数 */
    10 static void xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
    11 {
    12     ...
    13 }
    14
    15 /* 字符设备 file_operations open 成员函数 */
    16 static int xxx_open(struct inode *inode, struct file *file)
    17 {
    18     /* 申请中断，注册中断处理程序 */
    19     request_irq(xxx_irq, &xxx_interrupt, ...));
    20     ...
    21 }
    22
    23 /* 字符设备 file_operations ioctl 成员函数 */
    24 static int xxx_ioctl(struct inode *inode, struct file *file, unsigned
int cmd, unsigned long arg)
    25 {
    26    ...
    27 }
    28
```

```
29 ... /* 字符设备 file_operations read、write、mmap 等成员函数 */
30
31 /* 设备文件操作接口 */
32 static struct file_operations xxx_fops = {
33        owner:     THIS_MODULE,       /* xxx_fops 所属的设备模块 */
34        read:      xxx_read,      /* 读设备操作*/
35        write:     xxx_write,         /* 写设备操作*/
36        ioctl:     xxx_ioctl,         /* 控制设备操作*/
37        mmap:      xxx_mmap,          /* 内存重映射操作*/
38        open:      xxx_open,          /* 打开设备操作*/
39        release:    xxx_release       /* 释放设备操作*/
40       /* ... */
41 };
42
43 /* pci_driver 的 probe 成员函数 */
44 static int __init xxx_probe(struct pci_dev *pci_dev, const struct
pci_device_id *pci_id)
45 {
46          pci_enable_device(pci_dev); //启动 PCI 设备
47
48           /* 读取 PCI 配置信息 */
49          iobase = pci_resource_start (pci_dev,1);
50          ...
51
52          pci_set_master(pci_dev);//设置成总线主 DMA 模式
53
54          pci_request_regions(pci_dev);//申请 I/O 资源
55
56          /* 注册字符设备 */
57          cdev_init(xxx_cdev,&xxx_fops);
58          register_chrdev_region(xxx_dev_no, 1, ...);
59          cdev_add(xxx_cdev);
60
61          return 0;
62 }
63
64 /* pci_driver 的 remove 成员函数 */
65 static int __init xxx_release(struct pci_dev *pdev)
66 {
67       pci_release_regions(pdev);//释放 I/O 资源
```

```
68        pci_disable_device (pdev);//禁止 PCI 设备
69        unregister_chrdev_region(xxx_dev_no, 1); //释放占用的设备号
70        cdev_del(&xxx_dev.cdev); //注销字符设备
71        ...
72        return 0;
73 }
74
75 /* 设备模块信息 */
76 static struct pci_driver xxx_pci_driver = {
77        name:      xxx_MODULE_NAME,          /* 设备模块名称 */
78        id_table:   xxx_pci_tbl,             /* 能够驱动的设备列表 */
79        probe:     xxx_probe,                /* 查找并初始化设备 */
80        remove:    xxx_remove            /* 卸载设备模块 */
81 };
82
83 static int __init xxx_init_module (void)
84 {
85        pci_register_driver(&xxx_pci_driver);
86 }
87 static void __exit xxx_cleanup_module (void)
88 {
89        pci_unregister_driver(&xxx_pci_driver);
90 }
91 /*驱动模块加载函数 */
92 module_init(xxx_init_module);
93 /* 驱动模块卸载函数 */
94 module_exit(xxx_cleanup_module);
```

将代码清单 21.10 中的各个函数进行归类，可得出该驱动的组成，如图 21.5 所示。

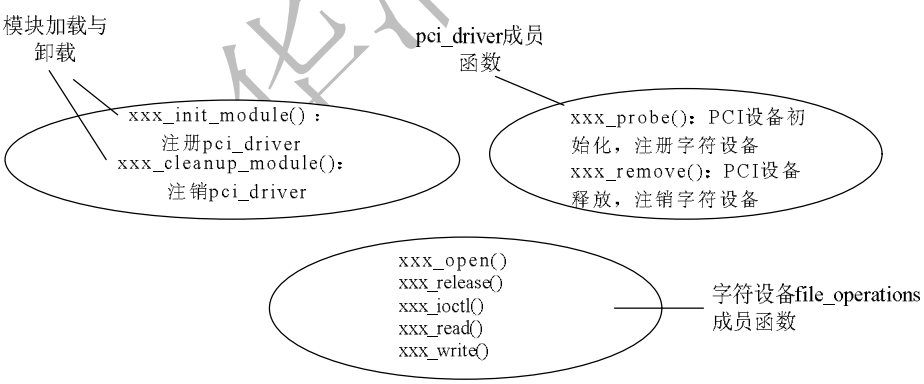

图 21.5　PCI 字符设备驱动的组成

图 21.6 所示的树中，树根是主机/PCI 桥，树叶是具体的 PCI 设备，树叶本身与树枝通过 pci_driver 连接，而树叶本身的驱动，读写、控制树叶则需要通过其树叶设备本身所属类设备驱动来完成。

由此我们看出，对于 USB、PCI 设备这种挂接在总线上的设备而言，USB、PCI 只是它们的“工作单位”，它们需要向“工作单位”注册（注册 usb_driver、pci_driver），

并接收“工作单位”的管理（被调入 probe()、调出 disconnect()/remove()、放假 shutdown()、继续上班 resume()等），但是其本身可能是一个工程师、一个前台或一个经理，因此，做好工程师、前台或经理是其主体工作，这部分对应于字符设备驱动、tty 设备驱动、网络设备驱动等。

图 21.6　PCI 设备驱动结构

21.2.3　旧版内核的 PCI 设备探测

21.1.1~21.1.2 节给出的 PCI 设备驱动通过 pci_register_driver()注册 pci_driver，从而导致其 probe()成员函数被调用。但是，在 Linux 2.6 之前的内核中，PCI 设备的探测主要依靠驱动程序在模块加载函数中调用如下函数来完成：

