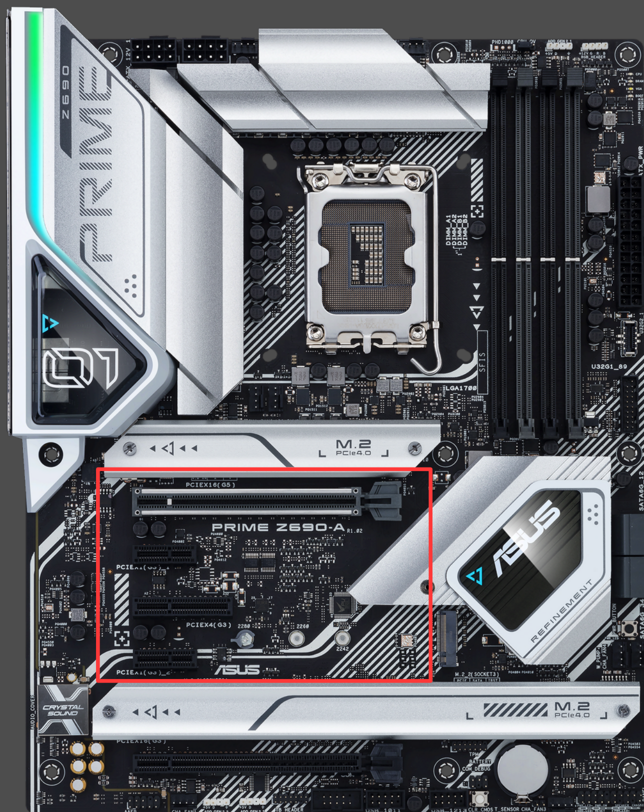
LunaixOS

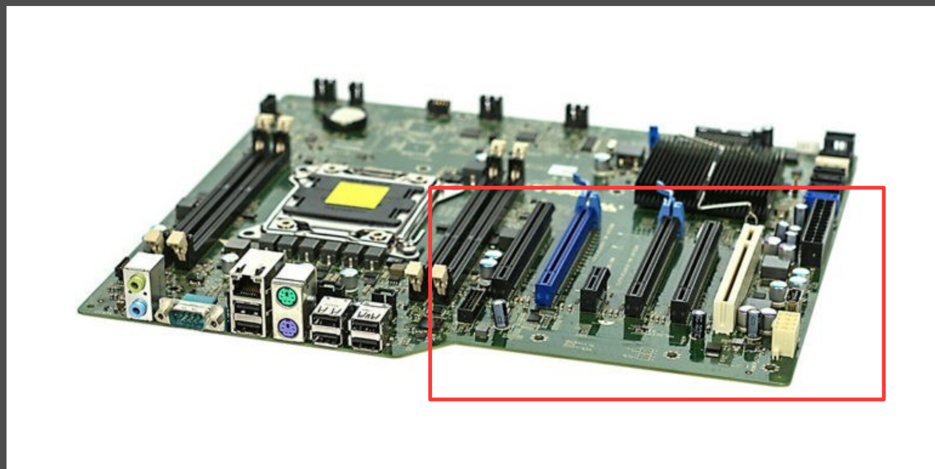
从零开始 自制操作系统

接入 PCI EXPRESS

EP 12

一个家喻户晓的东西……

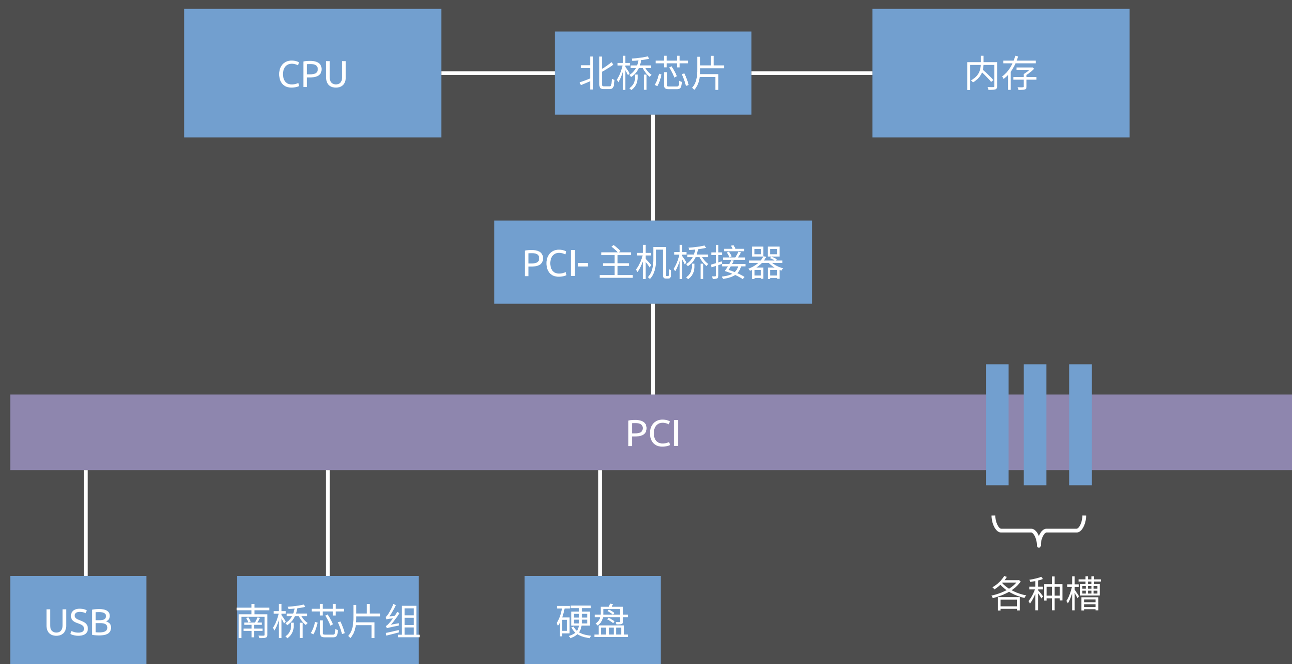




但凡攒过机的，多多少少都知道这个东西。

PCI 可不只是那几个槽！

PCI： 外围设备互联总线



从我们 OS 开发者的角度

我们主要通过 PCI 去……

1. 发现设备
2. 初始化和配置设备
3. 操控设备

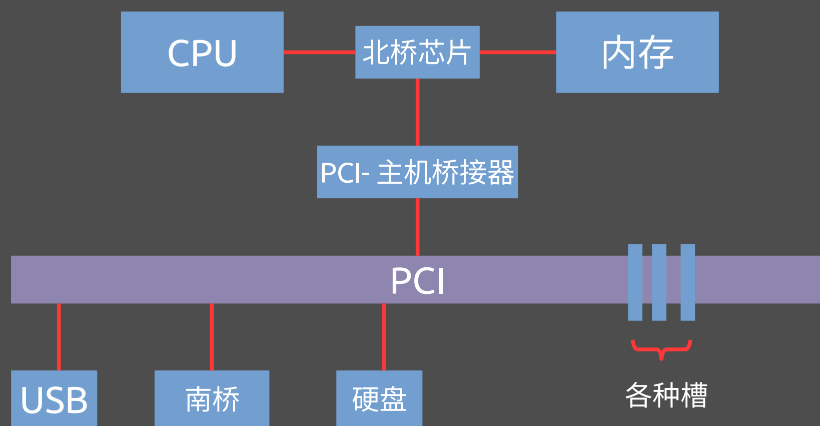
不需要关心……

1. 如何协调设备之间的通讯
