# 第Ⅲ篇　Linux 与 PCI 总线

本篇主要讲述 Linux 系统与 PCI/PCIe 总线相关的一些内容，其重点在于 Linux 系统 PCI/PCIe 总线驱动程序的实现。并以此为基础说明 PCI 总线控制器及其相关设备在系统软件的初始化过程。本篇并不会拘泥于 Linux 系统的实现细节，但是仍将介绍一些与 Linux 系统相关的基本知识。本篇内容基于 Linux 2.6.31.6 内核。

值得注意的是，在不同处理器体系结构中，Linux 系统初始化 PCI 总线的过程并不相同。如在 Linux x86 系统中，BIOS 为 PCI 总线的初始化做出了许多辅助工作，而在 Linux PowerPC 或者 Linux ARM 中使用的 Firmware，如 U-Boot，并没有做类似的工作。

从系统软件的角度来看，PCI 总线与 PCIe 总线的初始化过程和资源分配较为类似，为节约篇幅，本篇将 PCI 和 PCIe 总线统称为 PCI 总线，并将 Linux 系统的 PCI 和 PCIe 子系统简称为 Linux PCI。

在第 12.3 节中讲述了一个最基本的、基于 PCI 总线的 Linux 设备驱动程序。这个 PCI 设备驱动程序使用了一些 Linux 系统提供的标准 API 和数据结构，例如使用 pci_resource_start 和 pci_resource_len 函数获得该设备 BAR 空间的基地址和长度，并在 request_irq 函数中使用 pci_dev→irq 参数注册该设备使用的中断服务例程。

该 PCI 设备（Capric 卡）在驱动程序中使用的这些存储器资源，由系统软件对 PCI 总线进行初始化时确定，而中断资源在使能相应的 PCI 设备时由系统软件分配。这个系统软件包括操作系统和 Firmware[⊖]。

与其他处理器系统相比，x86 处理器作为一个通用处理器平台，始终强调向前兼容的重要性。而实现向前兼容需要做出许多牺牲，这也造成了 Linux x86 对 PCI 总线的初始化过程最为复杂也最为繁琐，x86 处理器在引入了 ACPI（Advanced Configuration and Power Interface Specification）机制之后，方便了处理器系统对“不规范外部设备”的管理，但是使得 PCI 总线的初始化过程更为复杂。

下文将以 Linux x86 为主线说明 PCI 总线的初始化过程。Linux x86 在对 PCI 总线进行初始化之前，BIOS 对 PCI 总线做出了部分初始化工作，如创建 ACPI 表、预先分配 PCI 设备使用的存储器资源，并执行 PCI 设备 ROM 中的初始化代码等一系列步骤。

Linux x86 将继承 BIOS 对 PCI 总线的初始化成果，并在此基础上进行 Linux PCI 子系统

⊖ x86 处理器使用的 BIOS 也是 Firmware 的一种。

的初始化，并执行 PCI 设备的 Linux 驱动程序的初始化模块。在 Linux x86 中，PCI 总线的初始化由一系列模块协调完成。

Linux x86 首先使用“make menuconfig”命令对内核进行必要的配置，然后产生 . config 文件。假定在 . config 文件中，CONFIG_PCI、CONFIG_PCI_MSI、CONFIG_PCI_GOANY、CONFIG_PCI_BIOS、CONFIG_PCI_DIRECT、CONFIG_PCI_MMCONFIG 等一些必要的参数为“y”，即使能 PCI 总线驱动、使能 MSI 中断请求机制等，而且对 x86 处理器非常重要的 CONFIG_ACPI 参数也为“y”。

在 Linux PCI 中，有两个常用的数据结构，分别为 pci_dev 和 pci_bus 结构。这两个数据结构的定义在 . /include/linux/pci. h 文件中。其中 pci_dev 结构描述 PCI 设备，包括这个 PCI 设备的配置寄存器信息，使用的中断资源，还有一些和 SR-IOV 相关的参数。而 pci_bus 结构描述 PCI 桥，包括这个 PCI 桥的配置寄存器信息和一些状态信息。该结构中 self 参数值得注意，pci_bus→self 指向一个 pci_dev 结构，该结构用于 PCI 桥的上游总线访问 PCI 桥，此时 PCI 桥被当作一个设备。

# 第 14 章　Linux PCI 的初始化过程

Linux PCI 初始化的主要工作是遍历当前处理器系统中的所有 PCI 总线树，并初始化 PCI 总线树上的全部设备，包括 PCI 桥与 PCI Agent 设备。在 Linux 系统中，多次使用了 DFS 算法对 PCI 总线树进行遍历查找，并分配相关的 PCI 总线号与 PCI 总线地址资源。

单纯从一种处理器系统的角度来看，Linux PCI 的实现机制远非完美。其中有许多冗余的代码和多余的步骤，比如 Linux PCI 中对 PCI 总线树的遍历次数过多，从而影响 Linux PCI 的初始化代码的执行效率。产生这些不完美的主要原因是，Linux PCI 首先以 x86 处理器为蓝本编写，而后作为通用代码逐渐支持其他处理器，如 ARM、PowerPC 和 MIPS 等。不同的处理器对 PCI 总线树的遍历机制并不完全相同，而 Linux PCI 作为通用代码必须兼顾这些不同，从而在某种程度上造成了这段代码的混乱。这种混乱是通用代码的无奈之举。

本章以 x86 处理器系统为例，介绍 Linux PCI 的执行流程。目前 ACPI 机制在 x86 处理器系统已经得到大规模的普及，而且在 x86 处理器中，只能使用 ACPI 机制支持处理器最新的特性。因此掌握 ACPI 机制，对于深入理解 x86 处理器的软件架构，已经不可或缺。为此本章将重点介绍 Linux PCI 在 ACPI 机制下的初始化过程，而不再介绍 Linux PCI 的传统初始化方式。

## 14.1　Linux x86 对 PCI 总线的初始化

一个处理器系统首先从 Firmware 开始执行，并由 Firmware 开始引导 Linux 内核。Linux 系统首先从 ./init/main.c 文件的 start_kernel 函数开始执行。不同的处理器系统使用的 Firmware 并不相同，如 x86 处理器系统使用 BIOS，而 PowerPC 处理器系统使用 U-Boot。有些处理器系统，最初的初始化操作可能由 $E^2$PROM 完成，之后执行 Firmware 中的程序。值得注意的是，在 x86 处理器中常用的 Grub 并不是 Firmware，而是 Linux 系统的引导程序。

start_kernel 函数在调用 rest_init 函数之前，其主要工作与操作系统核心层相关，包括进程调度、内存管理和中断系统等主要模块的初始化。而 rest_init 函数将创建 kernel_init 进程，并由该进程调用 do_basic_setup→do_initcalls 函数[⊖]完成所有外部设备的初始化，包括 PCI 总线的初始化，该函数如源代码 14-1 所示。

**源代码 14-1　do_initcalls 函数**

```
static void _init do_initcalls(void)
{
    initcall_t *call;

    for (call = _early_initcall_end; call < _initcall_end; call++)
        do_one_initcall(*call);
```

⊖ Linux PowerPC 在 setup_arch 函数中也做了一些有关 PCI 总线的初始化工作，如初始化 HOST 主桥。

```
    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_work();
}
```

do_initcalls 函数的主体是将_early_initcall_end 和_initcall_end 指针之间的函数全部执行一遍，这两个指针在 vmlinux.lds 文件中定义。在生成操作系统内核时，一些需要在 Linux 系统初始化时执行的函数指针被加入到_early_initcall_end 和_initcall_end 参数之间，之后由 do_initcalls 函数统一调用这些函数。Linux 系统定义了一系列需要在系统初始化时执行的模块，如源代码 14-2 所示。这段代码在 ./include/linux/init.h 文件中。

**源代码 14-2　Linux 系统的初始化模块**

```
/**
 * module_init() - driver initialization entry point
 * @x: function to be run at kernel boot time or module insertion
 *
 * module_init() will either be called during do_initcalls() (if
 * builtin) or at module insertion time (if a module). There can only
 * be one per module.
 */
#define module_init(x) __initcall(x);
...
#define __define_initcall(level,fn,id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" level ".init"))) = fn

/*
 * Early initcalls run before initializing SMP.
 *
 * Only for built-in code, not modules.
 */
#define early_initcall(fn)              __define_initcall("early",fn,early)

/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)               __define_initcall("0",fn,0)

#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
```

```
#define arch_initcall_sync(fn)              _define_initcall("3s",fn,3s)
#define subsys_initcall(fn)                 _define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)            _define_initcall("4s",fn,4s)
#define fs_initcall(fn)                     _define_initcall("5",fn,5)
#define fs_initcall_sync(fn)                _define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)                 _define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)                 _define_initcall("6",fn,6)
#define device_initcall_sync(fn)            _define_initcall("6s",fn,6s)
#define late_initcall(fn)                   _define_initcall("7",fn,7)
#define late_initcall_sync(fn)              _define_initcall("7s",fn,7s)

#define _initcall(fn) device_initcall(fn)

#define _exitcall(fn) \
    static exitcall_t _exitcall_##fn _exit_call = fn
```

以上初始化模块按照_define_initcall 定义的顺序执行，首先执行 early_initcall 初始化模块，之后是 pure_initcall 模块、core_initcall 模块等，最后执行 late_initcall_sync。如果 Linux 设备驱动程序采用 built-in[㊀]的方式而不是作为 Module 形式加载时，将使用 device_initcall 函数或者 device_initcall_sync 函数进行加载。

在 Linux 系统初始化时运行的模块需要使用以上的 xxx_initcall 宏，定义该模块的函数指针，之后该模块的函数指针将加入到 Linux 内核的_early_initcall_end 和_initcall_end 之间。我们以 xyz_init 模块的加载为例说明这些 xxx_initcall 函数的使用，xyz_init 函数用来加载某个模块。该函数的初始化过程如源代码 14-3 所示。

**源代码 14-3　xxx_initcall 函数**

```
static int _init xyz_init(void)
{
...
}
xxx_initcall (xyz_init);
```

这段代码首先使用宏 xxx_initcall 定义了一个_initcall_xyz_initx 函数，该函数存放 xyz 函数的指针。在生成 Linux 系统内核时，链接器将这个函数指针存放在_early_initcall_end 和_initcall_end 参数之间。

Linux 系统在初始化时，将在 do_initcalls 函数中执行_initcall_xyz_initx 函数，从而执行 xyz_init 函数。Linux 系统使用这种方法规范初始化模块的执行，并保证这些模块可以按照指定的顺序依次执行。

在 Linux 内核的 System. map[㊁]文件中，可以找到在_early_initcall_end 和_initcall_end 之间

㊀ 将设备驱动程序编译到 Linux 内核中。

㊁ System. map 文件存放 Linux 内核使用的符号表，包括当前 Linux 系统使用的所有函数指针和全局变量。

所有的函数指针，其中与 PCI 总线初始化相关的函数如源代码 14-4 所示，这些函数将按照在以下源代码中出现的顺序依次执行。

**源代码 14-4　System. map 文件中与 PCI 总线初始化相关的函数**

```
c0836ba4 t _initcall_pcibus_class_init2
c0836ba8 t _initcall_pci_driver_init2
c0836bd4 t _initcall_acpi_pci_init3
c0836bec t _initcall_pci_arch_init3
c0836c1c t _initcall_pci_slot_init4
c0836c34 t _initcall_acpi_pci_root_init4
c0836c38 t _initcall_acpi_pci_link_init4
c0836c70 t _initcall_pci_subsys_init4
c0836ca4 t _initcall_pci_iommu_init5
c0836cf0 t _initcall_pcibios_assign_resources5
c0836ebc t _initcall_pci_init6
c0836ec0 t _initcall_pci_proc_init6
c0836ec4 t _initcall_pcie_portdrv_init6
c0836ecc t _initcall_pci_hotplug_init6
c083706c t _initcall_pci_sysfs_init7
c0837084 t _initcall_pci_mmcfg_late_insert_resources7
```

每一次编译 Linux 内核时，都可能会产生一个新的 System. map，但是源代码 14-4 中函数指针的顺序不会发生变化，其执行顺序也不会发生变化。下面将依次分析这些函数的功能。并在后续章节，逐步解析这些函数的实现方法。

### 14. 1. 1　pcibus_class_init 与 pci_driver_init 函数

pcibus_class_init 函数在 . /driver/pci/probe. c 文件中，如源代码 14-5 所示。该函数的主要作用是注册一个名为“pci_bus”的 class 结构。在 Linux 系统中，为了便于测试将所有的设备使用一个文件系统进行管理，这个文件系统也被称为 sysfs 文件系统。

最初 Linux 系统将与设备相关的信息都存放在 proc 文件系统中，而随着 Linux 系统的不断演变，proc 文件系统变得异常混乱而复杂，难以维护，于是 sysfs 文件系统应运而生。与 proc 文件系统相比，sysfs 文件系统的组织结构较为清晰。

目前与设备相关的模块基本上都由 sysfs 文件系统维护，而 proc 文件系统留给真正的系统进程使用。本书不会详细介绍 sysfs 文件系统的详细实现机制，因为 sysfs 文件系统与 PCI 体系结构并没有太大的关系，只是 Linux 系统使用的一种对设备模块进行管理的方法。

**源代码 14-5　pcibus_class_init 函数**

```
static struct class pcibus_class = {
    . name        = "pci_bus",
    . dev_release        = &release_pcibus_dev,
};
```

```
static int _init pcibus_class_init(void)
{
    return class_register(&pcibus_class);
}
postcore_initcall(pcibus_class_init);
```

pcibus_class_init 函数执行完毕后，将会在/sys/class 目录下产生一个“pci_bus”的目录，有兴趣的读者可以使用“ls -l /sys/class”命令找到这个目录。该函数执行完毕后，将很快执行 pci_driver_init 函数，如源代码 14-6 所示。

**源代码 14-6　pci_driver_init 函数**

```
struct bus_type pci_bus_type = {
    .name           = "pci",
    .match          = pci_bus_match,
    .uevent         = pci_uevent,
    .probe          = pci_device_probe,
    .remove         = pci_device_remove,
    .shutdown       = pci_device_shutdown,
    .dev_attrs      = pci_dev_attrs,
    .bus_attrs      = pci_bus_attrs,
    .pm             = PCI_PM_OPS_PTR,
};

static int _init pci_driver_init(void)
{
    return bus_register(&pci_bus_type);
}

postcore_initcall(pci_driver_init);
```

该函数也与 sysfs 文件系统相关，该函数执行完毕后，将在/sys/bus 目录下建立一个“pci”目录，之后当 Linux 系统的 PCI 设备使用 device_register 函数注册一个新的 pci 设备时，将在/sys/bus/pci/drivers 目录下创建这个设备使用的目录。

如在第 12 章源代码 12-1 中，pci_register_driver 函数将最终调用 device_register 函数，并在/sys/bus/pci/drivers 下建立“capric”目录。在这个 capric 目录里包含 capric 卡在 Linux 系统中使用的一系列资源。

在源代码 14-4 中也有一些和 ACPI 机制初始化相关的函数，包括 acpi_pci_init、acpi_pci_root_init 和 acpi_pci_link 函数。有关 ACPI 机制的介绍见第 14.2 节。

### 14.1.2　pci_arch_init 函数

pci_arch_init 函数是 Linux x86 系统执行的第一个与 PCI 总线初始化相关的函数。该函数的定义在 ./arch/x86/pci/init.c 文件中，如源代码 14-7 所示。

源代码 14-7　pci_arch_init 函数

```
/* arch_initcall has too random ordering, so call the initializers
   in the right sequence from here. */
static _init int pci_arch_init(void)
{
#ifdef CONFIG_PCI_DIRECT
    int type = 0;

    type = pci_direct_probe();
#endif

    if (! (pci_probe & PCI_PROBE_NOEARLY))
        pci_mmcfg_early_init();

#ifdef CONFIG_PCI_OLPC
    if (! pci_olpc_init())
        return 0; /* skip additional checks if it's an XO */
#endif
#ifdef CONFIG_PCI_BIOS
    pci_pcbios_init();
#endif
    /*
     * don't check for raw_pci_ops here because we want pcbios as last
     * fallback, yet it's needed to run first to set pcibios_last_bus
     * in case legacy PCI probing is used. otherwise detecting peer busses
     * fails.
     */
#ifdef CONFIG_PCI_DIRECT
    pci_direct_init(type);
#endif
    if (! raw_pci_ops && ! raw_pci_ext_ops)
        printk(KERN_ERR
        "PCI: Fatal: No config space access function found\n");

    dmi_check_pciprobe();

    dmi_check_skip_isa_align();

    return 0;
}
arch_initcall(pci_arch_init);
```

该函数使用了一些编译选项，如果使能 CONFIG_PCI_BIOS 选项表示 Linux x86 系统将使用 BIOS 对 PCI 总线的枚举结果；如果使能 CONFIG_PCI_DIRECT 选项表示由 Linux x86 系统重新枚举 PCI 总线；如果使能 CONFIG_PCI_OLPC 选项表示当前处理器系统属于 OLPC(One Laptop per Child)。本节仅讲述使能 CONFIG_PCI_DIRECT 选项的情况。pci_arch_init 函数首先调用 pci_direct_probe 函数，pci_direct_probe 函数如源代码 14-8 所示。

源代码 14-8　pci_direct_probe 函数

```
int _init pci_direct_probe(void)
{
    struct resource  * region,  * region2;

    if ((pci_probe & PCI_PROBE_CONF1) ==0)
        goto type2;
    region = request_region(0xCF8, 8, "PCI conf1");
    if (! region)
        goto type2;

    if (pci_check_type1()) {
        raw_pci_ops = &pci_direct_conf1;
        port_cf9_safe = true;
        return 1;
    }
    release_resource(region);

type2:
    if ((pci_probe & PCI_PROBE_CONF2) ==0)
        return 0;
    region = request_region(0xCF8, 4, "PCI conf2");
    if (! region)
        return 0;
    region2 = request_region(0xC000, 0x1000, "PCI conf2");
    if (! region2)
        goto fail2;

    if (pci_check_type2()) {
        raw_pci_ops = &pci_direct_conf2;
        port_cf9_safe = true;
        return 2;
    }

    release_resource(region2);
fail2:
    release_resource(region);
    return 0;
}
```

pci_direct_probe 函数首先根据全局变量 pci_probe 判断 raw_pci_ops 函数使用的函数指针。全局变量 pci_probe 的缺省值在 ./arch/x86/pci/common.c 中定义，如下所示。

```
unsigned int pci_probe = PCI_PROBE_BIOS | PCI_PROBE_CONF1 | PCI_PROBE_CONF2 |
            PCI_PROBE_MMCONF;
```

如果 Boot loader 程序(如 Grub)在引导 Linux 内核时没有加入“pci = xxxx”参数，全局变

量 pci_probe 将使用缺省值。此时 pci_direct_probe 函数仅使用“conf1 类型”而不使用“conf2 类型”对 raw_pci_ops 函数赋值。

x86 处理器提供了三种方式访问 PCI 设备的配置空间。一种方法是使用“0xCF8 和 0xCFC”这两个 I/O 端口，这两个端口的详细描述见第 2.2.4 节，Linux x86 系统使用 pci_conf1_read 和 pci_conf1_write 函数操作这两个 I/O 端口，这两个函数的定义见 ./arch/x86/pci/direct.c 文件。

另一种方法是使用“conf2”方法，目前这种方法不再被 Linux x86 继续使用，对这种方法有兴趣的读者可以参考 pci_conf2_read 和 pci_conf2_write 函数，本节对这种方法不做介绍。

inux x86 使用 pci_mmcfg_read 和 pci_mmcfg_write 函数实现 ECAM 方式，这两个函数的定义见 ./arch/x86/pci/mmconfig_32.c 文件中。

其中使用 pci_conf1_read 和 pci_conf1_write 函数只能访问 PCI 设备配置空间的前 256 个字节，而使用 pci_mmcfg_read 和 pci_mmcfg_write 函数可以访问 PCI 设备的全部配置空间。在 Linux 系统中，可以使用这两种方式访问不同的配置空间。

pci_direct_probe 函数执行完毕，pci_arch_init 函数将继续调用 pci_direct_init 函数，然后依次调用 dmi_check_pciprobe() 和 dmi_check_skip_isa_align() 函数，这两个 dmi_xxx 函数与 x86 处理器的 DMI(Desktop Management Interface)接口和 SM(System Management)总线相关，本节对此不做进一步说明。

### 14.1.3 pci_slot_init 和 pci_subsys_init 函数

Linux x86 系统执行完毕 pci_arch_init 函数后，将调用 pci_slot_init 函数，该函数的主要作用是在 sysfs 文件系统中，建立 slots 目录及其 kobject 结构。pci_subsys_init 函数是一个重要的函数，其定义在 ./arch/x86/pci/legacy.c 文件中，如源代码 14-9 所示。

**源代码 14-9　pci_subsys_init 函数**

```
int _init pci_subsys_init(void)
{
#ifdef CONFIG_X86_NUMAQ
    pci_numaq_init();
#endif
#ifdef CONFIG_ACPI
    pci_acpi_init();
#endif
#ifdef CONFIG_X86_VISWS
    pci_visws_init();
#endif
    pci_legacy_init();
    pcibios_fixup_peer_bridges();
    pcibios_irq_init();
    pcibios_init();

    return 0;
}
subsys_initcall(pci_subsys_init);
```

本书并不关心 CONFIG_X86_NUMAQ 和 CONFIG_X86_VISWS 选项。在第 14.3.3 节将详细介绍 CONFIG_ACPI 选项使能时使用的 pci_acpi_init 函数。

pci_legacy_init 函数完成对 PCI 总线的枚举，并在 proc 文件系统和 sysfs 文件系统中建立相应的结构。如果当前处理器系统没有使能 ACPI 机制，则该函数是 Linux x86 对 PCI 总线进行初始化的一个重要函数，其实现机制如源代码 14-10 所示。

**源代码 14-10　pci_legacy_init 函数**

```
static int _init pci_legacy_init(void)
{
    if (! raw_pci_ops) {
        printk("PCI: System does not support PCI\n");
        return 0;
    }

    if (pcibios_scanned ++)
        return 0;

    printk("PCI: Probing PCI hardware\n");
    pci_root_bus = pcibios_scan_root(0);
    if (pci_root_bus)
        pci_bus_add_devices(pci_root_bus);

    return 0;
}
```

pci_legacy_init 函数首先调用 pcibios_scan_root 函数完成对 PCI 总线树的枚举，该函数的输入参数为 0 表示这次枚举将从总线号 0 开始进行。在完成 PCI 总线的枚举后，该函数将调用 pci_bus_add_devices 函数将 PCI 总线上的设备加入到 sysfs 文件系统中。

Linux x86 引入 ACPI 机制之后，pcibios_scanned 参数将被置为 1，从而 pci_legacy_init 函数将直接使用 0 作为返回值，并不会执行 pcibios_scan_root 和 pci_bus_add_devices 函数。

当 pci_legacy_init 函数执行完毕后，pcibios_irq_init 函数将使用 BIOS 提供的中断路由表，初始化当前处理器系统的中断路由表，同时确定 PCI 设备使用的中断向量，本章并不会对该函数进行详细分析，因为 Linux x86 目前大多使用 ACPI 提供的中断路由表，而不再使用 BIOS 中的中断路由表。如果 ACPI 机制被使能，该函数也将直接使用 0 作为返回值，并不会被完全执行。

pcibios_init 函数的主要工作是调用 pcibios_resource_survey 函数，检查 PCI 设备使用的存储器及 I/O 资源。pcibios_resource_survey 函数将在第 14.3.3 节中详细介绍。

### 14.1.4　与 PCI 总线初始化相关的其他函数

pci_iommu_init 函数在 ./arch/x86/kernel/pci-dma.c 文件中，该函数用来初始化处理器系统的 IOMMU，可以配置 IBM X-Series 刀片服务器使用的 Calgary IOMMU、Intel 的 Vtd 和 AMD 的 IOMMU 使用的 I/O 页表。如果在 Linux 系统中没有使能 IOMMU 选项，pci_iommu_init 函

数将调用 no_iommu_init 函数，并将 dma_ops 函数设置为 nommu_dma_ops。本节不进一步介绍该函数的详细实现机制。

pcibios_assign_resources 函数主要处理 PCI 设备使用的 ROM 空间和 PCI 设备使用的存储器和 I/O 资源。该函数的主要功能是调用 pci_assign_unassigned_resources 函数对 PCI 设备使用的存储器和 I/O 资源进行设置。对于 Linux x86 而言，BIOS 已经将 PCI 设备使用的存储器和 I/O 资源设置完毕，而其他 Linux 系统，如 Linux PowerPC，需要使用该函数设置 PCI 设备使用的存储器和 I/O 资源。

pci_init 函数的主要作用是对已经完成枚举的 PCI 设备进行修复工作，用于修补一些 BIOS 中对 PCI 设备有影响的 Bugs。

pci_proc_init 函数的主要功能是在 proc 文件系统中建立 ./bus/pci 目录，并将 proc_fs 默认提供的 file_operations 更换为 proc_bus_pci_dev_operations。

pcie_portdrv_init 函数首先在 ./sys/bus 中建立 pci_express 目录，然后使用 pci_register_driver 函数向内核注册一个名为 pcie_portdriver 的 pci_driver 结构。在 Linux x86 中，pci_express 目录中的设备都是从 sysfs 文件系统的 pci 目录中链接过来的。该函数的实现较为简单。

pci_hotplug_init 函数主要用来支持 CompactPCI 的热插拔功能。CompactPCI 总线在通信系统中较为常见。

而 pci_sysfs_init 函数与 sysfs 文件系统相关，主要功能是将每一个 PCI 设备加入到 sysfs 文件系统的相应目录中，本节对此不做进一步介绍。pci_mmcfg_late_insert_resources 函数的主要功能是将 MMCFG 使用的资源放入系统的 Resource Tree 中，并标记这些资源已经被使用，之后其他驱动程序不能再使用这个资源。