```
    struct pci_dev *pci_find_device (unsigned int vendor, unsigned int
device, const struct pci_dev *from);
    struct pci_dev *pci_find_device_reverse (unsigned int vendor, unsigned
int device, const struct pci_dev *from);
```

pci_find_device()、pci_find_device_reverse()函数在设备列表中查找匹配厂商及设备 ID 的 PCI 设备，若是第 1 次搜索，则 from 置 NULL，若是第 *n* 次搜索，则使用第 *n*−1 次搜索获得的 pci_dev。

```
    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);
```

该函数用于查找匹配厂商及设备 ID、子系统厂商及设备 ID 的 PCI 设备。

```
    struct pci_dev * pci_find_class (unsigned int class, const struct
pci_dev * from);
```

该函数用于查找参数 class 类 PCI 设备。

旧的 PCI 设备驱动模块加载函数通常采用如图 21.7 所示的流程，代码清单 21.11 则给出了相应的函数模板。

代码清单 21.11　旧版内核下 PCI 探测函数范例

```
1  int xxx_init_module()
2  {
3    int ret;
4    struct pci_dev *pdev = NULL;
5    struct XXX_DEV *dev;
6    //判断是否有总线
7    if (!pci_present())
8      return  - ENODEV;
9    // PCI
10   //查找该类卡
11     while  ((pdev  =  (struct  pci_dev*)pci_find_device(VENDOR_ID,
DEVICE_ID, pdev)))
12     dev = &my_device[dev_num];
13   //得到中断号和基址
14   dev->irq = pdev->irq;
15   dev->iobase = pdev->base_address[0] &PCI_BASE_ ADDRESS_IO_MASK;
16   //检查地址空间是否可用
17   if (check_region(dev->iobase, DEVICE_IO_EXTENT) < 0)
18     continue;
19   //注册中断处理函数
20   if (request_irq(dev->irq, my_interrupt, SA_SHIRQ | SA_INTERRUPT,
dev
21     ->dev_name, dev))
22     return  - EAGAIN;
23   //申请空间
24   request_region(pci_ioaddr, DEVICE_IO_EXTENT, dev->dev _name);
25   ...
26   //注册该设备
27   ret = register_xxxdev(...);//字符、USB、网络等
28   ...
29 }
```

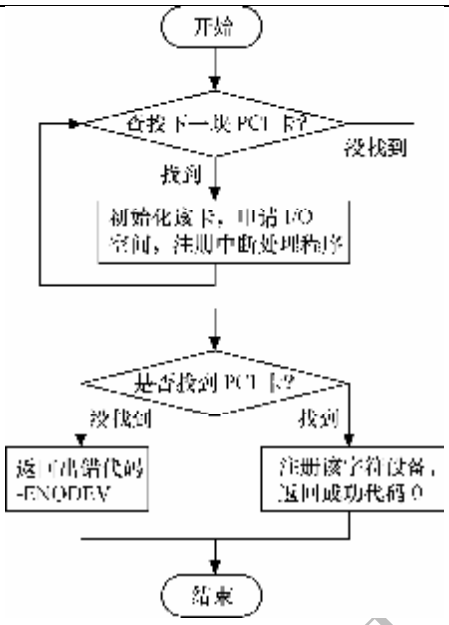

图 21.7　旧的 PCI 设备驱动的模块加载函数流程

在 Linux 2.6 内核中，不宜再采用这些函数搜寻系统中的 PCI 设备，如果驱动程序中确实需要查找特定的 PCI 设备，则可以使用如下函数：

