rk3399 PCIe rc device enumeration device resource identification analysis

原创 garden of idyllic poets Modified at 2022-05-16 23:52:27 1181 Favorite 15 copyright
Category column: PCI/PCIe topic Article tags: driver development PCIe RC PCIe host

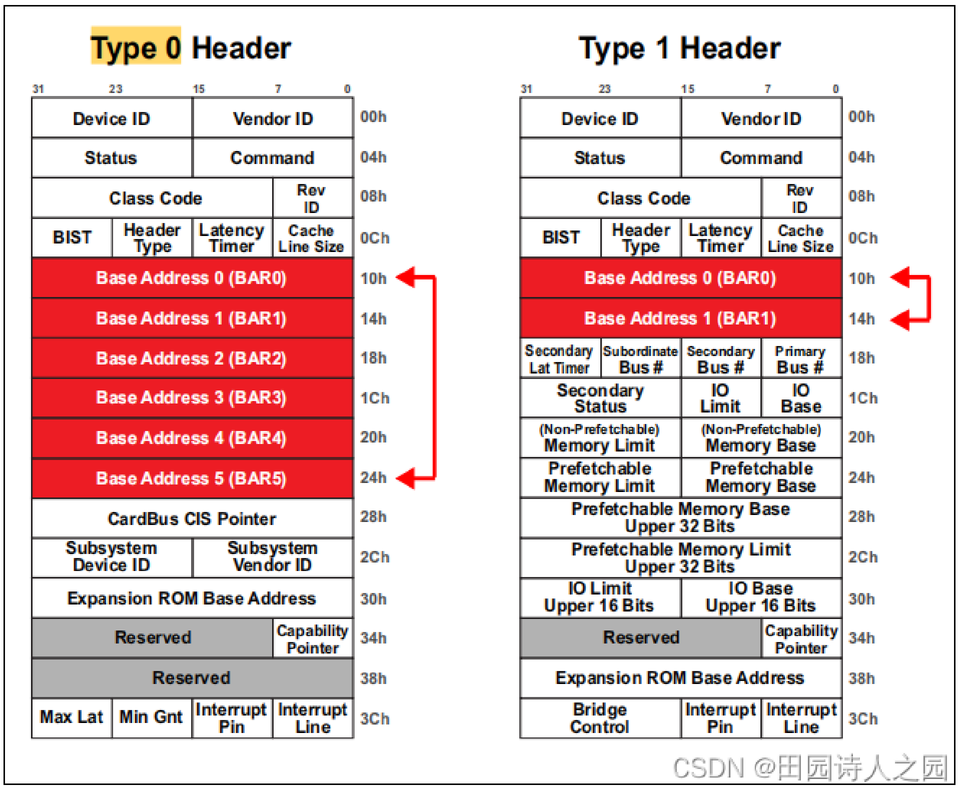
rk3399 PCIe rc device enumeration device resource identification analysis

Based on the learning summary of Mr. Wei Dongshan's PCIe learning topic

1. Device configuration space

Figure 4-2 in "PCI Express Technology 3.0.pdf" shows the configuration space diagram of PCIe devices and switch (or bridge);

Figure 4-2: BARs in Configuration Space


Type 0 Header
31 23 15 7 0
Device ID | Vendor ID 00h
Status | Command 04h
Class Code | Rev ID 08h
BIST | Header Type | Latency Timer | Cache Line Size 0Ch
Base Address 0 (BAR0) 10h
Base Address 1 (BAR1) 14h
Base Address 2 (BAR2) 18h
Base Address 3 (BAR3) 1Ch
Base Address 4 (BAR4) 20h
Base Address 5 (BAR5) 24h
CardBus CIS Pointer 28h
Subsystem Device ID | Subsystem Vendor ID 2Ch
Expansion ROM Base Address 30h
Reserved | Capability Pointer 34h
Reserved 38h
Max Lat | Min Gnt | Interrupt Pin | Interrupt Line 3Ch
Type 1 Header
31 23 15 7 0
Device ID | Vendor ID 00h
Status | Command 04h
Class Code | Rev ID 08h
BIST | Header Type | Latency Timer | Cache Line Size 0Ch
Base Address 0 (BAR0) 10h
Base Address 1 (BAR1) 14h
Secondary Lat Timer | Subordinate Bus # | Secondary Bus # | Primary Bus # 18h
Secondary Status | IO Limit | IO Base 1Ch
(Non-Prefetchable) Memory Limit | (Non-Prefetchable) Memory Base 20h
Prefetchable Memory Limit | Prefetchable Memory Base 24h
Prefetchable Memory Base Upper 32 Bits 28h
Prefetchable Memory Limit Upper 32 Bits 2Ch
IO Limit Upper 16 Bits | IO Base Upper 16 Bits 30h
Reserved | Capability Pointer 34h
Expansion ROM Base Address 38h
Bridge Control | Interrupt Pin | Interrupt Line 3Ch
CSDN @田园诗人之园

