Understanding of pci_read_bases of linux PCI/PCIe driver

Understanding of pci_read_bases of linux PCI/PCIe driver

1 pci_read_bases

The process of device enumeration is as follows. The pci_read_bases function will read the relevant information of the PCIe device, mainly related to the resource size information required by the PCI/PCIe device.

1.1 The calling process of pci_read_bases

```
    pci_scan_root_bus(&pdev->dev, 0, &rockchip_pcie_ops, rockchip, &res);
                pci_scan_root_bus_msi
            pci_scan_child_bus
                pci_scan_slot
                        dev = pci_scan_single_device(bus, devfn);
                                                dev = pci_scan_device(bus, devfn);
                                                        struct pci_dev *dev;
                                                        dev = pci_alloc_dev(bus);
                                                        pci_setup_device
                                pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
                        pci_device_add(dev, bus);
```

1.2 pci_read_bases function definition

- pci_read_bases first traverses the BAR configuration of howmany PCI/PCIe devices in turn
- struct resource *res = &dev->resource[pos] Obtain the resource structure of the BAR corresponding to the corresponding PCI/PCIe device, and fill in the resource information obtained later
- __pci_read_base is used for specific PCI/PCIe device resource acquisition and analysis, which will be further analyzed later
- If the rom is valid, it will further parse the resource information of the extended configuration space of PCI_ROM_RESOURCE (#6: expansion ROM resource)

```
static void pci_read_bases(struct pci_dev *dev, unsigned int howmany, int rom)
{
        unsigned int pos, reg;

        if (dev->non_compliant_bars)
                return;

        /* Per PCIe r4.0, sec 9.3.4.1.11, the VF BARs are all RO Zero */
        if (dev->is_virtfn)
                return;

        for (pos = 0; pos < howmany; pos++) {
                struct resource *res = &dev->resource[pos];
```

```
                reg = PCI_BASE_ADDRESS_0 + (pos << 2);
                pos += __pci_read_base(dev, pci_bar_unknown, res, reg);
        }

        if (rom) {
                struct resource *res = &dev->resource[PCI_ROM_RESOURCE];
                dev->rom_base_reg = rom;
                res->flags = IORESOURCE_MEM | IORESOURCE_PREFETCH |
                                IORESOURCE_READONLY | IORESOURCE_SIZEALIGN;
                __pci_read_base(dev, pci_bar_mem32, res, rom);
        }
}
```