```
    struct pci_dev *pci_get_device (unsigned int vendor, unsigned int device,
struct pci_dev *from);
```

这个函数从系统中当前存在的 PCI 设备列表中寻找与 vendor 和 device 参数匹配的 PCI 设备，如果找到，就增加对应 pci_dev 的引用计数并返回该结构体。使用 pci_get_device()函数后，记住要调用 pci_put_dev()函数减小使用计数。

类似于 pci_get_device()的函数还包括：

```
    struct pci_dev *pci_get_subsys (unsigned int vendor, unsigned int device,
unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
```

ss_vendor 参数为子系统厂商 ID，ss_device 参数为子系统设备 ID。因此这个函数用于从系统中当前存在的 PCI 设备列表中寻找与 vendor、device、ss_vendor 和 ss_device 参数匹配的 PCI 设备。

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

这个函数在指定的 PCI 总线上查找指定功能编号的 PCI 设备。

pci_get_device()、pci_get_subsys()和 pci_get_slot()函数只是查找的限制条件不一样，它们都会增加 PCI 设备的引用计数，因此，在调用者结束使用 pci_dev 之前，它必须调用 pci_dev_put()函数减少引用计数，如代码清单 21.12 所示。

代码清单 21.12　pci_get_device 等函数的使用

```
1 struct pci_dev *dev;
2 dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
3 if (dev)
4 {
```

```
5   /* 使用 PCI 设备 */
6   ...
7   pci_dev_put(dev); /* 减小引用计数 */
8 }
```

21.3 实例：Intel 810 主板声卡驱动

Intel 810 主板的集成声卡符合 AC97 标准，其驱动的 PCI 相关部分包含 pci_driver 的定义、支持设备列表的 pci_device_id 数组定义，对 pci_driver 的注册和注销发生在模块加载函数与卸载函数中，代码清单 21.13 所示为这一部分的实现。

代码清单 21.13　Intel 810 主板声卡驱动的 PCI 数据结构定义及模块加载与卸载函数

```
1  /* pci_driver 结构体 */
2  static struct pci_driver i810_pci_driver =
3  {
4    .name       = I810_MODULE_NAME,
5    .id_table   = i810_pci_tbl,
6    .probe      = i810_probe,
7    .remove     = __devexit_p(i810_remove),
8  #ifdef CONFIG_PM
9    .suspend    = i810_pm_suspend,
10   .resume     = i810_pm_resume,
11 #endif /* CONFIG_PM */
12 };
13
14 /* 支持的设备列表 */
15 static struct pci_device_id i810_pci_tbl [] = {
16  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801AA_5,
17   PCI_ANY_ID, PCI_ANY_ID, 0, 0, ICH82801AA},
18  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801AB_5,
19   PCI_ANY_ID, PCI_ANY_ID, 0, 0, ICH82901AB},
20  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_440MX,
21   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTEL440MX},
22  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801BA_4,
23   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH2},
24  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801CA_5,
```