2. 如何具体的读写 PCI 总线

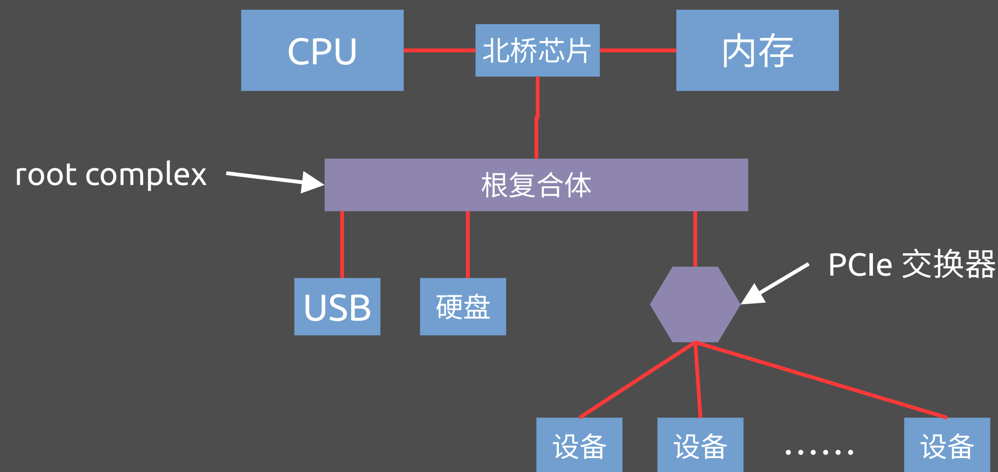
LunaixOS 将实现 PCI 3.0 的规范标准

PCI vs. PCIe ?

PCI 是实打实的总线结构



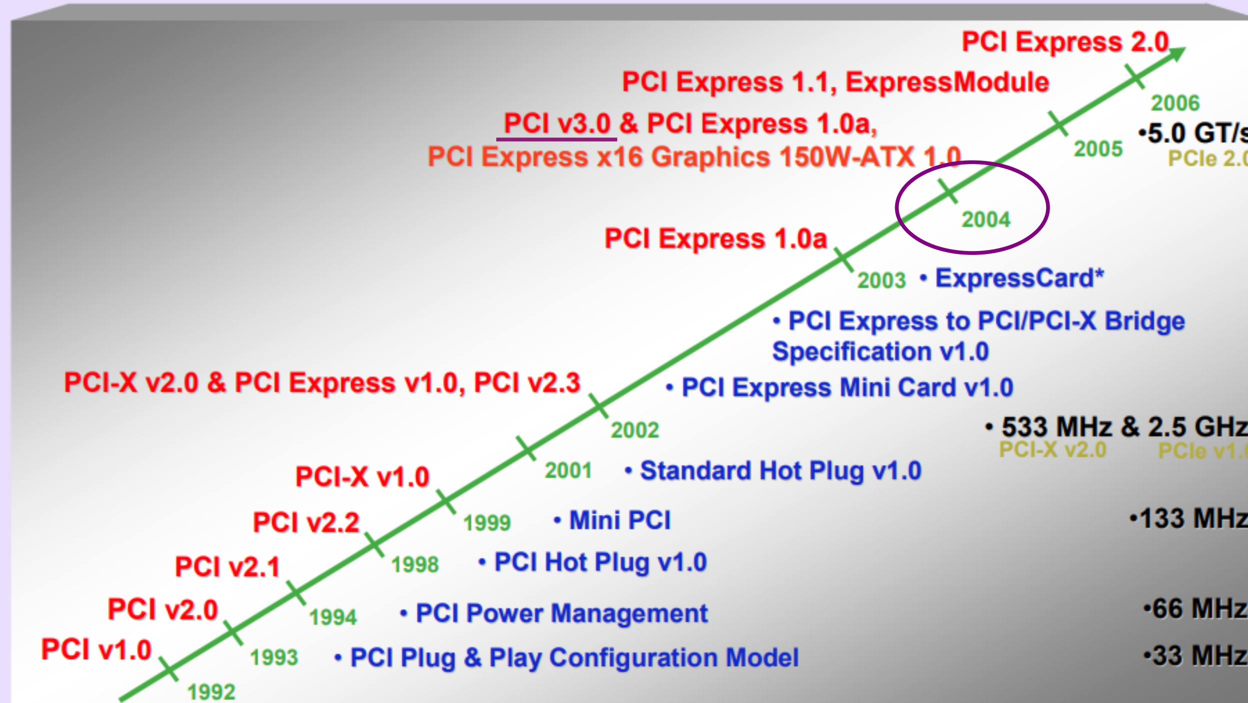
PCIe 基于报文交换机制，类似网络



从软件层面上来看， PCI 和 PCIe 是可以做到无缝兼容的

我们大致的位置

PCI Family History



* "ExpressCard" is a trademark, or registered trademark of PCMCIA in the United States and/or other countries.