本章并不会对 Linux x86 使用的 Legacy PCI 总线枚举方法进一步描述，x86 处理器为了实现向前兼容，付出了巨大的努力。x86 处理器在实现新的功能的同时，需要向前兼容古董级别的功能，有时 BIOS 无所适从。Linux x86 对 PCI 总线进行初始化时，使用了许多不完美的源代码。而这些貌似不完美的源代码背后，都有许多与向前兼容有关的故事。

## 14.2 x86 处理器的 ACPI

在 x86 处理器中，ACPI(Advanced Configuration and Power Interface)是一个非常重要而且较为复杂的概念。最初 ACPI 规范由 Intel、Microsoft 和 Toshiba 公司共同制定，后来 HP 和 Phoenix 公司也参与了 ACPI 规范的制定，该规范主要包括 x86 处理器系统的资源配置和电源管理两方面内容。

ACPI 规范整合了之前的 OSPM(Operating System directed Power Management)、MultiProcessor 规范和 Plug and Play BIOS 规范，并定义了一系列数据结构与电源管理状态，提供了电源管理接口、硬件及其 Firmware 接口，以描述处理器系统的设备和电源管理策略。ACPI 规范在 1996 年 12 月发布 1.0 版本，目前的稳定版是 4.0 版。

ACPI 规范是 x86 处理器使用的 Firmware 接口标准，操作系统需要获得的处理器底层信息基本上都可以从 ACPI 表中获得。在 ACPI 诞生之前，基于 x86 处理器的操作系统需要使用 BIOS 才能获得相应的信息。而不同厂商提供的 BIOS 之间并没有一个统一标准，从而在某种程度上造成硬件资源管理与使用上的混乱。

产生这种混乱的主要原因是由于x86处理器系统为了实现向前兼容，有许多不得已；而部分原因是在x86处理器系统中，有许多外部设备本身就挂接在一些“不可配置的”总线上，如ISA/EISA总线，还有一些外部设备本身就是不标准的，如电源按钮和在笔记本上使用的一些“特殊功能键”。

在ACPI规范没有出现之前，BIOS厂商通常按照某种“自定义”的方式使用这些外部设备，因此需要为操作系统提供各类驱动程序，并由操作系统集成这些并不属于任何标准的驱动程序，从而给操作系统的开发与维护带来了极大的困难。

ACPI机制在这种背景下应运而生。ACPI机制提供了一组与处理器硬件和操作系统的相关接口，对处理器平台以及设备的电源进行管理，并可以配置和管理外部设备使用的系统资源。ACPI机制主要管理以下系统资源。

- Legacy PNP设备[㊀]，如ISA设备、串并口等设备。
- 笔记本使用的一些外部设备，如电源开关、风扇、电源和一些快捷键。
- 系统电源管理，包括处理器和外部设备的电源管理。这部分内容也是ACPI规范的设计重点。
- 系统的热插拔管理。热插拔是ACPI规范的设计重点，ACPI系统热插拔可以覆盖小到“从笔记本插拔CDROM”，大到“NUMA(Non-uniform Memory Access)结构处理器系统热插拔PE(Processor Element)、存储器节点(Memory Node)和I/O节点”这些应用。不过许多NUMA处理器系统并没有使用ACPI规范进行热拔插管理。本节对ACPI的热插拔管理不做深入介绍。
- PCI设备的中断向量分配。在x86处理器平台中，中断向量的分配始终是一个问题。x86处理器由于一些历史遗留问题，中断向量的分配并不尽善尽美，而x86处理器为了实现向前兼容，必须保留这些不完美。本章将在第15.1.1节详细介绍中断向量的分配。
- 一些集成在MCH和ICH中的外部设备。x86处理器使用PCI配置空间存放这些外部设备的寄存器，但是这些设备并不都是严格意义上的PCI设备，甚至不是一个外部设备。如在x86处理器中的使用存储器映射寻址的一些寄存器，这些寄存器被存放在某个PCI设备的配置空间中，如TOLUD寄存器存放在Bus号、Device号和Function号都为0的PCI设备中，但是这个PCI设备并不是处理器系统的标准PCI设备。ACPI规范需要管理这类“伪PCI设备”中的寄存器。

目前在x86处理器中，新引入的一些与处理器体系结构相关的特性，基本上都只使用ACPI机制进行描述，而不再使用BIOS。因此为了深入理解x86处理器平台，需要了解一些与ACPI相关的基本知识。ACPI规范所涉及的内容非常广泛，与ACPI规范有关的全部知识可以独立成书，本节仅简要介绍ACPI规范中与PCI总线相关的部分内容。ACPI规范的各部分内容相对较为独立，除了ACPI规范的开发者和从事与此相关的系统程序员之外，绝大多数程序员不需要了解与ACPI规范相关的全部知识。

从系统软件的角度上看，ACPI的组成结构如图14-1所示。从图中可以发现，ACPI用以连接系统硬件平台与操作系统，在屏蔽了硬件的实现细节的同时，提供了一系列系统资源，包括ACPI寄存器(ACPI Registers)、ACPI BIOS和ACPI表(ACPI Tables)。

---

㊀ 微软规定了一系列PnP设备规范，详见http://www.microsoft.com/whdc/system/pnppwr/pnp/default.mspx。

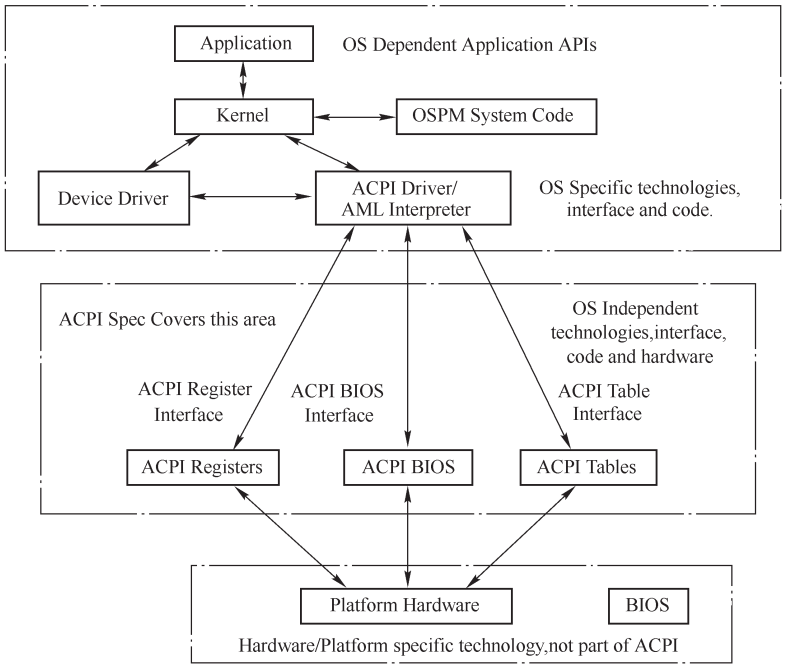



图 14-1　ACPI 的组成结构

当一个处理器系统使能 ACPI 机制后，操作系统访问 ACPI 管理的“非标”外设时，首先通过 ACPI 机制提供的一套标准 API，并由这些 API 将访问存放在系统内存中的 ACPI 表，并通过执行 ACPI 表中的程序读写 ACPI 寄存器，或者对这些寄存器进行操作。在 ACPI 表中，除了含有处理器系统的资源信息之外，还包括管理这些资源信息的操作函数，因此 BIOS 厂商通过 ACPI 表即可实现一些简单的设备驱动程序。

操作系统使用这些 API，并通过 ACPI Driver/AML Interpreter 访问 ACPI 提供的系统资源，包括 ACPI Registers、ACPI BIOS 和 ACPI 表。ACPI 表对于理解 ACPI 机制较为重要，在第 14.2.2 节和第 14.2.3 节将专门介绍该表的组成与实现。

在 x86 处理器系统中，系统资源由 ACPI BIOS 维护，操作系统使用 ACPI 提供的标准 API 从 ACPI BIOS 中获得这些资源，而不必关心这些“非标”外设的具体实现方式，因此也不需要使用特定的驱动程序访问这些“非标”外设。

因为这些“非标”外设的管理由 ACPI BIOS 完成，x86 处理器使用 ACPI 表描述这些“非标”外设，并将这个 ACPI 表存放到 BIOS 中。OEM 厂商需要在标准 PC 处理器平台上添加一些“自定义”的功能时，只需改动 ACPI 表即可，而不需要改动操作系统，从而极大地降低了操作系统的集成与维护难度。

ACPI 机制使用的“标准 API”在 ACPICA(ACPI Component Architecture Programmer Reference)规范中定义。这些标准 API 与操作系统相对独立，从而便于 ACPICA 在不同操作系统中的实现。

如图 14-1 所示，ACPI 机制所覆盖的内容包括 ACPI 寄存器组、ACPI BIOS 和 ACPI 表。在 ACPI 寄存器组中含有电源管理、处理器控制和 GPE(General-Purpose Event)寄存器组。其中处理器控制寄存器组是可选的，而电源管理和 GPE 寄存器组是必须实现的。这些与 ACPI

机制相关的寄存器在 Intel 的 ICH 中定义。

电源管理寄存器组由 PM1a_STS、PM1a_EN、PM1b_STS、PM1b_EN、PM1_CNTa、PM1_CNTb 等寄存器组成。在这些寄存器中，PM1a_STS、PM1_CNTa 寄存器和 PM1a_EN 是必须支持的，而其他寄存器是可选的。在 Intel 的 ICH9 中，PM1a_STS 寄存器与 PM1_STS 寄存器对应，PM1_CNTa 寄存器与 PM1_CNT 寄存器对应，而 PM1a_EN 寄存器与 PM1_EN 寄存器对应。这些寄存器的简单描述如下，有兴趣的读者可以参考[Intel I/O Controller Hub 9]的第 13.8 节以获得这些寄存器的详细说明。

- PM1_STS 寄存器包含当前处理器被唤醒的原因，以及一些与电源按键(Power Button)相关的信息。
- 通过操作 PM1_CNT 寄存器可以使处理器进入不同的休眠状态，并确定 ACPI 中断请求使用 SCI 中断请求还是 SMI[⊖]中断请求。
- PM1_EN 寄存器设置 SCI(System Control Interrupt)中断使能位，决定当 PM1_STS 寄存器的状态有效时，是否向处理器提交 SCI 中断请求。SCI 中断请求由 ACPI 规定的相应事件使用，并缺省使用中断向量 9 向处理器提交中断请求，操作系统使用中断服务程序进一步处理来自 ACPI 的中断请求。

GPE 寄存器组是 ACPI 寄存器组的一个重要组成部分，由 GPE0_STS、GPE0_EN、GPE1_STS 和 GPE1_EN 寄存器组成。在 Intel 的 ICH9 中，由 General Purpose I/O 寄存器组实现 ACPI 规范的 GPE 寄存器组。General Purpose I/O 寄存器组的使用方法非常简单，对此有兴趣的读者可以参考[Intel I/O Controller Hub 9]的第 13.10 节。

ACPI 的 GPEx_STS 和 GPEx_EN 寄存器的使用方法较为简单。其中 GPEx_STS 寄存器的每一位在 GPEx_EN 寄存器中都有一个使能位，当 GPEx_STS 寄存器的某位有效时，表示产生了一个 GPE 事件，如果与该位相对应的使能位也有效时，ICH 将向处理器提交 SCI 中断请求，由中断服务程序进一步处理这个 GPE 事件。

ACPI BIOS 与操作系统中的 ACPI 驱动程序/AML 解释器(ACPI Driver/ACPI Machine Language Interpreter)密切相关。操作系统可以使用 ACPICA 提供的标准 API，再通过 ACPI 驱动程序/AML 解释器访问 ACPI BIOS，并从 ACPI BIOS 获得相应的信息后执行与底层硬件相关的代码操纵实际的设备，有关这部分内容的详细说明见下文。

ACPI 表描述处理器平台使用的资源和管理这些资源的执行操作，操作系统可以通过标准的 API 函数访问 ACPI 表并执行相关的程序，从而维护整个 ACPI 系统的运转。ACPI 表是 BIOS 提供给系统软件的重要资源。

### 14.2.1 ACPI 驱动程序与 AML 解释器

ACPI 驱动程序与 AML 解释器与操作系统实现相关，其主要目的是将操作系统与 ACPI 提供的资源进行隔离。如上文所述，ACPI 使用一组标准的 API 函数访问 ACPI 表，目前 Unix/Linux 系统使用 ACPICA 规范实现这些接口函数，ACPICA 的组成结构如图 14-2 所示。值得注意的是 Windows 使用了其他方式实现这些接口函数。

---

⊖ x86 处理器接收 SMI 中断请求后，将进入 SMM 模式，此时将由 BIOS 处理 ACPI 中断请求。而 SCI 中断也被称为 ACPI 中断，下文将详细介绍该中断的实现机制。

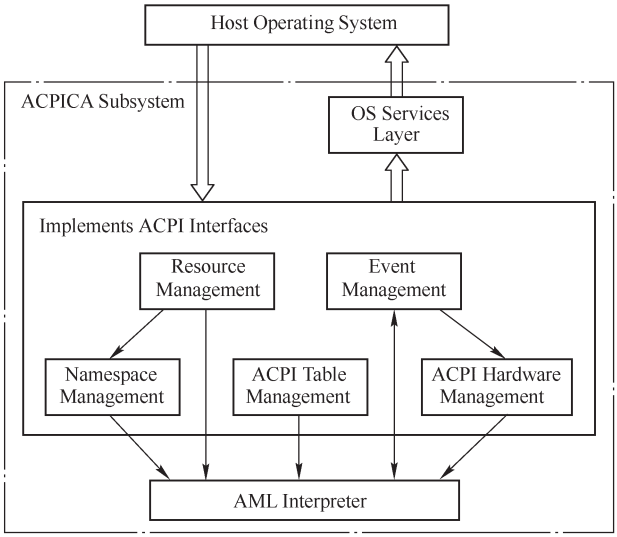



图 14-2　ACPICA 的组成结构

由上图所示，当操作系统需要访问 ACPI 表时，将首先通过 ACPICA 接口函数，因此在一个操作系统中，仅需要关注 ACPICA 接口函数，而不必了解硬件的具体实现细节，并由 BIOS 管理硬件的具体实现细节。从而使得某 x86 处理器平台引入的某些“自定义”功能仅与 BIOS 有关，而与操作系统无关。

### 1. ACPICA 接口函数

ACPICA 子系统提供了一系列标准的函数接口，Host OS 可以通过这些函数接口向 AML 解释器传递数据，并由 AML 解释器访问 ACPI 的相关资源。在 Linux 系统的 ./drivers/acpi/acpica 目录中，定义了这些 ACPI 接口函数，这些 ACPI 接口函数包括以下几大类。本章并不会详细介绍这些接口函数，而仅在分析相关代码时简要介绍对应的接口函数。

- ACPI Table Management 接口函数。这组 API 负责分析和管理在操作系统中存放的 DSDT、FAT 等描述符，ACPI 表在操作系统引导时放入系统内存中，有关 ACPI 表的详细描述见第 14.2.2 节。
- Namespace Management 接口函数。这组 API 负责创建 APCI 表在操作系统中存放的名字空间，并管理这些名字空间。
- Resource Management 接口函数。在名字空间中包含一些硬件资源，如 I/O 地址空间和中断向量等，这组 API 负责管理名字空间中使用的各类资源。
- Event Management 接口函数。这组 API 负责处理 ACPI 中的各类 Event，包括 GPE 事件和 PM 事件。
- ACPI Hardware Management 接口函数。这组 API 负责访问 ACPI 提供的各类硬件资源，包括寄存器和中断。

Host OS 通过 ACPICA 提供的接口函数最终可以访问 AML 解释器，并由 AML 解释器访问底层硬件。当一个系统使能了 ACPI 机制后，底层硬件的实现细节将被屏蔽，AML 解释器并不会直接访问底层硬件，而是直接访问 ACPI 提供的硬件抽象层。ACPI 使用 AML 语言描述这个硬件抽象层。

在一个处理器系统中，ACPI BIOS 将提供 ACPI 表，并由 BIOS 存放到处理器系统的特定存储器空间中。操作系统在初始化时，需要通过某个地址找到这个 ACPI 表。

值得注意的是 ACPI 主要管理“非标准”外部设备的硬件资源，并不会管理一些标准外部设备，如标准 PCI 设备，因此在 ACPI 表中并不包含标准 PCI 设备使用的 BAR 地址空间等一系列标准信息，但是会管理 PCI 总线中的中断路由表。

ACPI 表由 AML 解释器负责分析并维护，在 ACPI 表中除了存放底层硬件的资源描述外，还可以操作这些资源。有关 ACPI 表的详细介绍见下文。

如图 14-2 所示，在 ACPICA 中还存在一个 OS 服务层(OS Services Layer)接口，该接口的主要目的是保证 ACPICA 实现的“系统独立性”。目前在 Unix/Linux 系统中，ACPICA 接口函数使用 OS 服务层接口，访问与 ACPI 相关的硬件资源或者执行相应的操作。

### 2. OS 服务层接口函数

ACPICA 的实现与操作系统无关，但是 ACPI 接口函数仍然需要使用一些操作系统资源，比如 ACPI 接口函数需要使用操作系统提供的分配与释放内存资源，访问 PCI 设备配置空间等一系列 API 函数。

但是在不同的操作系统中，访问系统内存资源、访问 PCI 配置空间所需要调用的 API 函数并不相同。为了保证 ACPICA 的实现与操作系统无关，ACPICA 抽象了 OS 服务层接口函数。ACPICA 的接口函数使用 OS 服务层接口函数，而不是操作系统提供的函数访问 HOST OS 的资源，以保证 ACPICA 的独立性。值得注意的是，在不同的操作系统中，ACPICA 定义这组函数的实现并不相同，目前 Linux/Unix 系统使用了 ACPICA 定义的这组函数，而 Windows 使用其他的方法实现这些功能。

在 Linux 系统中，OS 服务层接口函数的实现在 ./drivers/acpi/osl.c 文件中，在该文件中提供了一系列访问系统资源的标准函数，这些函数实现的功能相对简单，如 acpi_os_printf 函数、acpi_os_sleep 函数、acpi_os_write_memory、acpi_os_read_pci_configuration。在该文件中，还包含一些最基本的与访问外部设备、内存管理和进程调度相关的操作函数。

例如在 ACPICA 程序释放中断服务例程时，需要调用 acpi_os_remove_interrupt_handler 函数，而不能直接调用 Linux 系统提供的 free_irq 函数，即便在 Linux 系统中，这两个函数几乎等价。因为 acpi_os_remove_interrupt_handler 函数的主要工作就是调用 free_irq 函数。该函数的实现如源代码 14-11 所示。

**源代码 14-11　acpi_os_remove_interrupt_handler 函数**

```
acpi_status
acpi_os_remove_interrupt_handler(u32 irq, acpi_osd_handler handler)
{
    if (irq) {
        free_irq(irq, acpi_irq);
        acpi_irq_handler = NULL;
        acpi_irq_irq = 0;
    }

    return AE_OK;
}
```

但是在开发与 ACPICA 相关的函数时，需要调用 acpi_os_remove_interrupt_handler 函数，而不能直接使用 free_irq 函数，以保证 ACPICA 的平台无关性，因为在不同的操作系统中，释放中断服务例程使用的函数并不相同。

ACPICA 使用 OS 服务层接口函数，极大降低了 ACPICA 接口函数的移植难度，在 Linux 系统中实现的 ACPICA 接口函数可以方便地移植到其他操作系统中。

### 14.2.2 ACPI 表

ACPI 规范使用了一系列描述符表管理处理器系统的部分硬件信息，而且包含与这些硬件相关的操作，并使用 RSDP 指针（Root System Description Pointer）指向这些描述符表。ACPI 规范定义了以下描述符表。

- XSDT（Extended System Description Table）。XSDT 包含 ACPI 规范的版本号和一些与 OEM 相关的信息，并含有其他描述符表的 64 位物理地址，如 FADT（Fixed ACPI Description Table）和 SSDT（Secondary System Description Table）等。
- RSDT（Root System Description Table）。RSDT 包含的信息与 XSDT 基本一致，只是在 RSDT 中存放的物理地址为 32 位。在 V1.0 之后的 ACPI 版本中，该描述符表被 XSDT 取代。但是有些 BIOS 可能会为操作系统同时提供 RSDT 和 XSDT，并由操作系统选择使用 RSDT 还是 XSDT。
- FADT。FADT 包含 ACPI 寄存器组使用的系统 I/O 端口地址、FACS（Firmware ACPI Control Structure）和 DSDT（Differentiated System Description Table）的基地址等信息。FADT 中还存放了一个“Boot Architecture Flags”字段，在这个字段中存放一些有关处理器系统初始化的基本信息，详见[Advanced Configuration and Power Interface Specification 4.0]的 Table 5-11。值得注意的是，FADT 的识别标识是“FACP”，在 ACPI 表中，FACP.dat 文件存放处理器系统的 FADT 表。
- FACS。FACS 包含 OS 与 BIOS 进行数据交换使用的一些参数，包括处理器系统的硬件签名，以及 Firmware 在处理器系统被唤醒后使用的、用来通知操作系统 Firmware 工作已经告一段落的中断向量，即 Firmware Waking Vector。在处理器被唤醒之后，Firmware 将执行一些基本的加载操作，并通过 Firmware Waking 中断向量，将控制权交还给操作系统，由操作系统完成其他的唤醒操作。在 FACS 中，还包含一个全局锁（Global Lock），当 Firmware 和操作系统对某些临界资源进行访问时，需要使用该锁。
- DSDT。该表是 ACPI 规范最复杂，同时也是最重要的一个表。该表包含处理器系统使用的硬件资源以及对这些硬件资源的管理操作。SSDT 可以对 DSDT 进行补充，在一个处理器系统中可以存在多个 SSDT。
- ACPI 规范还定义了一些其他表项，如 MADT（Multiple APIC Description Table）、SBST（Smart Battery Table）、SRAT（System Resource Affinity Table）和 SLIT（System Locality Information Table）等一系列表项。其中 MADT 描述处理器系统的中断资源和多处理器相关的配置信息；SBST 与电池的管理相关；而 SRAT 和 SLIT 与 NUMA 系统的资源管理相关。

在 ACPI 4.0 中，上述这些表的组成结构如图 14-3 所示。

如上图所示，在 RSDP 中提供了两个物理地址分别指向 RSDT 和 XSDT。其中在 RSDT 和在 RSDT 指向的其他描述符表中，如 SSDT 和 FADT 都使用 32 位物理地址，而在 XSDT 和在 XSDT 指向的其他描述符表中都使用 64 位物理地址。在 ACPI 2.0 规范之后的版本，均提供

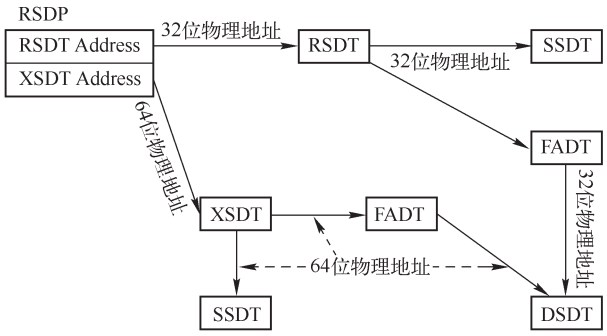



图 14-3　ACPI 表的组成结构

对 XSDT 的支持，即使用 64 位物理地址。

ACPI 表存放在处理器的主存储器中，当处理器系统初始化时，BIOS 将这些表放入特定物理内存，之后系统软件可以访问这些表项。Linux 系统提供了一系列操作 ACPI 表的工具，用户可以使用这些工具读取在系统内存中的 ACPI 表，并将其分解为 DSDT、XSDT 等描述符表。其使用方法如源代码 14-12 所示。

**源代码 14-12　ACPI 表的提取方法**

```
$ acpidump > tylersburg - hedt. out
$ acpixtract - a tylersburg - hedt. out
$ iasl - d APCI. dat
$ iasl - d DSDT. dat
...
$ iasl - d XSDT. dat
```

首先用户可以使用 acpidump 命令将 ACPI 表从内存读出，之后存放到 tylersburg-hedt. out 文件中；然后使用 acpixtract 命令将 tylersburg-hedt. out 文件存放的 ACPI 表全部分解，并得到一系列后缀为 . dat 的文件，其中 RSDP. dat 文件存放 RSDT 和 XSDT 表的物理地址；RSDT. dat 文件存放对 RSDT 的描述；而 XSDT. dat 文件存放对 XSDT 的描述。

这些 . dat 文件使用 AML 语法规范，操作系统中的 AML 解释器可以分析这些在 . dat 文件中的数据。但是这些 . dat 文件并不适合阅读，用户可以使用 iasl 命令将这些 . dat 文件转换为相应的 . dsl 文件。在 . dsl 文件中存放 ASL( ACPI Source Language)源代码，ASL 是一种高级语言，便于阅读和编写。在 Linux 系统中，可以使用以下方法调试 ACPI 表。通过源代码 14-12，可以得到 DSDT. dsl 文件，之后可以使用源代码 14-13 所示的方法调试 DSDT 表。

**源代码 14-13　DSDT 表的调试**

```
$ iasl - tc DSDT. dsl                    \\产生一个 DSDT. hex 文件
$ cp DSDT. hex $ SRC/include/            \\将这个文件复制到 Linux 源代码的 include 文件夹下
...
向 . config 文添加以下描述
CONFIG_STANDALONE = n                    \\将原 . config 的 y 改写为 n
CONFIG_ACPI_CUSTOM_DSDT = y
CONFIG_ACPI_CUSTOM_DSDT_FILE = "DSDT. hex"
```

经过以上操作，重新编译 Linux 内核，并用这个内核重新引导 Linux 系统后，Linux 系统将使用源代码 14-13 指定的 DSDT. hex 替代 BIOS 提供的 DSDT 表。采用这种方法，可以对 DSDT 表进行调试。

### 14.2.3 ACPI 表的使用实例

在 ACPI 提供的各类表中，DSDT 描述符表最为重要。DSDT 描述符表包含当前处理器系统使用的一些硬件资源，如某些外部设备使用的地址空间，以及对这些硬件资源的操作等其他描述信息。

当操作系统收到 ACPI 中断请求，即 SCI 中断请求时，将根据 DSDT 中提供的代码对相应的 ACPI 寄存器进行操作，从而完成所需的功能。DSDT、ACPI 寄存器和操作系统之间的关系如图 14-4 所示。

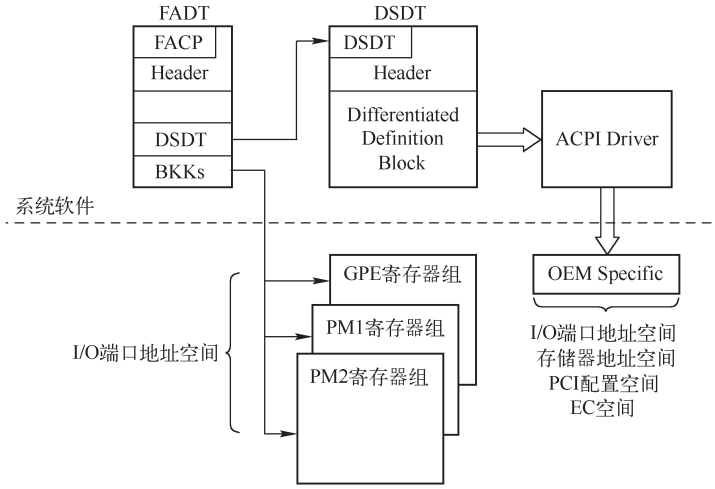



图 14-4 DSDT、ACPI 寄存器和操作系统之间的关系

下文将举例说明图 14-4 中各模块之间的关系，以及系统软件如何处理 ACPI 表。假设在一个 x86 处理器系统中，电源按钮(Power Button)使用 GPIO(General Purpose I/O)方式与处理器系统连接，而不是使用 Fixed hardware 方式。

在 ICH9 中定义了一个 PWRBTN#信号，该信号用来处理电源按钮。如果在一个处理器系统中，电源按钮信号连接到 ICH9 的 PWRBTN#信号时，处理器系统将使用 PM 寄存器组处理这个电源按钮，即电源按钮使用了 Fixed hardware 方式与处理器系统进行连接，而不是 GPE 方式与处理器系统进行连接。

使用 Fixed hardware 方式处理电源按钮的主要缺点是不够灵活，有时 OEM 厂商(Original Equipment Manufacturer)可以利用电源按键实现一些自定义的功能。此时主板设计者可以将电源按钮与 EC[㊀](Embedded Controller)直接相连，并将电源按钮事件与 GPE 联系在一起。使用这种方式时 OEM 厂商可以灵活地控制电源按钮。

㊀ EC 可以连接 PS2 口鼠标、键盘，并可以扩展其他自定义接口。在 x86 处理器中，EC 由一个简单的 MCU(Micro Control Unit)实现。

当用户按下电源按钮时，EC 可以根据其按键时间的长短产生不同的电源按钮事件，如“按键小于 4s”和“按键大于 4s”所对应的电源按钮事件。而且在 x86 处理器系统处于不同的运行状态，如 G0、G1 和 G2 时，对电源按钮事件的解释也并不相同。

x86 处理器规定了一系列休眠状态，其中 G0 为工作状态，与 S0 状态对应；G1 为休眠状态，G1 状态分为 4 个等级，分别为 S1～S4，其中编号越大，休眠的程度越深；G2 状态为 Soft Off 状态，该状态与 S5 状态对应，表示当前处理器处于软下电状态，此时处理器除了一些最基本模块，如 EC、ICH 中的部分逻辑和 Wake-on-LAN 机制仍然保持电源供应之外，其他所有模块均不供电；G3 状态为 Mechanical Off 状态[㊀]，此时处理器处于完全断电状态，全部模块均不上电。

下文仅讨论 x86 处理器处于 G0 和 G1 状态时，如何处理电源按钮事件。在 x86 处理器中，处理电源按钮事件的解释程序在 DSDT 表中，如源代码 14-14[㊁]所示。

**源代码 14-14　与电源按钮事件相关的 ASL 程序**

```
// Define a control method power button
Device( \_SB. PWRB) {
    Name( _HID, EISAID( "PNP0C0C" ) )
    Name( _PRW, Package( ) {0, 0x4} )
    OperationRegion( \PHO, SystemIO, 0x200, 0x1)
    Field( \PHO, ByteAcc, NoLock, WriteAsZeros) {
        PBP, 1, // sleep/off request
        PBW, 1 // wakeup request
    }
} // end of power button device object

Scope( \_GPE) { // Root level event handlers
    Method( _L00) { // uses bit 0 of GP0_STS register
        If( \PBP) {
            Store( One, \PBP) // clear power button status
            Notify( \_SB. PWRB, 0x80) // Notify OS of event
        }
        If( \PBW) {
            Store( One, \PBW)
            Notify( \_SB. PWRB, 0x2)
        }
    } // end of _L00 handler
} // end of \_GPE scope
```

这段程序的说明如下：

- 创建一个“PWRB”设备，其标识(_HID)为“PNP0C0C”，即 PWRB 设备与电源按钮对应。在 ACPI 中，每一个设备使用的标识不相同。

㊀ 处理器系统没有打开电源开关时，处理器系统处于 S5 状态。通常这个开关在 ATX 电源模块中。

㊁ 这段源代码来自 ACPI 规范 4.0，在一个实际的 x86 处理器系统中，电源按钮的处理可能与此不同。

- 将_PRW[㊀]定义为 Package( ){0, 0x4}。其含义为在处理器处于 S1 ~ S4 状态时，电源按钮可以将处理器系统唤醒，而且此时使用 GPEx_STS 寄存器的第 0 位作为唤醒状态位。处理器在即将进入休眠模式时，需要检查对应设备的_PRW 参数，并保证处理器系统进入的休眠等级不大于设备在_PRW 的定义，同时需要保证 GPEx_EN 寄存器的对应位是有效的，否则处理器进入休眠模式时，不能被这个按键事件唤醒。对于本节所提供的实例，如果处理器进入的休眠等级大于 S4，或者在进入休眠状态之前 GPEx_EN 寄存器的第 0 位没有使能时，用户将不能使用“ACPI 机制提供的电源按钮事件”激活处理器系统。
- 声明一个 PHO 变量，使用的 I/O 端口地址为 0x200，这个 I/O 端口的第 0 位和第 1 位分别与 PBP 和 PBW 位对应。其中 PBP 位为 1 表示处理器系统处于 S0 状态时电源按钮被按下，此时处理器需要进入休眠状态；PBW 位为 1 表示处理器系统处于 S1 ~ S4 状态时，电源按钮被按下，此时处理器需要被唤醒。
- 从 Scope( _GPE)开始的这段程序描述对电源按钮事件的处理过程。当 GPEx_STS 寄存器的第 0 位为 1 时，将进一步检查 PBP 和 PBW 位。
- 如果 PBP 位为 1，则首先向 PBP 位写 1，清除这个状态位，之后通知 OSPM 当前电源按钮对应的回调号为 0x80[㊁]。回调号为 0x80 表示在处理器处于 S0 状态时，电源按钮被按下。
- 如果 PBW 位为 1，则首先向 PBW 位写 1，清除这个状态位，之后通知 OSPM 当前电源按钮对应的回调号为 0x02。回调号为 0x02 表示设备发出了一个唤醒信号。

由以上描述，可以发现在源代码 14-14 中，除了定义了一个 PWRB 设备之外，还使用 ASL 语言简单描述了当有 PBW 或者 PBP 事件发生时，处理器的执行操作。对于一个具体的操作系统，如 Linux，需要将 PBW/PBP 事件和处理这些事件的执行操作联系在一起。

当电源按钮被按下时，如果 GPEx_EN 寄存器的相应位被使能，则处理器的 GPIO 接口将置 GPEx_STS 寄存器的对应位为 1，同时向处理器提交 SCI 中断请求。Linux 系统首先需要提供处理这个 SCI 中断请求的中断服务例程，然后进一步处理这些 SCI 中断请求。

在 Linux 系统中，这个中断服务例程为 acpi_ev_sci_xrupt_handler 函数，该函数即为 SCI 中断服务例程。SCI 中断服务例程具有 3 个输入参数，如下所示。

- gsi 参数为 acpi_gbl_FADT. sci_interrupt，缺省值为 0x09。
- handler 参数为 acpi_ev_sci_xrupt_handler。
- context 参数为 acpi_gbl_gpe_xrupt_list_head。

Linux 系统进行初始化，该中断服务例程由 acpi_early_init 函数调用 acpi_enable_subsystem 函数挂接到 Linux 系统的中断处理服务主程序( do_IRQ 函数)中，acpi_ev_sci_xrupt_handler 函数的详细说明在 . /drivers/acpi/acpica/evsci. c 文件中。

acpi_enable_subsystem 函数的详细实现在 . /drivers/acpi/acpica/utxface. c 文件中，该函数将调用 acpi_ev_install_xrupt_handlers→acpi_ev_install_sci_handler 函数注册 SCI 中断服务例程。acpi_ev_install_sci_handler 函数如源代码 14-15 所示。

---

㊀ _PRW 是 Power Resources for Wake 的缩写，可以将处理器从休眠状态唤醒的设备需要使用_PRW。

㊁ ACPI 4.0 规范的第 5.6 节 APIC Event Programming Model 中定义了一系列回调号。

源代码 14-15　acpi_ev_install_sci_handler 函数

```
u32 acpi_ev_install_sci_handler(void)
{
    u32 status = AE_OK;

    ACPI_FUNCTION_TRACE(ev_install_sci_handler);

    status =
        acpi_os_install_interrupt_handler((u32) acpi_gbl_FADT.sci_interrupt,
                                          acpi_ev_sci_xrupt_handler,
                                          acpi_gbl_gpe_xrupt_list_head);
    return_ACPI_STATUS(status);
}
```

acpi_ev_install_sci_handler 函数将调用 acpi_os_install_interrupt_handler 函数，并将 acpi_gbl_FADT.sci_interrupt、acpi_ev_sci_xrupt_handler 和 acpi_gbl_gpe_xrupt_list_head 参数传递给该函数。acpi_gbl_FADT.sci_interrupt 为 SCI 中断使用的 irq 号，acpi_ev_sci_xrupt_handler 即为 SCI 中断处理函数，acpi_gbl_gpe_xrupt_list_head 为 SCI 中断处理函数使用的入口参数。

acpi_os_install_interrupt_handler 函数的实现如源代码 14-16 所示。

源代码 14-16　acpi_os_install_interrupt_handler 函数

```
acpi_status
acpi_os_install_interrupt_handler(u32 gsi, acpi_osd_handler handler,void *context)
{
    unsigned int irq;

    acpi_irq_stats_init();

    /*
     * Ignore the GSI from the core, and use the value in our copy of the
     * FADT.  It may not be the same if an interrupt source override exists
     * for the SCI.
     */
    gsi = acpi_gbl_FADT.sci_interrupt;
    if (acpi_gsi_to_irq(gsi, &irq) < 0) {
        printk(KERN_ERR PREFIX "SCI (ACPI GSI %d) not registered\n",gsi);
        return AE_OK;
    }

    acpi_irq_handler = handler;
    acpi_irq_context = context;
    if (request_irq(irq, acpi_irq, IRQF_SHARED, "acpi", acpi_irq)) {
        printk(KERN_ERR PREFIX "SCI (IRQ%d) allocation failed\n", irq);
        return AE_NOT_ACQUIRED;
```

```
        }
        acpi_irq_irq = irq;

        return AE_OK;
    }
```

这段程序首先调用 acpi_irq_stats_init 函数建立 sysfs 中的 kobject，之后从 FADT[㊀]中获得 ACPI 使用的中断向量，在绝大多数 x86 处理器系统中，SCI 中断使用的 irq 号为 0x9。之后这段程序调用 request_irq，将 acpi_irq 函数与 irq 号 0x9 联系在一起。之后 Linux 系统将使用 acpi_irq 函数处理 SCI 中断请求，该函数的实现如源代码 14-17 所示。

**源代码 14-17　acpi_irq 函数**

```
static irqreturn_t acpi_irq(int irq, void *dev_id)
{
    u32 handled;

    handled = (*acpi_irq_handler)(acpi_irq_context);

    if (handled) {
        acpi_irq_handled++;
        return IRQ_HANDLED;
    } else {
        acpi_irq_not_handled++;
        return IRQ_NONE;
    }
}
```

acpi_irq 函数的主要作用是执行(*acpi_irq_handler)(acpi_irq_context)函数，并检查执行结果是否正确。acpi_irq_handler 函数指针在 acpi_os_install_interrupt_handler 函数中被赋值为 acpi_ev_sci_xrupt_handler 函数。因此在 Linux ACPI 的实现中，acpi_ev_sci_xrupt_handler 函数为真正的 SCI 中断服务例程，该函数在./drivers/acpi/acpica/evsci.c 文件中，如源代码 14-18 所示。

**源代码 14-18　acpi_ev_sci_xrupt_handler 函数**

```
/******************************************************************************
 *
 * FUNCTION:       acpi_ev_sci_xrupt_handler
 *
 * PARAMETERS:  Context  -  Calling Context
 *
 * RETURN:         Status code indicates whether interrupt was handled.
 *
 * DESCRIPTION:  Interrupt handler that will figure out what function or
```

㊀ 在 FACP.dsl 文件中存放 SCI Interrupt 使用的中断向量。

```
 *                  control method to call to deal with a SCI.
 *
 ******************************************************************************/
static u32 ACPI_SYSTEM_XFACE acpi_ev_sci_xrupt_handler(void *context)
{
    struct acpi_gpe_xrupt_info *gpe_xrupt_list = context;
    u32 interrupt_handled = ACPI_INTERRUPT_NOT_HANDLED;

    ACPI_FUNCTION_TRACE(ev_sci_xrupt_handler);

    /*
     * We are guaranteed by the ACPI CA initialization/shutdown code that
     * if this interrupt handler is installed, ACPI is enabled.
     */

    /*
     * Fixed Events:
     * Check for and dispatch any Fixed Events that have occurred
     */
    interrupt_handled |= acpi_ev_fixed_event_detect();

    /*
     * General Purpose Events:
     * Check for and dispatch any GPEs that have occurred
     */
    interrupt_handled |= acpi_ev_gpe_detect(gpe_xrupt_list);

    return_UINT32(interrupt_handled);
}
```

该函数首先调用 acpi_ev_fixed_event_detect 函数检查 PM 寄存器组，判断是否存在 PM 事件，之后调用 acpi_ev_gpe_detect 函数检查是否存在 GPE 事件。上文中描述的 PBW 和 PBP 事件由 acpi_ev_gpe_detect 函数处理。

acpi_ev_gpe_detect 函数的执行逻辑较为简单，本节不再列出该函数的源代码，该函数在 ./drivers/acpi/acpica/evgpe.c 文件中，属于 ACPICA 提供的 Event Management 接口函数。该函数首先获得一个自旋锁 acpi_gbl_gpe_lock，之后检查 GPEx_STS 寄存器和 GPEx_EN 寄存器以确定处理器系统中存在的 GPE 事件，然后调用 acpi_ev_gpe_dispatch 函数执行源代码 14-14 中 Method(_L00)之后的程序。

acpi_ev_gpe_dispatch 函数在执行源代码 14-14 中的 ASL 程序时，采用解释执行的方法。而解释执行相比编译执行而言，执行效率较低，为此该函数调用 acpi_os_execute 函数[㊀]，使用 Linux 系统提供的 Work Queue 机制，脱离中断处理程序的上下文环境，“异步”地分析并解释执行这些 ASL 程序。

㊀ 该函数为 ACPICA 提供的 OS 服务层接口函数。

在 Linux 系统中，与 ACPI 机制相关的程序虽然数量众多，处理的事务也较多，但是其逻辑结构较为简单，本章对此不做进一步分析和说明。

## 14.3 基于 ACPI 机制的 Linux PCI 的初始化

本节重点介绍 Linux 系统如何使用 ACPI 机制，对 PCI 总线树进行枚举。Linux 的 ACPI 系统的初始化较为复杂。本节重点介绍与 PCI 总线相关的一些基本模块，并不会介绍与 ACPI 系统初始化相关的全部内容。

在 Linux 系统中，ACPI 系统的初始化由两部分组成，一部分由 start_kernel→setup_arch 函数执行，另一部分作为模块由 do_initcalls 函数执行。

### 14.3.1 基本的准备工作

setup_arch 函数将分别调用 acpi_boot_table_init、early_acpi_boot_init 和 acpi_boot_init 函数完成 ACPI 系统的初始化，这几个函数的源代码在 ./arch/x86/kernel/acpi/boot.c 文件中。

acpi_boot_table_init 函数调用 acpi_table_init 函数在内存中找到 RSDP 和 RSDT/XSDT，从而定位 ACPI 表。BIOS 在系统初始化时将 ACPI 表放到一块固定物理地址区域中；early_acpi_boot_init 函数调用 early_acpi_process_madt 函数进一步处理 MADT；而 acpi_boot_init 函数依次分析 SBFT[㊀]（Simple Boot Flag Table）、FADT 和 HPET（IA-PC High Precision Event Timer Table），其中 HPET 是 Intel 定义的一个高精度定时器。

setup_arch 函数执行完毕后，Linux 系统将调用 do_initcalls 函数执行与 ACPI 系统相关的一些模块，其中与 PCI 总线有关的模块有 acpi_pci_init、acpi_pci_root_init 和 acpi_pci_link_init 函数。这些函数的说明如下。

1. acpi_pci_init 函数

acpi_pci_init 函数的执行过程较为简单，该函数在 ./drivers/pci/pci-acpi.c 文件中，如源代码 14-19 所示。

**源代码 14-19　acpi_pci_init 函数**

```
static int _init acpi_pci_init(void)
{
    int ret;

    if (acpi_gbl_FADT.boot_flags & ACPI_FADT_NO_MSI) {
        printk(KERN_INFO"ACPI FADT declares the system doesn't support MSI,
                so disable it\n");
        pci_no_msi();
    }

    if (acpi_gbl_FADT.boot_flags & ACPI_FADT_NO_ASPM) {
        printk(KERN_INFO"ACPI FADT declares the system doesn't support PCIe
```

㊀ 该表由 Microsoft 定义，详情见 http://www.microsoft.com/whdc/resources/respec/specs/simp_boot.mspx。

```
                    ASPM, so disable it\n");
            pcie_no_aspm();
        } ret = register_acpi_bus_type(&acpi_pci_bus);
        if (ret)
            return 0;
        pci_set_platform_pm(&acpi_pci_platform_pm);
        return 0;
    }
    arch_initcall(acpi_pci_init);
```

该函数首先分析“Boot Architecture Flags”字段，确定当前处理器系统是否需要使能 MSI 中断机制和 PCIe 设备的 ASPM(Active State Power Management)机制，ASPM 机制的详细描述见第 8.3 节，而 MSI 机制的详细说明见第 10 章。该函数调用 register_acpi_bus_type 函数，将 acpi_pci_bus 结构加入到全局链表 bus_type_list，最后调用 pci_set_platform_pm 函数将全局变量 pci_platform_pm 赋值为 acpi_pci_platform_pm。

### 2. acpi_pci_root_init 函数

acpi_pci_root_init 函数调用 acpi_pci_root_add 和 acpi_pci_root_start 函数遍历处理器系统中的 PCI 总线树。在 Linux 系统中，acpi_pci_root_init 函数的调用关系较为复杂，本节仅介绍其调用过程，并不详细介绍其实现机制。

acpi_pci_root_init 函数的调用过程如源代码 14-20 所示。

**源代码 14-20　acpi_pci_root_init 函数的调用过程**

```
acpi_pci_root_init -> acpi_bus_register_driver -> driver_register
 -> bus_add_driver -> driver_attach -> _driver_attach
 -> driver_probe_device -> really_probe -> (dev -> bus -> probe)
```

由以上过程可见 acpi_pci_root_init 函数将调用 really_probe 函数中的(dev -> bus -> probe)函数，而 dev -> bus -> probe 函数在 acpi_device_register 函数中被赋值为 acpi_device_probe 函数。

acpi_device_probe 函数又经过了一系列复杂的调用，最终调用 acpi_pci_root_add 和 acpi_pci_root_start 函数，其调用过程如源代码 14-21 所示。

**源代码 14-21　acpi_pci_root_init 函数的调用过程**

```
acpi_device_probe
    |----> acpi_bus_driver_init
    |        |----> driver -> ops. add
    |----> acpi_start_single_object
             |----> driver -> ops. start
```

其中 driver -> ops. add 函数与 acpi_pci_root_add 函数对应；而 driver -> ops. start 函数与 acpi_pci_root_start 函数对应。acpi_pci_root_add 函数在 ./drivers/acpi/pci_root. c 文件中，该函数的主要功能是遍历 PCI 总线树，如源代码 14-22 ~ 23 和源代码 14-31 所示。

源代码 14-22　acpi_pci_root_add 函数片段 1

```
static int _devinit acpi_pci_root_add(struct acpi_device *device)
{
    unsigned long long segment, bus;
    acpi_status status;
    int result;
    struct acpi_pci_root *root;
    acpi_handle handle;
    struct acpi_device *child;
    u32 flags, base_flags;

    segment = 0;
    status = acpi_evaluate_integer(device->handle, METHOD_NAME__SEG, NULL,
                &segment);
    if (ACPI_FAILURE(status) && status != AE_NOT_FOUND) {
        printk(KERN_ERR PREFIX "can't evaluate _SEG\n");
        return -ENODEV;
    }

    /* Check _CRS first, then _BBN. If no _BBN, default to zero. */
    bus = 0;
    status = try_get_root_bridge_busnr(device->handle, &bus);
    if (ACPI_FAILURE(status)) {
        status = acpi_evaluate_integer(device->handle, METHOD_NAME__BBN,
                NULL,&bus);
        if (ACPI_FAILURE(status) && status != AE_NOT_FOUND) {
            printk(KERN_ERR PREFIX
                "no bus number in _CRS and can't evaluate _BBN\n");
            return -ENODEV;
        }
    }

    root = kzalloc(sizeof(struct acpi_pci_root), GFP_KERNEL);
    if (!root)
      return -ENOMEM;

    INIT_LIST_HEAD(&root->node);
    root->device = device;
    root->segment = segment & 0xFFFF;
    root->bus_nr = bus & 0xFF;
    strcpy(acpi_device_name(device), ACPI_PCI_ROOT_DEVICE_NAME);
    strcpy(acpi_device_class(device), ACPI_PCI_ROOT_CLASS);
    device->driver_data = root;

    /*
     * All supported architectures that use ACPI have support for
```

```
         * PCI domains, so we indicate this in _OSC support capabilities.
         */
        flags = base_flags = OSC_PCI_SEGMENT_GROUPS_SUPPORT;
        acpi_pci_osc_support(root, flags);

        /*
         * TBD: Need PCI interface for enumeration/configuration of roots.
         */

        /* TBD: Locking */
        list_add_tail(&root->node, &acpi_pci_roots);

        printk(KERN_INFO PREFIX "%s [%s] (%04x:%02x)\n",
            acpi_device_name(device), acpi_device_bid(device),
            root->segment, root->bus_nr);
```

这段代码通过 ACPI 表中的_SEG 和_BBN 参数获得 HOST 主桥使用的 Segment 和 Bus 号，创建一个 acpi_pci_root 结构，并对该结构进行初始化，随后将 acpi_pci_root 结构加入到 acpi_pci_roots 队列中。acpi_pci_root 结构的主要功能是对当前 HOST 主桥控制器进行描述，而在 acpi_pci_roots 队列中包含当前 x86 处理器系统所有 HOST 主桥[⊖]的信息。

当 x86 处理器系统中只有一个 HOST 主桥时，acpi_pci_root_add 函数仅会被 Linux 调用一次，此时 acpi_pci_roots 队列中只有一个数据成员，即 root，其 Segment 和 Bus 号均为 0；如果存在多个 HOST 主桥时，acpi_pci_root_add 函数将在 PCI 总线初始化时被调用多次，并将所有主桥信息加入到 acpi_pci_roots 队列中。

这段代码还将 HOST 主桥的_OSC 参数的 PCI Segment Groups supported 位设置为 1，该参数在 ACPI 规范中定义，该位为 1 时表示当前处理器系统支持 PCI Segment Group。

**源代码 14-23　acpi_pci_root_add 函数片段 2**

```
    /*
     * Scan the Root Bridge
     * --------------------
     * Must do this prior to any attempt to bind the root device, as the
     * PCI namespace does not get created until this call is made (and
     * thus the root bridge's pci_dev does not exist).
     */
    root->bus = pci_acpi_scan_root(device, segment, bus);
    if (!root->bus) {
        printk(KERN_ERR PREFIX
            "Bus %04x:%02x not present in PCI namespace\n",
            root->segment, root->bus_nr);
        result = -ENODEV;
        goto end;
    }
```

⊖ Itanium 处理器系统含有多个对等 HOST 主桥；而在多数 x86 处理器系统中，仅含有一个 HOST 主桥。

在一个 x86 处理器系统中，如果没有使能 ACPI 机制，则 Linux 系统调用 pci_legacy_init→pcibios_scan_root 函数枚举 PCI 设备。如果 Linux 系统使能了 ACPI 机制，则由这段程序调用 pci_acpi_scan_root 函数完成 PCI 设备的枚举。pci_acpi_scan_root 和 pcibios_scan_root 函数对 PCI 总线树的枚举过程类似。

pci_acpi_scan_root 函数在 ./arch/x86/pci/acpi.c 文件中，如源代码 14-24 所示。

**源代码 14-24　pci_acpi_scan_root 函数**

```
struct pci_bus * __devinit
pci_acpi_scan_root(struct acpi_device *device, int domain, int busnum)
{
    struct pci_bus *bus;
    struct pci_sysdata *sd;
    int node;
...
    /* Allocate per-root-bus (not per bus) arch-specific data.
     * TODO: leak; this memory is never freed.
     * It's arguable whether it's worth the trouble to care.
     */
    sd = kzalloc(sizeof(*sd), GFP_KERNEL);
    if (!sd) {
        printk(KERN_ERR "PCI: OOM, not probing PCI bus %02x\n", busnum);
        return NULL;
    }

    sd->domain = domain;
    sd->node = node;
    /*
     * Maybe the desired pci bus has been already scanned. In such case
     * it is unnecessary to scan the pci bus with the given domain,busnum.
     */
    bus = pci_find_bus(domain, busnum);
    if (bus) {
        /*
         * If the desired bus exits, the content of bus->sysdata will
         * be replaced by sd.
         */
        memcpy(bus->sysdata, sd, sizeof(*sd));
        kfree(sd);
    } else {
        bus = pci_create_bus(NULL, busnum, &pci_root_ops, sd);
        if (bus) {
            if (pci_probe & PCI_USE__CRS)
                get_current_resources(device, busnum, domain,bus);
```

```
                bus -> subordinate = pci_scan_child_bus(bus);
            }
        }
...
        return bus;
}
```

这段代码首先判断当前总线号是否已经存在，如果存在说明这条总线已经被遍历过，该函数将直接退出。否则将首先调用 pci_create_bus 函数，pci_create_bus 函数的源代码在 ./drivers/pci/probe.c 文件中，其主要作用是为当前 HOST 主桥创建 pci_bus 结构，并初始化这个 pci_bus 结构的部分参数如 resource[0/1]，secondary 参数[㊀]等，然后将这个 pci_bus 结构加入到全局链表 pci_root_buses 中，最后进行一些与 sysfs 相关的初始化工作。

之后调用 pci_scan_child_bus 函数对当前 PCI 总线上的设备进行枚举，pci_scan_child_bus 函数将完成对 PCI 总线树的枚举操作，该函数是 Linux 遍历 PCI 总线树的要点，下一节将专门介绍讨论该函数的实现机制。

### 14.3.2 Linux PCI 初始化 PCI 总线号

PCI 总线树的枚举由 pci_scan_child_bus 函数完成，该函数的主要作用是分配 PCI 总线树的 PCI 总线号，而并不初始化 PCI 设备使用的 BAR 空间。

pci_scan_child_bus 函数在第一次执行时[㊁]，首先遍历当前 HOST 主桥之下所有的 PCI 设备，如果在 HOST 主桥下含有 PCI 桥，将再次遍历这个 PCI 桥下的 PCI 设备。并以此递归，直到将当前 PCI 总线树遍历完毕，并返回当前 HOST 主桥的 subordinate 总线号。subordinate 总线号记载当前 PCI 总线树中最后一个 PCI 总线号，因此只有完成了对 PCI 总线树的枚举后，才能获得该参数。pci_scan_child_bus 函数如源代码 14-25 和源代码 14-29 所示。

**源代码 14-25　pci_scan_child_bus 函数片段 1**

```
unsigned int _devinit pci_scan_child_bus(struct pci_bus *bus)
{
    unsigned int devfn, pass, max = bus -> secondary;
    struct pci_dev *dev;

    pr_debug("PCI: Scanning bus %04x:%02x\n", pci_domain_nr(bus),
            bus -> number);

    /* Go find them, Rover! */
    for (devfn = 0; devfn < 0x100; devfn += 8)
        pci_scan_slot(bus, devfn);
...
```

㊀ resource 参数存放 HOST 主桥管理的存储器和 I/O 地址空间，secondary 参数存放 Secondary 总线号。

㊁ Linux PCI 将递归调用 pci_scan_child_bus 函数。

该函数首先调用 pci_scan_slot 函数，扫描当前 PCI 总线的所有设备，并将其加入到对应总线的设备队列中。在 pci_scan_bus_parented 函数调用 pci_scan_child_bus 函数时，其输入参数为 HOST 主桥的 pci_bus 结构，此时 pci_scan_slot 函数首先初始化与 HOST 主桥直接相连的 PCI 设备，即 Bus 号为 0 的 PCI 设备。

### 1. pci_scan_slot 函数

一条 PCI 总线上最多有 32 个设备，每个设备最多有 8 个 Function。pci_scan_child_bus 函数需要枚举每一个可能存在的 Function。因此对于一条 PCI 总线，pci_scan_child_bus 函数需要调用 0x100 次 pci_scan_slot 函数。而 pci_scan_slot 函数调用 pci_scan_single_device 函数配置对当前 PCI 总线上的所有 PCI 设备。

pci_scan_single_device 函数进一步调用了 pci_scan_device 和 pci_device_add 函数。其中 pci_scan_device 函数主要对 PCI 设备的配置寄存器进行读写操作，侧重于 PCI 设备进行硬件层面的初始化操作，而 pci_device_add 函数侧重于软件层面的初始化。pci_scan_device 函数如源代码 14-26 所示。

**源代码 14-26　pci_scan_device 函数**

```
static struct pci_dev * pci_scan_device(struct pci_bus * bus, int devfn)
{
    struct pci_dev * dev;
    u32 l;
    int delay = 1;

    if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
        return NULL;

    /* some broken boards return 0 or ~0 if a slot is empty: */
    if (l == 0xffffffff || l == 0x00000000 ||
        l == 0x0000ffff || l == 0xffff0000)
        return NULL;

    /* Configuration request Retry Status */
    while (l == 0xffff0001) {
        msleep(delay);
        delay * = 2;
        if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
            return NULL;
        /* Card hasn't responded in 60 seconds? Must be stuck. */
        if (delay > 60 * 1000) {
            printk(KERN_WARNING "pci %04x:%02x:%02x.%d: not "
                    "responding\n", pci_domain_nr(bus),
                    bus->number, PCI_SLOT(devfn),
                    PCI_FUNC(devfn));
            return NULL;
        }
    }
```

```
        dev = alloc_pci_dev( );
        if ( ! dev)
            return NULL;

        dev -> bus = bus;
        dev -> devfn = devfn;
        dev -> vendor = l & 0xffff;
        dev -> device = (l >> 16) & 0xffff;

        if (pci_setup_device(dev)) {
            kfree(dev);
            return NULL;
        }

        return dev;
    }
```

pci_scan_device 函数首先读取 PCI 设备的 Vendor ID 和 Header Type 寄存器，并根据这两个寄存器的内容对 PCI 设备进行完整性检查，之后创建 pci_dev 结构，并对该结构进行基本的初始化。

set_pcie_port_type 函数的主要作用是处理 PCI Express Extended Capabilities 结构，并将其保存在 pci_dev→pcie_type 参数中，该结构的详细描述见第 4.3.2 节。值得注意的是，在 Linux 系统中，许多 PCIe 设备并没有提供该结构。在这段源代码的最后将调用 pci_setup_device 函数，其实现如源代码 14-27 所示。

**源代码 14-27　pci_setup_device 函数**

```
    static int pci_setup_device(struct pci_dev * dev)
    {
        u32 class;

    ...
        switch (dev -> hdr_type) {                  /* header type */
        case PCI_HEADER_TYPE_NORMAL:        /* standard header */
            if (class == PCI_CLASS_BRIDGE_PCI)
                goto bad;
            pci_read_irq(dev);
            pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
            pci_read_config_word(dev,
                    PCI_SUBSYSTEM_VENDOR_ID, &dev -> subsystem_vendor);
            pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev -> subsystem_device);
    ...
            }
            break;
        case PCI_HEADER_TYPE_BRIDGE:        /* bridge header */
```

```
            if (class != PCI_CLASS_BRIDGE_PCI)
                goto bad;
            /* The PCI-to-PCI bridge spec requires that subtractive
               decoding (i.e. transparent) bridge must have programming
               interface code of 0x01. */
            pci_read_irq(dev);
            dev->transparent = ((dev->class & 0xff) == 1);
            pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
            break;

        case PCI_HEADER_TYPE_CARDBUS:           /* CardBus bridge header */
            if (class != PCI_CLASS_BRIDGE_CARDBUS)
                goto bad;
            pci_read_irq(dev);
            pci_read_bases(dev, 1, 0);
            pci_read_config_word(dev,
                    PCI_CB_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
            pci_read_config_word(dev,
                    PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
            break;
...
        }
        return 0;
    }
```

pci_setup_device 函数首先根据 Header Type 寄存器，判断当前 PCI 设备是 PCI Agent 设备、PCI 桥还是 Card Bus。PCI Agent 设备使用的配置空间与 PCI 桥所使用的配置空间并不相同，因此 Linux PCI 需要区别处理这两种配置空间。本节忽略 Card Bus 的处理过程。

pci_setup_device 函数需要调用 pci_read_irq 和 pci_read_bases 函数访问 PCI 设备的配置空间，并进一步初始化 pci_dev 结构的其他参数。

pci_read_irq 函数的主要作用是读取 PCI 设备配置空间的 Interrupt Pin 和 Interrupt Line 寄存器，并将结构赋值到 pci_dev→pin 和 irq 参数中。其中 pin 参数记录当前 PCI 设备使用的中断引脚，而 irq 参数存放系统软件使用的 irq 号。

值得注意的是，在 pci_setup_devic 函数中初始化的 pci_dev→irq 参数并不一定是 PCI 设备驱动程序在 request_irq 函数中使用的 irq 入口参数。如果当前 Linux x86 系统使用了 I/O A-PIC 控制器时，Linux 设备驱动程序调用 pci_enable_device 函数将会改变 pci_dev→irq 参数，详见第 15.1.1 节。

而如果 PCIe 设备使能了 MSI/MSI-X 中断处理机制，pci_dev→irq 参数在设备驱动程序调用 pci_enable_msi/pci_enable_msix 函数后也将会发生变化，详见第 15.2 节。只有 x86 处理器使用 8259A 中断控制器处理 PCI 设备的中断请求时，pci_dev→irq 参数才与 Interrupt Line 寄存器中的值一致。

pci_read_bases 函数访问 PCI 设备的 BAR 空间和 ROM 空间，并初始化 pci_dev→resource

参数。在第 12.3.2 节 Capric 卡的初始化中使用的 pci_resource_start 和 pci_resource_len 函数就是从 pci_dev→resource 参数中获得 BAR 空间使用的基地址与长度。

这里有一个细节需要提醒读者注意，在 pci_dev→resource 参数中存放的 BAR 空间的基地址属于存储器域，而在 PCI 设备的 BAR 寄存器中存放的基地址属于 PCI 总线域。在 x86 处理器中，这两个值虽然相同，但是所代表的含义不同。

pci_read_bases 函数调用_pci_read_base 函数对 pci_dev→resource 参数进行初始化，_pci_read_base 函数的实现方式如源代码 14-28 所示。

**源代码 14-28　_pci_read_base 函数**

```
int _pci_read_base(struct pci_dev *dev, enum pci_bar_type type,
                   struct resource *res, unsigned int pos)
{
    u32 l, sz, mask;

    mask = type ? ~PCI_ROM_ADDRESS_ENABLE : ~0;

    res->name = pci_name(dev);

    pci_read_config_dword(dev, pos, &l);
    pci_write_config_dword(dev, pos, mask);
    pci_read_config_dword(dev, pos, &sz);
    pci_write_config_dword(dev, pos, l);
...
    if (type == pci_bar_mem64) {
...
    } else {
        sz = pci_size(l, sz, mask);

        if (!sz)
            goto fail;

        res->start = l;
        res->end = l + sz;
...
    }
out:
    return (type == pci_bar_mem64) ? 1 : 0;
fail:
    res->flags = 0;
    goto out;
}
```

_pci_read_base 函数的实现较为简单，本节仅介绍该函数获取 BAR 空间长度的方法。PCI 总线规范规定了获取 BAR 空间的标准实现方法。其步骤是首先向 BAR 寄存器写全 1，之后再读取 BAR 寄存器的内容，即可获得 BAR 空间的大小。

我们以 Capric 卡为例说明该过程，由上文所示 Capric 卡的 BAR0 空间为不可预读的存储器空间，大小为 0x10000 字节。这个设备在被初始化之前，其 BAR0 寄存器的值由硬件预置，

其值为0xFFFF-0000，其中BAR0寄存器的第15～0位只读，其15～4字段为0表示所申请的空间大小为64 KB；第3位为0表示不可预读；第2～1字段为0x00表示BAR0空间必须映射到PCI总线域的32位地址空间中；第0位为0表示为存储器空间。

当系统初始化完毕后，将BAR0寄存器重新进行赋值，其值为PCI总线域的地址，如0x9030-0000。当软件对这个寄存器写入“～0x0”之后，该寄存器的值将变为0xFFFF-0000，因为最后16位只读。采用此方法可以获得Capric卡BAR0空间的大小。在Linux系统中，可以使用pci_size函数将0xFFFF-0000转换为BAR0空间使用的实际大小，即64 KB。这段程序在获得BAR空间的基地址和长度后，继续判断当前BAR空间为64位PCI总线地址空间，还是32位PCI总线地址空间。为简化程序，本节仅列出处理“32位PCI总线地址这种情况”的源代码。

如果是当前PCI设备使用32位地址空间，则这段程序将初始化pci_dev→resource的start和end参数；如果是64位地址空间，该函数也需要初始化pci_dev→resource的start和end参数，只是过程稍微复杂。这段代码留给读者分析。

细心的读者在分析_pci_read_base函数后，会对“pci_read_config_dword(dev, pos, &l)”语句产生疑问。因为从Linux PCI的初始化过程，我们并没有发现处理器何时将PCI设备的BAR寄存器初始化，此时读到变量l的究竟是什么数值？

在x86处理器系统中，虽然Linux PCI并没有对PCI设备的BAR空间进行初始化操作，但是BIOS已经完成了对PCI总线树的枚举过程，因此变量l将保存有效的BAR空间基地址。对于其他处理器体系，负责初始化引导的Firmware可能并没有实现PCI总线树的枚举[㊀]，此时变量l将保存PCI设备的硬件复位值。

无论对于哪种处理器系统，执行_pci_read_base函数总能获得正确BAR空间的大小。但是如果有些处理器系统的Firmware没有对PCI总线树进行枚举时，PCI设备的BAR空间中仅为上电复位值。在这些处理器系统中，_pci_read_base函数执行完毕后，在pci_dev→resource中保存的start和end参数仅是PCI设备从E2PROM中获得的初始值。

### 2. pci_scan_bridge函数

再次回到pci_scan_child_bus函数，分析剩余的程序，如源代码14-29所示。

**源代码14-29　pci_scan_child_bus函数片段2**

```
...
    /*
     * After performing arch-dependent fixup of the bus, look behind
     * all PCI-to-PCI bridges on this bus.
     */
    if (!bus->is_added) {
        pr_debug("PCI: Fixups for bus %04x:%02x\n",
                pci_domain_nr(bus), bus->number);
        pcibios_fixup_bus(bus);
        if (pci_is_root_bus(bus))
            bus->is_added = 1;
    }
```

㊀ 这些处理器的Linux系统，将在pcibios_assign_resources函数中初始化BAR空间，详见下文。

```
        for (pass = 0; pass < 2; pass++)
            list_for_each_entry(dev, &bus->devices, bus_list) {
                if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
                    dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
                    max = pci_scan_bridge(bus, dev, max, pass);
            }

        /*
         * We've scanned the bus and so we know all about what's on
         * the other side of any bridges that may be on this bus plus
         * any devices.
         *
         * Return how far we've got finding sub-buses.
         */
        pr_debug("PCI: Bus scan for %04x:%02x returning with max = %02x\n",
            pci_domain_nr(bus), bus->number, max);
        return max;
    }
```

pci_scan_child_bus 函数执行完毕 pci_scan_slot 函数后，将首先调用 pcibios_fixup_bus 函数。pcibios_fixup_bus 函数的主要目的是为一些 PCI 设备中的 errata 提供 work-around，但是在该函数中还含有一个非常重要的函数，即 pci_read_bridge_bases 函数。

因为历史原因 pci_read_bridge_bases 函数一直存在于 pcibios_fixup_bus 函数中，但是这个函数更应该直接放入到 pci_scan_child_bus 函数中。pci_read_bridge_bases 函数将读取当前 PCI 桥[㊀]的 I/O Limit、I/O Base、Memory Limit、Memory Base、Prefetchable Memory Limit 和 Prefetchable Memory Base 寄存器，并根据这些寄存器的值，初始化 pci_bus→resource 参数，该参数存放当前 PCI 桥所能管理的地址空间。

之后 pci_scan_child_bus 函数将调用 pci_scan_bridge 函数处理当前 PCI 总线上所挂接的 PCI 桥，并初始化在这个桥片 Secondary PCI 总线上的 PCI 设备。值得注意的是 pci_scan_bridge 函数被调用了两次，一次 pass 参数等于 0，另外一次 pass 参数等于 1。

在一个处理器系统中，有些负责初始化引导的 Firmware 可能已经完成对 PCI 总线树的枚举操作，而有些 Firmware 没有做这样的操作。当 pass 参数等于 0 时，pci_scan_bridge 函数处理“已完成枚举”的 PCI 桥；当 pass 参数等于 1 时，pci_scan_bridge 函数处理“尚未完成枚举”的 PCI 桥。对于 x86 处理器系统而言，BIOS 将预先对 PCI 总线树进行枚举；而对于其他处理器系统，如 PowerPC 处理器系统，U-Boot 并没有进行这个枚举操作；当然还存在一种可能，就是 Firmware 完成了部分枚举。无论是哪种情况，通过两次调用 pci_scan_bridge 函数，都将完成对处理器系统中所有 PCI 桥的处理。

在 Linux PCI 中有许多函数都是通用函数，即各类处理器系统都需要使用的函数，这些

---

㊀ 如果当前 PCI 桥为 HOST 主桥，pci_read_bridge_bases 函数将直接返回。因为 HOST 主桥使用的 pci_bus 结构已经在 pci_create_bus 函数中进行了初始化。

通用函数给 Linux PCI 的设计带来了不小的麻烦。为不同的处理器平台开发通用架构，是对任何资深系统程序员的巨大考验。在 Linux PCI 中有许多这样的程序。pci_scan_bridge 函数是其中之一，该函数的主体实现如源代码 14-30 所示。

**源代码 14-30　pci_scan_bridge 函数**

```
int _devinit pci_scan_bridge(struct pci_bus *bus, struct pci_dev *dev, int max, int pass)
{
    struct pci_bus *child;
    int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);
    u32 buses, i, j = 0;
    u16 bctl;
    int broken = 0;

    pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
...
    if ((buses & 0xffff00) && !pcibios_assign_all_busses()
            && !is_cardbus && !broken) {
...
        if (pass)
            goto out;
        busnr = (buses >> 8) & 0xFF;

            goto out;
        busnr = (buses >> 8) & 0xFF;
...
        child = pci_find_bus(pci_domain_nr(bus), busnr);
        if (!child) {
            child = pci_add_new_bus(bus, dev, busnr);
            if (!child)
                goto out;
            child->primary = buses & 0xFF;
            child->subordinate = (buses >> 16) & 0xFF;
            child->bridge_ctl = bctl;
        }

        cmax = pci_scan_child_bus(child);
        if (cmax > max)
            max = cmax;
        if (child->subordinate > max)
            max = child->subordinate;
    } else {
        if (!pass) {
...
```

```
        if (! pass) {
            if (pcibios_assign_all_busses() || broken)
...
                pci_write_config_dword(dev, PCI_PRIMARY_BUS,
                                       buses & ~0xffffff);
            goto out;
        }

        /* Clear errors */
        pci_write_config_word(dev, PCI_STATUS, 0xffff);

        /* Prevent assigning a bus number that already exists.
         * This can happen when a bridge is hot-plugged */
        if (pci_find_bus(pci_domain_nr(bus), max+1))
            goto out;
        child = pci_add_new_bus(bus, dev, ++max);
        buses = (buses & 0xff000000)
            | ((unsigned int)(child->primary)     << 0)
            | ((unsigned int)(child->secondary)   << 8)
            | ((unsigned int)(child->subordinate) << 16);
...
        pci_write_config_dword(dev, PCI_PRIMARY_BUS, buses);

...
        child->subordinate = max;
        pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, max);
    }
...
out:
    pci_write_config_word(dev, PCI_BRIDGE_CONTROL, bctl);
    return max;
}
```

pci_scan_bridge 函数首先读取当前 PCI/HOST 主桥配置空间的第 21 ~ 18 字节，这段数据的描述如第 2.3 节所示。在这段数据中，依次存放 PCI 桥配置寄存器的 Secondary Latency Timer、Subordinate Bus Number、Secondary Bus Number 和 Primary Bus Number 寄存器。

这段程序通过判断 PCI 桥的 Subordinate 和 Secondary 总线号是否为 0，判断当前 PCI 桥是否已经被初始化。如果 Subordinate 或者 Secondary 总线号不为 0，则表示该 PCI 桥已经被 Firmware 遍历；如果为 0，表示没有被 Firmware 遍历。

如果当前 PCI 桥已经被 Firmware 遍历，即((bus & 0xffff00)...)的计算结果为 True 时，这段程序将继续判断 pass 参数，如果为 1 则跳出；否则这段程序将直接调用 pci_add_new_bus 函数为这个 PCI 桥创建 pci_bus 结构，然后递归调用 pci_scan_child_bus 函数初始化该 PCI 桥管理的 PCI 子树。当 pci_scan_child_bus 函数递归执行完毕后，这段程序将重新修正 pci_bus →subordinate 参数。

如果当前 PCI 桥没有被 Firmware 遍历，即((bus & 0xffff00)...)的计算结果为 False 时，这段程序将执行“else”分支，并首先判断 pass 参数是否为 0，如果为 0 则跳出；否则这段程

序将调用 pci_add_new_bus 函数为这个 PCI 桥创建并初始化 pci_bus 结构，同时还需要初始化 PCI 桥的 Subordinate Bus Number、Secondary Bus Number 和 Primary Bus Number 寄存器，之后这段程序也递归调用 pci_scan_child_bus 函数。当 pci_scan_child_bus 函数递归完毕后，重新修正 pci_bus→subordinate 参数。

### 3. acpi_pci_root_add 函数的剩余操作

当 pci_scan_bridge 函数执行完毕后，我们再次回到 acpi_pci_root_add 函数，如源代码 14-31 所示。

**源代码 14-31　acpi_pci_root_add 函数片段 3**

```
        result = acpi_pci_bind_root(device);
        if (result)
            goto end;
    ...
        status = acpi_get_handle(device -> handle, METHOD_NAME_PRT, &handle);
        if (ACPI_SUCCESS(status))
            result = acpi_pci_irq_add_prt(device -> handle, root -> bus);
    ...
        list_for_each_entry(child, &device -> children, node)
            acpi_pci_bridge_scan(child);

        /* Indicate support for various _OSC capabilities.  */
        if (pci_ext_cfg_avail(root -> bus -> self))
            flags | = OSC_EXT_PCI_CONFIG_SUPPORT;
        if (pcie_aspm_enabled())
            flags | = OSC_ACTIVE_STATE_PWR_SUPPORT |
                OSC_CLOCK_PWR_CAPABILITY_SUPPORT;
        if (pci_msi_enabled())
            flags | = OSC_MSI_SUPPORT;
        if (flags ! = base_flags)
            acpi_pci_osc_support(root, flags);

        return 0;

    end:
        if (! list_empty(&root -> node))
            list_del(&root -> node);
        kfree(root);
        return result;
    }
```

这段代码首先调用 acpi_pci_bind_root 函数绑定 acpi_device 与 pci_bus 结构。该函数还将 acpi_device→ops.bind 和 ops.unbind 参数分别赋值为 acpi_pci_bind 和 acpi_pci_unbind。然后这段代码调用 acpi_pci_irq_add_prt 和 acpi_pci_bridge_scan 函数分析当前处理器系统的中断路由表，这部分内容将在第 15.1.2 节介绍。

这段代码在 pcie_aspm_enabled、pci_msi_enabled 函数成功返回后将 HOST 主桥的_OSC 参数的“MSI supported 位”和“Active State Power Management supported”位设置 1。

### 4. acpi_pci_root_start 函数

acpi_pci_root_add 函数执行完毕后，Linux x86 将调用 acpi_pci_root_start 函数。该函数首先扫描 acpi_pci_roots 链表，并调用 pci_bus_add_devices 函数处理这个链表中的每一个 HOST 主桥。pci_bus_add_devices 函数在 ./driver/pci/bus.c 文件中，其实现如源代码 14-32 所示。

**源代码 14-32　pci_bus_add_devices 函数**

```
void pci_bus_add_devices(const struct pci_bus *bus)
{
    struct pci_dev *dev;
    struct pci_bus *child;
    int retval;

    list_for_each_entry(dev, &bus->devices, bus_list) {
        /* Skip already-added devices */
        if (dev->is_added)
            continue;
        retval = pci_bus_add_device(dev);
        if (retval)
            dev_err(&dev->dev, "Error adding device, continuing\n");
    }

    list_for_each_entry(dev, &bus->devices, bus_list) {
        BUG_ON(!dev->is_added);

        child = dev->subordinate;
...
        if (!child)
            continue;
        if (list_empty(&child->node)) {
            down_write(&pci_bus_sem);
            list_add_tail(&child->node, &dev->bus->children);
            up_write(&pci_bus_sem);
        }
        pci_bus_add_devices(child);
...
        if (child->is_added)
            continue;
        retval = pci_bus_add_child(child);
        if (retval)
            dev_err(&dev->dev, "Error adding bus, continuing\n");
    }
}
```

这段代码首先调用 pci_bus_add_device 函数，将当前 PCI 总线(pci_bus 结构)上的所有 PCI 设备的相关信息(pci_dev 结构)加入到 proc 和 sysfs 文件系统中。

之后这段代码递归调用 pci_bus_add_devices 函数遍历当前 PCI 总线上所有 PCI 子桥。这段代码最后调用 pci_bus_add_child 函数初始化 PCI 子桥 pci_bus 结构的 dev.parent 参数，并将一些相关信息加入到 sysfs 文件系统中。

当 acpi_pci_root_start 函数返回后，acpi_pci_root_init 函数将执行完毕。Linux 系统将继续调用 acpi_pci_link_init 函数进一步初始化 PCI 总线，该函数与 PCI 总线的中断路由相关，在第 15.1.3 节将详细介绍该函数的实现。

### 14.3.3 Linux PCI 检查 PCI 设备使用的 BAR 空间

当 acpi_pci_link_init 函数执行完毕后，Linux PCI 开始执行 pci_subsys_init 函数。在第 14.1.3 节曾简要介绍了该函数的实现，该函数如源代码 14-9 所示。

当一个处理器系统使能了 ACPI 机制，pci_subsys_init 函数的执行路径将会发生变化。该函数将首先执行 pci_acpi_init 函数，并跳过 pci_legacy_init 和 pcibios_irq_init 函数之后，执行 pcibios_init 函数。pci_acpi_init 函数的实现较为简单，其源代码在 ./arch/x86/pci/acpi.c 文件中，如源代码 14-33 所示。

**源代码 14-33　pci_acpi_init 函数**

```
int __init pci_acpi_init(void)
{
    struct pci_dev *dev = NULL;

    if (pcibios_scanned)
        return 0;

    if (acpi_noirq)
        return 0;

    printk(KERN_INFO "PCI: Using ACPI for IRQ routing\n");
    acpi_irq_penalty_init();
    pcibios_scanned++;
    pcibios_enable_irq = acpi_pci_irq_enable;
    pcibios_disable_irq = acpi_pci_irq_disable;

    if (pci_routeirq) {
...
        for_each_pci_dev(dev)
            acpi_pci_irq_enable(dev);
    }

    return 0;
}
```

该函数首先调用 acpi_irq_penalty_init 函数更新 acpi_irq_penalty 表，该函数与 Linux 系统使用的 IRQ Balance 技术相关，对此感兴趣的读者可以从 http://www.irqbalance.org 网站获得更多的信息，本书并不关心这部分内容。

这段程序将 pcibios_scanned 参数置 1，并将 pcibios_enable_irq 和 pcibios_disable_irq 参数初始化为 acpi_pci_irq_enable 和 acpi_pci_irq_disable。这也是 Linux 系统使能 ACPI 机制后，

Linux PCI 并不执行 pci_legacy_init[㊀]和 pcibios_irq_init[㊁]函数的原因。最后这段程序使用 acpi_pci_irq_enable 函数为当前 PCI 总线树上的所有 PCI 设备分配 irq 号。

如果当前处理器系统使能了 ACPI 机制，pci_acpi_init 函数执行后，pci_subsys_init 函数将执行 pcibios_init 函数。pcibios_init→pcibios_resource_survey 函数将检查当前处理器系统的所有 PCI 设备的 BAR 空间，该函数并不会操作 PCI 设备的 BAR 寄存器，而只是检查当前处理器系统中所有 PCI 设备的 pci_dev→resource 参数是否合法。

由第 14.3.2 节所示，pci_scan_slot 函数已经将 pci_dev→resource 参数进行基本的初始化工作，但是对于不同的处理器系统，resource→start 参数的值并不一定有效。

pcibios_resource_survey 函数在 ./arch/x86/pci/i386.c 文件中，如源代码 14-34 所示。

**源代码 14-34　pcibios_resource_survey 函数**

```
void _init pcibios_resource_survey(void)
{
    DBG("PCI: Allocating resources\n");
    pcibios_allocate_bus_resources(&pci_root_buses);
    pcibios_allocate_resources(0);
    pcibios_allocate_resources(1);

    e820_reserve_resources_late();
    /*
     * Insert the IO APIC resources after PCI initialization has
     * occured to handle IO APICS that are mapped in on a BAR in
     * PCI space, but before trying to assign unassigned pci res.
     */
    ioapic_insert_resources();
}
```

在 Linux x86 中，所有 PCI 总线树的根节点使用一个双向链表连接在一起，pci_root_buses 指向这个链表的起始地址。pcibios_allocate_bus_resources 函数使用 DFS 算法检查并分配 PCI 总线树中的所有 PCI 桥使用的系统资源，函数的源代码在 ./arch/x86/pci/i386.c 文件中，如源代码 14-35 所示。

**源代码 14-35　pcibios_allocate_bus_resources 函数**

```
static void _init pcibios_allocate_bus_resources(struct list_head *bus_list)
{
    struct pci_bus *bus;
    struct pci_dev *dev;
    int idx;
    struct resource *r;
    /* Depth-First Search on bus tree */
    list_for_each_entry(bus, bus_list, node) {
```

㊀ 如第 14.1.3 节所示，pci_legacy_init 函数在执行过程中需要检查 pcibios_scanned 参数，当该参数为 1 时，该函数将直接返回。

㊁ pcibios_irq_init 函数需要检查 pcibios_enable_irq 参数，如果该参数不为 NULL，该函数也将直接返回。

```
            if ((dev = bus -> self)) {
                for (idx = PCI_BRIDGE_RESOURCES;
                     idx < PCI_NUM_RESOURCES; idx ++) {
                    r = &dev -> resource[idx];
                    if (! r -> flags)
                        continue;
                    if (! r -> start ||
                        pci_claim_resource(dev, idx) < 0) {
...
                        r -> flags = 0;
                    }
                }
            }
            pcibios_allocate_bus_resources(&bus -> children);
        }
```

pcibios_allocate_bus_resources 函数首先遍历链表 pci_root_buses 中的所有 pci_bus 结构，之后调用 pci_claim_resource→pci_find_parent_resource 函数对 pci_bus 结构进行检查。pci_find_parent_resource 函数在 ./driver/pci/pci.c 文件中，如源代码 14-36 所示，该函数成功返回时，将获得当前 PCI 桥的上游 PCI 桥使用的 resource 参数。

**源代码 14-36　pci_find_parent_resource 函数**

```
struct resource *
pci_find_parent_resource(const struct pci_dev *dev, struct resource *res)
{
    const struct pci_bus *bus = dev -> bus;
    int i;
    struct resource *best = NULL;

    for(i = 0; i < PCI_BUS_NUM_RESOURCES; i ++) {
        struct resource *r = bus -> resource[i];
        if (! r)
            continue;
        if (res -> start && ! (res -> start >= r -> start && res -> end <= r -> end))
            continue; /* Not contained */
        if ((res -> flags ^ r -> flags) & (IORESOURCE_IO | IORESOURCE_MEM))
            continue; /* Wrong type */
        if (! ((res -> flags ^ r -> flags) & IORESOURCE_PREFETCH))
            return r; /* Exact match */
        if ((res -> flags & IORESOURCE_PREFETCH)
                && ! (r -> flags & IORESOURCE_PREFETCH))
            best = r; /* Approximating prefetchable by non-prefetchable */
    }
    return best;
}
```

pci_find_parent_resource 首先对 PCI 桥管理的地址空间进行检查。如图 3-2 所示，每一个 PCI 桥都管理一段 PCI 总线地址空间，而且这段地址空间必须隶属于上游 PCI 桥管理的地址空间，其中 PCI 桥 2 管理的地址空间隶属于 PCI 桥 1，而 PCI 桥 1 管理的地址空间隶属于 HOST 主桥，而且这些地址空间的类型需要一致。

之后这段代码检查上下游 PCI 桥的预读设置位，PCI 总线规定下游设备“不可预读空间”不能使用 PCI 桥的“可预读空间”；而下游设备“可预读空间”可以使用 PCI 桥的“不可预读空间”和“可预读空间”，下游设备的“可预读空间”优先使用 PCI 桥的“可预读空间”。

当完成这些检查后 pcibios_allocate_bus_resources → request_resource 函数将从上游 PCI 桥管理的地址空间中为当前 PCI 桥分配地址空间，如果该函数返回失败，则将 r→flags 参数置 0，标记资源没有被正确分配，这种情况可能是因为 BIOS 的 bug，也可能因为其他原因。之后 pcibios_allocate_bus_resources 函数将递归调用 pcibios_allocate_bus_resources 函数遍历其下游的 PCI 总线树。

我们再次回到 pcibios_resource_survey 函数，发现该函数分别使用两个不同的入口参数 0 和 1 调用了 pcibios_allocate_resources 函数。当入口参数为 0 时，pcibios_allocate_resources 函数为“在 BIOS 中已经启用了 PCI 设备”优先分配资源；当入口参数为 1 时，该函数为其他 PCI 设备分配资源。

该函数的实现较为简单，其主要过程依然是调用 pci_find_parent_resource 函数获得上游 PCI 桥管理的资源，并使用 request_resource 函数为当前 PCI 设备分配地址空间。值得注意的是，当入口参数为 0 时，pcibios_resource_survey 函数将暂时禁止 PCI 设备的 ROM 空间，ROM 空间的初始化将在下文介绍。

pcibios_init 函数主要操作 Linux 系统中的数据结构，并没有对 PCI 设备的 BAR 寄存器进行读写操作。在 x86 处理器系统中，BIOS 会枚举 PCI 总线树，并初始化 PCI 设备的 BAR 寄存器；但是在其他处理器系统中，Firmware 可能并没有做出这些操作，为此 Linux 系统将继续遍历 PCI 总线树，并初始化这些 PCI 设备的 BAR 寄存器。

### 14.3.4 Linux PCI 分配 PCI 设备使用的 BAR 寄存器

pci_subsys_init 函数执行完毕后，Linux PCI 将调用 pcibios_assign_resources 函数，设置 PCI 设备的 BAR 寄存器。pcibios_assign_resources 函数首先处理 PCI 设备的 ROM 空间，并进行资源分配，之后调用 pci_assign_unassigned_resources 函数设置 PCI 设备的 BAR 寄存器。该函数在 ./drivers/pci/setup - bus. c 文件中，如源代码 14-37 所示。

**源代码 14-37　pci_assign_unassigned_resources 函数**

```
void _init
pci_assign_unassigned_resources(void)
{
    struct pci_bus * bus;

    /* Depth first, calculate sizes and alignments of all
       subordinate buses. */
    list_for_each_entry(bus, &pci_root_buses, node) {
```

```
            pci_bus_size_bridges(bus);
        }
        /* Depth last, allocate resources and update the hardware. */
        list_for_each_entry(bus, &pci_root_buses, node) {
            pci_bus_assign_resources(bus);
            pci_enable_bridges(bus);
        }

        /* dump the resource on buses */
        list_for_each_entry(bus, &pci_root_buses, node) {
            pci_bus_dump_resources(bus);
        }
    }
```

该函数依次调用 pci_bus_size_bridges、pci_bus_assign_resources 和 pci_enable_bridges 函数，下文将分别讨论这些函数。而 pci_bus_dump_resources 函数的主要作用是将 Linux 系统分配的 PCI 设备的资源信息打印出来，本节对该函数不做介绍。

### 1. pci_bus_size_bridges 函数

pci_bus_size_bridges 函数的主要作用是修复和对界当前 PCI 总线树下的所有 PCI 设备（包括 PCI 桥）所使用的 I/O 和存储器地址空间。该函数的实现如源代码 14-38 所示。

**源代码 14-38　pci_bus_size_bridges 函数**

```
    void __ref pci_bus_size_bridges(struct pci_bus *bus)
    {
        struct pci_dev *dev;
        unsigned long mask, prefmask;

        list_for_each_entry(dev, &bus->devices, bus_list) {
            struct pci_bus *b = dev->subordinate;
...
            switch (dev->class >> 8) {
...
            case PCI_CLASS_BRIDGE_PCI:
            default:
                pci_bus_size_bridges(b);
                break;
            }
        }
        /* The root bus? */
        if (!bus->self)
            return;

        switch (bus->self->class >> 8) {
...
```

```
        case PCI_CLASS_BRIDGE_PCI:
            pci_bridge_check_ranges(bus);
        default:
            pbus_size_io(bus);
...
            mask = IORESOURCE_MEM;
            prefmask = IORESOURCE_MEM | IORESOURCE_PREFETCH;
            if (pbus_size_mem(bus, prefmask, prefmask))
                mask = prefmask; /* Success, size non-prefetch only. */
            pbus_size_mem(bus, mask, IORESOURCE_MEM);
            break;
        }
    }
```

这段代码首先递归调用 pci_bus_size_bridges 函数，直到找到当前 PCI 总线树最底层的 PCI 桥，然后调用 pci_bridge_check_ranges 函数检查这个 PCI 桥所管理的地址空间是否支持 I/O 或者可预读的存储器空间，如果支持，则将当前 PCI 桥的 pci_bus→self→resource 参数的相应状态位置 1。这段代码随后调用 pbus_size_io 和 pbus_size_mem 函数修复并对界当前 PCI 桥的 I/O 空间和存储器空间，并从低到高逐层递归调用 pci_bus_size_bridges 函数。

### 2. pci_bus_assign_resources 函数

pci_bus_assign_resources 函数在./drivers/pci/setup-bus.c 文件中，如源代码 14-39 和源代码 14-41 所示。

**源代码 14-39　pci_bus_assign_resources 函数片段 1**

```
void __ref pci_bus_assign_resources(const struct pci_bus *bus)
{
    struct pci_bus *b;
    struct pci_dev *dev;

    pbus_assign_resources_sorted(bus);
```

该函数首先调用 pbus_assign_resources_sorted 函数遍历并初始化当前 PCI 总线上的所有 PCI 设备的 BAR 寄存器，包括 PCI Agent 设备和 PCI 桥，之后递归调用自身遍历当前 PCI 总线的所有下游总线，最后调用 pci_setup_bridge 函数初始化 PCI 桥的存储器和 I/O Base、Limit 寄存器。

在第 14.3.2 中曾经使用 pci_scan_bridge 函数，将 PCI 桥的 Primary Bus Number、Secondary Bus Number 和 Subordinate Bus Number 寄存器初始化完毕，此时 Linux PCI 可以访问 HOST 主桥之下的所有 PCI 设备的配置空间，但是不能访问“未初始化的 PCI 设备”的 BAR 空间。PCI/PCIe 总线规定使用 ID 寻址方式访问配置空间，而使用地址寻址方式访问存储器空间，因此处理器虽然不能访问 BAR 空间，但是依然能够访问 PCI 设备的配置空间。

值得注意的是这段代码中的一个细节问题。其中 pbus_assign_resources_sorted 函数在 pci_bus_assign_resources 函数递归调用之前执行，而 pci_setup_bridge 函数在递归调用之后执行。Linux PCI 采用这种方式，可以保证 PCI 设备 BAR 寄存器初始化是从上游 PCI 总线到下游 PCI

总线，而 PCI 桥 Base、Limit 寄存器的初始化是从下游 PCI 总线到上游 PCI 总线。

这一细节对 PCI 设备的初始化非常重要，因为 PCI 桥所管理的地址空间是其下所有 PCI 设备使用地址空间的合集，因此 PCI 桥 Base、Limit 寄存器的初始化需要从下而上进行。而 PCI 设备的 BAR 寄存器的初始化的方向并没有严格规定，PCI 规范并没有对此做具体的要求，在实现中只要保证系统软件在初始化 PCI 桥的 Base、Limit 寄存器之前，其下所有 PCI 设备的 BAR 寄存器已经完成初始化即可。目前 Linux 系统使用从上游总线到下游总线的方法初始化 PCI 设备的 BAR 寄存器。

对于 x86 处理器系统，PCI 设备的 BAR 空间已经被 BIOS 初始化，因此只要 BIOS 正确分配了 PCI 设备的 BAR 寄存器，Linux 系统不执行 pci_assign_unassigned_resources 函数也没有什么关系。不过对于一些处理器系统，其 Firmware 并没有完全枚举 PCI 总线树上的 PCI 设备，此时必须调用 pci_assign_unassigned_resources 中的 pci_bus_assign_resources[0]函数初始化“未初始化 BAR 空间”的 PCI 设备。

pbus_assign_resources_sorted 函数负责分配“未初始化 PCI 设备的 BAR 寄存器”，该函数将对这些 PCI 设备的 BAR 寄存器进行写操作。该函数的实现如源代码 14-40 所示。

**源代码 14-40　pbus_assign_resources_sorted 函数**

```
static void pbus_assign_resources_sorted(const struct pci_bus *bus)
{
    struct pci_dev *dev;
    struct resource *res;
    struct resource_list head, *list, *tmp;
    int idx;

    head.next = NULL;
    list_for_each_entry(dev, &bus->devices, bus_list) {
        u16 class = dev->class >> 8;
...
        pdev_sort_resources(dev, &head);
    }

    for (list = head.next; list;) {
        res = list->res;
        idx = res - &list->dev->resource[0];
        if (pci_assign_resource(list->dev, idx)) {
            res->start = 0;
            res->end = 0;
            res->flags = 0;
        }
        tmp = list;
        list = list->next;
        kfree(tmp);
    }
}
```

这段代码首先调用 pdev_sort_resources 函数，该函数的实现过程较为简单，其主要作用是将“未初始化”的 PCI 设备使用的资源进行排序对齐，然后加入到 head 链表中，随后调用 pci_assign_resource 函数初始化这些 PCI 设备的 BAR 寄存器。

pci_assign_resource 函数在 ./drivers/pci/setup-res.c 文件中，该函数的实现逻辑较为简单，本节并不列出这段源代码。该函数两次调用了 pci_bus_alloc_resource 函数，第一次试图从上游总线的可预读存储器空间为当前 PCI 设备分配资源，第二次从“不可预读的存储器空间”分配资源。当资源分配成功后，pci_assign_resource→pci_update_resource 函数将初始化 PCI 设备的 BAR 寄存器，这些代码并不复杂，本节将这些代码留给读者。

**源代码 14-41　pci_bus_assign_resources 函数片段 2**

```
        list_for_each_entry(dev, &bus->devices, bus_list) {
            b = dev->subordinate;
            if (!b)
                continue;

            pci_bus_assign_resources(b);

            switch (dev->class >> 8) {
            case PCI_CLASS_BRIDGE_PCI:
                pci_setup_bridge(b);
                break;

            case PCI_CLASS_BRIDGE_CARDBUS:
                pci_setup_cardbus(b);
                break;

            default:
                dev_info(&dev->dev, "not setting up bridge for bus "
                    "%04x:%02x\n", pci_domain_nr(b), b->number);
                break;
            }
        }
    }
```

再次回到 pci_bus_assign_resources 函数，该函数开始递归调用自身，寻找当前 PCI 总线子树的最后一个 PCI 桥，之后调用 pci_setup_bridge 函数初始化这个 PCI 桥的 Base、Limit 寄存器。pci_setup_bridge 函数的源代码在 ./drivers/pci/set-bus.c 文件中，本节对此不作介绍。

当 Linux PCI 执行 pci_setup_bridge 函数初始化当前 PCI 桥之后，这个桥的上游设备和这个桥管理的 PCI 设备的 BAR 寄存器已经初始化完毕。因此 pci_setup_bridge 函数通过简单的计算，即可得出当前 PCI 桥 Base、Limit 寄存器的值，之后调用 pci_write_config_dword 函数将这个数据对 PCI 桥的这些寄存器更新即可。

至此，PCI 总线树上的所有 PCI 设备的 BAR 寄存器，以及 PCI 桥的 Base、Limit 寄存器全部初始化完毕，从硬件的角度来看，PCI 总线系统已经初始化完毕。

### 3. pci_enable_bridges 函数

我们再次回到 pci_assign_unassigned_resources 函数，如源代码 14-37 所示，该函数将调用 pci_enable_bridges 函数，使能所有 PCI 桥设备。pci_enable_bridges 函数的实现如源代码 14-42 所示。

**源代码 14-42　pci_enable_bridges 函数**

```
void pci_enable_bridges(struct pci_bus *bus)
{
    struct pci_dev *dev;
    int retval;

    list_for_each_entry(dev, &bus->devices, bus_list) {
        if (dev->subordinate) {
            if (!pci_is_enabled(dev)) {
                retval = pci_enable_device(dev);
                pci_set_master(dev);
            }
            pci_enable_bridges(dev->subordinate);
        }
    }
}
```

该函数的实现较为简单，分别调用 pci_enable_device、pci_set_master 函数启动当前 PCI 桥，之后递归调用 pci_enable_bridges 函数启动当前 PCI 桥下游的 PCI 桥。至此 Linux PCI 完成对当前 PCI 总线树的主要初始化工作。

## 14.4　Linux PowerPC 如何初始化 PCI 总线树

Linux PowerPC 初始化 PCI 总线树的步骤与 Linux x86 类似，也调用了一些 Linux 系统中与 PCI 总线相关的通用函数。但是 PowerPC 处理器使用的 HOST 主桥与 x86 处理器并不相同，因此 Linux PowerPC 初始化 PCI 总线树的过程与 Linux x86 有些差别。本节以 MPC8572 处理器为例，说明 Linux PowerPC 初始化 PCI 总线树的过程。

MPC8572 处理器共有三个 PCIe 总线控制器，其中每一个总线控制器都可以管理一个独立的 PCI 总线树。在每一个总线控制器中都包含一组独立的寄存器，MPC8572 处理器可以通过设置 Inbound 和 Outbound 寄存器，访问对应 PCI 总线树上所有 PCI 设备的配置空间。这组寄存器与 MPC8548 处理器提供的对应寄存器较为类似，详见第 2.2 节。

Linux PowerPC 在引入了 Open Firmware 机制[㊀]后，使用 dts 文件管理 PCI 总线控制器。MPC8572 处理器系统使用的 dts 文件为 ./arch/powerpc/boot/dts/mpc8572ds.dts 文件，其中与 PCI 总线控制器相关的部分如源代码 14-43 所示。

㊀ 该机制由 Sun Microsystems 引入，广泛应用于 Sun、Apple、IBM 和 Freescale 的非 x86 处理器系统中。

源代码 14-43　MPC8572 处理器系统使用的 dts 文件

```
        pci0: pcie@ffe08000 {
            compatible = "fsl,mpc8548-pcie";
            device_type = "pci";
            #interrupt-cells = <1>;
            #size-cells = <2>;
            #address-cells = <3>;
            reg = <0 0xffe08000 0 0x1000>;
            bus-range = <0 255>;
            ranges = <0x2000000 0x0 0x80000000 0 0x80000000 0x0 0x20000000
                      0x1000000 0x0 0x00000000 0 0xffc00000 0x0 0x00010000>;
    ...
        }
        pci1: pcie@ffe09000 {
            compatible = "fsl,mpc8548-pcie";
            device_type = "pci";
            #interrupt-cells = <1>;
            #size-cells = <2>;
            #address-cells = <3>;
            reg = <0 0xffe09000 0 0x1000>;
            bus-range = <0 255>;
            ranges = <0x2000000 0x0 0xa0000000 0 0xa0000000 0x0 0x20000000
                      0x1000000 0x0 0x00000000 0 0xffc10000 0x0 0x00010000>;
    ...
        }
        pci2: pcie@ffe0a000 {
            compatible = "fsl,mpc8548-pcie";
            device_type = "pci";
            #interrupt-cells = <1>;
            #size-cells = <2>;
            #address-cells = <3>;
            reg = <0 0xffe0a000 0 0x1000>;
            bus-range = <0 255>;
            ranges = <0x2000000 0x0 0xc0000000 0 0xc0000000 0x0 0x20000000
                      0x1000000 0x0 0x00000000 0 0xffc20000 0x0 0x00010000>;
    ...
        }
```

以上代码分别描述了 MPC8572 处理器系统的 3 个 PCIe 控制器，其使用的寄存器空间为 0xFFE08000 ~ 0xFFE08FFFF、0xFFE09000 ~ 0xFFE09FFFF 和 0xFFE0A000 ~ 0xFFE0AFFFF。这三个 PCIe 控制器分别管理 3 棵 PCI 总线树，其 PCI 总线的编号都为 0 ~ 255，在这个 dts 文件中还包含了 PCIe 控制器的其他信息，本节并不关心这些内容。

Linux PowerPC 在调用 setup_arch→mpc85xx_cds_setup_arch 函数时，分别初始化这三个 PCIe 控制器，该函数的实现如源代码 14-44 所示。

源代码 14-44　mpc85xx_cds_setup_arch 函数

```
static void __init mpc85xx_cds_setup_arch(void)
{
#ifdef CONFIG_PCI
    struct device_node * np;
#endif
...
#ifdef CONFIG_PCI
    for_each_node_by_type(np, "pci") {
        if (of_device_is_compatible(np, "fsl,mpc8540-pci") ||
            of_device_is_compatible(np, "fsl,mpc8548-pcie")) {
            struct resource rsrc;
            of_address_to_resource(np, 0, &rsrc);
            if ((rsrc.start & 0xfffff) == 0x8000)
                fsl_add_bridge(np, 1);
            else
                fsl_add_bridge(np, 0);
        }
    }

    ppc_md.pci_irq_fixup = mpc85xx_cds_pci_irq_fixup;
    ppc_md.pci_exclude_device = mpc85xx_exclude_device;
#endif
}
```

mpc85xx_cds_setup_arch 函数分析 mpc8572ds.dts 文件，并将“pci0”作为主 PCI 总线控制器，并调用 fsl_add_bridge(np, 1)函数进行初始化操作，“pci1”和“pci2”作为从 PCI 总线控制器调用 fsl_add_bridge(np, 0)函数进行初始化操作。在 MPC8572 处理器中，主 PCI 总线控制器需要处理 ISA 总线使用的存储器和 I/O 地址空间。

fsl_add_bridge 函数在./arch/powerpc/sysdev/fsl_pci.c 文件中，该函数的实现如源代码 14-45～26 所示。Linux PowerPC 使用 pci_controller 结构描述 HOST 主桥，包括这个主桥管理的 PCI 总线域地址范围、PCI 总线号和访问配置寄存器的方法等一系列信息，pci_controller 结构在./arch/powerpc/include/asm/pci-bridge.h 文件中。

源代码 14-45　fsl_add_bridge 函数片段 1

```
int __init fsl_add_bridge(struct device_node * dev, int is_primary)
{
    int len;
    struct pci_controller * hose;
    struct resource rsrc;
    const int * bus_range;
...
    /* Fetch host bridge registers address */
    if (of_address_to_resource(dev, 0, &rsrc)) {
```

```
            printk(KERN_WARNING "Can't get pci register base!");
            return -ENOMEM;
        }
...
        ppc_pci_add_flags(PPC_PCI_REASSIGN_ALL_BUS);
        hose = pcibios_alloc_controller(dev);
        if (!hose)
            return -ENOMEM;

        hose->first_busno = bus_range ? bus_range[0] : 0x0;
        hose->last_busno = bus_range ? bus_range[1] : 0xff;

        setup_indirect_pci(hose, rsrc.start, rsrc.start + 0x4,
            PPC_INDIRECT_TYPE_BIG_ENDIAN);
```

这段代码首先分析 mpc8572ds.dts 文件，然后获得 PCIe 主桥管理的 PCI 总线范围，对于 pci0 控制器，bus_range[0]为 0，而 bus_range[1]为 255。之后为当前 PCIe 主桥使用的 hose 结构分配空间，并初始化其 first_busno 和 last_busno 参数。

setup_indirect_pci 函数设置在 Linux PowerPC 中间接访问 PCI 设备配置空间的函数，如第 2.2 节所示，在 PowerPC 处理器中，访问 PCI 设备配置空间有两种方式，一种是使用间接访问方式，一种是使用 ECAM 方式。与 x86 处理器略有不同，PowerPC 处理器使用间接访问方式也可以访问 PCIe 设备的扩展配置空间，因此 ECAM 方式对于 PowerPC 处理器而言，并不是必须的。

**源代码 14-46　fsl_add_bridge 函数片段 2**

```
...
    /* Interpret the "ranges" property */
    /* This also maps the I/O region and sets isa_io/mem_base */
    pci_process_bridge_OF_ranges(hose, dev, is_primary);

    /* Setup PEX window registers */
    setup_pci_atmu(hose, &rsrc);

    return 0;
}
```

pci_process_bridge_OF_ranges 函数分析 mpc8572ds.dts 文件的“ranges”字段，在 dts 文件中，ranges 字段的解释如下。

```
ranges = <0x2000000 0x0 0x80000000 0 0x80000000 0x0 0x20000000
          0x1000000 0x0 0x00000000 0 0xffc00000 0x0 0x00010000>;
```

ranges 字段共由 14 个双字组成，每 7 个双字为 1 组，每一组描述一段 PCI 总线域地址空间与存储器域地址空间的对应关系。

- 每一组的第一个双字代表 pci_space，为 0x200-0000 表示这段 PCI 总线地址空间为存储器地址空间，为 0x100-0000 表示这段 PCI 总线地址空间为 I/O 地址空间。

- 每一组的第 2 ~ 3 个双字存放 pci_address，即 PCI 域地址空间。
- 每一组的第 4 ~ 5 个双字存放 cpu_address，即存储器域地址空间。
- 每一组的第 6 ~ 7 个双字存放 size，即这段地址空间的大小。

pci_process_bridge_OF_ranges 函数的主要作用就是根据 dts 文件中的 ranges 字段，初始化 hose 结构的对应参数，本节对该函数不做进一步介绍。

setup_pci_atmu 函数首先设置 MPC8572 处理器中的 Outbound 和 Inbound 寄存器组，这两组寄存器的描述见第 2.2 节。然后设置 PEXCSRBAR 寄存器。

如果 MPC8572 处理器作为 RC[㊀]，而且支持 MSI 中断机制时，需要设置 PCIe 主桥的 BAR0 寄存器，即 PEXCSRBAR（PCI Express Base Address Register）寄存器。在 PCI 规范中，MSI 中断机制以存储器写的方式实现，当这个 MSI 存储器写最终到达 RC 时，需要能够被 RC 接收。在 PowerPC 处理器中，MSI 存储器写的目的地址为 MSIIR 寄存器在 PCI 总线域的物理地址。此时 PowerPC 处理器可以采用两种方式接收这个 MSI 存储器写，一种是设置 Inbound 寄存器，映射 MSIIR 寄存器所在的 PCI 总线空间，另一种是设置 RC 的 BAR0 寄存器。Linux PowerPC 使用了后一种方式。

Linux PowerPC 执行完毕 setup_arch 函数后，还会执行一些和 PCI 总线初始化相关的函数，如下所示。

```
c053e04c t _initcall_pcibus_class_init2
c053e050 t _initcall_pci_driver_init2
c053e088 t _initcall_pcibios_init4
c053e0ac t _initcall_pci_slot_init4
c053e28c t _initcall_pci_init6
c053e290 t _initcall_pci_proc_init6
c053e3ec t _initcall_pci_resource_alignment_sysfs_init7
c053e3f0 t _initcall_pci_sysfs_init7
```

这些函数在第 14.3 节中都有介绍，虽然 Linux PowerPC 执行这些函数的过程与 Linux x86 略有不同，但大体类似，本章对此不做进一步说明。

## 14.5 小结

本章使用了一定的篇幅介绍 Linux PCI 的实现过程。Linux PCI 中的源代码对于读者理解 PCI 体系结构有较大的帮助，但希望读者不要拘泥于此。Linux PCI 只是 PCI 软件体系结构的一种实现方式，这种实现并不是最合理的。

Linux PCI 子系统在其发展过程中，遇到了各种各样的问题与 Bug，这些代码经历了一遍又一遍的修改。这种修改有如向一个满是补丁的衣服上继续打补丁，最后已无法识别衣服的原本模样。同许多通用代码类似，Linux PCI 需要兼容各类处理器系统，目前的实现远非完美，而这些不完美将继续。

---

㊀ MPC8572 的 PCIe 总线控制器可以作为 RC，也可以作为 EP。

# 第 15 章　Linux PCI 的中断处理

Linux PCI 的中断处理包含两部分内容，一部分是 PCI 设备使用 INTx 信号，包括 PCIe 设备使用 INTx 消息，向处理器提交的中断请求，这种中断请求方式也被称为 PCI 设备的传统中断请求；而另一部分是处理 MSI/MSI-X 中断机制。

Linux PCI 在处理传统中断请求时，需要考虑 PCI 总线的中断路由。本章将首先介绍 PCI 总线的中断路由，并在第 15.2 节介绍 MSI 和 MSI-X 中断机制，而不再详细介绍 PCI 设备的传统中断请求。

## 15.1　PCI 总线的中断路由

在多数 x86 处理器系统中，PCI 设备的 INTA ~ D#四个中断请求信号与 LPC 接口提供的外部引脚 PIRQA ~ D#相连，之后 PIRQA ~ D#与 I/O APIC 的中断请求信号 IRQ_PIN16 ~ 19#相连。如果 PCIe 设备没有使用 MSI 中断请求机制，而是使用了 Legacy INTx 方式[⊖]模拟 INTA ~ D#信号时，这些 Assert INTx 和 Deassert INTx 消息也由 Chipset 处理，并由 Chipset 将这些消息转换为一根硬件引脚，然后将这个硬件引脚与 I/O APIC 的中断输入引脚相连。其连接关系如图 15-1 所示。I/O APIC 最终使用 REDIR_TBL 表，将来自输入引脚的中断请求发送至 Local APIC，并由 CPU 进一步处理这个中断请求。

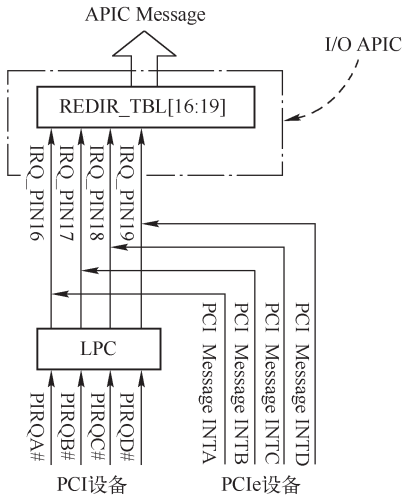



图 15-1　I/O APIC 如何处理 PCI 设备的中断请求

⊖　I/O APIC 将这种方式称为 PCI Message。PCIe 设备可以通过 INTx 中断消息报文，向 I/O APIC 提交中断请求，详见第 6.3.4 节。

本书并不关心 I/O APIC 如何使用 APIC Message 将中断消息传递给 Local APIC，而重点关注 PCI 和 PCIe 设备使用的中断信号与 I/O APIC 输入引脚 IRQ_PIN16 ~ 19 的连接关系。如图 15-1 所示，LPC 的 PIRQA ~ D#分别与 IRQ_PIN16 ~ 19 对应，但是 PCI 设备的 INTA ~ D#与 PIRQA ~ D#的连接关系并不是唯一的，图 15-1 所示的 PCI 设备与中断控制器连接方法只是其中一种连接方法。

而无论硬件采用何种连接结构，系统软件都需要能够正确识别是哪个 PCI 设备发出的中断请求，为此系统软件使用 PCI 中断路由表（PCI Interrupt Routing Table）记录 PCI 设备使用的 INTA ~ D#与 I/O APIC 中断输入引脚 IRQ16 ~ 19 的对应关系。

如果在 x86 处理器系统中存在 Switch，而这个 Switch 的每一个端口都相当于一个虚拟 PCI 桥，此时该 Switch 的下游端口连接的 PCIe 设备，在使用 PCI Message INTx 消息提交中断请求时，虚拟 PCI 桥可能将其转换为其他 PCI Message INTx 消息。在虚拟 PCI 桥中，Primary 总线和 Secondary 总线 PCI Message INTx 消息的对应关系如表 15-1 所示。

**表 15-1　虚拟 PCI 桥 Primary 总线与 Secondary 总线间 INTx 消息间的映射关系**

| 设备号 | PCI 桥 Secondary 总线的虚拟中断信号 INTx# | PCI 桥 Primary 总线的虚拟中断信号 INTx# |
|---|---|---|
| 0, 4, 8, 12, 16, 20, 24, 28 | INTA# | INTA# |
| | INTB# | INTB# |
| | INTC# | INTC# |
| | INTD# | INTD# |
| 1, 5, 9, 13, 17, 21, 25, 29 | INTA# | INTB# |
| | INTB# | INTC# |
| | INTC# | INTD# |
| | INTD# | INTA# |
| 2, 6, 10, 14, 18, 22, 26, 30 | INTA# | INTC# |
| | INTB# | INTD# |
| | INTC# | INTA# |
| | INTD# | INTB# |
| 3, 7, 11, 15, 19, 23, 27, 31 | INTA# | INTD# |
| | INTB# | INTA# |
| | INTC# | INTB# |
| | INTD# | INTC# |

PCIe 设备发送的 PCI Message INTx 消息首先到达虚拟 PCI 桥的 Secondary 总线，之后虚拟 PCI 桥根据 PCIe 设备的设备号将这些 PCI Message INTx 消息转换为 Primary 总线合适的虚拟中断信号。如设备号为 1 的 PCIe 设备使用 PCI Message INTA 消息进行中断请求时，该消息在通过虚拟 PCI 桥后，将被转换为 PCI Message INTB 消息，然后继续传递该消息报文，最终 PCI Message INTx 消息将到达 RC，并由 RC 将该消息报文转换为虚拟中断信号 INTx，并与 I/O APIC 的中断请求引脚 IRQ_PIN16 ~ 19 相连。

然而直接使用 PCIe 总线提供的标准方法会带来一些问题。因为一条 PCIe 链路只能挂接

一个 EP，这个 EP 的设备号通常为 0，而这些设备使用的虚拟中断信号多为 INTA#，因此这些 PCIe 设备通过 Switch 的虚拟 PCI-to-PCI 桥进行中断路由后，将使用虚拟中断信号 INTA#，并与 I/O APIC 的 IRQ_PIN16 引脚相连，并不会使用其他 IRQ_PIN 引脚，这造成了 IRQ_PIN16 的负载过重。其连接拓扑结构如图 15-2 所示。

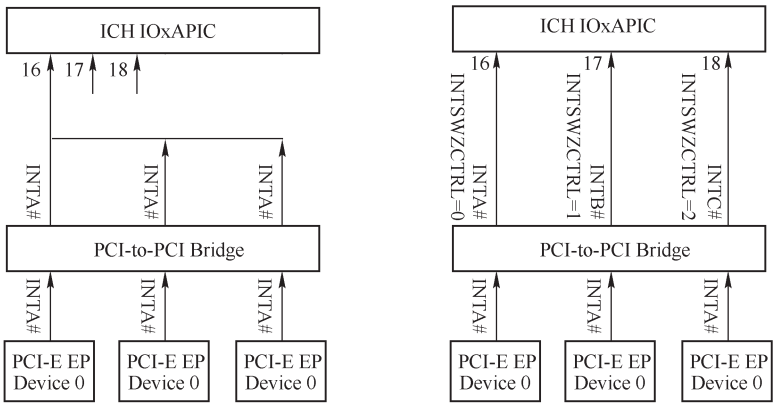



图 15-2　PCI Message 中断路由

如上图所示，PCIe 设备使用的 INTx 中断请求都最终使用 I/O APIC 的 IRQ_PIN16 引脚，从而造成了这个引脚所申请的中断过于密集，因此采用这种中断路由方法并不合理。为此 Intel 在 5000 系列的 Chipset 中使用了 Interrupt Swizzling 技术将这些来自 PCIe 设备的中断请求平均分配到 I/O APIC 的 IRQ_PIN16 ~ 19 引脚中。

在图 15-2 中，Chipset 设置了一个 INTSWZCTRL 寄存器，通过这些寄存器可以将 PCIe 设备提交的中断请求均衡地发送至 I/O APIC 中。如果一个 EP 对应的 INTSWZCTRL 位为 0，则该设备的 INTA#将与 IRQ_PIN16 相连；如果为 1，将与 IRQ_PIN17 相连，并以此类推，最终实现中断请求的负载均衡。

在一个 x86 处理器系统中，PCI 设备或者 PCIe 设备使用的中断信号 INTA ~ D#与 I/O APIC 的 IRQ_PIN16 ~ 19 之间的对应关系并不明确，各个厂商完全可以按照需要定制其映射关系。这为系统软件的设计制造了不小的困难。为此 BIOS 为系统软件提供了一个 PCI 中断路由表，存放这个映射关系，ACPI 规范将这个中断路由表存放在 DSDT 中。

值得注意的是，每一个 HOST 主桥和每一条 PCI 总线都含有一个中断路由表。在讲述 PCI 中断路由表之前，我们简要回顾 Linux 系统如何为 PCI 设备分配中断向量。

### 15.1.1　PCI 设备如何获取 irq 号

在 Linux 系统中，PCI 设备使用的 irq 号存放在 pdev→irq 参数中，该参数在 Linux 设备驱动程序进行初始化时，由 pci_enable_device 函数设置。本书在第 12.3.2 节曾简要介绍过这个函数，下文进一步说明如何使用该函数设置 PCI 设备的 irq 号。pci_enable_device 函数将依次调用__pci_enable_device_flags→do_pci_enable_device→pcibios_enable_device 函数设置 PCI 设备使用的 irq 号。

pcibios_enable_device 函数将调用 pcibios_enable_irq 函数，设置 PCI 设备使用的 irq 号。如果处理器系统使能了 ACPI 机制，pcibios_enable_irq 函数将被赋值为 acpi_pci_irq_enable。acpi_pci_irq_enable 函数在./drivers/acpi/pci_irq.c 文件中，其实现过程如源代码 15-1 所示。

源代码 15-1　acpi_pci_irq_enable 函数

```
int acpi_pci_irq_enable(struct pci_dev *dev)
{
    struct acpi_prt_entry *entry;
    int gsi;
    u8 pin;
    int triggering = ACPI_LEVEL_SENSITIVE;
    int polarity = ACPI_ACTIVE_LOW;
    char *link = NULL;
    char link_desc[16];
    int rc;

    pin = dev->pin;
    if (!pin) {
        ACPI_DEBUG_PRINT((ACPI_DB_INFO,
            "No interrupt pin configured for device %s\n",
            pci_name(dev)));
        return 0;
    }

    entry = acpi_pci_irq_lookup(dev, pin);
    if (!entry) {
        /*
         * IDE legacy mode controller IRQs are magic. Why do compat
         * extensions always make such a nasty mess.
         */
        if (dev->class >> 8 == PCI_CLASS_STORAGE_IDE &&
                (dev->class & 0x05) == 0)
            return 0;
    }

    if (entry) {
        if (entry->link)
            gsi = acpi_pci_link_allocate_irq(entry->link,
                    entry->index,
                    &triggering, &polarity,
                    &link);
        else
            gsi = entry->index;
    } else
        gsi = -1;
...
```

```
        if (gsi < 0) {
            dev_warn(&dev->dev, "PCI INT %c: no GSI", pin_name(pin));
            /* Interrupt Line values above 0xF are forbidden */
            if (dev->irq > 0 && (dev->irq <= 0xF)) {
                printk(" - using IRQ %d\n", dev->irq);
                acpi_register_gsi(&dev->dev, dev->irq,
                    ACPI_LEVEL_SENSITIVE,
                    ACPI_ACTIVE_LOW);
                return 0;
            } else {
                printk("\n");
                return 0;
            }
        }

        rc = acpi_register_gsi(&dev->dev, gsi, triggering, polarity);
        if (rc < 0) {
            dev_warn(&dev->dev, "PCI INT %c: failed to register GSI\n",
                pin_name(pin));
            return rc;
        }
        dev->irq = rc;
...
        return 0;
    }
```

该函数首先调用 acpi_pci_irq_lookup→ acpi_pci_irq_find_prt_entry 函数，从 acpi_prt_list 链表中获得一个 acpi_prt_entry 结构的 Entry。在 acpi_prt_list 链表中存放 PCI 总线的中断路由表，本章将在第 15.1.2 节进一步介绍该表。在这个 Entry 中，存放 PCI 设备使用的 Segment、Bus、Device 和 Function 号，PCI 设备使用的中断请求信号（INTA# ~ INTD#）和 GSI（Global System Interrupt）号。

这段程序在获得 Entry 后，将判断 Entry→link 是否为空，如果为空，表示当前 x86 处理器系统使用 I/O APIC 管理外部中断，而不是使用 8259A。在 Intel 的 ICH9 中集成了两个中断控制器，一个是 8259A，另一个是 I/O APIC。Linux x86 通过软件配置，决定究竟使用哪个中断控制器，在绝大多数情况下，Linux x86 使用 I/O APIC 而不是 8259A 管理外部中断请求 200。本章不再关心 8259A 中断控制器，因此也不再关心 Entry→link 不为空的处理情况[⊖]。

这段程序在获得 GSI 号之后，将调用 acpi_register_gsi 函数，将 GSI 号转换为系统软件使

⊖ Linux IA 在引导时可以加入“noapic”参数关闭 I/O APIC，此时处理器系统将使用 8259A 中断控制器。

用的 irq 号。acpi_register_gsi 函数使用三个入口参数，分别为 GSI 号，中断触发方式和采用电平触发时的极性。其中 PCI 设备使用低电平触发方式。

acpi_register_gsi 函数执行完毕后，将为 PCI 设备分配一个 irq 号，这个 irq 号是系统软件使用的，之后 PCI 设备的驱动程序可以使用 request_irq 函数将中断服务例程与 irq 号建立映射关系；该函数还将设置 I/O APIC 的 REDIR_TBL 表，将 GSI 号与 REDIR_TBL 表中的中断向量建立对应关系，同时初始化与操作系统相关的 irq 结构[⊖]。为了深入理解 acpi_register_gsi 函数，读者需要理解 GSI 号、I/O APIC 的 REDIR_ TBL 表、IRQ_PIN 引脚和 Linux 使用的 irq 号之间的对应关系。

GSI 号是 ACPI 规范引入的，用于记录 I/O APIC 的 IRQ_PIN 引脚号的参数。如果 x86 处理器系统使用 I/O APIC 管理外部中断请求，而且在这个处理器系统中具有多个 I/O APIC 控制器，那么 GSI 号与 I/O APIC 中断引脚号的对应关系如图 15-1 所示。

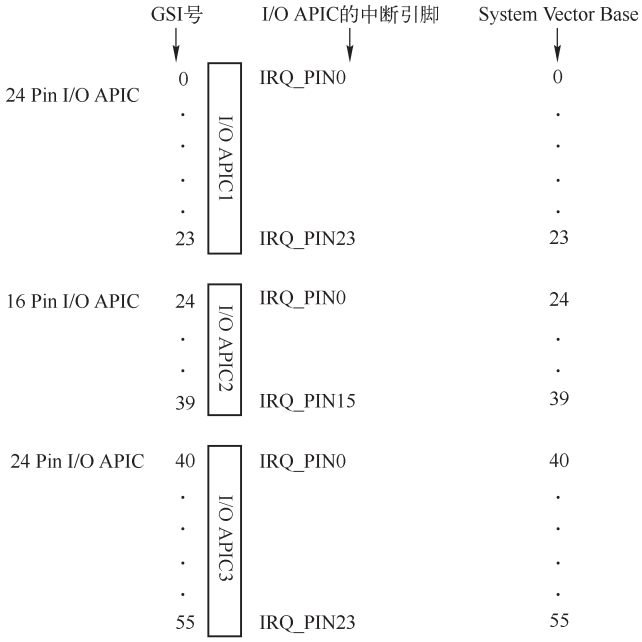



图 15-3　GSI 和 IO APIC 中断引脚号的对应关系

假设在一个 x86 处理器系统中存在 3 个 I/O APIC，其中有两个 I/O APIC 的外部中断引脚数为 24 根，另外一个 I/O APIC 的外部中断引脚数为 16 根。其中 GSI 号的 0 ~ 23 与 I/O APIC1 的 IRQ_PIN0 ~ 23 对应；GSI 号的 24 ~ 39 与 I/O APIC2 的 IRQ_PIN0 ~ 15 对应；而 GSI 号的 40 ~ 55 与 I/O APIC3 的 IRQ_PIN0 ~ 23 对应。ACPI 规范为统一起见使用 GSI 号描述外部设备与 I/O APIC 中断引脚的连接关系。

I/O APIC 的 IRQ_PIN 引脚与外部设备的中断请求引脚相连，如 I/O APIC1 的 IRQ_PIN16 与某个 PCI 设备的 INTA#相连。值得注意的是，PCI 设备的 INTA#信号首先与 LPC 的 PIRQA#信号相连，而 PIRQA#信号再与 I/O APIC1 的 IRQ_PIN16 相连。其中 I/O APIC 集成

⊖ 在 Linux 系统中，该结构为 irq_desc，该结构记录与 irq 号相关的信号。

在 ICH 中，因此这些 IRQ_PIN 引脚并没有从 ICH 中引出。

REDIR_TBL 表中存放对 IRQ_PIN 引脚的描述，一个 I/O APIC 具有多少个 IRQ_PIN 引脚，REDIR_TBL 表就由多少项组成。该表的每一个 Entry 由多个字段组成，其中本节仅对这个 Entry 的 Vector 字段感兴趣，Vector 字段是这个 Entry 的第 7 ~ 0 位，存放对应 IRQ_PIN 引脚使用的中断向量。

在 Linux 系统中，与 IRQ_PIN 引脚对应的中断向量由 acpi_register_gsi 函数设置，当 x86 处理器系统使用 I/O APIC 管理外部中断时，acpi_register_gsi 函数将调用 mp_register_gsi 函数。mp_register_gsi 函数在 ./drivers/acpi/boot.c 文件中定义，其实现机制如源代码 15-2 所示。我们假定在 Linux 系统中使能了 CONFIG_X86_32 选项。

**源代码 15-2　mp_register_gsi 函数**

```
int mp_register_gsi(struct device *dev, u32 gsi, int trigger, int polarity)
{
    int ioapic;
    int ioapic_pin;
...
    ioapic = mp_find_ioapic(gsi);
    if (ioapic < 0) {
        printk(KERN_WARNING "No IOAPIC for GSI %u\n", gsi);
        return gsi;
    }

    ioapic_pin = mp_find_ioapic_pin(ioapic, gsi);

#ifdef CONFIG_X86_32
    if (ioapic_renumber_irq)
        gsi = ioapic_renumber_irq(ioapic, gsi);
#endif

    if (ioapic_pin > MP_MAX_IOAPIC_PIN) {
        printk(KERN_ERR "Invalid reference to IOAPIC pin "
            "%d-%d\n", mp_ioapics[ioapic].apicid,
            ioapic_pin);
        return gsi;
    }

    if (enable_update_mptable)
        mp_config_acpi_gsi(dev, gsi, trigger, polarity);

    set_io_apic_irq_attr(&irq_attr, ioapic, ioapic_pin,
```

```
                trigger == ACPI_EDGE_SENSITIVE ? 0 : 1,
                polarity == ACPI_ACTIVE_HIGH ? 0 : 1);
        io_apic_set_pci_routing(dev, gsi, &irq_attr);

        return gsi;
    }
```

这段程序首先根据 GSI 号，使用 mp_find_ioapic 和 mp_find_ioapic_pin 函数，确定当前 PCI 设备与 I/O APIC 中断控制器的哪个 IRQ_PIN 引脚相连（GSI 号与 I/O APIC 和 IRQ_PIN 引脚的对应关系如图 15-1 所示）。

然后 mp_register_gsi 函数调用 io_apic_set_pci_routing 函数设置 I/O APIC 中的寄存器。在 Linux x86 的源代码中，mp_register_gsi 函数调用 io_apic_set_pci_routing 函数时，有一个并不恰当的处理，在 mp_register_gsi 函数中使用 GSI 号作为 io_apic_set_pci_routing 函数的第二个入口参数，但是 io_apic_set_pci_routing 函数要求的这个输入参数是 irq 号。

在 Linux x86 系统中，irq 号是一个纯软件[㊀]概念，而这段代码的作用实际上是令 GSI 号直接等于 irq 号。笔者认为这种方法并不十分恰当，因为 GSI 号用来描述 I/O APIC 的 IRQ_PIN 输入引脚，而 irq 号是设备驱动程序用来挂接中断服务例程的。

本节在此强调这个问题，主要为了读者辨明 GSI 号和 irq 号的关系，目前在 Linux x86 系统中，PCI 设备使用的 GSI 号与 irq 号采用了“直接相等”[㊁]的一一映射关系，实际上，GSI 号并不等同于 irq 号。在系统软件的实现中，两者只要建立一一映射的对应关系即可，并不一定要“直接相等”。还有一点需要提醒读者注意，就是不同的 PCI 设备可以共享同一个 GSI 号，即共享 I/O APIC 的一个 IRQ_PIN 引脚，从而在 Linux 系统中共享同一个 irq 号。

io_apic_set_pci_routing 函数调用__io_apic_set_pci_routing→setup_IO_APIC_irq 操作 I/O APIC 中的寄存器。setup_IO_APIC_irq 是一个重要函数，如源代码 15-3 所示。

**源代码 15-3　setup_IO_APIC_irq 函数**

```
static void setup_IO_APIC_irq(int apic_id, int pin, unsigned int irq,
                struct irq_desc *desc, int trigger, int polarity)
{
    struct irq_cfg *cfg;
    struct IO_APIC_route_entry entry;
    unsigned int dest;
...
    cfg = desc->chip_data;
    if (assign_irq_vector(irq, cfg, TARGET_CPUS))
        return;
...
    if (setup_ioapic_entry(mp_ioapics[apic_id].apicid, irq, &entry,
```

㊀ 如果 Linux x86 并没有使用 8825A 作为中断控制器，irp 号和中断向量并没有直接的对应关系。

㊁ 如果在一个处理器系统中，irp 号大于 16，那么 irp 号等于 CSI 号。

```
                dest, trigger, polarity, cfg->vector, pin)) {
            printk("Failed to setup ioapic entry for ioapic %d, pin %d\n",
                mp_ioapics[apic_id].apicid, pin);
            __clear_irq_vector(irq, cfg);
            return;
        }

        ioapic_register_intr(irq, desc, trigger);
        if (irq < NR_IRQS_LEGACY)
            disable_8259A_irq(irq);

        ioapic_write_entry(apic_id, pin, entry);
    }
```

该函数首先调用 assign_irq_vector→_assign_irq_vector 函数将外部设备使用的 GSI 号与 I/O APIC 中 REDIR_TBL 表建立联系，并将其结果记录到 CPU 的 vector_irq 表中。这个步骤非常重要，在 Linux x86 系统中，如果存在多个 CPU，那么每一个 CPU 都有一个 vector_irq 表，这张表中包含了 vector 号与 irq 号的对应关系。这张表也是处理器硬件与系统软件联系的桥梁。

处理器硬件并不知道 irq 号的存在，而仅仅知道 vector 号，而 Linux x86 系统使用的是 irq 号。在处理外部中断请求时，Linux 系统需要通过 vector_irq 表将 vector 号转换为 irq 号才能通过 irq_desc 表找到相关设备的中断服务例程。

setup_ioapic_entry 函数将初始化 entry 参数。该参数是一个 IO_APIC_route_entry 类型的结构。而 ioapic_register_intr 函数调用 set_irq_chip_and_handler_name 函数设置 irq_desc[irq] 变量，并将这个变量的 chip 参数设置为 ioapic_chip，handle_irq 参数设置为 handle_fasteoi_irq，这个步骤对于 Linux x86 中断处理系统非常重要。

ioapic_write_entry 函数将保存在 entry 参数中的数据写入到与 GSI 号对应的 REDIR_TBL 表中，该函数将直接操作 I/O APIC 的寄存器。

由以上描述，我们可以发现当 acpi_pci_irq_enable 函数执行完毕后，Linux 系统将 GSI 号与 irq 号建立映射关系，同时又将 irq 号与 I/O APIC 中的 vector 号进行映射，并将这个映射关系记录到 vector_irq 表中，这个映射表由操作系统使用。之后该程序还将初始化 I/O APIC 的 REDIR_TBL 表，将 PCI 设备使用的 GSI 号与 I/O APIC 的 vector 号联系在一起。

在 x86 处理器系统中，PCI 设备的 INTx 引脚首先与 LPC 的 PIRQA ~ H 引脚直接相连，而 LPC 中的 PIRQA ~ H 引脚将与 I/O APIC 的 IRQ_PIN16 ~ 23 引脚相连。当 PCI 设备通过 INTx 引脚提交中断请求时，最终将传递到 IRQ_PIN16 ~ 23 引脚。而 I/O APIC 接收到这个中断请求后，将根据 REDIR_TBL 表与“IRQ_PIN16 ~ 23 引脚”对应的 Entry 向 Local APIC 发送中断请求消息，处理器通过 Local APIC 收到这个中断请求后，将执行中断处理程序进一步处理这个来自 PCI 设备的中断请求。

Linux x86 系统使用 do_IRQ 函数处理外部中断请求，该函数在 ./arch/x86/kernel/irq.c 文件中，如源代码 15-4 所示。

源代码 15-4　do_IRQ 函数

```
unsigned int __irq_entry do_IRQ(struct pt_regs * regs)
{
    struct pt_regs * old_regs = set_irq_regs(regs);

    /* high bit used in ret_from_ code */
    unsigned vector = ~regs->orig_ax;
    unsigned irq;

    exit_idle();
    irq_enter();

    irq = __get_cpu_var(vector_irq)[vector];

    if (!handle_irq(irq, regs)) {
        ack_APIC_irq();

        if (printk_ratelimit())
            pr_emerg("%s: %d.%d No irq handler for vector (irq %d)\n",
                __func__, smp_processor_id(), vector, irq);
    }

    irq_exit();

    set_irq_regs(old_regs);
    return 1;
}
```

do_IRQ 函数首先获得 vector 号，这个 vector 号由 I/O APIC 传递给 Local APIC，并与某个 IRQ_PIN 引脚对应，其描述在 I/O APIC 的 REDIR_TBL 表中。vector 号是一个硬件概念，x86 处理器系统在处理外部中断请求时，仅仅知道 vector 号的存在，而不知道 irq 号。

Linux x86 系统通过 vector_irq 表，将 vector 号转换为 irq 号，之后执行 handle_irq 函数进一步处理这个中断请求。对于 PCI 设备，这个 handle_irq 函数将调用 handle_fasteoi_irq 函数，而 handle_fasteoi_irq 函数将最终执行 PCI 设备使用的中断服务例程。handle_fasteoi_irq 函数的源代码在 ./kernel/irq/chip.c 文件中，本节对该函数不做进一步分析。

在 PCI 设备的 Linux 驱动程序中，将使用 request_irq 函数将其中断服务例程挂接到系统中断服务处理程序中。

### 15.1.2　PCI 中断路由表

上节简要介绍了 PCI 设备如何获取中断向量。由上文所述，PCI 设备在获取中断向量之前需要从 acpi_prt_list 链表获得 GSI 号，在 acpi_prt_list 链表中存放 PCI 总线的中断路由表，

而这个中断路由表中存放 PCI 设备所使用的 GSI 号。

这个 PCI 中断路由表由 BIOS 提供，如果 x86 处理器系统支持 ACPI 机制，这个中断路由表存在于 DSDT. dsl 文件中，如源代码 15-5 所示。ACPI 规范使用 ASL 语言描述 PCI 中断路由表。

**源代码 15-5　DSDT 表中的 PCI 中断路由表**

```
Device (PCI0)
{
...
    Method (_PRT, 0, NotSerialized)
    {
        If (LEqual (GPIC, Zero))
        {
            Package (0x04) {0x0001FFFF, 0x00, \_SB. PCI0. LPC. LNKA, 0x00},
            Package (0x04) {0x0001FFFF, 0x01, \_SB. PCI0. LPC. LNKB, 0x00},
            Package (0x04) {0x0001FFFF, 0x02, \_SB. PCI0. LPC. LNKC, 0x00},
            Package (0x04) {0x0001FFFF, 0x03, \_SB. PCI0. LPC. LNKD, 0x00},
        ...
        }
        Else
        {
            Return (Package (0x47) {
                ...
                Package (0x04) { 0x001CFFFF, Zero, Zero, 0x11 },
                Package (0x04) { 0x001CFFFF, One, Zero, 0x10 },
                Package (0x04) { 0x001CFFFF, 0x02, Zero, 0x12 },
                Package (0x04) { 0x001CFFFF, 0x03, Zero, 0x13 },

                Package (0x04) { 0x001DFFFF, Zero, Zero, 0x17 },
                Package (0x04) { 0x001DFFFF, One, Zero, 0x13 },
                Package (0x04) { 0x001DFFFF, 0x02, Zero, 0x12 },
                Package (0x04) { 0x001DFFFF, 0x03, Zero, 0x10 },
            ...})
        }
    }
}
```

在以上源代码中，_PRT 存放 x86 处理器系统 PCI 总线 0 的中断路由表，在 x86 处理器体系结构中，每一条 PCI 总线都有一个中断路由表，因此在 DSDT 中，将存在多个中断路由表。在以上源代码中，我们仅列出 PCI 总线 0 使用的中断路由表，即 RC 使用的中断路由表，在一个处理器系统中还可能有其他中断路由表，如 PCIe 桥使用的中断路由表等。

在以上源代码中，首先判断 GPIC 是否为 0，如果为 0 表示当前 x86 处理器系统使用 PIC

模式，即使用8259A中断控制器管理外部中断，在第15.1.3节将介绍这种情况；如果为1表示当前x86处理器系统使用I/O APIC管理外部中断，此时“Package（0x04）”[㊀]204中含有四个参数，这四个参数的定义如表15-1所示。

表15-2　PCI中断路由表使用的参数

| 参数 | 类型 | 描　　述 |
| --- | --- | --- |
| Address | DWORD | 设备地址，其中高两个字节表示PCI设备的Device号，低两个字节表示PCI设备的Function号，如果低两字节为0xFFFF表示全部Function号 |
| Pin | Byte | 其中0～3分别与INTA～D#引脚#对应 |
| Source | Name Path或者Byte | 该字段为0表示使用GSI号描述PCI设备使用的中断资源，否则该字段存放该设备与LPC的哪个PIRQ引脚，如LPC的PIRQA信号连接，在第15.1.3节将讲述LPC的PIRQ引脚的描述 |
| Source Index | DWORD | Source字段为0时，该字段存放PCI设备使用的GSI号 |

通过以上描述，发现“Package（0x04）｛0x0001FFFF，0x00，_SB.PCI0.LPC.LNKA，0x00｝[㊁]”的含义为，PCI总线0的某个设备，其Device号为0x01，而且这个设备的INTA#引脚与LPC的PIRQA相连，INTB#引脚与PIRQB相连，INTC#引脚与PIRQC相连，而INTD#引脚与PIRQD相连。

而“Package（0x04）｛0x001CFFFF，Zero，Zero，0x11｝[㊂]...”这段代码的含义为，PCI总线0的某个PCI设备，其Device号为0x1C，而且这个设备的INTA#引脚使用的GSI号为0x11；这个PCI设备的INTB#引脚使用的GSI号为0x10；这个PCI设备的INTC#使用的GSI号为0x12，这个PCI设备的INTD#引脚使用的GSI号为0x13。

Linux x86系统进行初始化时，将_PRT表加载到acpi_prt_list链表中，操作系统首先执行acpi_pci_root_init函数，之后调用acpi_device_probe→acpi_bus_driver_init→acpi_pci_root_add函数。acpi_pci_root_add函数将调用acpi_pci_irq_add_prt→acpi_pci_irq_add_entry函数将_PRT表中的中断路由表的每一个Entry加载到acpi_prt_list链表。

通过上文的分析，可以发现在每一个PCI桥中，包括Switch的虚拟PCI桥中都有一个中断路由表，因此acpi_pci_root_add还会调用acpi_pci_bridge_scan函数分析并加载每一个PCI桥的中断路由表。对Linux x86系统初始化PCI中断路由表感兴趣的读者可以自行分析这段代码，本节对此不做进一步介绍。

在Linux x86系统中，PCI设备在获取irq号时，将从这个链表中获得GSI号，从而最终获得irq号，具体过程见第15.1.1节。

### 15.1.3　PCI插槽使用的irq号

在x86处理器系统中，还有一类特殊的PCI设备，即PCI插槽。PCI插槽无法确定其上的PCI设备如何使用INTA#～INTD#信号，因此必须处理全部中断请求引脚，而在其上的

㊀ 对应Else之后的这段代码。

㊁ 使用8859A中断控制器的情况。

㊂ 使用APIC中断控制器的情况。

PCI 设备有选择地使用这些信号。

PCI 插槽使用的中断请求信号将与 LPC 的 PIRQA ~ F 相连，如果处理器系统使能了 I/O APIC，LPC 的这些中断请求引脚将与 IRQ_PIN16 ~ 23 相连，否则中断控制器 8259A 将管理这些中断引脚。在 ACPI 表中含有对这些 PCI 插槽中断请求信号的描述，这些描述主要针对处理器系统没有使用 I/O APIC 的处理情况，如源代码 15-6 所示。

**源代码 15-6　PCI 插槽使用中断请求信号**

```
Device (LPC)
{
...
    Device (LNKA)
    {
        Name (_HID, EisaId ("PNP0C0F"))
        Name (_UID, 0x01)
        Method (_STA, 0, NotSerialized)
        {
            If (And (PIRA, 0x80))
            {
                Return (0x09)
            }
            Else
            {
                Return (0x0B)
            }
        }
        Method (_DIS, 0, NotSerialized)
        {
            Or (PIRA, 0x80, PIRA)
        }
        Method (_CRS, 0, NotSerialized)
        {
            Name (BUF0, ResourceTemplate ()
            {
                IRQ (Level, ActiveLow, Shared, _Y02)
                {0}
            })
            CreateWordField (BUF0, \_SB. PCI0. LPC. LNKA. _CRS. _Y02. _INT, IRQW)
            If (And (PIRA, 0x80))
            {
                Store (Zero, Local0)
            }
            Else
```

```
            {
                Store (One, Local0)                }
            ShiftLeft (Local0, And (PIRA, 0x0F), IRQW)
            Return (BUF0)
        }
        Name (_PRS, ResourceTemplate ()
        {
            IRQ (Level, ActiveLow, Shared, )
            {3,4,5,7,9,10,11,12}
        })
        Method (_SRS, 1, NotSerialized)
        {
            CreateWordField (Arg0, 0x01, IRQW)
            FindSetRightBit (IRQW, Local0)
            If (LNotEqual (IRQW, Zero))
            {
                And (Local0, 0x7F, Local0)
                Decrement (Local0)
            }
            Else
            {
                Or (Local0, 0x80, Local0)
            }
                Store (Local0, PIRA)
            }
        }
    }
```

在 ACPI 规范中，PCI 插槽的中断请求信号的标识符“PNP0C0F”。在这段源代码中 LNKA 与 LPC 的 PIRQA 引脚对应，这段代码的作用是描述 LPC 的 PIRQA 引脚。在 ICH 中，使用 PIRQA_ROUT 寄存器描述 PIRQA 引脚。在以上这段源程序中，“_STA”、“_DIS”、“_CRS”、“_PRS”和“_SRS”可以操作 PIRQA_ROUT 寄存器，具体含义如下所示。

_STA 用来测试当前 PIRQA 引脚的状态，这段代码判断 PIRQA_ROUT 寄存器的第 7 位是否为 1，如果为 1 表示当前 PIRQ 引脚并没有与 8259A 相连，此时 I/O APIC 将管理该引脚，_STA 将返回 0x09 表示 PIRQA 没有与 8259A 相连；否则返回 0x0B，表示 PIRQA 与 8259A 相连。_STA 的返回值在 ACPI 规范中具有明确的定义。

_DIS 用来关闭 PIRQA 引脚与 8259A 的联系，即使用 I/O APIC 管理该引脚。_DIS 的作用是将 PIRQA_ROUT 寄存器的第 7 位置 1。

_CRS 用来获得当前资源的描述，对于 PIRQA 引脚而言，这段描述表示 PIRQA 引脚使用“低电平有效的共享中断请求”，随后通过 PIRQ[A]_ROUT 寄存器的最高位判断，该中断信号是由 8259A 中断控制器还是 APIC 中断控制器接管，最后将 IRQW 根据 PIRQ[A]_

ROUT 寄存器的 IRQ Routing 字段赋值，IRQ Routing 字段可以使用的资源在 {3, 4, 5, 7, 9, 10, 11, 12} 集合中。

_PRS 描述 PCI 插槽的中断请求信号可能使用的中断资源，对于 PIRQA 而言，可能使用的 irq 号为 {3, 4, 5, 7, 9, 10, 11, 12}。这些 irq 号由 x86 处理器系统规定，这些 irq 号与 ISA 总线兼容，如果一个系统使用了 I/O APIC，这些规定将不再有效。

在 Linux 系统中，acpi_pci_link_init 函数处理 PCI 插槽的中断请求，该函数在 ./drivers/acpi/pci_link.c 文件中，其实现如源代码 15-7 所示。

**源代码 15-7　acpi_pci_link_init 函数**

```
static int __init acpi_pci_link_init(void)
{
...
    if (acpi_bus_register_driver(&acpi_pci_link_driver) < 0)
        return -ENODEV;

    return 0;
}

subsys_initcall(acpi_pci_link_init);
```

acpi_pci_link_init 函数调用 acpi_bus_register_driver→...→ acpi_pci_link_add 函数将 LPC 的 PIRQA ~ H 引脚与 irq 号对应在一起。acpi_pci_link_add 函数的执行过程较为简单，首先该函数调用 acpi_pci_link_get_possible 函数，运行_ PRS 代码获得 {3, 4, 5, 7, 9, 10, 11, 12} 这个集合；之后调用 acpi_pci_link_get_current 函数，运行_ CRS 代码并从 {3, 4, 5, 7, 9, 10, 11, 12} 集合中获得 irq 号。acpi_pci_link_init 函数执行完毕后，Linux 系统将显示以下信息。

```
ACPI: PCI Interrupt Link [LNKA] (IRQs 3 4 5 7 9 10 *11 12)
ACPI: PCI Interrupt Link [LNKB] (IRQs 3 4 5 7 9 *10 11 12)
ACPI: PCI Interrupt Link [LNKC] (IRQs 3 4 5 7 9 10 *11 12)
ACPI: PCI Interrupt Link [LNKD] (IRQs 3 4 5 7 9 10 *11 12)
ACPI: PCI Interrupt Link [LNKE] (IRQs 3 4 5 7 *9 10 11 12)
ACPI: PCI Interrupt Link [LNKF] (IRQs 3 4 5 7 9 *10 11 12)
ACPI: PCI Interrupt Link [LNKG] (IRQs 3 4 5 7 *9 10 11 12)
ACPI: PCI Interrupt Link [LNKH] (IRQs 3 4 5 7 9 10 *11 12)
```

其中 LNKA 使用 IRQ11，LNKB 使用 IRQ10，并以此类推。如果一个处理器系统使能了 I/O APIC，acpi_pci_link_init 函数的执行结果并不重要，因为 PCI 设备在执行 pci_enable_device 函数后，该设备使用的 irq 号，还将发生变化。

目前 Linux x86 系统在大多数情况下，都会使能 I/O APIC，在这种情况下，即便不执行 acpi_pci_link_add 函数对系统也没有什么影响，也正是基于这个考虑，本节对 acpi_pci_link_init 函数并不做深入研究。

## 15.2 使用 MSI/MSIX 中断机制申请中断向量

上文讲述了 ACPI 如何为 PCI 设备或者“使用 INTx Emulation 方式”的 PCIe 设备分配中断向量。本节讲述 PCIe 设备使用 MSI/MSIX 中断机制时，Linux 系统如何分配中断向量。对于 PCI 设备，MSI/MSIX 中断机制是可选的，但是 PCIe 设备必须支持 MSI 或者 MSI-X 中断机制，或者同时支持这两种中断机制。

### 15.2.1 Linux 如何使能 MSI 中断机制

如果 PCI/PCIe 设备需要使用 MSI 中断机制，将调用 pci_enable_msi 函数，在 Linux 2.6.31 内核中，pci_enable_msi 函数使用 pci_enable_msi_block(pdev, 1)实现。pci_enable_msi_block 函数在./drivers/pci/msi.c 文件中，如源代码 15-8 所示。pci_enable_msi_block 函数具有两个入口参数，其中 dev 参数存放 PCIe 设备的 pci_dev 结构，而 nvec 参数为申请的 irq 号个数。

该函数返回值为 0 时，表示成功返回，此时该函数将更新 pci_dev→irq 参数，此时在 Linux 设备驱动程序中，可以使用的 irq 号在 pci_dev→irq ~ pci_dev→irq + nvec - 1 之间；当函数返回值为负数时，表示出现错误；而为正数时，表示 pci_enable_msi_block 函数没有成功返回，返回值为该 PCIe 设备 MSI Cabalibities 结构的 Multiple Message Capable 字段。

源代码 15-8 pci_enable_msi_block 函数

```
int pci_enable_msi_block(struct pci_dev *dev, unsigned int nvec)
{
    int status, pos, maxvec;
    u16 msgctl;

    pos = pci_find_capability(dev, PCI_CAP_ID_MSI);
    if (!pos)
        return -EINVAL;
    pci_read_config_word(dev, pos + PCI_MSI_FLAGS, &msgctl);
    maxvec = 1 << ((msgctl & PCI_MSI_FLAGS_QMASK) >> 1);
    if (nvec > maxvec)
        return maxvec;

    status = pci_msi_check_device(dev, nvec, PCI_CAP_ID_MSI);
    if (status)
        return status;

    WARN_ON(!!dev->msi_enabled);

    /* Check whether driver already requested MSI-X irqs */
    if (dev->msix_enabled) {
```

```
            dev_info(&dev->dev, "can't enable MSI "
                "(MSI-X already enabled)\n");
            return -EINVAL;
        }

        status = msi_capability_init(dev, nvec);
        return status;
    }
```

这段代码首先检查 PCI 设备是否支持 MSI 中断机制，如果不支持将直接退出该函数。否则检查 nvec 参数和 Multiple Message Capable 字段的大小，如果 nvec 的值较大时，该函数直接使用 Multiple Message Capable 字段返回。

如果 pci_enable_msi_block 函数通过了这些检查，将调用 pci_msi_check_device 函数，检查 Linux 系统是否能够使能 PCI 设备的 MSI 中断机制。这个检查包含两方面内容，一方面是纯软件层面的，包括检查全局变量 pci_msi_enable、pci_dev→no_msi 参数等；一方面是硬件层面的检测，包括当前 PCI 设备的上游 PCI 桥是否支持 MSI 报文的转发，PCI 设备是否具有 Capabilities 链表，是否具有 MSI Capability 结构。完成这些检查后，pci_enable_msi 将进一步调用 msi_capability_init 函数，完成与 MSI 中断相关的设置，msi_capability_init 函数的实现如源代码 15-9～10 所示。

**源代码 15-9　msi_capability_init 函数片段 1**

```
static int msi_capability_init(struct pci_dev *dev, int nvec)
{
    struct msi_desc *entry;
    int pos, ret;
    u16 control;
    unsigned mask;

    pos = pci_find_capability(dev, PCI_CAP_ID_MSI);
    msi_set_enable(dev, pos, 0);/* Disable MSI during set up */

    pci_read_config_word(dev, msi_control_reg(pos), &control);
    /* MSI Entry Initialization */
    entry = alloc_msi_entry(dev);
    if (!entry)
        return -ENOMEM;

    entry->msi_attrib.is_msix = 0;
    entry->msi_attrib.is_64 = is_64bit_address(control);
    entry->msi_attrib.entry_nr = 0;
    entry->msi_attrib.maskbit = is_mask_bit_support(control);
```

```
        entry -> msi_attrib.default_irq = dev -> irq;/* Save IOAPIC IRQ */
        entry -> msi_attrib.pos = pos;

        entry -> mask_pos = msi_mask_reg(pos, entry -> msi_attrib.is_64);
        /* All MSIs are unmasked by default, Mask them all */
        if (entry -> msi_attrib.maskbit)
            pci_read_config_dword(dev, entry -> mask_pos, &entry -> masked);
        mask = msi_capable_mask(control);
        msi_mask_irq(entry, mask, mask);

        list_add_tail(&entry -> list, &dev -> msi_list);
```

msi_capability_init 函数具有两个入口参数，在 Linux 2.6.30 内核中，该函数具有一个入口参数，仅能获得一个 irq 号。msi_capability_init 函数参考了 msix_capability_init 函数的实现机制。这段代码置 PCI 设备的 MSI Capability 结构的 Enable 位为 0，msi_capability_init 函数需要对 MSI Capability 结构进行读写操作，因此需要暂时禁止当前设备使用 MSI 中断机制。

这段程序随后读取 MSI Capabilty 结构的 Message Control 字段，并暂时保存在 control 变量中，在 control 变量中存放 PCIe 设备使用的 MSI Capability 结构的格式，如图 10-1 所示，MSI Capability 结构可以使用 4 种格式。

最后这段程序调用 alloc_msi_entry 函数分配一个 msi_desc 结构的 entry 参数，并将其初始化后，加入到 pci_dev→msi_list 链表中。在 entry 参数中存放该 PCI 设备使用的 MSI 中断机制的详细信息。

**源代码 15-10　msi_capability_init 函数片段 2**

```
        /* Configure MSI capability structure */
        ret = arch_setup_msi_irqs(dev, nvec, PCI_CAP_ID_MSI);
        if (ret) {
            msi_mask_irq(entry, mask, ~mask);
            msi_free_irqs(dev);
            return ret;
        }

        /* Set MSI enabled bits */
        pci_intx_for_msi(dev, 0);
        msi_set_enable(dev, pos, 1);
        dev -> msi_enabled = 1;

        dev -> irq = entry -> irq;
        return 0;
    }
```

这段代码继续调用 arch_setup_msi_irqs 函数设置 MSI Capability 结构的其他字段，并设置

entry 结构的 irq 参数，arch_setup_msi_irqs 函数的实现与体系结构相关，下文将分别介绍 x86 和 PowerPC 处理器的实现方式。

然后这段代码调用 pci_intx_for_msi 函数，关闭 PCI 设备配置空间 Command 寄存器的 Interrupt Disable 位，因为该 PCI 设备将使用 MSI 中断机制，而不是传统的 INTx 中断机制；并调用 msi_set_enable 函数使能 MSI Capability 结构的 Enable 位；最后对 pci_dev→msi_enabled 位置 1，并将 pci_dev→irq 参数赋值。

### 1. Linux x86

Linux x86 使用的 arch_setup_msi_irqs 函数在 ./arch/x86/kernel/apic/io_apic.c 文件中，其实现如源代码 15-1 所示，本节并不关心 intr_remapping_enabled 参数为 1 的情况，该参数与 IOMMU 机制的 IRQ Remapping 相关。

**源代码 15-11　Linux x86 使用的 arch_setup_msi_irqs 函数**

```
int arch_setup_msi_irqs(struct pci_dev *dev, int nvec, int type)
{
    /* x86 doesn't support multiple MSI yet */
    if (type == PCI_CAP_ID_MSI && nvec > 1)
        return 1;

    node = dev_to_node(&dev->dev);
    irq_want = nr_irqs_gsi;
    sub_handle = 0;
    list_for_each_entry(msidesc, &dev->msi_list, list) {
        irq = create_irq_nr(irq_want, node);
        if (irq == 0)
            return -1;
        irq_want = irq + 1;
        if (!intr_remapping_enabled)
            goto no_ir;
...
no_ir:
        ret = setup_msi_irq(dev, msidesc, irq);
        if (ret < 0)
            goto error;
        sub_handle++;
    }
    return 0;

error:
    destroy_irq(irq);
    return ret;
}
```

这段代码首先判断 type 是否为 PCI_CAP_ID_MSI，而且 nvec 参数是否大于 1，如果满足这两个条件，该函数将直接返回 1。通过这段代码可以发现，虽然在 Linux 2.6.31 内核中定义了一个新的 pci_enable_msi_block 函数，但是 PCIe 设备依然只能使用一个中断向量号。

这段程序随后调用 create_irq_nr 函数，分配 PCI 设备使用的 irq 号，并将其保存到 irq 变量中，之后调用 setup_msi_irq 函数初始化当前 PCI 设备的 MSI Capability 结构。setup_msi_irq 函数是一个重要函数，其实现如源代码 15-12 所示。

**源代码 15-12　setup_msi_irq 函数**

```
static int setup_msi_irq(struct pci_dev *dev, struct msi_desc *msidesc, int irq)
{
    int ret;
    struct msi_msg msg;

    ret = msi_compose_msg(dev, irq, &msg);
    if (ret < 0)
        return ret;

    set_irq_msi(irq, msidesc);
    write_msi_msg(irq, &msg);

    if (irq_remapped(irq)) {
        struct irq_desc *desc = irq_to_desc(irq);
...
        desc->status |= IRQ_MOVE_PCNTXT;
        set_irq_chip_and_handler_name(irq, &msi_ir_chip,
            handle_edge_irq, "edge");
    } else
        set_irq_chip_and_handler_name(irq, &msi_chip,
            handle_edge_irq, "edge");

    dev_printk(KERN_DEBUG, &dev->dev, "irq %d for MSI/MSI-X\n", irq);

    return 0;
}
```

这段代码首先调用 msi_compose_msg 函数，初始化 msg 结构的 address_hi、address_lo 和 data 参数，与 MSI Capability 结构的 Message Upper Address、Message Address 和 Message Data 字段对应。

对于 x86 处理器系统，Message Address 字段的格式见图 10-1，其中 Destination ID 字段与 CPU 的 ACPI ID 相关，Linux x86 使用 cpu_mask_to_apicid_and 函数获得该字段的值；而 Message Data 字段的格式如图 10-1 所示，其 Vector 字段由 assign_irq_vector 函数设置，Trig-

ger Mode 字段为 0x00 表示使用边沿触发方式，而 Delivery Mode 字段为“Fixed Mode”或者“Lowest Priority”。值得注意的是，在 Message Data 字段中存放的是中断向量号（vector），是一个硬件的概念，而设备驱动程序中使用的 irq 号是一个软件关系，两者之间存在对应关系，但是并不等同。

set_irq_msi 函数设置 PCIe 设备使用的 irq_desc 结构；而 write_msi_msg 函数将 msg 结构中的参数写入到 PCIe 设备的对应寄存器中；而 set_irq_chip_and_handler_name 函数设置 MSI 中断使用的中断处理程序。本节对这些函数不做进一步介绍。

2. PowerPC

Linux PowerPC 使用的 arch_setup_msi_irqs 函数在 ./arch/powerpc/kernel/msi.c 文件中，对于 Freescale 的 PowerPC 处理器，该函数等效与 fsl_setup_msi_irqs。fsl_setup_msi_irqs 函数的实现，如源代码 15-13 所示。

**源代码 15-13　fsl_setup_msi_irqs 函数**

```
static int fsl_setup_msi_irqs(struct pci_dev * pdev, int nvec, int type)
{
...
    list_for_each_entry(entry, &pdev->msi_list, list) {
        hwirq = msi_bitmap_alloc_hwirqs(&msi_data->bitmap, 1);
...
        virq = irq_create_mapping(msi_data->irqhost, hwirq);
...
        set_irq_msi(virq, entry);
        fsl_compose_msi_msg(pdev, hwirq, &msg);
        write_msi_msg(virq, &msg);
    }
    return 0;
out_free:
    return rc;
}
```

该函数的实现机制与 x86 处理器类似，值得提醒读者注意的是在 fsl_compose_msi_msg 函数中 msg→address_hi 等于 fsl_msi→msi_addr_lo，其值为 MSIIR 寄存器在 PCI 总线域的物理地址。在该函数中使用的 address_hi、address_lo 和 data 参数的详细描述见第 10.2 节。本节对此不一一叙述。目前在 PowerPC 处理器系统中，PCIe 设备也只能使用一个中断向量号。

## 15.2.2　Linux 如何使能 MSI-X 中断机制

在 Linux 系统中，如果 PCI/PCIe 设备需要使用 MSI-X 中断机制，需要调用 pci_enable_msix 函数，pci_enable_msix 函数调用的大多数函数与 pci_enable_msi 类似，本节并不会重复解释这些函数，该函数的实现如源代码 15-14 所示。

源代码 15-14　pci_enable_msix 函数

```
int pci_enable_msix(struct pci_dev * dev, struct msix_entry *entries, int nvec)
{
    int status, nr_entries;
    int i, j;

    if (! entries)
        return -EINVAL;

    status = pci_msi_check_device(dev, nvec, PCI_CAP_ID_MSIX);
    if (status)
        return status;

    nr_entries = pci_msix_table_size(dev);
    if (nvec > nr_entries)
        return nr_entries;

    /* Check for any invalid entries */
    for (i = 0; i < nvec; i++) {
        if (entries[i].entry >= nr_entries)
            return -EINVAL;/* invalid entry */
        for (j = i + 1; j < nvec; j++) {
            if (entries[i].entry == entries[j].entry)
                return -EINVAL;/* duplicate entry */
        }
    }
    WARN_ON(!! dev->msix_enabled);

    /* Check whether driver already requested for MSI irq */
    if (dev->msi_enabled) {
        dev_info(&dev->dev, "can't enable MSI-X "
            "(MSI IRQ already assigned)\n");
        return -EINVAL;
    }
    status = msix_capability_init(dev, entries, nvec);
    return status;
}
```

与 pci_enable_msi_block 函数不同，pci_enable_msix 函数的入口参数包括一个 msix_entry 结构的 entries 链表（在使用这个 entries 链表之前需要将 msix_entry.entry 参数赋值），而 nvec 参数保存 entries 链表的长度。该函数首先对入口参数进行检查，然后调用 msix_capability_init 函数为 PCIe 设备分配多个中断向量号。msix_capability_init 函数的实现与 msi_capabil-

ity_init 函数的实现方法类似，本章对此不做进一步描述。

该函数成功返回后，PCIe 设备将得到多个中断向量，并将结果放入 pci_dev→msi_list 和 entries 链表中，之后 PCIe 设备的 Linux 驱动程序可以使用多个 request_irq 函数注册相应的中断服务例程。

下文将以 Intel 的 e1000e 网卡驱动程序说明如何使用 MSI-X 中断机制挂接中断服务例程。在 Linux 中，与 e1000e 网卡相关的驱动程序在 ./drivers/net/e1000e/netdev.c 文件中。其中 MSI-X 中断机制的初始化在 e1000_probe→e1000_sw_init→e1000e_set_interrupt_capability 函数中，该函数的实现如源代码 15-15 所示。

**源代码 15-15　e1000e_set_interrupt_capability 函数**

```
void e1000e_set_interrupt_capability(struct e1000_adapter *adapter)
{
...
    switch (adapter->int_mode) {
    case E1000E_INT_MODE_MSIX:
        if (adapter->flags & FLAG_HAS_MSIX) {
            numvecs = 3; /* RxQ0, TxQ0 and other */
            adapter->msix_entries = kcalloc(numvecs,
                                sizeof(struct msix_entry),
                                GFP_KERNEL);
            if (adapter->msix_entries) {
                for (i=0; i < numvecs; i++)
                    adapter->msix_entries[i].entry = i;
                err = pci_enable_msix(adapter->pdev,
                                adapter->msix_entries,
                                numvecs);
                if (err == 0)
                    return;
            }
...
        }
    }
}
```

当 e1000e_set_interrupt_capability 函数返回后，MSI-X 中断机制使用的中断向量将被保存在 adapter->msix_entries 数组中，之后 e1000_open→ e1000_request_irq→ e1000_request_msix 函数将多次调用 request_irq 函数将 e1000e 使用的中断服务例程挂接到系统中断服务程序中，e1000_request_msix 函数的实现如源代码 15-16 所示。

**源代码 15-16　e1000_request_msix 函数**

```
static int e1000_request_msix(struct e1000_adapter *adapter)
{
```

```
...
        err = request_irq(adapter -> msix_entries[vector].vector,
              &e1000_intr_msix_rx, 0, adapter -> rx_ring -> name,
              netdev);
...
        err = request_irq(adapter -> msix_entries[vector].vector,
              &e1000_intr_msix_tx, 0, adapter -> tx_ring -> name,
              netdev);
...
        err = request_irq(adapter -> msix_entries[vector].vector,
              &e1000_msix_other, 0, netdev -> name, netdev);
...
}
```

e1000_request_msix 函数将“接收完成中断请求 e1000_intr_msix_rx”、“发送完成中断请求 e1000_intr_msix_tx”和“其他中断请求 e1000_msix_other”分别注册。当有中断事件发生时，驱动程序不需要读取中断状态寄存器之后再进行处理，从而有效降低了系统延时。

## 15.3 小结

本节主要介绍了 PCI 设备的中断请求在 Linux 系统中的处理过程。Linux 系统的更新速度较快，并不断加入新的功能。本章的内容基于 Linux 系统，目前 Linux 系统支持多种架构的处理器系统，而且得到了极大的普及。对于有志于学习体系结构的工程师而言，深入了解几种操作系统是必须的。而在这些操作系统中，Linux 无疑最为开放，读者也最容易了解其实现细节。但是值得注意的是，在体系结构的学习过程中，不要拘泥于 Linux 系统本身，Linux 系统仅包含了体系结构的部分内容，也只是一种实现方法。

本书到此告一段落，而 PCIe 总线仍然继续向前发展，PCIe V3.0 规范即将发布，其中增加了许多新的功能，而这些新的功能在许多处理器系统中并没有意义。这些新的功能在本书中多有提及，但并不是本书的重点。本书的重点是以 PCIe 总线为例说明处理器的体系结构，是对 PCIe 体系结构进行导读，更准确地说，是以 PCIe 总线为例说明处理器体系结构中局部总线的设计原理与使用方法。

# 参 考 文 献


[1] PCISIG. PCI Local Bus Specification, Revision 3.0 [S]. 2003.

[2] PCISIG. PCI-to-PCI Bridge Architecture Specification, Revision 1.2 [S]. 2003.

[3] PCISIG. PCI Express Base Specification, Revision 2.1 [S]. 2009.

[4] PCISIG. PCI Express Base Specification, Revision 3.0, Version 0.7 [S]. 2009.

[5] PCISIG. PCI Express to PCI/PCI-X Bridge Specification, Revision 1.0 [S]. 2003.

[6] PCISIG. PCI Bus Power Management Interface Specification, Revision 1.2 [S]. 2004.

[7] PCISIG. Address Translation Services, Revision 1.1 [S]. 2009.

[8] PCISIG. Single Root I/O Virtualization and Sharing Specification, Revision 1.0 [S]. 2007.

[9] PCISIG. Multi-Root I/O Virtualization and Sharing Specification, Revision 1.0 [S]. 2008.

[10] Tom Shanley, Don Anderson . "Chapter 25: Transaction Ordering & Deadlocks", PCI System Architecture, Fourth Edition, 651-671 [M]. 1999.

[11] Ravi Budruk, Don Anderson, Tom Shanley. PCI Express System Architecture, Chapter 5 ACK/NAK Protocal [M]. 2003.

[12] Intel. 21555 Non-Transparent PCI-to-PCI Bridge User Manual [S]. 2001.

[13] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 [S]. 2008.

[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 [S]. 2008.

[15] Intel. Mobile Intel® 4 Series Express Chipset Family, Revision 2.1 [S]. 2008.

[16] Intel. Intel I/O Controller Hub 9(ICH9) Family Datasheet [S]. 2008.

[17] Intel. Intel Virtualization Technology for Directed I/O [S]. 2008.

[18] Intel. Interrupt Swizzling Solution for Intel 5000 Chipset Series-based Platforms [OL]. 2006.

[19] Intel. MultiProcessor Specification Version 1.4 [S]. 1997.

[20] Intel. ACPI Component Architecture Programmer Reference—OS-Independent Subsystem, Debugger, and Utilities, Revision 1.22 [S]. 2009.

[21] IBM. Power ISA, Version 2.05, October 2007 [S]. 2007

[22] Freescale. PowerPC e500 Core Family Reference Manual, Revision 1 [S]. April 2005.

[23] Freescale. MPC8548E PowerQUICC™ III Integrated Processor Family Reference Manual, Revision 2 [S]. 2007.

[24] Freescale. MPC8572E PowerQUICC™ III Integrated Processor Family Reference Manual, Revision A [S]. 2008.

[25] Freescale. QorIQ P4080 Communications Process Product Brief [S]. 2009.

[26] Freescale. Embedded Multicore: An introduction [S]. 2009.

[27] AMD. AMD64 Technology—AMD64 Architecture Programmer's Manual Volume 2: System Programming [S]. 2007.

[28] AMD. AMD I/O Virtualization Technology (IOMMU) Specification [S]. 2009.

[29] Xilinx. LogiCORE™ Endpoint PIPE v1.7 for PCI Express® User Guide [S]. 2007.

[30] HP, Intel, Microsoft, Phoenix and Toshiba . Advanced Configuration and Power Interface Specification4.0

[S]. 2009.

[31] Robert A Maddox, Gurbir Singh, Robert J Safranek. Weaving High Performance Multiprocessor Fabric [M]. Intel Press, ISBN 13: 978-1-934053-18-8. 2009.

[32] Elliot Garbus, Peter Sankhagowit, Marc Goldschmidt, Nick Eskandari. Architecture for an I/O processor that Integrates a PCI to PCI Bridge [P], March 1999. US Patent 5,884,027. 1999.

[33] Joe Winkles. Sizing of the Replay Buffer in PCI Express Devices [OL]. October, 2003. MindShare, Inc. 2003.

[34] Joe Winkles. Elastic Buffer Implementations in PCI Express Devices [OL]. November, 2003. MindShare, Inc. 2003.

[35] Roger E Tipley. Split transaction protocol for the peripheral component interconnect bus [P]. US Patent 5533204, Jul 2, 1996.

[36] James E Smith. A Study of Branch Prediction Strategies, Proceedings of the 8th Annual Symposium on Computer Architecture [J], p. 135-148, May 12-14, 1981, Minneapolis, Minnesota, United States. 1981.

[37] T Y Yeh, Y N Patt. Alternative implementation of Two-level Adaptive Branch prediction [J], Proc. 19th Ann. Int '1 Symp. Computer Architecture, pp. 124-134, 1992.

[38] Intel. The White Paper of Intel® Next Generation Nehalem Microarchitecture [OL]. 2009.

[39] Steven P Vanderwiel, David J Lilja. Data Prefetch Mechnisms [J], ACM Computing Surverys, Vol. 32, No. 2. June 2000.

[40] Norman P Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers [J], Proceedings of the 17th Annual International Symposium on Computer Architecture, p. 364-373, May 28-31, 1990, Seattle, Washington, United States. 1990.

[41] Mikko H Lipasti, Christopher B Wilkerson, John Paul Shen. Value locality and Load Value Prediction [J], Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, p. 138-147, October 01-04, 1996, Cambridge, Massachusetts, United States.

[42] F Bonomi, K W Fendick. The Rate-based Flow Control Framework for Available Bit-rate ATM Services [J]. IEEE Network, pp. 25-39, March/April 1995.

[43] William J Dally, Member, IEEE. Virtual-Channel Flow Control [J], IEEE Transcactions On Parallel and Distributed System, Vol. 3, No. 2, March 1992.

[44] H T Kung and Robert Morris. Credit-Based Flow Control for ATM Networks [J], IEEE Network , Pg. 40～48, March/Apricl 1995.

[45] H T Kung, Trevor Blackwell, Alan Chapman. Credit-based Flow Control for ATM Networks: Credit Update Protocal, Adaptive Credit Allocation, and Statistical Multiplexing [J]. Proceedings of the ACM SIGCOMM 1994 Symposium on Communications Architectures, Protocals and Applications, Pg. 101～104 Auguest 31-September 2, 1994.

[46] H T Kuang, Alan Chapman. The FCVC(Flow-controlled Virtual Channels) Proposal for ATM Networks [S], Version 1.1, 1993.

[47] A Parekh and R Gallager. A Generalized Processor Sharing Approach to Flow Control- The Single Node Case [J]. In Technical Report LIDS-TR-2040, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1991.

[48] A Parekh. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks [J], In Technical Report LIDS-TR-2089, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1992.

[49] P Kernami, L Kleinrock. Virtual Cut Through: A New Computer Communication Controller [J]. IEEE Computer Society Press, Oct. 1987, Pg. 230 ~ 234.

[50] M Shreedhar, George Varghese. Efficient fair queueing using deficit round-robin [J]. IEEE/ACM Transactions on Networking, Volume 4, Iuuse 3, June 1996.

[51] Ferguson P and Huston G. Quality of Service: Delivering QoS on the Internet and in Corporate Networks [J]. John Wiley & Sons, Inc., 1998. ISBN 0-471-24358-2.

[52] William J Dally. Performance Analysis of k-ary n-cube Interconnection Networks [J]. IEEE Transactions on Computers, v. 39 n. 6, p. 775-785, June 1990.

[53] Jack Regula. Using PCIe in a Variety of Multiprocessor System Configurations [OL]. http://www.plxtech.com/about/news/archive/articles2007. PLX Technology. 2007.

[54] Albert X, Widmer, Peter A Franaszak. A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code [J]. IBM J. RES. DEVELOP. Vol. 27, No. 5. 1983,

[55] James E J Bottomley. Dynamic DMA mapping using the generic device [OL]. http://www.kernel.org/doc/Documentation/DMA-API.txt.

[56] Jim Handy. Cache Memory Book [M]. the 2nd version. 1998.

[57] V Krishnan. Towards an Integrated IO and Clustering Solution using PCI Express [J]. IEEE International Conference on Cluster Computing (CLUSTER 2007), September, 2007.

[58] Hum, Herbert H J, Goodman, James R. US Patent 6922756-Forward State for Use in Cache Coherency in a Multiprocessor system[P]. 2005.

[59] Jack, Regula. Using PCIe in a Variety of Multiprocessor System Configurations [OL]. http://www.embedded.com/columns/technicalinsights/196902357? _requestid=349156. 2007.

[60] Brian Holden. Latency Comparision Between HyperTransport and PCI-Express In Communication Systems [OL]. http://enterprise2.amd.com/Downloads/Industry/Telecommunications/Latency_Comparison_HyperTransport.pdf. 2006.

[61] Alex Goldhammer, John Ayer Jr. Understanding Performance of PCI Express System [OL]. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf. 2008.

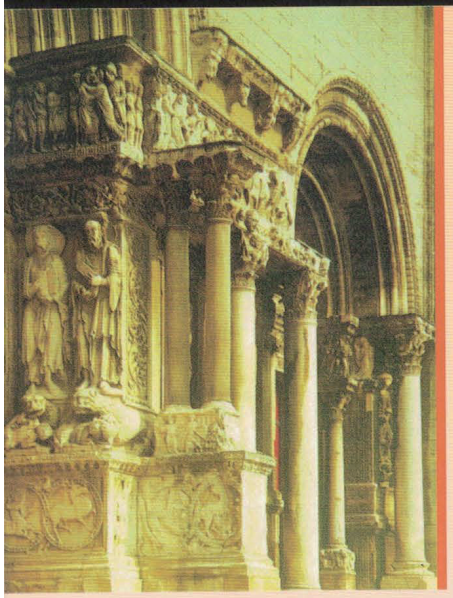
[62] SBS Implementers Forum. System Management Bus Specification Verison 2.0 [S]. August, 2000.

[63] Application Notes from Maxim. Comparing the $I^2C$ Bus to the SMBus [OL]. http://pdfserv.maxim-ic.com/en/an/AN476.pdf. 2000.

[64] PLX. ExpressLane PEX 8518AA/AB/AC 5-Port/16-Lane PCI Express Switch Data Book [S]. 2007.

ISBN 978-7-111-29822-9
策划编辑：时 静
封面设计：子时文化 ZiShi Culture

# PCI Express 体系结构导读



本书讲述了与PCI及PCI Express总线相关的最为基础的内容，并介绍了一些必要的、与PCI总线相关的处理器体系结构知识，这也是本书的重点所在。深入理解处理器体系结构是理解PCI与PCI Express总线的重要基础。

读者通过对本书的学习，可超越PCI 与PCI Express总线自身的内容，理解在一个通用处理器系统中局部总线的设计思路与实现方法，从而理解其他处理器系统使用的局部总线。

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
社服务中心：010-88361066
销售一部：010-68326294
销售二部：010-88379649
读者购书热线：010-88379203
网络服务
教材网：http://www.cmpedu.com
机工官网：http://www.cmpbook.com
机工官博：http://weibo.com/cmp1952
封面无防伪标均为盗版

ISBN 978-7-111-29822-9
9 787111 298229 01>

定价：75.00元