```
25   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH3},
26  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801DB_5,
27   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
28  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801EB_5,
29   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH5},
30  {PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_7012,
31   PCI_ANY_ID, PCI_ANY_ID, 0, 0, SI7012},
32  {PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP1_AUDIO,
33   PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
34  {PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP2_AUDIO,
35   PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
36  {PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP3_AUDIO,
37   PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
38  {PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_OPUS_7445,
39   PCI_ANY_ID, PCI_ANY_ID, 0, 0, AMD768},
40  {PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_8111_AUDIO,
41   PCI_ANY_ID, PCI_ANY_ID, 0, 0, AMD8111},
42  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ESB_5,
43   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
44  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ICH6_18,
45   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
46  {PCI_VENDOR_ID_NVIDIA,  PCI_DEVICE_ID_NVIDIA_CK804_AUDIO,
47   PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
48  {0,}
49 };
50
51 /* 模块加载，注册 pci_driver */
52 static int __init i810_init_module (void)
53 {
54  int retval;
55
56  printk(KERN_INFO "Intel 810 + AC97 Audio, version "
57          DRIVER_VERSION ", " __TIME__ " " __DATE__ "\n");
58
59  retval = pci_register_driver(&i810_pci_driver);
60  if (retval)
61      return retval;
62
63  if(ftsodell != 0) {
64      printk("i810_audio: ftsodell is now a deprecated option.\n");
```

```
    65  }
    66  if(spdif_locked > 0 ) {
    67      if(spdif_locked == 32000 || spdif_locked == 44100 || spdif_locked
== 48000) {
    68          printk("i810_audio: Enabling S/PDIF at sample rate %dHz.\n",
spdif_locked);
    69      } else {
    70          printk("i810_audio: S/PDIF can only be locked to 32000,
44100, or 48000Hz.\n");
    71          spdif_locked = 0;
    72      }
    73  }
    74
    75  return 0;
    76 }
    77
    78 /* 模块卸载，注销 pci_driver */
    79 static void __exit i810_cleanup_module (void)
    80 {
    81  pci_unregister_driver(&i810_pci_driver);
    82 }
    83
    84 module_init(i810_init_module);
    85 module_exit(i810_cleanup_module);
```

在 pci_driver 的 probe()成员函数中，将分析声卡占用的 I/O 资源、I/O 内存资源及其 IRQ，并申请这些资源，之后会注册 OSS 声卡驱动的/dev/dsp 接口和/dev/mixer 接口，如代码清单 20.14 所示。

代码清单 20.14　Intel 810 声卡驱动的探测函数

```
    1   static int __devinit i810_probe(struct pci_dev *pci_dev, const
struct
    2    pci_device_id *pci_id)
    3   {
    4    struct i810_card *card;
    5
    6    if (pci_enable_device(pci_dev))  //使能 PCI 设备
    7      return  - EIO;
    8
```

```
    9     /* 设置设备的 DMA 掩码 */
    10    if (pci_set_dma_mask(pci_dev, I810_DMA_MASK))
    11    {
    12      printk(KERN_ERR "i810_audio: architecture does not support"
    13        " 32bit PCI busmaster DMA\n");
    14      return  - ENODEV;
    15    }
    16
    17    /* 分配全局数据结构 i810_card 的内存 */
    18     if ((card = kmalloc(sizeof(struct i810_card), GFP_KERNEL)) ==
NULL)
    19    {
    20      printk(KERN_ERR "i810_audio: out of memory\n");
    21      return  - ENOMEM;
    22    } memset(card, 0, sizeof(*card));
    23
    24    card->initializing = 1;
    25    card->pci_dev = pci_dev;
    26    card->pci_id = pci_id->device;
    27    /* 寄存器地址 */
    28    card->ac97base = pci_resource_start(pci_dev, 0);
    29    card->iobase = pci_resource_start(pci_dev, 1);
    30    if (!(card->ac97base) || !(card->iobase))
    31    {
    32      card->ac97base = 0;
    33      card->iobase = 0;
    34    }
    35
    36    /* 如果芯片有 mmio 能力，检查它 */
    37    if (card_cap[pci_id->driver_data].flags &CAP_MMIO)
    38    {
    39      card->ac97base_mmio_phys = pci_resource_start(pci_dev, 2);
    40      card->iobase_mmio_phys = pci_resource_start(pci_dev, 3);
    41
    42      if ((card->ac97base_mmio_phys) && (card->iobase_mmio_phys))
    43      {
    44        card->use_mmio = 1;
    45      }
    46      else
    47      {
```