1.3 __pci_read_base

- Consistent with the meaning of the declaration of the function itself, read the BAR register of the PCI/PCIe device, parse the attributes of the PCI/PCIe device and resource requirement information
- The process of reading the BAR register
  - Read BAR, retain the original value
  - Write 0xFFFFFFFF to BAR
  - After reading it out, parse out the required address space size and record it in pci_dev->resource[ ]
    - pci_dev->resource[ ].start = 0;
    - pci_dev->resource[ ].end = size - 1;
  - Write the original value to the BAR register
- According to the l and sz read in the previous step, calculate the base address and size of the resources required by the current BAR of the device, and configure it into the resource of the corresponding device BAR through pcibios_bus_to_resource, and use it for resource allocation later

```

/**
 * pci_read_base - Read a PCI BAR
 * @dev: the PCI device
 * @type: type of the BAR
 * @res: resource buffer to be filled in
 * @pos: BAR position in the config space
 *
 * Returns 1 if the BAR is 64-bit, or 0 if 32-bit.
 */
int __pci_read_base(struct pci_dev *dev, enum pci_bar_type type,
                    struct resource *res, unsigned int pos)
{
        u32 l = 0, sz = 0, mask;
        u64 l64, sz64, mask64;
        u16 orig_cmd;
        struct pci_bus_region region, inverted_region;pci_bus_type

        mask = type ? PCI_ROM_ADDRESS_MASK : ~0;

        /* No printks while decoding is disabled! */
        if (!dev->mmio_always_on) {
                pci_read_config_word(dev, PCI_COMMAND, &orig_cmd);
                if (orig_cmd & PCI_COMMAND_DECODE_ENABLE) {
                        pci_write_config_word(dev, PCI_COMMAND,
                                orig_cmd & ~PCI_COMMAND_DECODE_ENABLE);
                }
        }

        res->name = pci_name(dev);

        /* 读取BAR寄存器的过程 */
        pci_read_config_dword(dev, pos, &l);
        pci_write_config_dword(dev, pos, l | mask);
        pci_read_config_dword(dev, pos, &sz);
        pci_write_config_dword(dev, pos, l);
```

```
        /*
         * All bits set in sz means the device isn't working properly.
         * If the BAR isn't implemented, all bits must be 0.  If it's a
         * memory BAR or a ROM, bit 0 must be clear; if it's an io BAR, bit
         * 1 must be clear.
         */
        if (sz == 0xffffffff)
                sz = 0;

        /*
         * I don't know how l can have all bits set.  Copied from old code.
         * Maybe it fixes a bug on some ancient platform.
         */
        if (l == 0xffffffff)
                l = 0;

        /* 对于初始不知道是什么类型PCI/PCIe 设备的，采用的是pci_bar_unknown的配置 */
        if (type == pci_bar_unknown) {
                res->flags = decode_bar(dev, l);
                res->flags |= IORESOURCE_SIZEALIGN;
                /* 对于IO类型的设备的配置解析 */
                if (res->flags & IORESOURCE_IO) {
                        /* l64表示数据原始配置信息，sz64表示的是IO设备需要的资源大小 */
                        l64 = l & PCI_BASE_ADDRESS_IO_MASK;
                        sz64 = sz & PCI_BASE_ADDRESS_IO_MASK;
                        mask64 = PCI_BASE_ADDRESS_IO_MASK & (u32)IO_SPACE_LIMIT;
                } else {
                        /* 对于MEM设备l64表示数据原始配置信息，sz64表示的是MEM设备需要的资源大小 */
                        l64 = l & PCI_BASE_ADDRESS_MEM_MASK;
                        sz64 = sz & PCI_BASE_ADDRESS_MEM_MASK;
                        mask64 = (u32)PCI_BASE_ADDRESS_MEM_MASK;
                }
        } else {
                /* 对于其他已知类型的设备则直接走该流程 */
                if (l & PCI_ROM_ADDRESS_ENABLE)
                        res->flags |= IORESOURCE_ROM_ENABLE;
                l64 = l & PCI_ROM_ADDRESS_MASK;
                sz64 = sz & PCI_ROM_ADDRESS_MASK;
                mask64 = PCI_ROM_ADDRESS_MASK;
        }

        /* 对于MEM64的设备，两个相邻的BAR组成一个对应的64位的有效地址，
           该处所读取出的配置作为对应的64位地址的高位，而MEM大小也作为高位 */
        if (res->flags & IORESOURCE_MEM_64) {
                pci_read_config_dword(dev, pos + 4, &l);
                pci_write_config_dword(dev, pos + 4, ~0);
                pci_read_config_dword(dev, pos + 4, &sz);
                pci_write_config_dword(dev, pos + 4, l);

                l64 |= ((u64)l << 32);
                sz64 |= ((u64)sz << 32);
                mask64 |= ((u64)~0 << 32);
        }

        if (!dev->mmio_always_on && (orig_cmd & PCI_COMMAND_DECODE_ENABLE))
                pci_write_config_word(dev, PCI_COMMAND, orig_cmd);

        if (!sz64)
                goto fail;

        /* 按照对齐的要求去调整PCI/PCIe设备的大小 */
        sz64 = pci_size(l64, sz64, mask64);
        if (!sz64) {
                pci_info(dev, FW_BUG "reg 0x%x: invalid BAR (can't size)\n",
                         pos);
                goto fail;
```

```
        }

        if (res->flags & IORESOURCE_MEM_64) {
                if ((sizeof(pci_bus_addr_t) < 8 || sizeof(resource_size_t) < 8)
                    && sz64 > 0x100000000ULL) {
                        res->flags |= IORESOURCE_UNSET | IORESOURCE_DISABLED;
                        res->start = 0;
                        res->end = 0;
                        pci_err(dev, "reg 0x%x: can't handle BAR larger than 4GB (size %#010llx)\n",
                                pos, (unsigned long long)sz64);
                        goto out;
                }

                if ((sizeof(pci_bus_addr_t) < 8) && l) {
                        /* Above 32-bit boundary; try to reallocate */
                        res->flags |= IORESOURCE_UNSET;
                        res->start = 0;
                        res->end = sz64 - 1;
                        pci_info(dev, "reg 0x%x: can't handle BAR above 4GB (bus address %#010llx)\n",
                                 pos, (unsigned long long)l64);
                        goto out;
                }
        }

        region.start = l64;
        region.end = l64 + sz64 - 1;

        pcibios_bus_to_resource(dev->bus, res, &region);
        pcibios_resource_to_bus(dev->bus, &inverted_region, res);

        /*
         * If "A" is a BAR value (a bus address), "bus_to_resource(A)" is
         * the corresponding resource address (the physical address used by
         * the CPU.  Converting that resource address back to a bus address
         * should yield the original BAR value:
         *
         *     resource_to_bus(bus_to_resource(A)) == A
         *
         * If it doesn't, CPU accesses to "bus_to_resource(A)" will not
         * be claimed by the device.
         */
        if (inverted_region.start != region.start) {
                res->flags |= IORESOURCE_UNSET;
                res->start = 0;
                res->end = region.end - region.start;
                pci_info(dev, "reg 0x%x: initial BAR value %#010llx invalid\n",
                         pos, (unsigned long long)region.start);
        }

        goto out;



fail:
        res->flags = 0;
out:
        if (res->flags)
                pci_info(dev, "reg 0x%x: %pR\n", pos, res);

        return (res->flags & IORESOURCE_MEM_64) ? 1 : 0;
}
```

1.4 pcibios_bus_to_resource

- The resource data corresponding to the bridge->windows used in this function is parsed out during the initialization process of the pci_parse_request_of_pci_ranges function. In the ARM architecture, the PCI domain address and the processor domain address are generally consistent. , so usually the value of window->offset is also 0

- res->start = region->start + offset and res->end = region->end + offset need to be used later in resource allocation as the basis for resource allocation.

```
void pcibios_bus_to_resource(struct pci_bus *bus, struct resource *res,
                             struct pci_bus_region *region)
{
        struct pci_host_bridge *bridge = pci_find_host_bridge(bus);
        struct resource_entry *window;
        resource_size_t offset = 0;

        resource_list_for_each_entry(window, &bridge->windows) {
                struct pci_bus_region bus_region;

                if (resource_type(res) != resource_type(window->res))
                        continue;

                bus_region.start = window->res->start - window->offset;
                bus_region.end = window->res->end - window->offset;

                if (region_contains(&bus_region, region)) {
                        offset = window->offset;
                        break;
                }
        }

        res->start = region->start + offset;
        res->end = region->end + offset;
}
```

The knowledge points of the article are matched with the official knowledge files, and relevant knowledge can be further learned