(3 条消息) 浅谈 Linux PCI 设备驱动（一）_Linux Driver 回忆录 - CSDN 博客

要弄清楚 Linux PCI 设备驱动，首先要明白，所谓的 Linux PCI 设备驱动实际包括 **Linux PCI 设备驱动和设备本身驱动**两部分。不知道读者理不理解这句话，本人觉得这句话很重要，对于 PCI、USB 这样的驱动来说，必须要理解这个概念，才能明白该如何看待 Linux 下的 PCI 和 USB 以及类似的总线型的驱动。理由也很简单，就是 Linux PCI 驱动是内核自带的，或者说内核帮你写好了！而我们需要完成的就是设备本身的驱动，比如网卡驱动等。当然，并不是说内核帮咱们写好了 Linux PCI 驱动我们什么就不用做了，至少你要明白内核大致都干了些什么，这样你才能明白你该干什么，如何完成设备本身的驱动。这跟我们学习操作系统时要学习很多系统调用接口一样的道理，不知道这些接口，怎么用操作系统或者说操作系统给你提供的功能呢？ 所以这里我们就来研究下 Linux PCI 驱动到底都干了些什么，以便我们在此基础上完成我们设备本身的驱动。

在 http://tldp.org/LDP/tlk/dd/pci.html 这篇文章里(整本书叫做 The Linux Kernel，中文翻译见 http://oss.org.cn/ossdocs/linux/kernel/ 本文也参考了该中文翻译) 提到了：

Linux PCI 初始化代码逻辑上分为三个部分：

(1)PCI 设备驱动程序（即上面提到的 Linux PCI 设备驱动）

这个伪设备驱动程序从总线 0 开始查询 PCI 系统并且定位系统中所有的 PCI 设备和 PCI 桥。它建立一个可以用来描述这个 PCI 系统拓朴层次的数据结构链表。并且对所有的发现的 PCI 桥编号。

(2)PCI BIOS

这个软件层提供在 bib-pci-bios 归约中描述的服务。虽然 Alpha AXP 不提供 BIOS 服务，在其 Linux 版本中包含了相应的功能。

(3)PCI Fixup

与特定系统相关的 PCI 初始化修补代码

而这里主要就是探讨 Linux PCI 设备驱动，会在最后列出一段包含设备本身驱动的示例代码，仅供参考。

一、概述及简介

PCI(Periheral Component Interconnect) 有三种地址空间：PCI I/O 空间、PCI 内存地址空间和 PCI 配置空间。其中，PCI I/O 空间和 PCI 内存地址空间由**设备驱动程序 (**即上面提到的设备本身驱动**)** 使用，而 PCI 配置空间由 **Linux 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。所以这里的 PCI 设备驱动就是要大致描述对于 PCI 设备驱动，Linux 内核都帮我们做了什么 (主)，接着就是我们应该完成什么 (次)。

(1)Linux 内核做了什么

简单的说，Linux 内核主要就做了对 PCI 设备的枚举和配置；这些工作都是在 Linux 内核初始化时完成的。

枚举：对于 PCI 总线，有一个叫做 PCI 桥的设备用来将父总线与子总线连接。作为一种特殊的 PCI 设备，PCI 桥主要包括以下三种：

1). 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)”。

2). PCI/ISA 桥： 用于连接旧的 ISA 总线。通常，PCI 中类似 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)”

3). PCI-to-PCI 桥 (以下称为 PCI-PCI 桥)：用于连接 PCI 主总线(Primary Bus) 和次总线(Secondary Bus)。PCI-PCI 桥所处的 PCI 总线称为主总线，即次总线的父总线；PCI-PCI 桥所连接的 PCI 总线称为次总线，即主总线的子总线。

下图摘自 PCI Local Bus Specification Revision 2.1，可以看到 PCI-PCI 桥的 Class Code(见图 3) 就是 0x060400。

Base Class 06h

This base class is defined for all types of bridge devices. A PCI bridge is any PCI device that maps PCI resources (memory or I/O) from one side of the device to the other. Several sub-class values are defined. There are no register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
06h	00h	00h	Host bridge.
	01h	00h	ISA bridge.
	02h	00h	EISA bridge.
	03h	00h	MCA bridge.
	04h	00h	PCI-to-PCI bridge.
	05h	00h	PCMCIA bridge.
	06h	00h	NuBus bridge.
	07h	00h	CardBus bridge.
	80h	00h	Other bridge device.