```
48        card->ac97base_mmio_phys = 0;
49        card->iobase_mmio_phys = 0;
50      }
51    }
52
53    /*I/O资源不可获得 */
54    if (!(card->use_mmio) && (!(card->iobase) || !(card->ac97base)))
55    {
56      printk(KERN_ERR "i810_audio: No I/O resources available.\n");
57      goto out_mem;
58    }
59
60    card->irq = pci_dev->irq;
61    card->next = devs;
62    card->magic = I810_CARD_MAGIC;
63    #ifdef CONFIG_PM
64      card->pm_suspended = 0;
65    #endif
66    spin_lock_init(&card->lock);
67    spin_lock_init(&card->ac97_lock);
68    devs = card;
69
70    pci_set_master(pci_dev); //设置为总线主DMA
71
72    printk(KERN_INFO "i810: %s found at IO 0x%04lx and 0x%04lx, "
73                     "MEM   0x%04lx   and   0x%04lx,   IRQ   %d\n",
card_names[pci_id->driver_data], card
74      ->iobase, card->ac97base, card->ac97base_mmio_phys,
75      card->iobase_mmio_phys,card->irq);
76
77    card->alloc_pcm_channel = i810_alloc_pcm_channel;
78    card->alloc_rec_pcm_channel = i810_alloc_rec_pcm_channel;
79    card->alloc_rec_mic_channel = i810_alloc_rec_mic_channel;
80    card->free_pcm_channel = i810_free_pcm_channel;
81
82    if ((card->channel = pci_alloc_consistent(pci_dev, sizeof(struct
83      i810_channel)*NR_HW_CH, &card->chandma)) == NULL)
84    {
```

```
    printk(KERN_ERR "i810: cannot allocate channel DMA memory\n");
    goto out_mem;
  }

  ...
  /* 申请 I/O 资源 */
                    if     (!request_region(card->iobase,     64,
card_names[pci_id->driver_data]))
  {
    printk(KERN_ERR "i810_audio: unable to allocate region %lx\n",
      card->iobase);
    goto out_region1;
  }
                  if     (!request_region(card->ac97base,     256,
card_names[pci_id->driver_data]))
  {
    printk(KERN_ERR "i810_audio: unable to allocate region %lx\n",
card
      ->ac97base);
    goto out_region2;
  }

  if (card->use_mmio)
  {
    /* 申请 I/O 内存资源 */
         if  (request_mem_region(card->ac97base_mmio_phys,  512,
"ich_audio MMBAR"))
    {
      ...
    }
    else
    {
      card->use_mmio = 0;
    }
  }

  /* 初始化 AC97 codec 并注册/dev/mixer 接口 */
  if (i810_ac97_init(card) <= 0)
    goto out_iospace;
  pci_set_drvdata(pci_dev, card); //设置 PCI 设备私有数据
```

```
121
122  if (clocking == 0)
123  {
124    clocking = 48000;
125    i810_configure_clocking();
126  }
127
128  /* 注册/dev/dsp */
129  if ((card->dev_audio = register_sound_dsp(&i810_audio_fops,  -
1)) < 0)
130  {
131    int i;
132    printk(KERN_ERR "i810_audio: couldn't register DSP device!\n");
133    for (i = 0; i < NR_AC97; i++)
134    if (card->ac97_codec[i] != NULL)
135    {
136      unregister_sound_mixer(card->ac97_codec[i]->dev_mixer);
137      ac97_release_codec(card->ac97_codec[i]);
138    }
139    goto out_iospace;
140  }
141
142  /* 申请 irq，注册中断处理程序 */
143       if   (request_irq(card->irq,   &i810_interrupt,   SA_SHIRQ,
card_names[pci_id
144    ->driver_data], card))
145  {
146     printk(KERN_ERR "i810_audio: unable to allocate irq %d\n",
card->irq);
147    goto out_iospace;
148  }
149
150  card->initializing = 0;
151  return 0;
152  ...
153 }
```

在 pci_driver 的 remove()成员函数中，将释放声卡所占用的 I/O 资源、I/O 内存资源及其 IRQ，之后会注销 OSS 声卡驱动的/dev/dsp 接口和/dev/mixer 接口，如代码清单 20.15 所示。

代码清单 20.15　Intel 810 声卡驱动的移除函数