Copyright © 2006, PCI-SIG, All Rights Reserved

发现 PCI 设备：配置空间

配置空间是……

- 一组 32 位寄存器组成的区域。
- 大小 256 字节（64 个寄存器）。
- 在前 64 字节包含了一个头部。
- 包括了设备的所有信息以及 PCI 相关设置

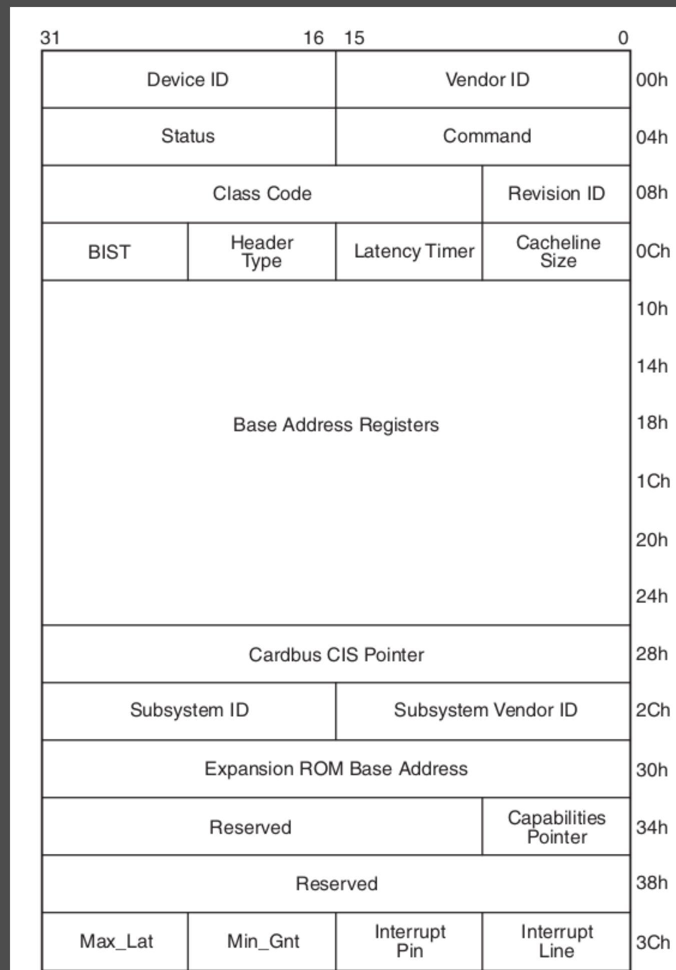
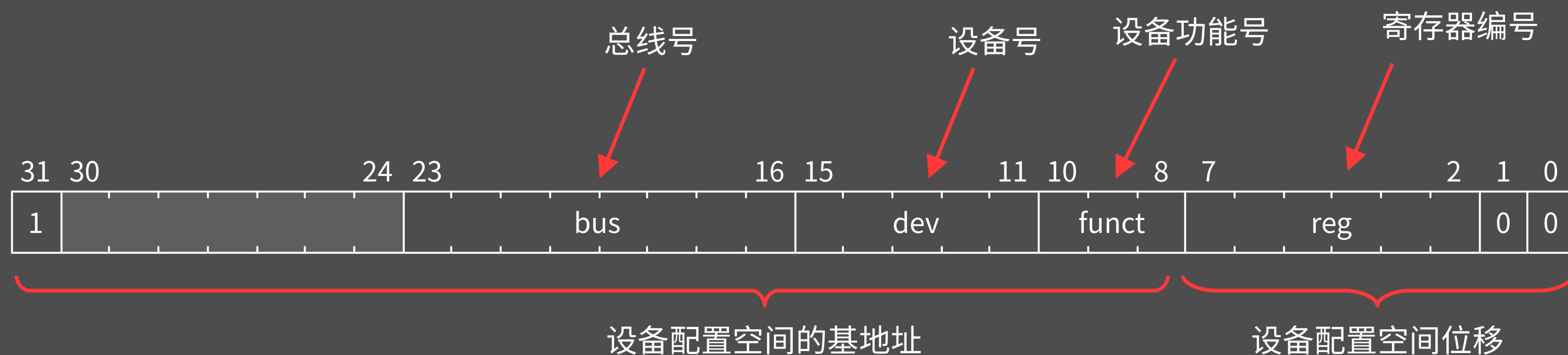
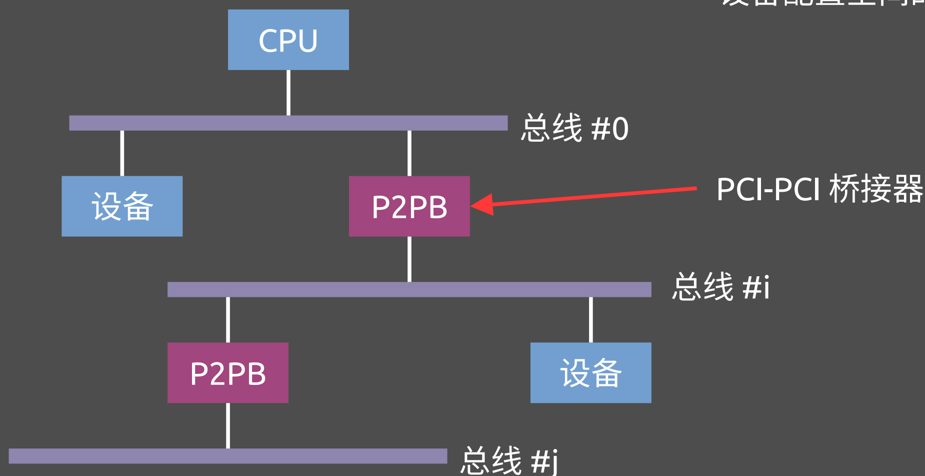


Figure 6-1: Type 00h Configuration Space Header

发现 PCI 设备：访问配置空间

每个设备在 PCI 上都有一个唯一的地址





PCI 是树状结构，最多 256 条总线

发现 PCI 设备：访问配置空间

PCI - 主机适配器提供的 I/O 口

```
#define PCI_CONFIG_ADDR 0xcf8
#define PCI_CONFIG_DATA 0xcfc
```

地址 I/O

寄存器 I/O

```
inline pci_reg_t
pci_read_cspace(uint32_t base, int offset)
{
    io_outl(PCI_CONFIG_ADDR, base | (offset & ~0x3));
    return io_inl(PCI_CONFIG_DATA);
}

inline void
pci_write_cspace(uint32_t base, int offset, pci_reg_t data)
{
    io_outl(PCI_CONFIG_ADDR, base | (offset & ~0x3));
    io_outl(PCI_CONFIG_DATA, data);
}
```

向适配器写入地址

发现 PCI 设备：总线扫描

穷举出所有可能的设备地址，进行探测。

通过检测生产商 ID 来确定设备是否存在。

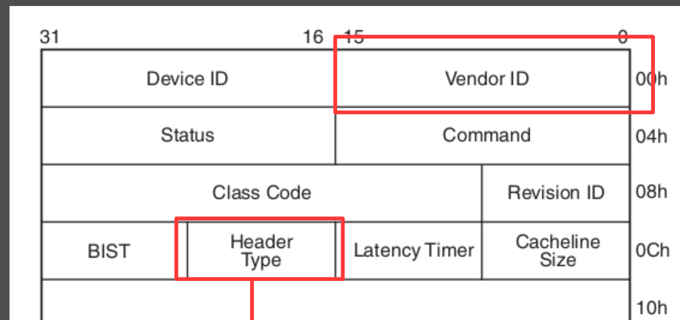
0xFFFF 则为不存在。

256 条总线

每条总线最多 32 个设备

每个设备最多 8 个功能

常数时间： T(n) = 256*32*8=65536



检查第七位是否置位来判断该设备是不是多功能设备。

大多数设备都只有一个功能。

平均时间： T(n) = 2048

发现 PCI 设备：总线扫描

```c
void
pci_probe()
{
    // 暴力扫描所有PCI设备
    // XXX: 尽管最多会有256条PCI总线，但就目前而言，只考虑bus #0就足够了
    for (int bus = 0; bus < 1; bus++) {
        for (int dev = 0; dev < 32; dev++) {
            pci_probe_device(bus, dev, 0);
        }
    }
}
```

```c
void
pci_probe_device(int bus, int dev, int funct)
{
    uint32_t base = PCI_ADDRESS(bus, dev, funct);
    pci_reg_t reg1 = pci_read_cspace(base, 0);

    // Vendor=0xffff则表示设备不存在
    if (PCI_DEV_VENDOR(reg1) == PCI_VENDOR_INVLD) {
        return;
    }

    pci_reg_t hdr_type = pci_read_cspace(base, 0xc);
    hdr_type = (hdr_type >> 16) & 0xff;

    // 防止堆栈溢出
    // QEMU的ICH9/Q35实现似乎有点问题，对于多功能设备的每一个功能的header type
    //  都将第七位置位。而virtualbox 就没有这个毛病。
    if ((hdr_type & 0x80) && funct == 0) {
        hdr_type = hdr_type & ~0x80;
        // 探测多用途设备（multi-function device）
        for (int i = 1; i < 7; i++) {
            pci_probe_device(bus, dev, i);
        }
    }

    if (hdr_type != PCI_TDEV) {
        // XXX: 目前忽略所有桥接设备，比如PCI-PCI桥接器，或者是CardBus桥接器
        return;
    }

    pci_reg_t intr = pci_read_cspace(base, 0x3c);
    pci_reg_t class = pci_read_cspace(base, 0x8);

    struct pci_device* device = lxmalloc(sizeof(struct pci_device));
    *device = (struct pci_device){ .cspace_base = base,
                                   .class_info = class,
                                   .device_info = reg1,
                                   .intr_info = intr };

    pci_probe_msi_info(device);

    llist_append(&pci_devices, &device->dev_chain);
```

初始化 PCI 设备

PCI 相关的初始化： Command

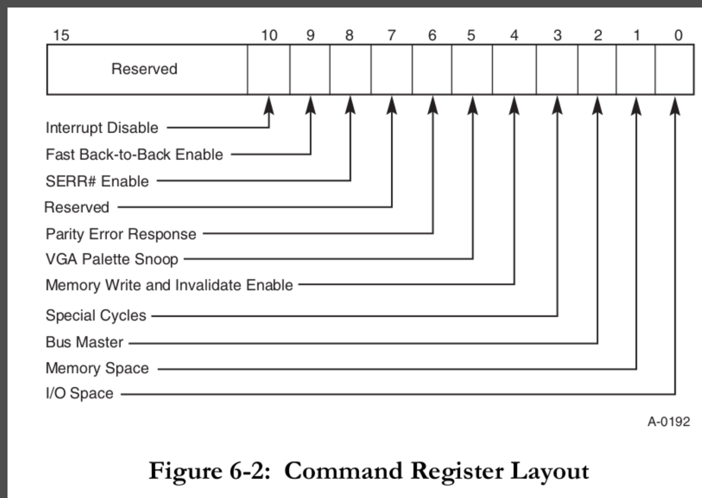


Figure 6-2: Command Register Layout

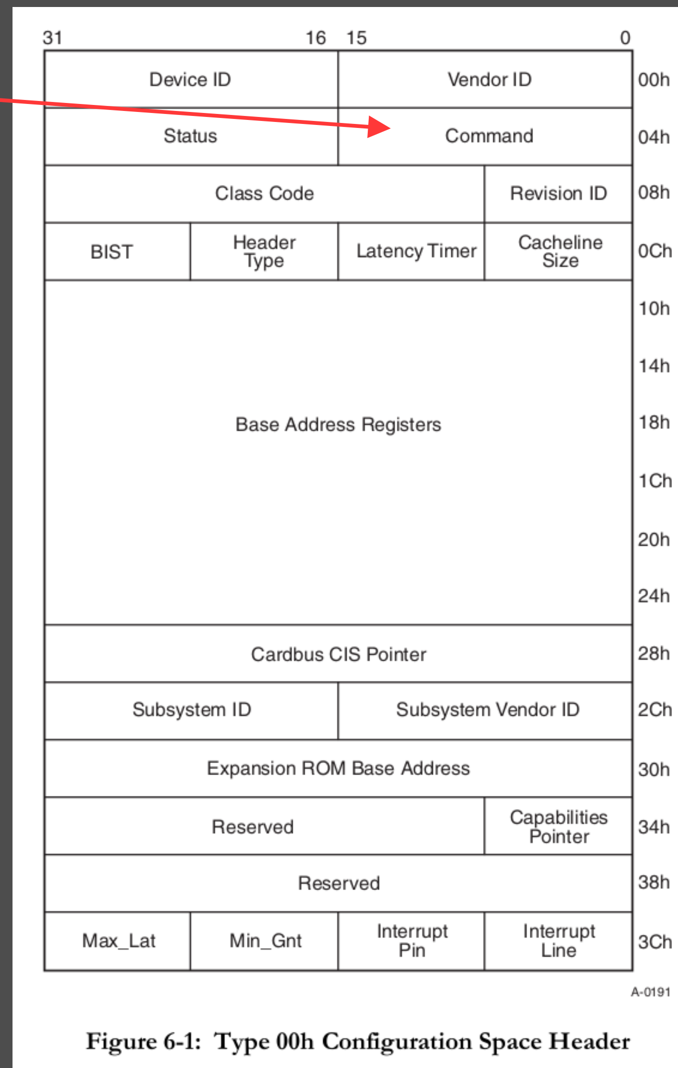


Figure 6-1: Type 00h Configuration Space Header

设备相关，不同的设备，对每个位的具体解释不一样。

操作 PCI 设备

也是设备相关的问题。

不同的设备会提供不同的寄存器， OS 通过往这些寄存器里写值，向设备发送命令。

设备相关的寄存器的基地址会在 **BAR** 中给出：

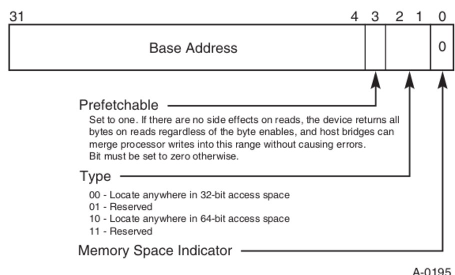


Figure 6-5: Base Address Register for Memory

对于内存映射的寄存器（MMIO）

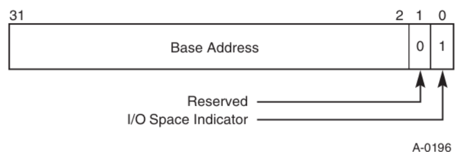


Figure 6-6: Base Address Register for I/O

对于 I/O 映射的寄存器

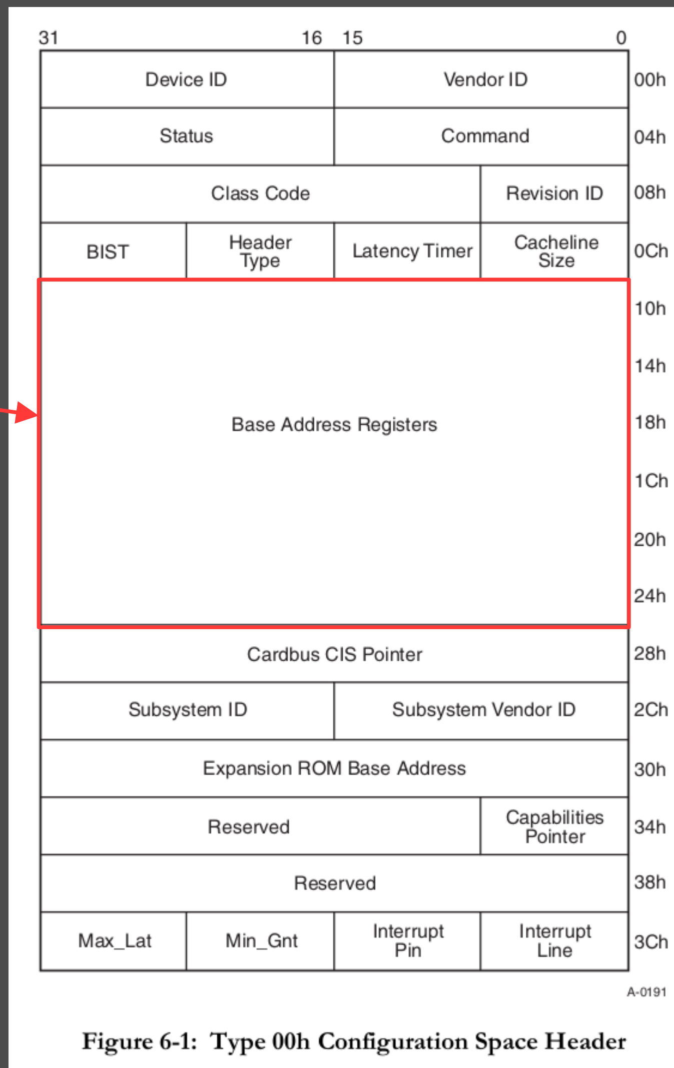


Figure 6-1: Type 00h Configuration Space Header

操作 PCI 设备：设置中断

一个设备需要通过中断去和 CPU 通讯。

Interrupt Line： 使用的 IRQ

那么我们可以通过 ACPI 查询到 IRQ → IOAPIC 针脚的映射关系，从而通过 IOAPIC 来管理这些中断。

但这种方法有两个很严重的问题……

1. IRQ 共用的情况
2. 竞态条件

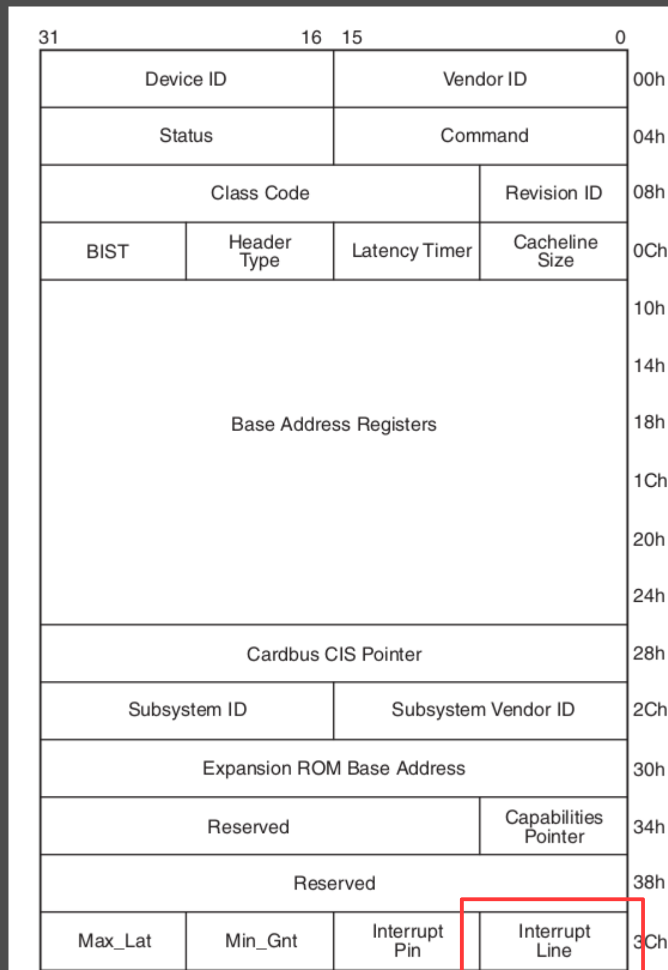
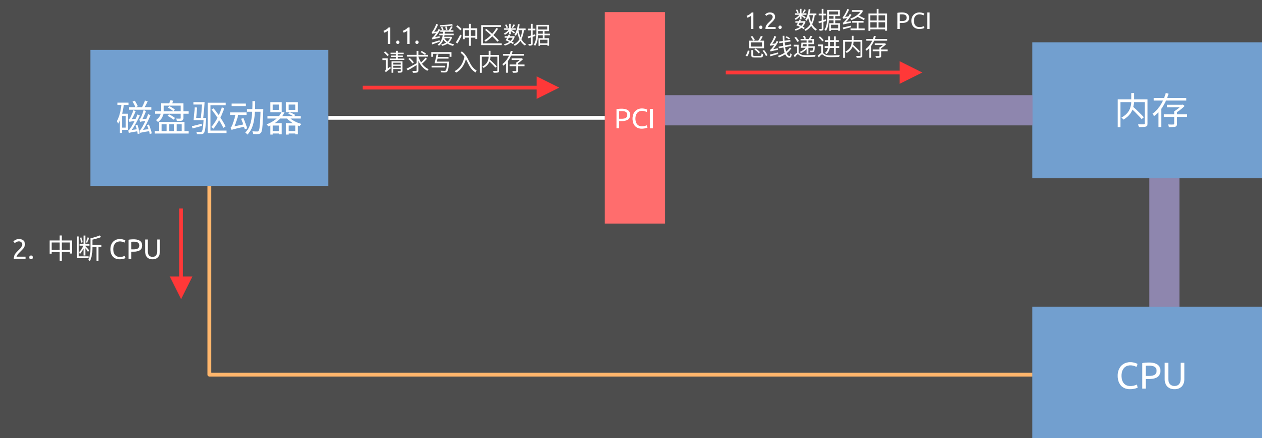


Figure 6-1: Type 00h Configuration Space Header

PCI 中断机制：竞态条件

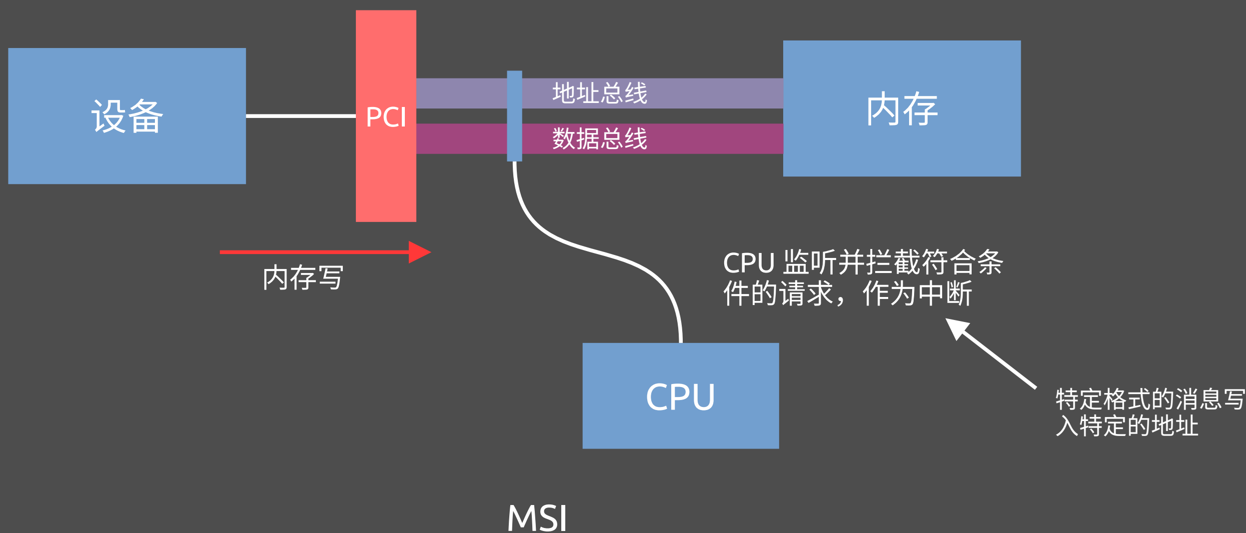


正常执行顺序： 1.1 → 1.2 → 2

如果 PCI 繁忙，决定稍后再写入数据：
1.1 → 2 → 1.1

另一种办法： MSI （信息中断）

传统针脚式中断



MSI

PCI 中断机制： MSI 解决竞态条件

将中断的产生纳入 PCI 的总线的仲裁机制。

利用 PCI 的总线锁进行同步。

→ 确保了 MSI 只会在数据送达内存后被送出

判断设备是否支持 MSI

在 PCI 下面： MSI 为可选功能。在 PCIe 里面则是强制要求。

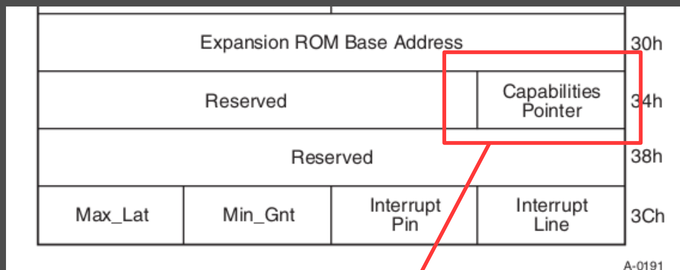
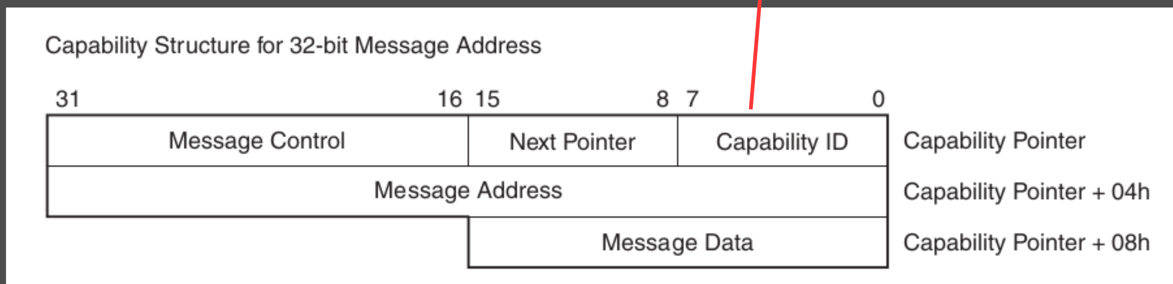


Figure 6-1: Type 00h Configuration Space Header

遍历“能力链表”去查找 MSI 。

0x5： MSI 功能

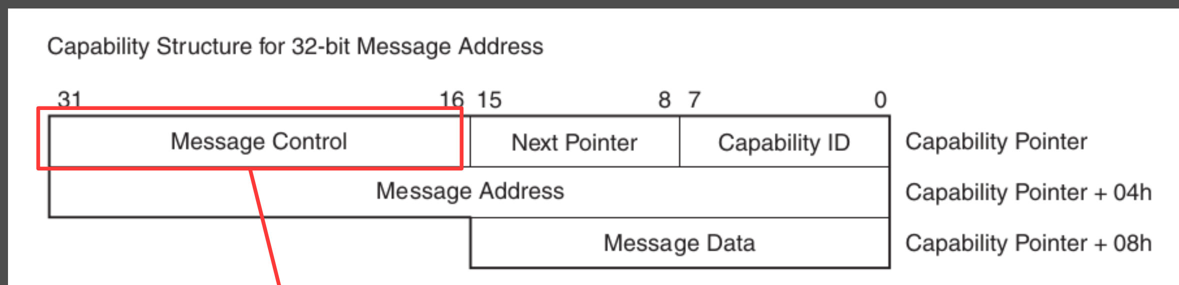


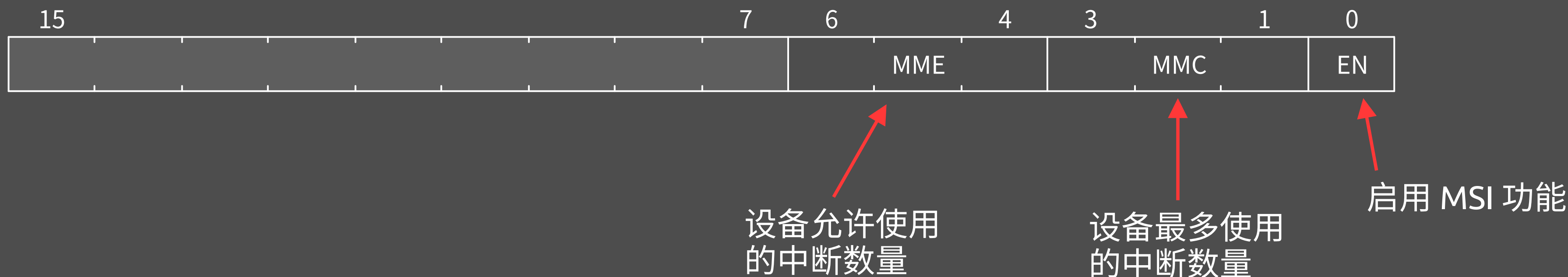
判断设备是否支持 MSI

status 寄存器第五位告诉我们该设备是否存在能力链表

```c
pci_reg_t status =
    pci_read_cspace(device->cspace_base, PCI_REG_STATUS_CMD) >> 16;

if (!(status & 0x10)) {
    device->msi_loc = 0;
    return;
}

pci_reg_t cap_ptr = pci_read_cspace(device->cspace_base, 0x34) & 0xff;
uint32_t cap_hdr;

while (cap_ptr) {
    cap_hdr = pci_read_cspace(device->cspace_base, cap_ptr);
    if ((cap_hdr & 0xff) == 0x5) {
        // MSI
        device->msi_loc = cap_ptr;
        return;
    }
    cap_ptr = (cap_hdr >> 8) & 0xff;
}
```

设置一个 MSI：启用 MSI 功能






```
// manipulate the MSI_CTRL to allow device using MSI to request service.
reg1 = ((((reg1 >> 16) & ~0x70) | 0x1) << 16) | (reg1 & 0xffff);
pci_write_cspace(device->cspace_base, device->msi_loc, reg1);
```

设置一个 MSI： 让 CPU 识别 MSI

Capability Structure for 32-bit Message Address

31 – 16	15 – 8	7 – 0	
Message Control	Next Pointer	Capability ID	Capability Pointer
Message Address			Capability Pointer + 04h
	Message Data		Capability Pointer + 08h

31 – 20	19 – 12	11 – 4	3	2	1 – 0
0FEEH	Destination ID	Reserved	RH	DM	XX
	0		1	0	

Figure 10-24. Layout of the MSI Message Address Register

63 – 32
Reserved

31 – 16	15	14	13 – 11	10 – 8	7 – 0
Reserved	0	0	Reserved	000	Vector

Trigger Mode
0 - Edge
1 - Level

Level for Trigger Mode = 0
X - Don't care
Level for Trigger Mode = 1
0 - Deassert
1 - Assert

Delivery Mode
000 - Fixed
001 - Lowest Priority
010 - SMI
011 - Reserved
100 - NMI
101 - INIT
110 - Reserved
111 - ExtINT

Figure 10-25. Layout of the MSI Message Data Register

设置一个 MSI：让 CPU 识别 MSI

0xFEE00000

```c
// Dest: APIC#0, Physical Destination, No redirection
uint32_t msi_addr = (__APIC_BASE_PADDR | 0x8);

// Edge trigger, Fixed delivery
uint32_t msi_data = vector;

pci_write_cspace(
  device->cspace_base, PCI_MSI_ADDR(device->msi_loc), msi_addr);
pci_write_cspace(
  device->cspace_base, PCI_MSI_DATA(device->msi_loc), msi_data & 0xffff);
```

配置 MSI： 全部封装在一起

```c
void
pci_setup_msi(struct pci_device* device, int vector)
{
    // Dest: APIC#0, Physical Destination, No redirection
    uint32_t msi_addr = (__APIC_BASE_PADDR | 0x8);

    // Edge trigger, Fixed delivery
    uint32_t msi_data = vector;

    pci_write_cspace(
      device->cspace_base, PCI_MSI_ADDR(device->msi_loc), msi_addr);
    pci_write_cspace(
      device->cspace_base, PCI_MSI_DATA(device->msi_loc), msi_data & 0xffff);

    pci_reg_t reg1 = pci_read_cspace(device->cspace_base, device->msi_loc);

    // manipulate the MSI_CTRL to allow device using MSI to request service.
    reg1 = ((((reg1 >> 16) & ~0x70) | 0x1) << 16) | (reg1 & 0xffff);
    pci_write_cspace(device->cspace_base, device->msi_loc, reg1);
}
```

一个额外的话题： PCIe 的支持

如同在一开始时所说，上述 PCI 的方法在 PCIe 中全部适用，不需要进行任何的更改。

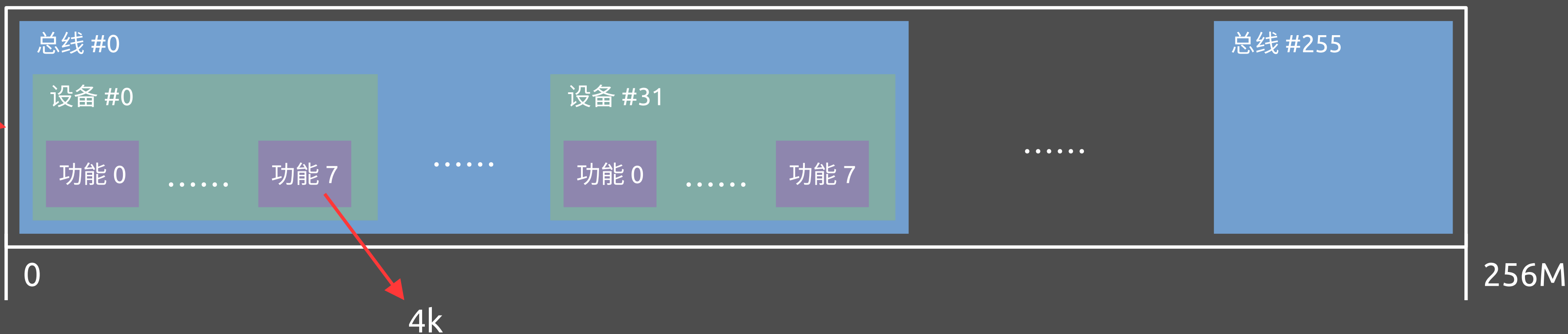
不同的一点： PCIe 的配置空间扩容到 4096 字节。需要使用 PCIe 专用的访问方式去访问。

映射到内存的配置空间

Table 4-2: MCFG Table to Support Enhanced Configuration Space Access

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	"MCFG". Signature for the Memory mapped configuration space base address Description Table. (refer to Note 1)
Length	4	4	Length, in bytes, of the entire MCFG Description table including the memory mapped configuration space base address allocation structures.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the MCFG Description Table, the table ID is the manufacture model ID
OEM Revision	4	24	OEM revision of MCFG table for supplied OEM Table ID
Creator ID	4	28	Vendor ID of utility that created the table
Creator Revision	4	32	Revision of utility that created the table
Reserved	8	36	Reserved
Configuration space base address allocation structure [n]	---	44	A list of the memory mapped configuration base address allocation structures. This list will contain one entry corresponding to each PCI Segment Group present in the platform. The structure of this entry is defined in Table 4-3.

基地址通过查询 ACPI 的 MCFG 表可知



下一步， SATA 驱动!