CPU 通过 Host/PCI 桥与一条 PCI 总线相连，处在这种位置上的 PCI 总线称为根总线。PC 机中通常只有一个 Host/PCI 桥，在一条 PCI 总线的基础上，可以再通过 PCI 桥连接到其他次一层的总线，例如通过 PCI-PCI 桥可以连接到另一条 PCI 总线，通过 PCI-ISA 桥可以连接到一条 ISA 总线。事实上，现代 PC 机中的 ISA 总线正是通过 PCI-ISA 桥连接在 PCI 总线上的。这样，通过使用 PCI-PCI 桥，就构筑起了一个层次的、树状的 PCI 系统结构。对于上层的总线而言，连接在这条总线上的 PCI 桥也是一个设备。但是这是一种特殊的设备，它既是上层总线上的一个设备，实际上又是上层总线的延伸。 所谓**枚举**，就是从 Host/PCI 桥开始进行探测和扫描，逐个 “枚举” 连接在第一条 PCI 总线上的所有设备并记录在案。如果其中的某个设备是 PCI-PCI 桥，则又进一步再探测和扫描连在这个桥上的次级 PCI 总线，就这样递归下

扫描连在这个桥上的次级 PCI 总线。就这样递归下去，直到穷尽系统中的所有 PCI 设备。其结果，是在内存中建立起一棵代表着这些 PCI 总线和设备的 PCI 树。每个 PCI 设备（包括 PCI 桥设备）都由一个 pci_dev 结构体来表示，而每条 PCI 总线则由 pci_bus 结构来表示。你有通过 PCI 桥建立起的硬件设备树，我有内存中通过数据结构构建的软件树，多么和谐 呵呵。

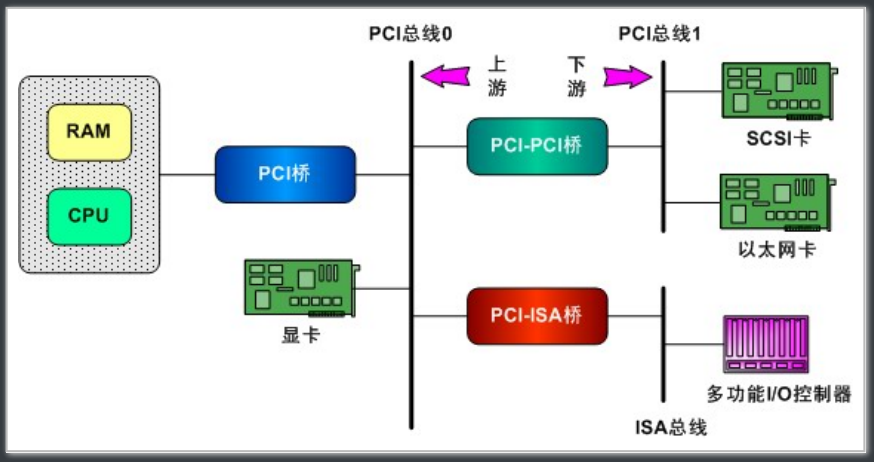

图 1　PCI 系统示意图

配置：PCI 设备中一般都带有一些 RAM 和 ROM 空间，通常的控制 / 状态寄存器和数据寄存器也往往以 RAM 区间的形式出现，而这些区间的地址在设备内部一般都是从 0 开始编址的，那么当总线上挂接了多个设备时，对这些空间的访问就会产生冲突。所以，这些地址都要先映射到系统总线上，再进一步映射到内核的虚拟地址空间。而所谓的配置就是通过对 PCI 配置空间的寄存器进行操作从而完成地址的映射 (只完成内部编址映射到总线地址的工作，而映射到内核的虚拟地址空间是由设备本身的驱动要做的工作)。

(2)Linux 内核怎么做的

这里首先要说明的是，对于 PCI 的设备初始化 (即上面提到的枚举和配置工作)，PC 机的 BIOS 和 Linux 内核都可以做。一般而言，只要是采用 PCI 总线的 PC 机，其 BIOS 就必须提供对 PCI 总线操作的支持，因而称为 PCI BIOS。而且最早 Linux 内核也是通过这种 BIOS 调用的方式来获取系统中的 PCI 设备信息的，但是不是所有的平台都有 BIOS(如某些嵌入式系统)，并且在实践中也发现有些母板上的 PCI BIOS 存在这样那样的问题，所以后来就改由 Linux 内核自己动手了，自己动手 丰衣足食 呵呵。不过，Linux 内核还是很体贴的在 make menuconfig 的选项里为我们提供了自己选择的权利，即 PCI access mode，里面提供了四个选项分别是 BIOS、MMconfig、Direct 和 Any。Direct 方式就是抛开 BIOS 而由内核自己完成初始化工作的意思。

二、开始我们的枚举与配置之路

注：为了更清晰，简单的描述 PCI 设备的初始化过程 (因为 2.4.18 中还没有引入设备驱动模型，这样可以让我们专心研究 PCI 设备驱动本身)。这里是对 Linux-2.4.18 的内核进行的分析，主要原因大家从参考资料中也应该能明白，这里很多就是参考了[1] 中的资料来分析的。如果想学 PCI 设备驱动，那么应该好好看看 [1] 的第八章中的 PCI 总线一节。然后再能找到一个驱动的例子代码看看，就可以说算是对 PCI 设备驱动入门了，当然，前提是都看懂了 呵呵。

废话少说，下面进入正题。前面提到了 PCI 有三种地址空间，其中的 PCI 配置空间是给 Linux 内核中的 PCI 初始化代码用的，也就是我们这里的枚举与配置时用到的。那么这个 PCI 配置空间里放的是什么东西呢，显然应该是寄存器，称为配置寄存器组。当 PCI 设备上电时，硬件保持未激活状态。即该设备只会对配置事务做出响应。上电时，设备上不会有内存和 I/O 端口映射到计算机的地址空间；其他设备相关的功能，例如中断报告，也被禁止。

PCI 标准规定每个设备的配置寄存器组最多可以有 256 字节的连续空间，其中开头的 64 字节的用途和格式是标准的，称为配置寄存器的头部。系统中提供一些与硬件有关的机制，使得 PCI 配置代码可以检测在一个给定的 PCI 总线上所有可能的 PCI 配置寄存器头部，从而知道哪个 PCI 插槽上目前有设备，哪个插槽上暂无设备。这是通过读 PCI 配置寄存器头部上的某个域完成的 (一般是“Vendor ID" 域)。如果一个插槽上为空，上述操作会返回一些错误返回值，如 0xFFFFFFFF。这种头部(指 64 字节头部) 又有三种，其中 “0 型”(type 0) 头部用于一般的 PCI 设备，“1 型”头部用于各种 PCI-PCI 桥，“2 型”头部是用于 PCI-CardBus 桥的，CardBus 是笔记本电脑中使用的总线，我们不关心。而 64 字节头部中的 16 个字节中又包含着有关头部的类型、设备的种类、设备的一些性质、由谁制造等等信息。根据这 16 个字节中提供的信息，来确定应该怎样进一步解释和处理剩余头部

中的 48 个字节。对于这 16 个字节的地址，include/linux/pci.h 中定义了这样一些常数。

```
#define PCI_VENDOR_ID  0x00 /* 16 bits */
#define PCI_DEVICE_ID  0x02 /* 16 bits */
#define PCI_COMMAND  0x04 /* 16 bits */

#define PCI_STATUS  0x06 /* 16 bits */

#define PCI_CLASS_REVISION 0x08 /* High 24 bits are class, low 8  revision */
#define PCI_REVISION_ID         0x08    /* Revision ID */
#define PCI_CLASS_PROG          0x09    /* Reg. Level Programming Interface */
#define PCI_CLASS_DEVICE        0x0a    /* Device class */

#define PCI_CACHE_LINE_SIZE 0x0c /* 8 bits */
#define PCI_LATENCY_TIMER 0x0d /* 8 bits */
#define PCI_HEADER_TYPE  0x0e /* 8 bits */
```

对应我们的图 3(见下) 中的前 16 字节。而且我们也看到了紧挨着 PCI_HEADER_TYPE (即存放头部类型的寄存器) 下面定义的就是上面提到的三种类型的头部：

```
#define  PCI_HEADER_TYPE_NORMAL   0
#define  PCI_HEADER_TYPE_BRIDGE     1
```

```
#define  PCI_HEADER_TYPE_CARDBUS  2
```

在 Linux 系统上，可以通过 cat /proc/pci 等命令查看系统中所有 PCI 设备的类别、型号以及厂商等等信息，那就是从这些寄存器来的。下面是在虚拟机中用 lspci -x 命令的信息截取 (lspci 命令也是使用 / proc 文件作为其信息来源)：

```
00:00.0 Host bridge: Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX Host bridge (rev 01)
00: 86 80 90 71 06 00 00 02 01 00 00 06 00 00 00 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 ad 15 76 19
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

首先要说明的是 PCI 寄存器是小端字节序格式的。那么根据最下面的 PCI 配置寄存器组的结构 (图 3)，显然这个 Host bridge 的 Vendor ID 是 0x8086，我不说你也能猜到这个 Vendor 就是 Intel 了。

这里有个问题要先说清楚，就是这些寄存器的地址问题，不然往后就进行不下去了。配置寄存器可以让我们来进行配置以便完成 PCI 设备上的存储空间的访问，但这些配置寄存器本身也位于 PCI 设备地址空间中，如何访问这部分空间也就成了我们整个初始化工作的一个入口点，就像每个可执行程序都要有入口点一样。PCI 采用的办法是让所有设备的配置寄存器组都采用相同的地址，由所在总线的 PCI 桥在访问时附加上其他条件来区分。而 CPU 则通过一个统一的入口地址向 “宿主 --PCI 桥” 发出命令，由相应的 PCI 桥

间接的完成具体的读写。对于 i386 结构的处理器，

PCI 总线的设计者在 I/O 地址空间保留了 8 个字节用于这个目的，那就是 0xCF8~0xCFF。这 8 个字节构成了两个 32 位的寄存器，第一个是“地址寄存器”0xCF8，第二个是“数据寄存器”0xCFC。要访问某个设备中的某个配置寄存器时，CPU 先往地址寄存器中写入目标地址，然后通过数据寄存器读写数据。不过，写入地址寄存器的目标地址是一种总线号、设备号、功能号以及设备寄存器地址在内的综合地址。格式如图 2：

图 2　写入地址寄存器 0xCF8 的综合地址

这里的总线号、设备号和功能号是对配置寄存器地址的扩充，就是上面提到的附加的其他条件。首先每个 PCI 总线都有个总线号，主总线的总线号为 0，其余的则由 CPU 在枚举阶段每当探测到一个 PCI 桥时便为其指定一个，依次递增。设备号一般代表着一块 PCI 接口卡 (更确切的说是 PCI 总线接口芯片)，通常取决于插槽的位置。每块 PCI 接口卡上可以有若干个功能模块，这些功能模块共用一个 PCI 总线接口芯片，包括其中用于地址映射的电子线路，以降低成

片，也指其中用于地址映射的电子线路，以降低成本。从逻辑的角度说，每个“功能” 实际上就是一个设备 (看过 USB 设备驱动的人很眼熟吧 呵呵)，所以设备号和功能号合在一起又可以称作“逻辑设备号”，而每块卡上最多可以容纳 8 个设备。显然，这些字段(指整个 32bit) 结合在一起就惟一确定了系统中的一项 PCI 逻辑设备。开始时，只有 0 号总线可以访问，在扫描 0 号总线时如果发现上面某个设备是 PCI 桥，就为之指定一个新的总线号，例如 1，这样 1 号总线就可以访问了，这就是枚举阶段的任务之一。

现在请读者考虑一个问题：当我们拿到一块 PCI 网卡，我们把它插到 PC 的主板上，打算写个这个网卡的驱动。那么第一步该干什么呢？读者可以回顾前面的内容，既然我们说 Linux 内核帮我们做了设备的枚举和配置工作，那么我在写网卡驱动之前是不是可以先看看 Linux 内核对我们的这个 PCI 网卡设备完成的枚举工作的结果呢？或者直白些说，我把网卡插上了，现在 Linux 内核有没有识别出这块设备呢？ 注意识别出设备跟能正常使用设备是不同的概念，这很好理解。安装过 PC 网卡驱动的人都知道，当设备的驱动没有安装时，我们在设备管理器中是可以看到这个设备的，不过上面是一个黄色的大问号。而在 Linux 系统中，我们可以通过 lspci 命令来查看。

下面是在 LDD3 的 PCI 驱动那一章截取的一段内容: lspci 的输出 (pciutils 的一部分, 在大部分发布中都有) 和在 /proc/pci 和 /porc/bus/pci 中的信息排布. PCI 设备的 sysfs 表示也显示了这种寻址方案, 还有 PCI 域信息. 当显示硬件地址时, 它可被显示为 2 个值(

PCI 域信息. 当显示设备地址时, 它们被显示为 2 个值(一个 8 - 位总线号和一个 8 - 位 设备和功能号), 作为 3 个值(bus, device, 和 function), 或者作为 4 个值 (domain, bus, device, 和 function); 所有的值常常用 16 进制显示.

例如, /proc/bus/pci/devices 使用一个单个 16 位字段 (来便于分析和排序), 而 /proc/bus/busnumber 划分地址为 3 个字段. 下面内容显示了这些地址如何显示, 只显示了输出行的开始:

```
$ 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
004a
00a0
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:
```

```
|-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:

|-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:
|-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:
`-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:
```

所有的 3 个设备列表都以相同顺序排列, 因为 lspci 使用 /proc 文件作为它的信息源. 拿 VGA 视频控制器作一个例子, 0x00a0 意思是 0000:00:14.0 当划分为域(16 位), 总线(8 位), 设备(5 位) 和功能 (3 位). 为什么 0x00a0 对应的是 0000:00:14.0 呢，这就要看图 2 中的内容了，根据图 2 中的寄存器对应 0x00a0 就代表着总线(8 位), 设备(5 位) 和功能(3 位).0x00a0=0000000010100000，很容易看出高 8 位是总线号也就是 0。剩下的 0xa0=10100000，可以看出如果低 3 位表示功能号，那么剩下的 10100 就是设备号，补全成 8 位的值就是 00010100 即 0x14.

Table 7. PCI Configuration Registers

Address	Access	Byte 3	Byte 2	Byte 1	Byte 0
00h	read only	Device ID		Vendor ID	
04h	read/write	Status		Command	
08h	read only	Class Code			Revision ID
0Ch	read/write	Reserved	Header Type	Latency Timer	Cache Line Size
10h	read/write	Base 0 Address (4M-byte prefetchable)			
14h	read/write	Base 1 Address (8M-byte nonprefetchable)			
18h	read/write	Base 2 Address (4 words I/O)			
24h	read only	Reserved			
2Ch	read only	Subsystem ID		Subsystem Vendor ID	
30h	read only	Reserved			
34h	read only	Reserved			Capabilities Pointer
38h	read only	Reserved			
3Ch	read/write	Max_Latency	Min_Grant	Interrupt Pin	Interrupt Line
40h	read only	Power Management Capabilities		Next Item Pointer	Capability ID
44h	read/write	Power Data	Reserved	Power Management Control/Status	
48h FFh	read only	Reserved			

图 2. PCI 配置寄存器组

参考资料：

[1] Linux 内核源代码情景分析（下册）

[2] Linux 设备驱动开发详解

[3] Linux 设备驱动 (第三版)

[4] 内核 Documentation 下的 pci.txt

[5] 精通 Linux 设备驱动开发

[6] http://tldp.org/LDP/tlk/dd/pci.html

[7] http://linux.die.net/man/8/lspci

[8] http://www.ibm.com/developerworks/cn/linux/l-pci/

全文完