```
1  static void __devexit i810_remove(struct pci_dev *pci_dev)
2  {
3    int i;
4    struct i810_card *card = pci_get_drvdata(pci_dev);
5    /* 释放硬件资源 */
6    free_irq(card->irq, devs);
7    release_region(card->iobase, 64);
8    release_region(card->ac97base, 256);
9        pci_free_consistent(pci_dev,  sizeof(struct  i810_channel)
*NR_HW_CH, card
10     ->channel, card->chandma);
11   if (card->use_mmio)
12   {
13     iounmap(card->ac97base_mmio);
14     iounmap(card->iobase_mmio);
15     release_mem_region(card->ac97base_mmio_phys, 512);
16     release_mem_region(card->iobase_mmio_phys, 256);
17   }
18
19   /* 注销/dev/mixer 和/dev/dsp */
20   for (i = 0; i < NR_AC97; i++)
21   if (card->ac97_codec[i] != NULL)
22   {
23     unregister_sound_mixer(card->ac97_codec[i]->dev_mixer);
24     ac97_release_codec(card->ac97_codec[i]);
25     card->ac97_codec[i] = NULL;
26   }
27   unregister_sound_dsp(card->dev_audio);
28   kfree(card);
29 }
```

代码清单 20.14 第 129 行注册 dsp 接口时，给出的参数 i810_audio_fops 是 dsp 接口文件操作结构体，内部包含了 dsp 接口的 i810_open()、i810_release()、i810_read()、i810_write()、i810_poll()、i810_ioctl()等成员函数。118 行调用的 i810_ac97_init()函数会通过 register_sound_mixer(&i810_mixer_fops, -1))注册/dev/mixer 接口，其内部包含了 i810_open_mixdev()、i810_ioctl_mixdev()函数。这些函数的实现与第 17 章所讲解的结构非常相似，由于本章的重点是讲解 PCI 设备驱动的 PCI 相关部分，因此，对于 OSS 驱动部分不再赘述。

由以上分析可知，Intel 810 主板集成声卡的驱动主要由两部分组成，在驱动的模块加载和卸载函数中，会分别注册和注销 pci_driver，pci_driver 的探测和移除函数会分别申请/释放资源、注册/注销/dev/dsp 和/dev/mixer 接口。Intel 810 主板集成声卡的 OSS 驱动部分主要包含/dev/dsp、/dev/mixer 接口的 file_operations 成员实现，如图 21.8 所示。

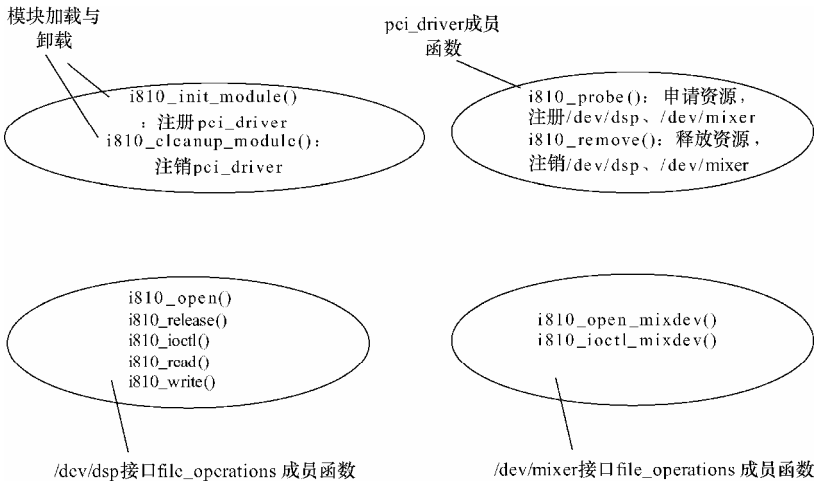

图 21.8　Intel 810 主板集成声卡驱动组成

21.4 总结

PCI 设备驱动只是字符设备、tty 设备、网络设备、音频设备等与系统的一个接口，因此，驱动将由两部分组成，一部分是 PCI 相关部分，另一部分是设备本身所属类驱动。PCI 驱动的核心数据结构是 pci_driver，在 probe()成员函数中将申请资源并注册对应的字符设备、tty 设备、网络设备、音频设备等，而 remove()成员函数中将释放资源并注销对应的字符设备、tty 设备、网络设备、音频设备等。

- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

  http://www.farsight.com.cn/courses/TS-LinuxApp.htm

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm