基于 MIPS 的 Linux 内核 PCI 子系统分析系列
— PCI 枚举

Current Version： V0.1
Date： 2011-10-31
Author： leewg <wgl.liwengang@gmail.com>

版本	作者	参与者	时间	备注
0.1	Li Wengang		2011-10-31	初始化版本

备注：

内核源代码版本：2.6.36

代码流程的分支：基于 MIPS 下的 loongson 分支

说明：文档参考了网上的诸多关于 linux PCI 的文章，整理而成。
存在问题请大家即使指出，共同探讨进步。

一、PCI 基础

PCI 是 Peripheral Component Interconnect 的缩写，它因为高性能、低成本以及良好的扩展性而在计算机系统中被广泛使用。上至服务器，下至嵌入式设备都能找到它的身影。图 1 显示了一个标准 PCI 总线的组织结构图。

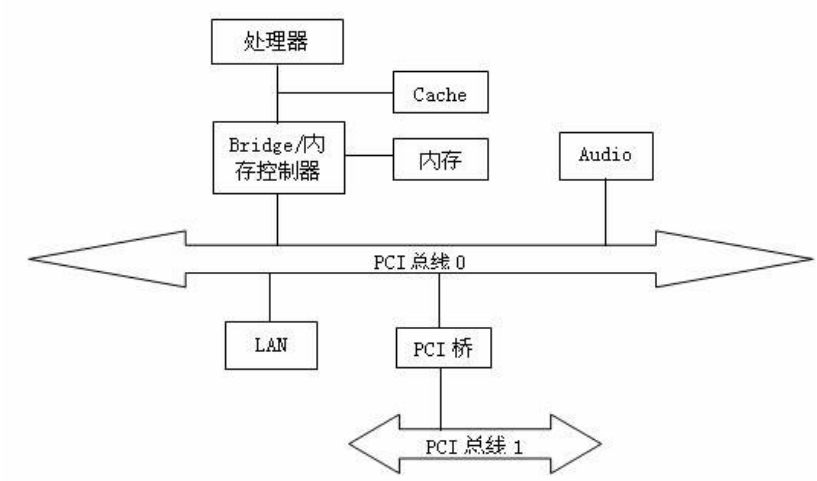


图 1 标准 PCI 总线的组织结构图

从图中我们可以看出 PCI 总线架构主要被分成三部分：

1、**PCI 设备**。符合 PCI 总线标准的设备就被称为 PCI 设备，PCI 总线架构中可以包含多个 PCI 设备。图中的 Audio、LAN 都是一个 PCI 设备。PCI 设备同时也分为主设备和目标设备两种，主设备是一次访问操作的发起者，而目标设备则是被访问者。

2、**PCI 总线**。PCI 总线在系统中可以有多条，类似于树状结构进行扩展，每条 PCI 总线都可以连接多个 PCI 设备/桥。上图中有两条 PCI 总线。

3、**PCI 桥**。当一条 PCI 总线的承载量不够时，可以用新的 PCI 总线进行扩展，而 PCI 桥则是连接 PCI 总线之间的纽带。图中的 PCI 桥有两个，一个桥用来连接处理器、内存以及 PCI 总线，而另外一条则用来连接另一条 PCI 总线。

PCI 总线操作

PCI 总线操作表示主设备向目标设备所发起的操作请求，最多有 16 种类型。主要类型有：IO 方式读/写，Memory 方式读/写，Configuration 方式读/写等。

PCI 配制空间

对于软件开发者来说，该如何对 PCI 设备进行编程呢？PCI 总线标准中定义了一套配置空间寄存器用于读取或者设置 PCI 设备的信息。每个 PCI 设备/桥都有自己的配置空间寄存器。

配置空间共有 256 字节，设备类型不同，其配置空间的布局也不尽相同。设备类型的区分可以通过配置空间内的 Header Type 寄存器（0Eh）进行，该寄存器值为 00h 表示当前设

备是一个 PCI 设备，01h 表示当前设备是一个 PCI 桥。

配置空间的前 64 字节是配置空间起始段，它对于每种类型的设备都是相同的。 显示了 PCI 设备的配置空间起始段。

图 2. PCI 设备的配置空间起始段

31 16	15 0	
Device ID	Vendor ID	00h
Status	Command	04h
Class Code	Revision ID	08h
BIST / Header Type	Latency Timer / Cacheline Size	0Ch
Base Address Register 1		10h
Base Address Register 2		14h
Base Address Register 3		18h
Base Address Register 4		1Ch
Base Address Register 5		20h
Base Address Register 6		24h
Cardbus CIS Pointer		28h
Subsystem ID	Subsystem Vendor ID	2Ch
Expansion ROM Base Address		30h
Reserved	Capabilities Pointer	34h
Reserved		38h
Max_Lat / Min_Gnt	Interrupt Pin / Interrupt Line	3Ch

图 3 显示了 PCI 桥的配置空间起始段。

31 16	15 0	
Device ID	Vendor ID	00h
Status	Command	04h
Class Code	Revision ID	08h
BIST / Header Type	Latency Timer / Cacheline Size	0Ch
Base Address Register 1		10h
Base Address Register 2		14h
Secondary Latency Timer / Subordinate Bus Number	Secondary Bus Number / Primary Bus Number	18h
Secondary Status	I/O Limit / I/O Base	1Ch
Memory Limit	Memory Base	20h
Prefetchable Memory Limit	Prefetchable Memory Base	24h
Prefetchable Base Upper 32 Bits		28h
Prefetchable Limit Upper 32 Bits		2Ch
I/O Limit Upper 16 Bits	I/O Base Upper 16 Bits	30h
Reserved	Capabilities Pointer	34h
Expansion ROM Base Address		38h
Bridge Control	Interrupt Pin / Interrupt Line	3Ch

配置空间寄存器有些是只读的，有些是可写的，下面介绍几个在编程时会用到的寄存器。

Device ID 和 Vendor ID 寄存器

这两个寄存器分别存放了设备信息和厂商信息（值在 0x0000 和 0xFFFF 之间，但不能取 0xFFFF），因此软件开发者可以通过读取这两个寄存器的值，并与 0xFFFF 比较，从而判断当前设备是否有效。

Command 和 Status 寄存器

Command 寄存器存放了设备的配置信息，比如是否允许 Memory/IO 方式的总线操作、

是否为主设备等。Status 寄存器存放了设备的状态信息，比如中断状态、错误状态等。

Header Type 寄存器

这个寄存器前面曾经提过，它定义了设备类型，比如 PCI 设备、PCI 桥等。

Base Address 寄存器

这个寄存器有三个作用。

1、该寄存器存放了 Memory/IO 访问空间的起始地址。

2、该寄存器存放了 Memory/IO 访问空间的大小，这个数据可以通过下面的方式读出：

a、往寄存器里写 0xFFFFFFFF；

b、读出寄存器的值，并取反；

c、将上一步的值加上 1 后就是该空间的大小。

3、该寄存器定义了这段地址空间的访问类型（Memory 方式还是 IO 方式）。

PCI 设备最多有 6 个 Base Address 寄存器，而 PCI 桥最多有 2 个 Base Address 寄存器。

Subordinate Bus Number，Secondary Bus Number 和 Primary Bus Number 寄存器

这三个寄存器只在 PCI 桥配置空间中存在，因为 PCI 桥会连接两条 PCI 总线，上行的总线被称为 Primary Bus，下行的总线被称为 Secondary Bus，Primary Bus Number 和 Secondary Bus Number 寄存器分别存储了上行和下行总线的编号，而 Subordinate Bus Number 寄存器则是存储了当前桥所能直接或者间接访问到的总线的最大编号。

二、Linux 内核对 PCI 的支持

Linux 内核（2.6 版本）在初始化之初就对所有 PCI 设备进行了扫描并且配制，具体操作分为下面几个步骤。

PCI 相关数据结构

Linux 提供了三类数据结构用以描述 PCI 控制器、PCI 设备以及 PCI 总线。

PCI 控制器

PCI 控制器用 pci_controller 结构来描述，对于多总线支持系统，它可能有多个 PCI controller 或一个 PCI controller 单支持多 channel，它有以下几个主要的属性：

- index：该属性标志 PCI 控制器的编号。
- next：该属性指向下一个 PCI 控制器，通过 next 属性，PCI 控制器可以形成一个单向链表。
- bus：该属性标志了当前 PCI 控制器所连接的 PCI 总线，它对应的数据结构是 pci_bus。
- pci_ops：该属性标志了当前 PCI 控制器所对应的 PCI 配制空间读写操作函数。
- mem_resource：该属性标志了当前 PCI 控制器的 memory 资源。
- io_resource：该属性标志了当前 PCI 控制器的 io 资源。
- io_map_base：PCI 设备的 IO map 基地址。

PCI 总线

PCI 总线用 pci_bus 结构来描述，它有以下几个主要的属性：

- node：连接到父总线的 children 链表中。
- parent：可通过该属性索引到上层 PCI 总线。

- self：该属性标志了连接的上行 PCI 桥（对应的数据结构是 pci_dev）。
- children：该属性标志了总线连接的所有 PCI 子总线链表。
- devices：该属性标志了总线连接的所有 PCI 设备链表。
- ops：该属性标志了总线上所有 PCI 设备的配制空间读写操作函数。
- number：该属性标志了当前 PCI 总线的编号。
- primary：该属性标志了 PCI 上行总线编号。
- secondary：该属性标志了 PCI 下行总线编号。
- subordinate：该属性标志了能够访问到的最大总线编号。
- resource：该属性标志了 Memory/IO 地址空间。

PCI 设备

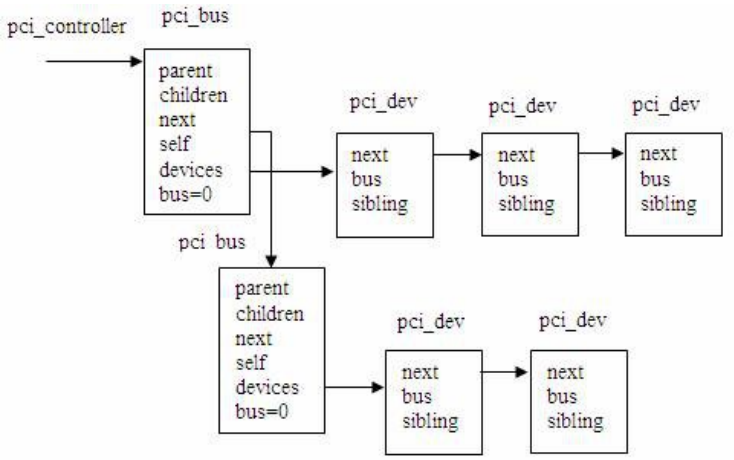
PCI 设备通过 pci_dev 来描述，它有一下主要属性：

- bus_list：通过它挂接到所在总线的 pci_dev 的 devices
- bus：该属性标志了当前设备所在的 PCI 总线（对应的数据结构是 pci_bus）。
- subordinate：桥设备使用。指向另一个 bus 的指针。
- devfn：该属性标志了设备编号和功能编号。
- vendor：该属性标志了供应商编号。
- device：该属性标志了设备编号。
- driver：该属性标志了设备对应的驱动代码（对应的数据结构是 pci_driver）。
- irq：该属性标志了中断号。
- resource：该属性标志了 Memory/IO 地址区间。

内核里的 PCI 数据结构图

当 Linux 内核在做 PCI 初始化工作时，它会根据图 4 建立一个由 pci_controller、pci_bus 和 pci_dev 三者组成的一个组织结构图。根据这个结构，软件开发者可以很方便的通过 PCI 控制器索引到每个 PCI 设备或者 PCI 总线。

图 4. 组织结构图



三、PCI 内部结构、枚举 分析

1、pci 内部结构分析

Linux PCI 子系统必须扫描系统中所有的 PCI 总线，寻找系统中所有的 PCI 设备(包括 PCI-PCI 桥设备)。系统中的每条 PCI 总线都有个编号 number，根 PCI 总线的编号为 0。系统当前存在的所有根总线(因为可能存在不止一个 Host/PCI 桥，那么就可能存在多条根总线) 都通过其 pci_bus 结构体中的 node 成员链接成一个全局的根总线链表，其表头由 struct list_head 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。

每个 PCI 设备都由一个 pci_dev 结构体表示，每个 pci_dev 结构体通过成员 bus_list 挂入其所在总线的 pci_dev 结构队列 devices(队列头是 pci_bus.devices，即该 pci 设备所在的 pci 总线的 devices 队列)，并且使指针 bus(指 pci_dev 结构体里的 bus 成员)指向代表着其所在总线的 pci_bus 结构。如果具体的设备是 PCI-PCI 桥，则还要使其指针 subordinate 指向代表着另一条 PCI 总线的 pci_bus 结构。

2、pci 枚举分析

系统如何知道当前连接了多少 PCI 设备？有多少根 PCI 总线？每个 PCI 设备的访问空间如何配置？等等。这些都得靠 PCI 自动扫描来完成。PCI 自动扫描主要做下面的工作：

1、扫描 PCI 总线，识别 PCI 总线上的所有设备。
2、对于连接在 PCI 总线上的所有 PCI 桥进行总线编号。
3、对于连接在 PCI 总线上的所有 PCI 设备和 PCI 桥进行 Memory/IO 访问空间的配置。

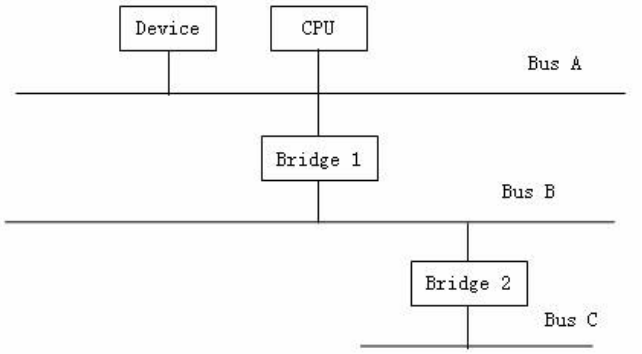
识别 PCI 总线上的设备

PCI 总线扫描的原理是从总线 0 扫描到总线 255，对于每条总线，系统都会扫描所有（总线号，设备号，功能号），通过 Configuration 方式读出每个设备的 Device ID 和 Vendor ID 寄存器，如果这两个寄存器的值是个有效值（非 0xFFFF），则说明当前设备是个有效的 PCI 设备/桥。进而再读取该设备的 Header Type 寄存器，如果该寄存器为 1，则表示当前设备是 PCI 桥，否则是 PCI 设备。

对所有 PCI 总线进行编号

PCI 桥如何知道它所连接的 PCI 总线情况呢？这就需要对 PCI 桥进行总线编号。前面介绍过 PCI 桥提供了 Primary Bus Number、Secondary Bus Number 和 Subordinate Bus Number 三个寄存器用于标志该桥所连接的 PCI 总线，下面通过一个示例来说明内核对于 PCI 总线是如何进行编号的。

图 6. 示例



1、系统运行初始，Bus A 为 0，通过上面的 PCI 总线扫描得到连接在 Bus A 上的 PCI 桥（即图中的 Bridge 1）。
2、下面开始设置 Bridge 1 的 Bus 寄存器。将 Primary Bus Number 寄存器设置成 Bus A 的编号，即 0。将 Secondary Bus Number 寄存器设置成 Bus B 的编号，它的值等于（Bus A + 1），也就是 1。由于暂时无法知道该桥所能访问的所有下行总线数目，Subordinate Bus Number 寄存器暂时设置成 0xFF。
3、当扫描完所有 Bus A 上所有（设备号，功能号）后，开始扫描 Bus B，Bus B 的编号在扫描完 Bus A 后已经得到，为 1。Bus B 的扫描方法同步骤（1），先扫描出 Bus B 上的 PCI 桥（即图中的 Bridge 2），然后配置 Primary Bus Number 寄存器为 1，Secondary Bus Number 寄存器为 2，而 Subordinate Bus Number 寄存器依然为 0xFF。
4、Bus B 扫描完后得到 Bus C 的编号，为 2。下面开始扫描 Bus C，因为 Bus C 上没有 PCI 桥，于是在扫描完其它（设备号，功能号）后，Bus C 的扫描结束。
5、由于 Bridge 2 所能访问到的最大 Bus 编号是 2，因此重新设置 Bridge 2 的 Subordinate Bus Number 寄存器为 2。
6、由于 Bridge 1 所能访问到的最大 Bus 编号也是 2，因此重新设置 Bridge 1 的

Subordinate Bus Number 寄存器为 2。

7、总线编号结束。

配置访问空间

当系统需要访问 PCI 设备时，它需要产生 Configuration、Memory 或者 IO 的读写操作，对于 Memory/IO 的访问方式来说，它们需要定义一个地址范围，落在这个地址范围的操作会被认为是相应的 Memory/IO 的读写操作。

通常 PCI 设备提供了最多 6 组 Base Address 寄存器，在 PCI 总线扫描时，每当扫描出一个可用的 PCI 设备后，会对该设备的 Base Address 寄存器进行 Memory/IO 访问空间的配置。

而对于 PCI 桥来说，它只提供了 2 组 Base Address 寄存器，当 PCI 总线扫描出一个 PCI 桥后，也会对该桥的 Base Address 寄存器进行 Memory/IO 访问空间的配置。

需要注意的是，在构建系统之初，需要明确当前系统的地址范围，划分出特定的物理地址作为 PCI Memory 或者 PCI IO 空间，在给 PCI 设备/桥进行访问空间配置时，就是取事先约定的地址空间中的某段地址进行配置，所有设备/桥的访问地址不能冲突。

下面使用网上一文章的例子，来说明 pci 枚举的过程。

配置一个 PCI-PCI 桥的时候，并不知道这个 PCI-PCI 桥的 subordinate bus number。那么就不知道该 PCI 桥下面是否还有其他的 PCI-PCI 桥。即使你知道，也不清楚如何对它们赋值。解决方法是利用上述的深度扫描算法来扫描每个总线。每当发现 PCI-PCI 桥就对它进行赋值。当发现一个 PCI-PCI 桥时，可以确定它的 secondary bus number。然后我们暂时先将其 subordinate bus number 赋值为 0xFF。紧接着，开始扫描该 PCI-PCI 桥的 downstream 桥。这个过程看起来有点复杂,下面的例子将给出清晰的解释：

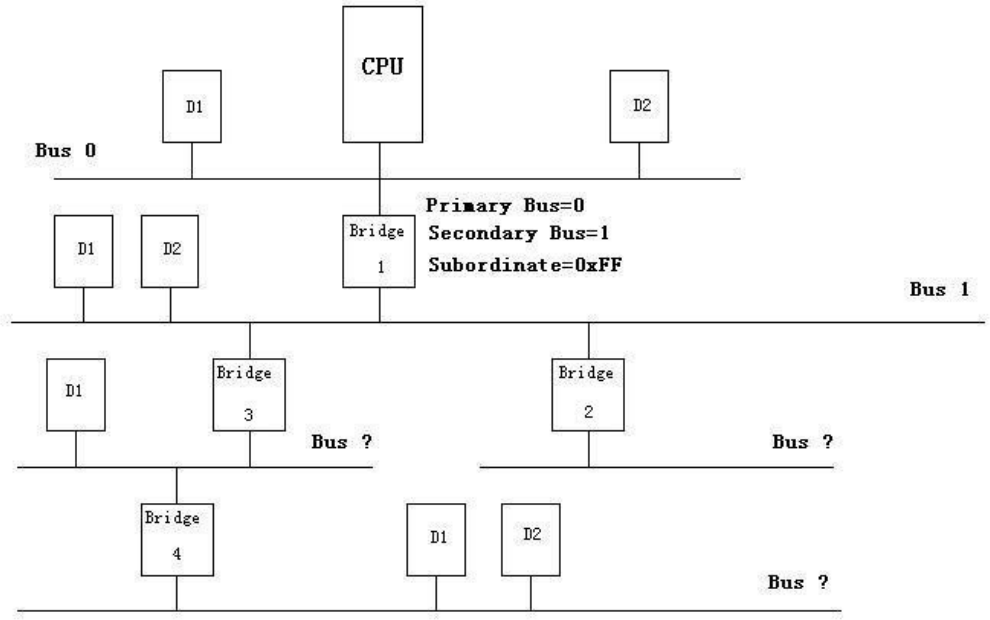


图 7　配置 PCI 系统 第一步

PCI-PCI 桥编号--第一步

以图 7 的拓扑结构为例，扫描时首先发现的桥是 Bridge1。Bridge 1 的 downstream PCI 总线号码被赋值 1。自然该桥的 secondary bus number 也是 1。其 subordinate bus number 暂时赋值为 0xFF。上述赋值的含义是所有类型 1 的含有 PCI 总线 1 或更高(<255)的号码的 PCI 配置地址将被 Bridge 1 传递到 PCI 总线 1 上。如果 PCI 总线号是 1，Bridge 1 还负责将配置地址的类型转换成类型 0(对于这里说的类型 0 和类型 1，请参考浅谈(一))。否则，就不做转换。上述动作就是开始扫描总线 1 时 Linux PCI 初始化代码所完成的对总线 0 的配置工作。

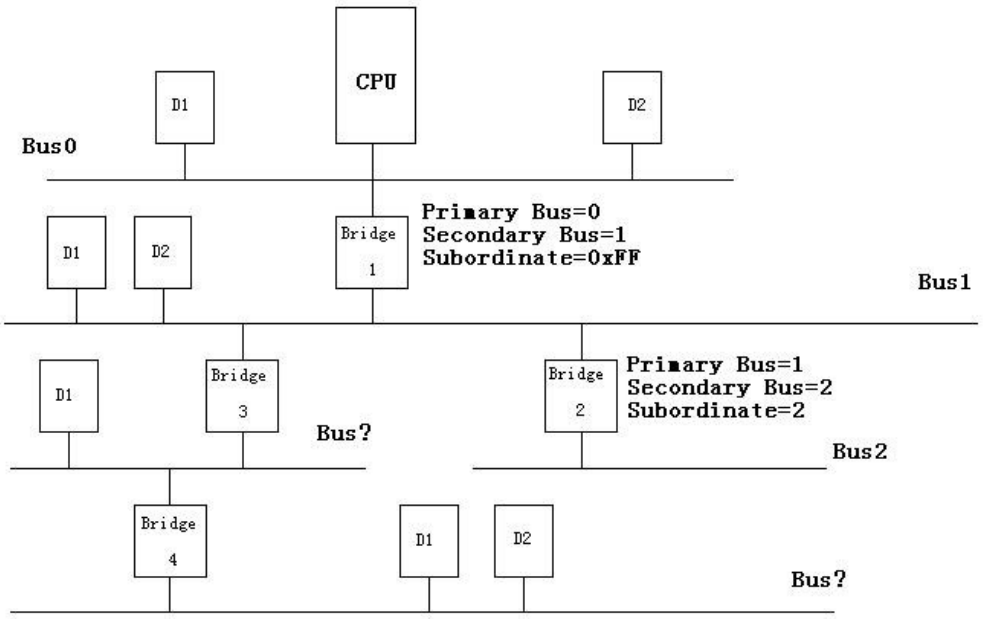


图 8 配置 PCI 系统 第二步

PCI-PCI 桥编号--第二步

由于 Linux PCI 设备驱动使用深度优先算法进行扫描，所以初始化代码开始扫描总线 1。从而 Bridge 2 被发现。因为在 Bridge 2 下面发现不再有 PCI-PCI 桥，所以 Bridge 2 的 subordinate bus number 是 2，等于它的 secondary bus number。图 8 显示了在这个时刻总线和 PCI-PCI 桥的赋值情况。

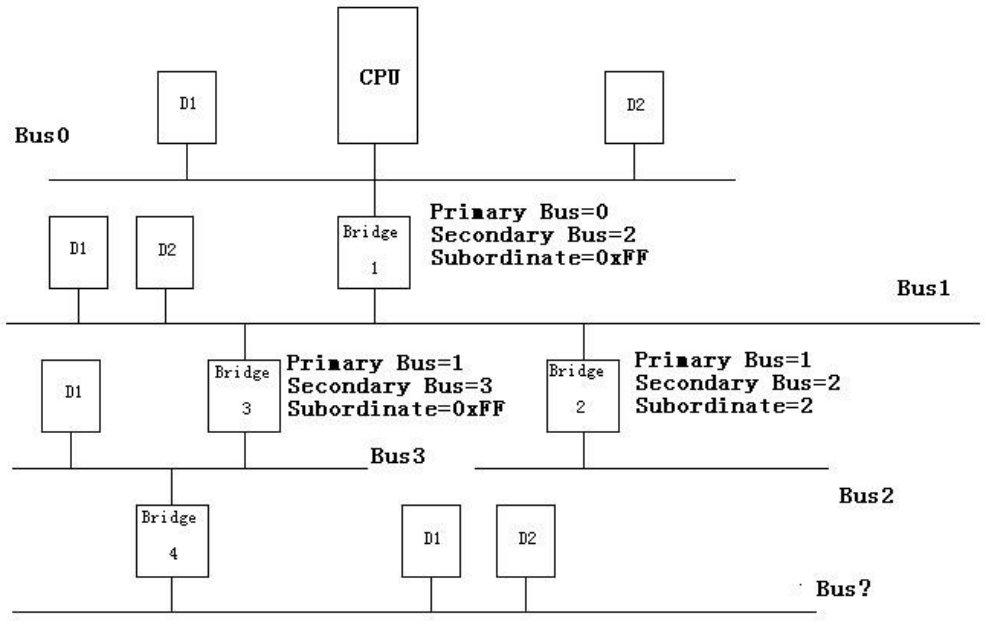


图 9　配置 PCI 系统 第三步

PCI-PCI 桥编号--第三步

Linux PCI 设备驱动代码从总线 2 的扫描中回来接着进行扫描总线 1，发现 Bridge 3。它的 primary bus number 被赋值为 1，secondary bus number 为 3。因为总线 3 上还发现了 PCI-PCI 桥，所以 Bridge 3 的 subordinate bus number 暂时赋值 0xFF。图 9 显示了这个时刻系统配置的状态。到目前为止，含有总线号 1，2，3 的类型 1 的 PCI 配置都可以正确地传送到相应的总线上。

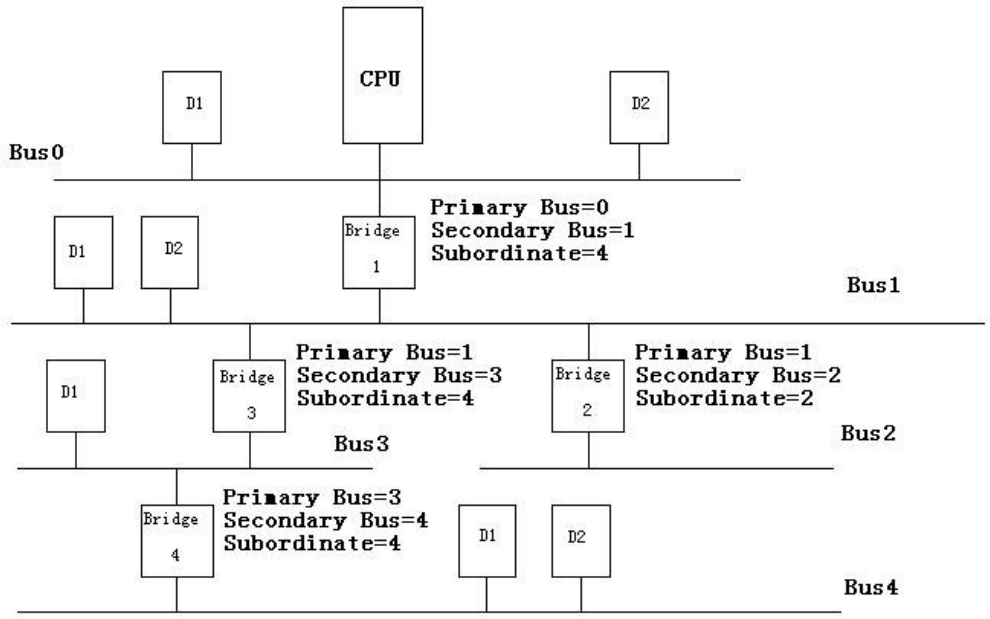


图 10　配置 PCI 系统 第四步

PCI-PCI 桥编号--第四步

现在 Linux 开始扫描 PCI 总线 3，Bridge 3 的 downstream。PCI 总线 3 上有另外一个 PCI-PCI 桥，Bridge 4。因此 Bridge 4 的 primary bus number 的值为 3，secondary bus number 为 4。由于 Bridge 4 下面没有别的桥设备，所以 Bridge 4 的 subordinate bus number 为 4。然后回到 PCI-PCI Bridge 3。这时就将 Bridge 3 的 subordinate bus number 从 0xFF 改为 4，表示总线 4 是从 Bridge 3 往下走的最远的 PCI-PCI 桥。最后，Linux PCI 设备驱动代码将 4 以同样的道理赋值给 Bridge 1 的 subordinate bus number。图 10 反映了系统最后的状态。

四、PCI 枚举代码分析

1、代码入口分析

pcibios_init 函数在两个文件中有定义。

a、arch/mips/loongson/common/pci.c arch_initcall()

b、arch/mips/pci/pci.c subsys_initcall()

由于 arch_initcall() 的执行优先级高于 subsys_initcall()申明，因此 a 中的函数为 pci 函数的入口函数，b 会晚于 a 在后续的初始化中执行。

关于 arch_initcall、subsys_initcall 会单独进行分析。（未分析）

2、pci_controller 注册

arch/mips/loongson/common/pci.c

pcibios_init—> register_pci_controller

完成 pci_controller 的注册；此过程是在 PCI subsystem initialization 之前执行的，因此只执行了将 pci controller resource 注册到 root resource 结构中去的操作（内核通过维护一个链表来维护 resource 模块；使用 child、sibling、parent 来建立 resource 结构，在 pci 系统中 root_resource 包含 mem 和 io 两部分、分别为 ioport_resource、iomem_resource），没有进行 pcibois_scanbus 函数对 pci bus 进行扫描。

3、总线枚举

arch/mips/pci/pci.c

```
pcibios_init
|→ pcibios_scanbus
    |→pci_scan_bus
    |   |→ pci_scan_bus_parented        （从根总线开始扫描）
    |       |→ pci_create_bus           （创建 bus0，并增加到 pci_root_buses 链表）
    |       |→ pci_scan_child_bus       （从 bus0 开始扫描其后面的总线）
    |       |   |→ pci_scan_slot        （bus0 上所有设备的枚举，32 设备×8 多功能）
    |       |       |→ pci_scan_single_device （扫描单功能设备）
    |       |       |   |→ pci_get_slot     （根据 bus-dev/fun 号从 bus-devices 表中搜索）
    |       |       |   |→ pci_scan_device  （根据 bus-dev/fun 号 进行真是硬件扫描）
    |       |       |   |→ pci_device_add   （将上面扫描的 dev 增加到 bus->devices 中）
    |       |       |→ pci_iov_bus_range    （忽略）
    |       |       |→ pcibios_fixup_bus    （根据 controller 资源，调整 devices 资源）
    |       |       |→ pci_scan_bridge      （对 bus0 上 bridge behind 后面 bus 进行枚举）
    |       |→ pci_bus_add_devices          （增加 dev 到 global list 和 sysfs、procfs 入口）
    |→ pci_bus_size_bridges
    |→ pci_bus_assign_resources
    |→ pci_enable_bridges
```

```
152 static int __init pcibios_init(void)
153 {
154     struct pci_controller *hose;
155
156     /* Scan all of the recorded PCI controllers.  */
157     for (hose = hose_head; hose; hose = hose->next)
158         pcibios_scanbus(hose);
159
160     pci_fixup_irqs(pci_common_swizzle, pcibios_map_irq);
161
162     pci_initialized = 1;
163
164     return 0;
165 }
```

每个 PCI controller 下面会挂接一个 PCI 子系统；通过多次调用 pcibios_scanbus 函数，来实现对每个控制器 behind 的 PCI 系统进行枚举。pci_fixup_irqs 利用两个回调函数实现对 INTA、INTB、INTC、INTD 的优化分配和对 device 的 irq PIN 和 LINE 的调整分配，具体的中断相关，会在 PCI 中断部分说明。

```
 79 static void __devinit pcibios_scanbus(struct pci_controller *hose)
 80 {
 81     static int next_busno;
 82     static int need_domain_info;
 83     struct pci_bus *bus;
 84
 85     if (!hose->iommu)
 86         PCI_DMA_BUS_IS_PHYS = 1;
 87
 88     if (hose->get_busno && pci_probe_only)
 89         next_busno = (*hose->get_busno)();
 90
 91     bus = pci_scan_bus(next_busno, hose->pci_ops, hose);  /* next_bus =0，hose 为 pci
controller */
 92     hose->bus = bus;
 93
 94     need_domain_info = need_domain_info || hose->index;
 95     hose->need_domain_info = need_domain_info;
 96     if (bus) {
```

```
 97        next_busno = bus->subordinate + 1;
 98        /* Don't allow 8-bit bus number overflow inside the hose -
 99           reserve some space for bridges. */
100        if (next_busno > 224) {
101            next_busno = 0;
102            need_domain_info = 1;
103        }
104
105        if (!pci_probe_only) {
106            pci_bus_size_bridges(bus);
107            pci_bus_assign_resources(bus);
108            pci_enable_bridges(bus);
109        }
110    }
111 }
```

控制器 host 利用 bus 元素将自己和 root bus 联系在一起 92 行。现在的 PCI 子系统中也引入了 domain 的概念，由于次系统中只有一个 pci 控制器股 domain 的值为 0。使用 pci_scan_bus 函数，来完成 root bus （总线 0）上所有设备的枚举，如果系统中存在 pci 总线，bus0 回被返回，否则返回 NULL。后面 106～108 行，主要是来完成总线、bridge 的 resource 的分配，具体关于 PCI resource 部分，会在 PCI resource 部分进行详细说明。下面继续 pci_scan_bus 函数内部分析。

```
643 static inline struct pci_bus * __devinit pci_scan_bus(int bus, struct pci_ops *ops,
644                     void *sysdata)
645 {
646    struct pci_bus *root_bus;
647    root_bus = pci_scan_bus_parented(NULL, bus, ops, sysdata);
648    if (root_bus)
649        pci_bus_add_devices(root_bus);
650    return root_bus;
651 }
```

pci_scan_bus_parented 函数用来继续完成 pci 枚举，如果枚举成功 root bus 会被返回，pci_bus_add_devices 根据枚举的情况，将系统中 bus->devices 中的设备增加到 global_list 链表中去，并增加 sysfs、procfs 文件入口。下面分析枚举相关的 pci_scan_bus_parented 函数。

```
1483 struct pci_bus * __devinit pci_scan_bus_parented(struct device *parent,
1484        int bus, struct pci_ops *ops, void *sysdata)
1485 {
1486    struct pci_bus *b;
1487
1488    b = pci_create_bus(parent, bus, ops, sysdata);
1489    if (b)
1490        b->subordinate = pci_scan_child_bus(b);
1491    return b;
1492 }
```

pci_create_bus 函数创建 bus 0，并将其增加到 pci_root_buses，并且将其归属到

pcibus_class 类中；此函数的逻辑较为简单此处就不进入分析了，此函数中 struce deivce dev 的用途理解的不是很明确。b->subordinate 表示 downstream 总线的最大总线号。下面调用 pci_scan_child_bus 函数枚举总线下的所有设备。

```
1364 unsigned int __devinit pci_scan_child_bus(struct pci_bus *bus)
1365 {
1366     unsigned int devfn, pass, max = bus->secondary;
1367     struct pci_dev *dev;
1368
1369     dev_dbg(&bus->dev, "scanning bus\n");
1370
1371     /* Go find them, Rover! */
         /* 每个 pci 总线最大可以支持 32 个设备，
          *  每个设备最多可支持 8 个逻辑设备
          *  0x100 即为 32×8 = 256 */
1372     for (devfn = 0; devfn < 0x100; devfn += 8)
1373         pci_scan_slot(bus, devfn);
1374
1375     /* Reserve buses for SR-IOV capability. */
1376     max += pci_iov_bus_range(bus);
1377
1378     /*
1379      * After performing arch-dependent fixup of the bus, look behind
1380      * all PCI-to-PCI bridges on this bus.
1381      */
        /* arch depend 相关，进行 pci resource 进行调整  */
1382     if (!bus->is_added) {
1383         dev_dbg(&bus->dev, "fixups for bus\n");
1384         pcibios_fixup_bus(bus);
1385         if (pci_is_root_bus(bus))
1386             bus->is_added = 1;
1387     }
1388
1389     for (pass=0; pass < 2; pass++)
1390         list_for_each_entry(dev, &bus->devices, bus_list) {
1391             if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
1392                 dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
1393                 max = pci_scan_bridge(bus, dev, max, pass);
1394         }
1395
1396     /*
1397      * We've scanned the bus and so we know all about what's on
1398      * the other side of any bridges that may be on this bus plus
1399      * any devices.
1400      *
1401      * Return how far we've got finding sub-buses.
1402      */
```

```
1403    dev_dbg(&bus->dev, "bus scan returning with max=%02x\n", max);
1404    return max;
1405 }
```

将 bus 0 上所有可能的 slot （32×8）进行统一进行扫描。这里有一个 pci spec 相关的知识需要说明：每个 pci 总线最大可支持 32 个设备，设备可分为单功能和多功能设备，单功能设备只能识别为一个设备而多功能设备每个最多可支持 8 个功能即可被识别为 8 个逻辑设备。我们可以将设备号和功能号组织成一个 9bit 的二进制（0-2 为 func 号、3-7 为 dev 号）。

```
1328 int pci_scan_slot(struct pci_bus *bus, int devfn)
1329 {
1330    unsigned fn, nr = 0;
1331    struct pci_dev *dev;
1332    unsigned (*next_fn)(struct pci_dev *, unsigned) = no_next_fn;
1333
1334    if (only_one_child(bus) && (devfn > 0))
1335       return 0; /* Already scanned the entire slot */
1336
       /* 枚举单功能设备  */
1337    dev = pci_scan_single_device(bus, devfn);
1338    if (!dev)
1339       return 0;
1340    if (!dev->is_added)
1341       nr++;
1342
       /* 根据不同的功能设置回调函数  */
1343    if (pci_ari_enabled(bus))
1344       next_fn = next_ari_fn;
1345    else if (dev->multifunction)
1346       next_fn = next_trad_fn;
1347
       /*  对每个设备的 8 个功能进行使用单功能枚举函数进行枚举
        *  如果枚举成功表示设备是多功能设备，multifunction 置 1 */
1348    for (fn = next_fn(dev, 0); fn > 0; fn = next_fn(dev, fn)) {
1349       dev = pci_scan_single_device(bus, devfn + fn);
1350       if (dev) {
1351          if (!dev->is_added)
1352             nr++;
1353          dev->multifunction = 1;
1354       }
1355    }
1356
1357    /* only one slot has pcie device */
1358    if (bus->self && nr)
1359       pcie_aspm_init_link_state(bus->self);
1360
1361    return nr;
```

```
1362 }
```

每个设备都会调用 pci_scan_single_device 来进行设备探测，如果是单功能设备至进行一次探测，如果是多功能设备，则对每一个功能（8）进行探测。下面对 pci_scan_single_device 函数进行分析。

```
1258 struct pci_dev *__ref pci_scan_single_device(struct pci_bus *bus, int devfn)
1259 {
1260     struct pci_dev *dev;
1261
        /* 根据 bus devfn 从 bus 的 devices 链表中寻找此设备  */
1262     dev = pci_get_slot(bus, devfn);
1263     if (dev) {
1264         pci_dev_put(dev);
1265         return dev;
1266     }
1267
        /* 通过读取 PCI 的具体设备来进行枚举  */
1268     dev = pci_scan_device(bus, devfn);
1269     if (!dev)
1270         return NULL;
1271
        /* 将探测到的具体设备增加到 bus 的 devices 链表中  */
1272     pci_device_add(dev, bus);
1273
1274     return dev;
1275 }
```

根据具体的 bus 号和 devfn 分别从 bus 的 devices 链表中和有 bus、devfn 三个元素具体决定的 pci 的 slot 中进行枚举；将从具体的 slot 中枚举到的设备通过 pci_devices_add 增加到 bus 的 devices 结构中去，此时的设备并没有增加到整个 pci 设备列表 global_list 中。

```
1155 static struct pci_dev *pci_scan_device(struct pci_bus *bus, int devfn)
1156 {
1157     struct pci_dev *dev;
1158     u32 l;
1159     int delay = 1;
1160
1161     if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
1162         return NULL;
1163
1164     /* some broken boards return 0 or ~0 if a slot is empty: */
1165     if (l == 0xffffffff || l == 0x00000000 ||
1166         l == 0x0000ffff || l == 0xffff0000)
1167         return NULL;
1168
1169     /* Configuration request Retry Status */
1170     while (l == 0xffff0001) {
```

```
1171        msleep(delay);
1172        delay *= 2;
1173        if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
1174            return NULL;
1175        /* Card hasn't responded in 60 seconds?  Must be stuck. */
1176        if (delay > 60 * 1000) {
1177            printk(KERN_WARNING "pci %04x:%02x:%02x.%d: not "
1178                    "responding\n", pci_domain_nr(bus),
1179                    bus->number, PCI_SLOT(devfn),
1180                    PCI_FUNC(devfn));
1181            return NULL;
1182        }
1183    }
1184
1185    dev = alloc_pci_dev();
1186    if (!dev)
1187        return NULL;
1188
1189    dev->bus = bus;
1190    dev->devfn = devfn;
1191    dev->vendor = l & 0xffff;
1192    dev->device = (l >> 16) & 0xffff;
1193
1194    if (pci_setup_device(dev)) {
1195        kfree(dev);
1196        return NULL;
1197    }
1198
1199    return dev;
1200 }
```

根据从某个具体的 slot 中读会的 config 空间的值来判断，是否存在设备；如果 slot 为空间，config 空间的 vendor_id 的值会为 0 或～0。如果设备存在，新分配一个 pci_dev 结构体，填充它的一些基本信息，并通过 pci_setup_device 完成它的必须信息的填充。

```
928 int pci_setup_device(struct pci_dev *dev)
929 {
930     u32 class;
931     u8 hdr_type;
932     struct pci_slot *slot;
933     int pos = 0;
934
935     if (pci_read_config_byte(dev, PCI_HEADER_TYPE, &hdr_type))
936         return -EIO;
937
938     dev->sysdata = dev->bus->sysdata;
939     dev->dev.parent = dev->bus->bridge;
```

```
        /* pci 的 driver 也是基于该类型的，在添加驱动的时候就可以
         * 匹配到此设备了 */
940     dev->dev.bus = &pci_bus_type;
941     dev->hdr_type = hdr_type & 0x7f;
        /* hdr_type 最高位为 0 表示是一个单功能设备 */
942     dev->multifunction = !!(hdr_type & 0x80);
943     dev->error_state = pci_channel_io_normal;
944     set_pcie_port_type(dev);
945
946     list_for_each_entry(slot, &dev->bus->slots, list)
947         if (PCI_SLOT(dev->devfn) == slot->number)
948             dev->slot = slot;
949
950     /* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
951        set this higher, assuming the system even supports it.  */
952     dev->dma_mask = 0xffffffff;
953
954     dev_set_name(&dev->dev, "%04x:%02x:%02x.%d", pci_domain_nr(dev->bus),
955             dev->bus->number, PCI_SLOT(dev->devfn),
956             PCI_FUNC(dev->devfn));
957
958     pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
959     dev->revision = class & 0xff;
960     class >>= 8;                    /* upper 3 bytes */
961     dev->class = class;
962     class >>= 8;
963
964     dev_dbg(&dev->dev, "found [%04x:%04x] class %06x header type %02x\n",
965         dev->vendor, dev->device, class, dev->hdr_type);
966
967     /* need to have dev->class ready */
        /* 设置 pci 的配置空间大小  pci 256byte, pcie 4096byte */
968     dev->cfg_size = pci_cfg_space_size(dev);
969
970     /* "Unknown power state" */
971     dev->current_state = PCI_UNKNOWN;
972
973     /* Early fixups, before probing the BARs */
        /* 执行一些 arch depend 相关的 fixup */
974     pci_fixup_device(pci_fixup_early, dev);
975     /* device class may be changed after fixup */
976     class = dev->class >> 8;
```

```
977
        /* 根据 hdr_type 的三种情况 pci dev、pci bridge、cardbus
         * 分别进行一些类型相关的配置 */
978     switch (dev->hdr_type) {          /* header type */
979     case PCI_HEADER_TYPE_NORMAL:          /* standard header */
980         if (class == PCI_CLASS_BRIDGE_PCI)
981             goto bad;
982         pci_read_irq(dev);
983         pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
984         pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID, &dev-
>subsystem_vendor);
985         pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);
986
987         /*
988          *  Do the ugly legacy mode stuff here rather than broken chip
989          *  quirk code. Legacy mode ATA controllers have fixed
990          *  addresses. These are not always echoed in BAR0-3, and
991          *  BAR0-3 in a few cases contain junk!
992          */
993         if (class == PCI_CLASS_STORAGE_IDE) {
994             u8 progif;
995             pci_read_config_byte(dev, PCI_CLASS_PROG, &progif);
996             if ((progif & 1) == 0) {
997                 dev->resource[0].start = 0x1F0;
998                 dev->resource[0].end = 0x1F7;
999                 dev->resource[0].flags = LEGACY_IO_RESOURCE;
1000                dev->resource[1].start = 0x3F6;
1001                dev->resource[1].end = 0x3F6;
1002                dev->resource[1].flags = LEGACY_IO_RESOURCE;
1003            }
1004            if ((progif & 4) == 0) {
1005                dev->resource[2].start = 0x170;
1006                dev->resource[2].end = 0x177;
1007                dev->resource[2].flags = LEGACY_IO_RESOURCE;
1008                dev->resource[3].start = 0x376;
1009                dev->resource[3].end = 0x376;
1010                dev->resource[3].flags = LEGACY_IO_RESOURCE;
1011            }
1012        }
1013        break;
1014
1015    case PCI_HEADER_TYPE_BRIDGE:          /* bridge header */
1016        if (class != PCI_CLASS_BRIDGE_PCI)
1017            goto bad;
```

```
1018        /* The PCI-to-PCI bridge spec requires that subtractive
1019           decoding (i.e. transparent) bridge must have programming
1020           interface code of 0x01. */
1021        pci_read_irq(dev);
1022        dev->transparent = ((dev->class & 0xff) == 1);
1023        pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
1024        set_pcie_hotplug_bridge(dev);
1025        pos = pci_find_capability(dev, PCI_CAP_ID_SSVID);
1026        if (pos) {
1027           pci_read_config_word(dev, pos + PCI_SSVID_VENDOR_ID, &dev-
>subsystem_vendor);
1028           pci_read_config_word(dev, pos + PCI_SSVID_DEVICE_ID, &dev-
>subsystem_device);
1029        }
1030        break;
1031
1032    case PCI_HEADER_TYPE_CARDBUS:          /* CardBus bridge header */
1033        if (class != PCI_CLASS_BRIDGE_CARDBUS)
1034           goto bad;
1035        pci_read_irq(dev);
1036        pci_read_bases(dev, 1, 0);
1037        pci_read_config_word(dev, PCI_CB_SUBSYSTEM_VENDOR_ID, &dev-
>subsystem_vendor);
1038        pci_read_config_word(dev, PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
1039        break;
1040
1041    default:                   /* unknown header */
1042        dev_err(&dev->dev, "unknown header type %02x, "
1043           "ignoring device\n", dev->hdr_type);
1044        return -EIO;
1045
1046    bad:
1047        dev_err(&dev->dev, "ignoring class %02x (doesn't match header "
1048           "type %02x)\n", class, dev->hdr_type);
1049        dev->class = PCI_CLASS_NOT_DEFINED;
1050    }
1051
1052    /* We found a fine healthy device, go go go... */
1053    return 0;
1054 }
```

主要完成设备的一些具体的配置。dev->dev.bus = &pci_bus_type; 非常重要，pci 设备驱动和设备的连接和它存在莫大的关系。由于这个世界本身的不完美，有一些 pci 设备势必会存在默写 bug 需要软件进行修复，为了解决一些 ugly pci device 可以正常工作，pci_fixup 出现了；它根据 vendor 和 device 或可以唯一识别某个 pci device 的 id 来适配设备，使用 hook 函数来进行 bug fixup；一共有 7 中级别的 pci_fixup（pci_fixup_early、pci_fixup_header、pci_fixup_final、pci_fixup_enable、pci_fixup_resume、pci_fixup_suspend、pci_fixup_resume_early）存在，每一种级别代表一种级别的 fixup

运行的时机。

pci_read_irq 函数完成 pci 的 IRQ 的读取。在 PCI_INTERRUPT_PIN 中存放的是 INTA~INTD 的哪一个引脚连接到了中断控制器，如果该值为零，说明并未将引脚连接至中断控制器，自然也就不能产生中断信号。 其实，在 PCI_INTERRUPT_LINE 存放的是该设备的中断线连接在中断控制器的哪一个 IRQ 线上，也就是对应设备的 IRQ。注意，这里的寄存器只读有意义，并不是更改寄存器的值就更改该设备的 IRQ。

pci_read_base 实现了 pci 内部存储区间的确定。pci 设备配置寄存器图中可以看到 6 个 base address 寄存器，里面存放的就是内部存储器的地起地址和长度、及其类型，还有一个 ROM address 寄存器。

首先，将对应寄存器的值取出，如果最低位为 1，则说明该区域是 I/O 端口，高 29 位是端口地址的高 29 位，低 3 位为零。否则，低位为 1，是存储映射区间，前 28 位是存储区的高 28 位，低四位为零。

然后，将该寄存器全部置 1，再读取得到长度信息，如果是 I/O 端口，屏蔽低三位；如果是存储区间，屏蔽低四位。最后取从 0 开始的第 1 个位为 1 的值，对应便是空间的大小，即为相应区间的长度。 例如，取出来的值是 0xC107，假设是 I/O 端口 屏蔽低三位，为 0xC100。第一个为 1 的值对应的值为 0x0100，即 0x100 。

另外，ROM 的操作和其类似.

```
1227 void pci_device_add(struct pci_dev *dev, struct pci_bus *bus)
1228 {
1229     device_initialize(&dev->dev);
1230     dev->dev.release = pci_release_dev;
1231     pci_dev_get(dev);
1232
1233     dev->dev.dma_mask = &dev->dma_mask;
1234     dev->dev.dma_parms = &dev->dma_parms;
1235     dev->dev.coherent_dma_mask = 0xffffffffull;
1236
1237     pci_set_dma_max_seg_size(dev, 65536);
1238     pci_set_dma_seg_boundary(dev, 0xffffffff);
1239
        /* 此处有一次调用了 pci_fixup，它是和 arch depend 的  */
1240     /* Fix up broken headers */
1241     pci_fixup_device(pci_fixup_header, dev);
1242
1243     /* Clear the state_saved flag. */
1244     dev->state_saved = false;
1245
1246     /* Initialize various capabilities */
1247     pci_init_capabilities(dev);
1248
1249     /*
1250      * Add the device to our list of discovered devices
1251      * and the bus list for fixup functions, etc.
1252      */
```

```
        /* 此函数的重点，将发现的设备增加到 bus 的 devices 列表  */
1253     down_write(&pci_bus_sem);
1254     list_add_tail(&dev->bus_list, &bus->devices);
1255     up_write(&pci_bus_sem);
1256 }

266 void __devinit pcibios_fixup_bus(struct pci_bus *bus)
267 {
268     /* Propagate hose info into the subordinate devices.  */
269
270     struct pci_controller *hose = bus->sysdata;
271     struct list_head *ln;
272     struct pci_dev *dev = bus->self;
273
274     if (!dev) {
275         bus->resource[0] = hose->io_resource;
276         bus->resource[1] = hose->mem_resource;
277     } else if (pci_probe_only &&
278            (dev->class >> 8) == PCI_CLASS_BRIDGE_PCI) {
279         pci_read_bridge_bases(bus);
280         pcibios_fixup_device_resources(dev, bus);
281     }
282
283     for (ln = bus->devices.next; ln != &bus->devices; ln = ln->next) {
284         dev = pci_dev_b(ln);
285
286         if ((dev->class >> 8) != PCI_CLASS_BRIDGE_PCI)
287             pcibios_fixup_device_resources(dev, bus);
288     }
289 }
290
```

pcibios_fixup_bus 函数完成对非 bus 0 的 pci bridge 的资源的调整。

Pci bridge 有 2 个（0x10 开始）存储区间和 1 个（0x38 开始）rom 区间；它的操作和前面介绍的 pci dev 的 6 个 address 和 1 个 rom address 的操作类似，但是 pci bridge 还有其自己特殊的特性：地址过滤功能（过滤窗口）。

过滤窗口决定了访问的方向。例如：如果 cpu 一侧要经过 pci bridge 访问 pci 总线，则它的地址必须要落在这个 pci 桥的过滤窗口内才可以通过。另外，pci bridge 下游的 pci bus 要访问 cpu 侧，则地址必须要落在过滤窗口外才可以。

此外，pci bridge 还提供了一个命令寄存器来控制“memory access enable“和“I/O access enable”两个位来控制两个功能。如果全为 0，则两个方向都会关闭。在 pci 初始化前，为了防止对 cpu 侧造成干扰, 这两个功能都关闭的。

Pci bridge 有三个这样的窗口，分别如下：

1：起始地址在 PCI_IO_BASE 中，长度在 PCI_IO_LIMIT 中。如果是 32 位，还要通过 PCI_IO_BASE_UPPER16 和 PCI_IO_LIMIT_UPPER16 提供高 16 位。

2：起始地址在 PCI_MEMORY_BASE，长度在 PCI_MEMORY_LIMIT 中。

这个是一个 16 位的窗口。

3：起始地址在 PCI_PREF_MEMORY_BASE，长度在 PCI_PREF_MEMORY_LIMIT。默认是 32 位。如果是 64，则需要 PCI_PREF_BASE_UPPER32 和 PCI_PREF_LIMIT_UPPER32 提供高 32 位。

到此为止，总线 0 上面的所有的 devices 都被枚举了一遍，每一类设备的信息都已经完全读取出来了，并存放在 pci_dev 的相关字段。此后在驱动中就可以直接找到 pci_dev.取得相应的信息，而不需要再次去枚举了。

接下来判断 bus 0 的 devices 中为 bridge 的 devices 下面的 bus 上的设备的枚举。我们知道，pci 总线可以通过 pci bridge 再连一层 pci 总线。这个问题显然是一个递归过程。我们接下来看 pci 桥的处理。

```
1364 unsigned int __devinit pci_scan_child_bus(struct pci_bus *bus)
1365 {
        ……
        ……
                /* pass = 0 扫描已经被 BIOS 配置过的 bus，
                 *  pass = 1 扫描新的 bus，并分配未被分配的 bus 号  */
        1389    for (pass=0; pass < 2; pass++)
        1390        list_for_each_entry(dev, &bus->devices, bus_list) {
        1391            if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
        1392                dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
        1393                max = pci_scan_bridge(bus, dev, max, pass);
        1394        }

        ……
        ……
1405 }
```

上面的操作基本上就是遍历挂在 pci_bus->devices 上面的设备(是否还记得上面在分析的时候，每枚举到一个设备都会加入到 pci_bus->device 呢)。如果是 pci 桥或 cardbus。就会调用 pci_scan_bridge()来遍历桥下面的设备。这里让人疑惑的是，为什么要遍历二次呢？

这是因为，在 x86 上，系统启动的时候，bios 会枚举一次 pci 设备。所以有些 pci bridge 是经过 bios 处理过的。而有些可能是 bios 没有枚举的。这就需要分两次处理。一次来处理那里已经由 bios 处理过的 pci bridge。一次是处理全新的 pci bridge。这样做是因为每次枚举总线后，要为其分配一个总线号，而 bios 处理后的 pci bridge 的总线号已经全部由 bios 分配好了，而要为新的 pci bridge 分配总线号，就必须要处理完旧的（bios 处理过的）pci bridge 才会知道那些 bus 号是可用的。

如果 bus 0 上扫描到的设备是 bridge，则配置该 bridge 并进入 bus0 的下级总线进行扫描；如果扫描到的设备是 cardbus，则配置它，但是不扫描下级 devices，cardbus bridge 驱动会对其后的设备进行处理。进入 pci_sacn_bridge()的代码进行分析。

```
679 int __devinit pci_scan_bridge(struct pci_bus *bus, struct pci_dev *dev, int max, int pass)
680 {
```

```
681    struct pci_bus *child;
682    int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);
683    u32 buses, i, j = 0; 684     u16 bctl;
685    u8 primary, secondary, subordinate;
686    int broken = 0;
687
        /* 从 config 空间提取表示总线号码的变量
         * primary：pci 上行总线号
         * secondary：pci 下行总线号
         * subordinate：总线可以访问的最大总线编号*/

688    pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
689    primary = buses & 0xFF;
690    secondary = (buses >> 8) & 0xFF;
691    subordinate = (buses >> 16) & 0xFF;
692
693    dev_dbg(&dev->dev, "scanning [bus %02x-%02x] behind bridge, pass %d\n",
694        secondary, subordinate, pass);
695
696    /* Check if setup is sensible at all */
697    if (!pass &&
698        (primary != bus->number || secondary <= bus->number)) {
699        dev_dbg(&dev->dev, "bus configuration invalid, reconfiguring\n");
700        broken = 1;
701    }
702
703    /* Disable MasterAbortMode during probing to avoid reporting
704       of bus errors (in some architectures) */
705    pci_read_config_word(dev, PCI_BRIDGE_CONTROL, &bctl);
706    pci_write_config_word(dev, PCI_BRIDGE_CONTROL,
707                bctl & ~PCI_BRIDGE_CTL_MASTER_ABORT);
708
        /* BIOS 配置过的分支，如果从 PCI_PRIMARY_BUS 中取出的
         * secondary、subordinate 有意义，说明该桥已经被配置，即被 BIOS 配置。因为
         * secondary 是在扫描到一个 bridge 加 1 的。
         * 由于 primary 有可能为 0（根总线的）所以此处不能用它来判断。
         */
709    if ((secondary || subordinate) && !pcibios_assign_all_busses() &&
710        !is_cardbus && !broken) {
711        unsigned int cmax;
712        /*
713         * Bus already configured by firmware, process it in the first
714         * pass and just note the configuration.
715         */
716        if (pass)
717            goto out;
```

```
718
        /* 如果该 bridge 是被 bios 处理过，那直接构造一个 pci_bus（pci_add_new_bus()），
         * 再递归枚举这个 pci_bus 下的设备就可以了。 */
719     /*
720      * If we already got to this bus through a different bridge,
721      * don't re-add it. This can happen with the i450NX chipset.
722      *
723      * However, we continue to descend down the hierarchy and
724      * scan remaining child buses.
725      */
726     child = pci_find_bus(pci_domain_nr(bus), secondary);
727     if (!child) {
728        child = pci_add_new_bus(bus, dev, secondary);
729        if (!child)
730           goto out;
731        child->primary = primary;
732        child->subordinate = subordinate;
733        child->bridge_ctl = bctl;
734     }
735
736     cmax = pci_scan_child_bus(child);
737     if (cmax > max)
738        max = cmax;
739     if (child->subordinate > max)
740        max = child->subordinate;
       /* 未被 BIOS 处理分支 */
741  } else {
       /* 第一次扫描是不会处理新的 bridge，只处理 bios 处理过的 bridge */
742     /*
743      * We need to assign a number to this bus which we always
744      * do in the second pass.
745      */
746     if (!pass) {
747        if (pcibios_assign_all_busses() || broken)
748           /* Temporarily disable forwarding of the
749              configuration cycles on all bridges in
750              this bus segment to avoid possible
751              conflicts in the second pass between two
752              bridges programmed with overlapping
753              bus ranges. */
754           pci_write_config_dword(dev, PCI_PRIMARY_BUS,
755                       buses & ~0xffffff);
756        goto out;
757     }
758
759     /* Clear errors */
760     pci_write_config_word(dev, PCI_STATUS, 0xffff);
```

```
761
        /* 阻止分配一个已经存在的 bus 号  */
762       /* Prevent assigning a bus number that already exists.
763        * This can happen when a bridge is hot-plugged */
764       if (pci_find_bus(pci_domain_nr(bus), max+1))
765           goto out;
        /* 创建一个新的 pci_bus */
766       child = pci_add_new_bus(bus, dev, ++max);
767       buses = (buses & 0xff000000)
768             | ((unsigned int)(child->primary)     <<  0)
769             | ((unsigned int)(child->secondary)   <<  8)
770             | ((unsigned int)(child->subordinate) << 16);
771
772       /*
773        * yenta.c forces a secondary latency timer of 176.
774        * Copy that behaviour here.
775        */
776       if (is_cardbus) {
777           buses &= ~0xff000000;
778           buses |= CARDBUS_LATENCY_TIMER << 24;
779       }
780
        /* 将三个总线号相关的值，写入 bridge 对应的 3 个寄存器中  */
781       /*
782        * We need to blast all three values with a single write.
783        */
784       pci_write_config_dword(dev, PCI_PRIMARY_BUS, buses);
785
        /* 更新 pci 的 subordinate 值，默认该值初始化为 0xff，当 dridge 的 behind 的 child bus
         * 被完全枚举了，subordinate 值也会被得到，此时更新此值从 0xff 到实际的最大总线
         * 号。
         * 此处会调用 2 次 pci_fixup_parent_subordinate_busnr 来更新 subordinate 的值。是因为
         * bios 有可能没有枚举完全所有设备。系统将 bios 认为的 subordinate 更新，并重新枚
         * 所有的设备得到真正的 subordinate 的值，并从 0xff 进行更新。
         */
786       if (!is_cardbus) {
787           child->bridge_ctl = bctl;
788           /*
789            * Adjust subordinate busnr in parent buses.
790            * We do this before scanning for children because
791            * some devices may not be detected if the bios
792            * was lazy.
793            */
794           pci_fixup_parent_subordinate_busnr(child, max);
795           /* Now we can scan all subordinate buses... */
796           max = pci_scan_child_bus(child);
```

```
797         /*
798          * now fix it up again since we have found
799          * the real value of max.
800          */
801         pci_fixup_parent_subordinate_busnr(child, max);
      /* 此段代码针对 cardbus 不进行详细分析，其作用也是更新 subordinate 值 */
802       } else {
803         /*
804          * For CardBus bridges, we leave 4 bus numbers
805          * as cards with a PCI-to-PCI bridge can be
806          * inserted later.
807          */
808         for (i=0; i<CARDBUS_RESERVE_BUSNR; i++) {
809           struct pci_bus *parent = bus;
810           if (pci_find_bus(pci_domain_nr(bus),
811                   max+i+1))
812             break;
813           while (parent->parent) {
814             if ((!pcibios_assign_all_busses()) &&
815               (parent->subordinate > max) &&
816               (parent->subordinate <= max+i)) {
817               j = 1;
818             }
819             parent = parent->parent;
820           }
821           if (j) {
822             /*
823              * Often, there are two cardbus bridges
824              * -- try to leave one valid bus number
825              * for each one.
826              */
827             i /= 2;
828             break;
829           }
830         }
831         max += i;
832         pci_fixup_parent_subordinate_busnr(child, max);
833       }
834       /*
835        * Set the subordinate bus number to its real value.
836        */
837       child->subordinate = max;
838       pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, max);
839     }
840
841     sprintf(child->name,
842       (is_cardbus ? "PCI CardBus %04x:%02x" : "PCI Bus %04x:%02x"),
```

```
843         pci_domain_nr(bus), child->number);
844
          /* debug 信息的打印*/
845       /* Has only triggered on CardBus, fixup is in yenta_socket */
846       while (bus->parent) {
847          if ((child->subordinate > bus->subordinate) ||
848             (child->number > bus->subordinate) ||
849             (child->number < bus->number) ||
850             (child->subordinate < bus->number)) {
851             dev_info(&child->dev, "[bus %02x-%02x] %s "
852                "hidden behind%s bridge %s [bus %02x-%02x]\n",
853                child->number, child->subordinate,
854                (bus->number > child->subordinate &&
855                 bus->subordinate < child->number) ?
856                   "wholly" : "partially",
857                bus->self->transparent ? " transparent" : "",
858                dev_name(&bus->dev),
859                bus->number, bus->subordinate);
860          }
861          bus = bus->parent;
862       }
863
864 out:
865       pci_write_config_word(dev, PCI_BRIDGE_CONTROL, bctl);
866
867       return max;
868 }
```

PCI_PRIMARY_BUS 寄存器中的值的含义为：从低到高位分别为：主总线号，次总线号，子层最大线号、各占两位。如果从 PCI_PRIMARY_BUS 取出来的值，次总线号（每枚举到一个 pci-bridge 就会增 1）和子层最大线号有意义，说明该 pci-bridge 是被 bios 处理过的。由于总主线号可能为零（如根总线）。

如果该 pci bridge 被 bios 处理过的，那直接构造一个 pci_bus（pci_add_new_bus()）。再递归枚举这个 pci_bus 下的设备就可以了。

相反，如果该 pci-bridge 没有被 bios 处理过，就需要我们手动去处理了。这时，为它分配一个可能的总线号，然后将总线号写入 PCI_PRIMARY_BUS 寄存器，再构造一个 pci_bus 递归枚举其下的设备。

特别注意，在遍其次级总线下层 pci 时，此时还不知道下层最大总号是多少。所以将 pci_bus->subordinate 赋值为 0xFF，即其下的所有设备都可以透过这个 pci-bridge(具体参考 pci_alloc_child_bus()中的处理)。然后，等下层的子总线遍历完了之后，再来确定子总线的最大总线号，将其更新至 pci_bus->subordinate 替换 0xff。

递归完成之后，pci 总线上的所有信息都被找到了。所有 pci_bus 被存放在 pci_root_buses 为根的倒立树中。总线上对应的 pci_dev 存放在 pci_bus->device 链表中。

三个总线号相关的值说明：

Primary Bus Number(主总线号)

该 PCI-PCI 桥所处的 PCI 总线称为主总线。

Secondary Bus Number(子总线号)

该 PCI-PCI 桥所连接的 PCI 总线称为子总线/次总线号。

Subordinate Bus Number

PCI 总线的下属 PCI 总线的总线编号最大值。

主总线号在访问下层总线是不会被使用到的，因为在配置的时候,只会比较 pci_bridge 的子总线号和下层最大总线号。如果总线号落在这个区间中，将其透传到下一层总线。否则，忽略这个请求。

到此，整个枚举过程代码分析完成。下面回到 pci_scan_bus 函数继续分析。

```
643 static inline struct pci_bus * __devinit pci_scan_bus(int bus, struct pci_ops *ops,
644                     void *sysdata)
645 {
646     struct pci_bus *root_bus;
647     root_bus = pci_scan_bus_parented(NULL, bus, ops, sysdata);
648     if (root_bus)
649         pci_bus_add_devices(root_bus);
650     return root_bus;
651 }
```

如果 pci 的根总线存在，即 root bus 存在。通过 pci_bus_add_devices 将所有的 pci devices 增加到系统的 pci dev 设备列表中 global_list。

```
190 void pci_bus_add_devices(const struct pci_bus *bus)
191 {
192     struct pci_dev *dev;
193     struct pci_bus *child;
194     int retval;
195
        /* 增加所有的设备到 pci_device 链表中，并创建 sysfs 和 procfs 文件入口*/
196     list_for_each_entry(dev, &bus->devices, bus_list) {
197         /* Skip already-added devices */
198         if (dev->is_added)
199             continue;
200         retval = pci_bus_add_device(dev);
201         if (retval)
202             dev_err(&dev->dev, "Error adding device, continuing\n");
203     }
204
        /* 如果有没有连接的 subordinate bus，连接它到 bus 的 children
         * 并枚举它下面的所有设备 */
205     list_for_each_entry(dev, &bus->devices, bus_list) {
206         BUG_ON(!dev->is_added);
207
208         child = dev->subordinate;
209         /*
210          * If there is an unattached subordinate bus, attach
```

```
211         * it and then scan for unattached PCI devices.
212         */
213        if (!child)
214            continue;
215        if (list_empty(&child->node)) {
216            down_write(&pci_bus_sem);
217            list_add_tail(&child->node, &dev->bus->children);
218            up_write(&pci_bus_sem);
219        }
220        pci_bus_add_devices(child);
221
222        /*
223         * register the bus with sysfs as the parent is now
224         * properly registered.
225         */
226        if (child->is_added)
227            continue;
228        retval = pci_bus_add_child(child);
229        if (retval)
230            dev_err(&dev->dev, "Error adding bus, continuing\n");
231    }
232 }
```

继续回到 pcibios_scanbus 向下分析。

```
 79 static void __devinit pcibios_scanbus(struct pci_controller *hose)
 80 {
 81     static int next_busno;
 82     static int need_domain_info;
 83     struct pci_bus *bus;
 84
 85     if (!hose->iommu)
 86         PCI_DMA_BUS_IS_PHYS = 1;
 87
 88     if (hose->get_busno && pci_probe_only)
 89         next_busno = (*hose->get_busno)();
 90
 91     bus = pci_scan_bus(next_busno, hose->pci_ops, hose);
        /* 将根总线的 bus 和 pci controller 通过 bus 连接起来 */
 92     hose->bus = bus;
 93
        /* 完成一些 pci domain 相关的信息设置，pci 结构中设备
          * 的定位采用如下结构：domain：pci：dev：func（eg：0000:00:00:0）*/
 94     need_domain_info = need_domain_info || hose->index;
 95     hose->need_domain_info = need_domain_info;
 96     if (bus) {
```

```
        /* 定义下一个域的 root bus 的 bus 号，为上一个 domain 的 bus 的 subordinate + 1 */
 97         next_busno = bus->subordinate + 1;
 98         /* Don't allow 8-bit bus number overflow inside the hose -
 99            reserve some space for bridges. */
100         if (next_busno > 224) {
101             next_busno = 0;
102             need_domain_info = 1;
103         }
104
        /* pci 的设备资源的调整和分配  */
105         if (!pci_probe_only) {
106             pci_bus_size_bridges(bus);
107             pci_bus_assign_resources(bus);
108             pci_enable_bridges(bus);
109         }
110     }
111 }
```

继续向上回到 pcibios_init 函数。

```
152 static int __init pcibios_init(void)
153 {
154     struct pci_controller *hose;
155
156     /* Scan all of the recorded PCI controllers.  */
157     for (hose = hose_head; hose; hose = hose->next)
158         pcibios_scanbus(hose);
159
        /* 完成体系相关的 pci 的中断的处理  */
160     pci_fixup_irqs(pci_common_swizzle, pcibios_map_irq);
161
162     pci_initialized = 1;
163
164     return 0;
165 }
```