The purpose of using PCI/PCIe is to simply access it: read and write PCI/PCIe devices like reading and writing memory.

Ask:

- Which addresses are used to read and write to the device?

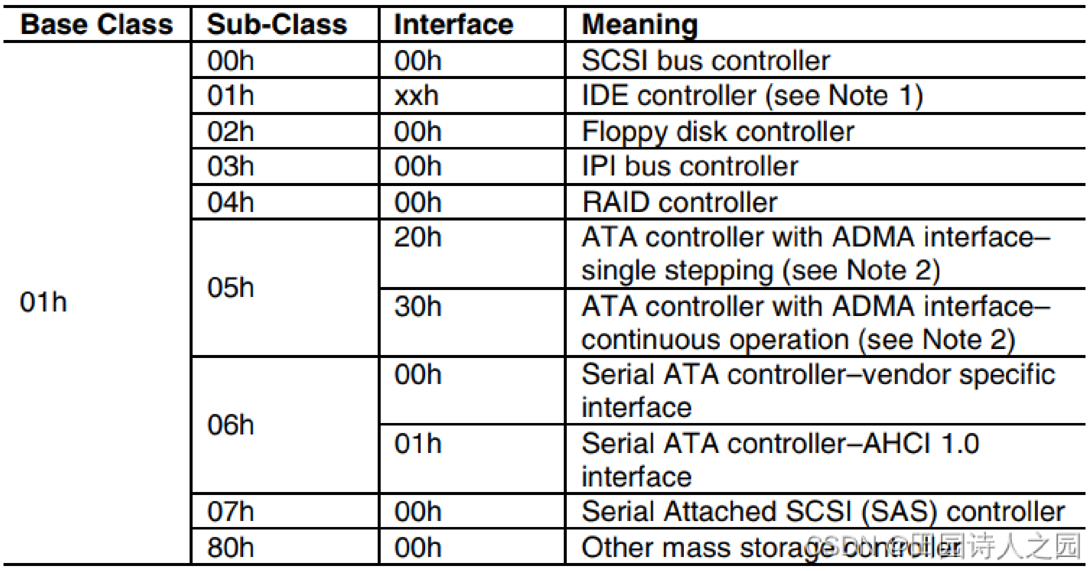
- What is the range of these addresses?
- Is it accessed like memory, or is it accessed like IO?

Each PCI/PCIe device has a configuration space, which is a series of registers. For ordinary devices, its configuration space includes:

- Device ID
- Vendor ID
- Class Code: What kind of equipment? Storage device? display screen? wait
- 6 Base Address Registers:

1.1Device information

- Vendor ID: Vendor ID, the PCI SIG organization assigns a unique ID to each vendor
- Device ID: The manufacturer assigns a Device ID to a certain type of product
- Revision ID: The version number customized by the manufacturer, which can be considered as an extension of Device ID
- Header Type:
  - bit[7]: 1 - it is a multifunction device ("multi-function"), 0 - it is a single function device ("single-function")
  - bit[6:0]: 00h-normal equipment, 01h-switch or bridge equipment, this value also determines the meaning of the beginning of the offset address 10h in the configuration space
- Class Code: This is a read-only register , it contains 3 bytes, used to indicate the function of the device, it is divided into 3 parts
  - Highest byte: Indicates "base class", which is used to indicate that it belongs to the memory card, graphics card, etc.
  - Middle byte: Indicates "sub-class", subdivide the category
  - Lowest byte: used to represent the programming interface "Interface" at the register level
  - The example is as follows: When the Base Class is 01h, it means that it is a storage device, but sub-class and Interface subdivisions can also be used



Base Class	Sub-Class	Interface	Meaning
01h	00h	00h	SCSI bus controller
	01h	xxh	IDE controller (see Note 1)
	02h	00h	Floppy disk controller
	03h	00h	IPI bus controller
	04h	00h	RAID controller
	05h	20h	ATA controller with ADMA interface–single stepping (see Note 2)
		30h	ATA controller with ADMA interface–continuous operation (see Note 2)
	06h	00h	Serial ATA controller–vendor specific interface
		01h	Serial ATA controller–AHCI 1.0 interface
	07h	00h	Serial Attached SCSI (SAS) controller
	80h	00h	Other mass storage controller

1.2 Base Address

Ordinary PCI/PCIe devices have 6 base address registers, referred to as BAR:

BARs are used to:

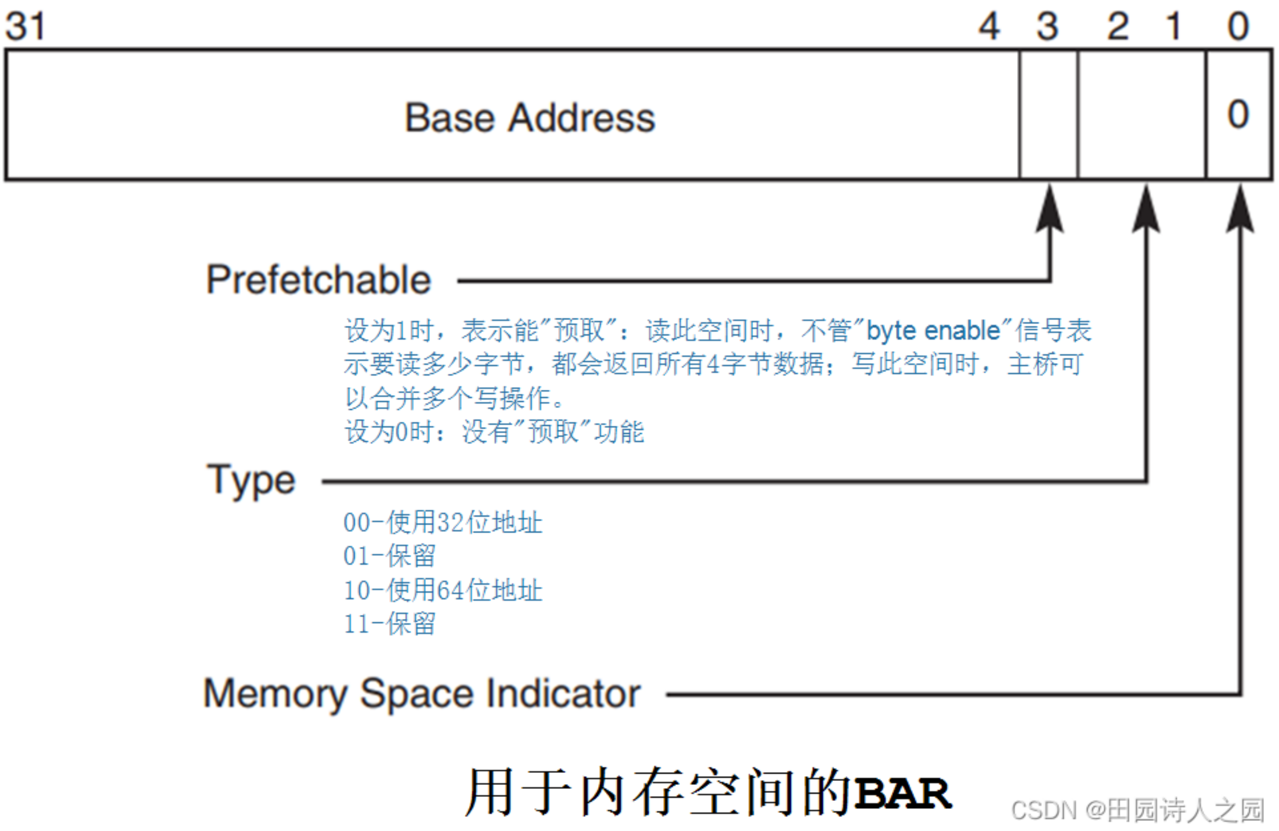
- What type of space is required for the statement: memory (32-bit address or 64-bit address), IO
- How much space is required to declare
- Save the PCI space base address assigned to it by the master

The address space can be divided into two categories: memory (Memory), IO:

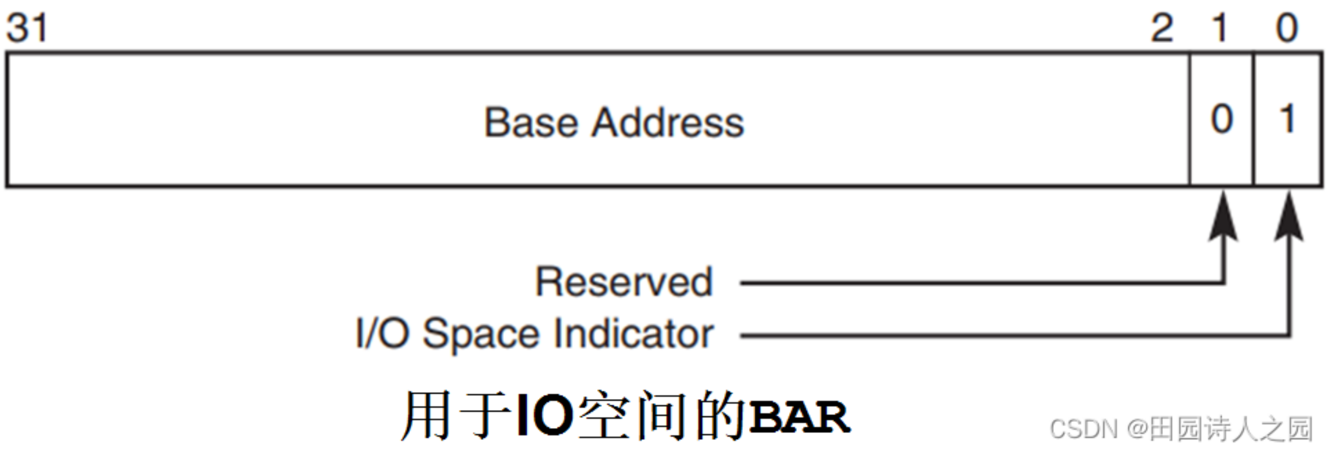
- For memory, whatever value is written is what value is read out, which can be read in advance
- For IO, it reflects the current state of the hardware, and the value read at each moment is not necessarily the same

The format of BAR is as follows:

- for memory space



- For IO space:



How does BAR indicate how much space it wants to apply for? Take a 32-bit address as an example:

- Software writes 0xFFFFFFFF to BAR
- Software to read BAR
- The value read is assumed to be 0xFFF0,000?, and the lowest 4 bits are ignored, and the result is: 0xFFF0,0000
  - This means that the "Base Address" that can be written in BAR is only the highest 12 bits
  - It also means that the lowest 20 bits are a variable range, so the size of this space is 2^20=1M Byte The code example is as follows:
    pos indicates the bar register to be accessed

```
        u32 l = 0, sz = 0, mask;
        ...
        mask = type ? PCI_ROM_ADDRESS_MASK : ~0;
        ...
        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);
```

If BAR indicates that it uses a 32-bit address, then BAR0~BAR5 can represent 6 address spaces respectively.

If BAR indicates that it uses a 64-bit address, then BAR0 and BAR1, BAR2 and BAR3, BAR4 and BAR5 represent three address spaces:

- The low-order BAR represents the lower 32 bits of the 64-bit address
- A high-order address represents the upper 32 bits of a 64-bit address.

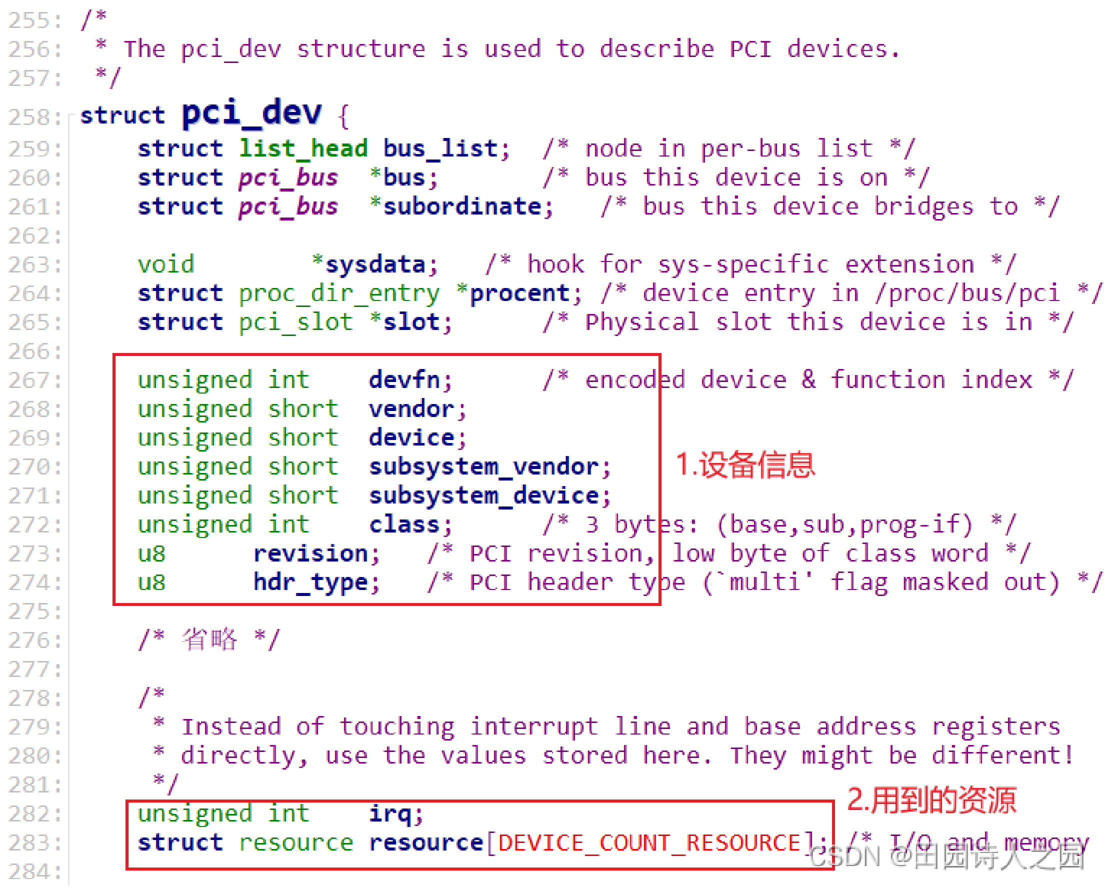
2. The process of scanning the device

2.1 Core: Construct pci_dev

Scan the PCIe bus, and construct a corresponding pci_dev for each PCIe bridge and PCIe device:

- Fill in the members of pci_dev, such as VID, PID, Class, etc.
- Allocate address space, write to PCIe device

The pci_dev structure is as follows:

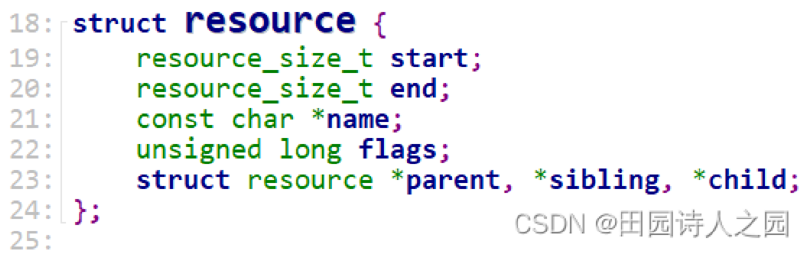


```
/*
 * The pci_dev structure is used to describe PCI devices.
 */
struct pci_dev {
    struct list_head bus_list;  /* node in per-bus list */
    struct pci_bus  *bus;       /* bus this device is on */
    struct pci_bus  *subordinate;   /* bus this device bridges to */

    void        *sysdata;   /* hook for sys-specific extension */
    struct proc_dir_entry *procent; /* device entry in /proc/bus/pci */
    struct pci_slot *slot;      /* Physical slot this device is in */

    unsigned int    devfn;      /* encoded device & function index */
    unsigned short  vendor;
    unsigned short  device;
    unsigned short  subsystem_vendor;
    unsigned short  subsystem_device;
    unsigned int    class;      /* 3 bytes: (base,sub,prog-if) */
    u8      revision;   /* PCI revision, low byte of class word */
    u8      hdr_type;   /* PCI header type (`multi' flag masked out) */

    /* 省略 */

    /*
     * Instead of touching interrupt line and base address registers
     * directly, use the values stored here. They might be different!
     */
    unsigned int    irq;
    struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory
```

Corresponding to the device information in the pci_dev structure: it can be obtained by reading the configuration space of the PCI device.

Corresponding to the resources in the pci_dev structure, this course will not analyze irq first. For the resource structure, each member corresponds to a BAR.

The resource structure is as follows. It should be noted that the start, end, etc. recorded in it are viewed from the perspective of the CPU. That is to say, if the memory address and IO address are recorded, it is the CPU address, not the PCI address. And these addresses are physical addresses, and to use them in software, ioremap must be executed first.



```
struct resource {
    resource_size_t start;
    resource_size_t end;
    const char *name;
    unsigned long flags;
    struct resource *parent, *sibling, *child;
};
```

2.2 Code Analysis

We need to find these 4 core codes:

- assign pci_dev
- Read the configuration space of the PCIe device and fill in the device information in pci_dev
- According to the BAR of the PCIe device, know what type of address it wants to apply for and how big it is
- Assign address, write to BAR

The key code is divided into two parts:

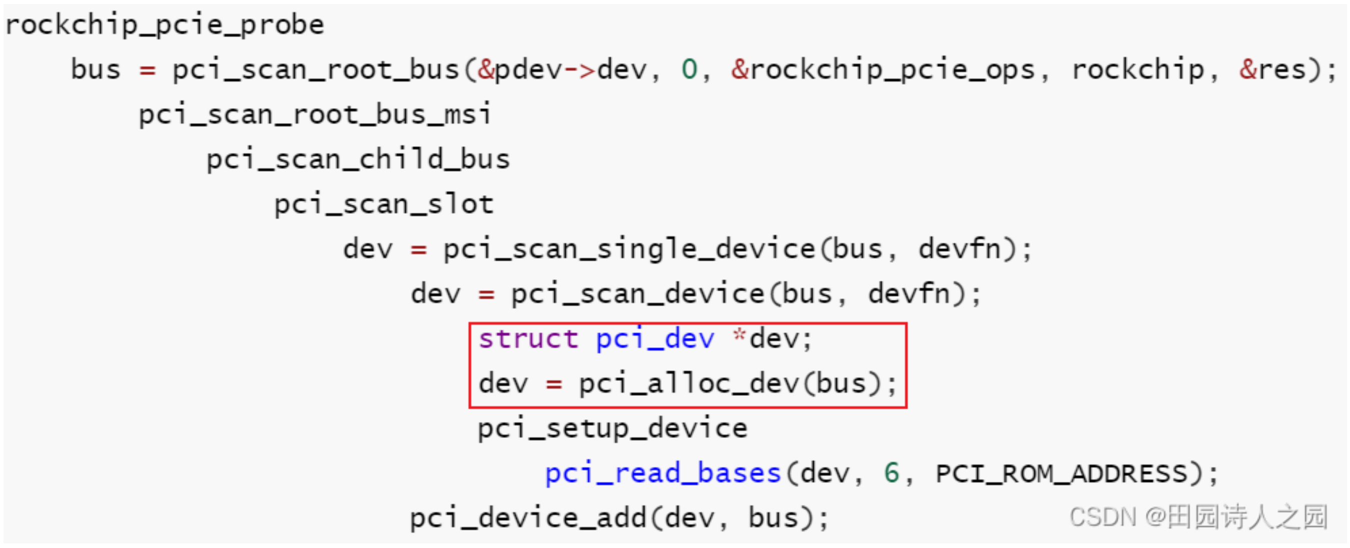
- Read the information and know how much space the PCIe device wants to apply for

```
rockchip_pcie_probe
    bus = 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);
                                        struct resource *res = &dev->resource[pos];
                                        __pci_read_base
                                                                pci_read_config_dword(dev
                                                                pci_write_config_dword(de
                                                                pci_read_config_dword(dev
                                                                pci_write_config_dword(de
                        pci_device_add(dev, bus);
```

- allocate space

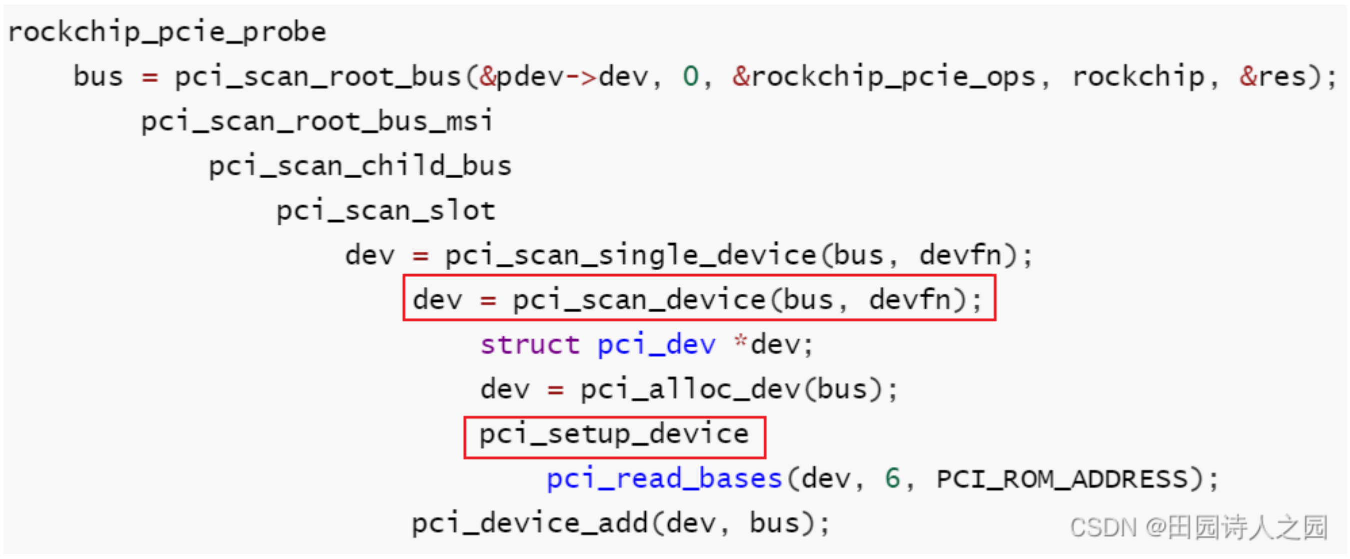
```
rockchip_pcie_probe
        pci_bus_size_bridges(bus);
        pci_bus_assign_resources(bus);
                __pci_bus_assign_resources
            pbus_assign_resources_sorted
                /* pci_dev->resource[]里记录有想申请的资源的大小,
                 * 把这些资源按对齐的要求排序
                 * 比如资源A要求1K地址对齐，资源B要求32地址对齐
                 * 那么资源A排在资源B前面, 优先分配资源A
                 */
                list_for_each_entry(dev, &bus->devices, bus_list)
                    __dev_sort_resources(dev, &head);
                                // 分配资源
                                __assign_resources_sorted
                    assign_requested_resources_sorted(head, &local_fail_head);
```

2.2.1 Allocating the pci_dev structure



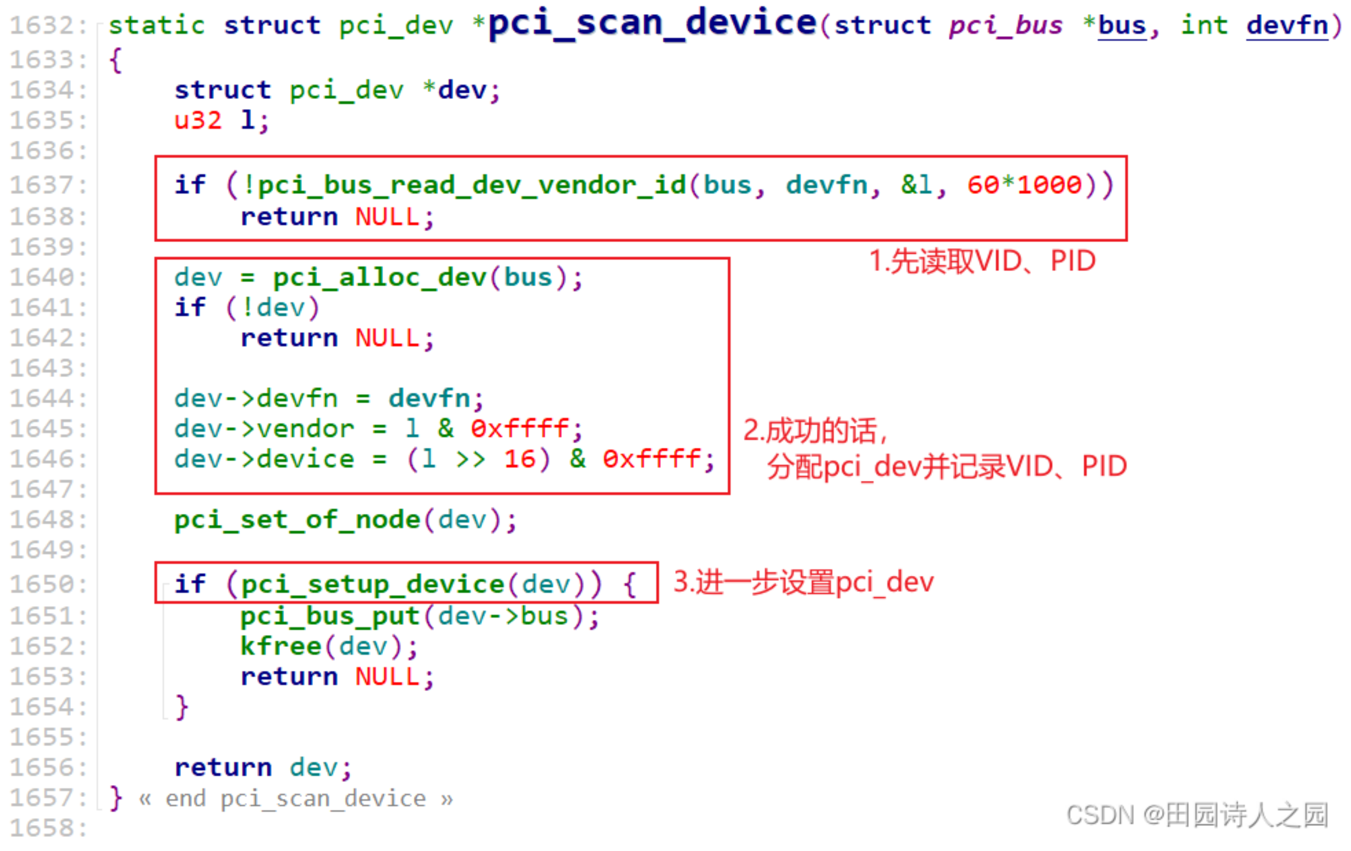
```
rockchip_pcie_probe
    bus = pci_scan_root_bus(&pdev->dev, 0, &rockchip_pcie_ops, rockchip, &res);
        pci_scan_root_bus_msi
            pci_scan_child_bus
                pci_scan_slot
                    dev = pci_scan_single_device(bus, devfn);
                        dev = pci_scan_device(bus, 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);
```

2.2.2 Read device information



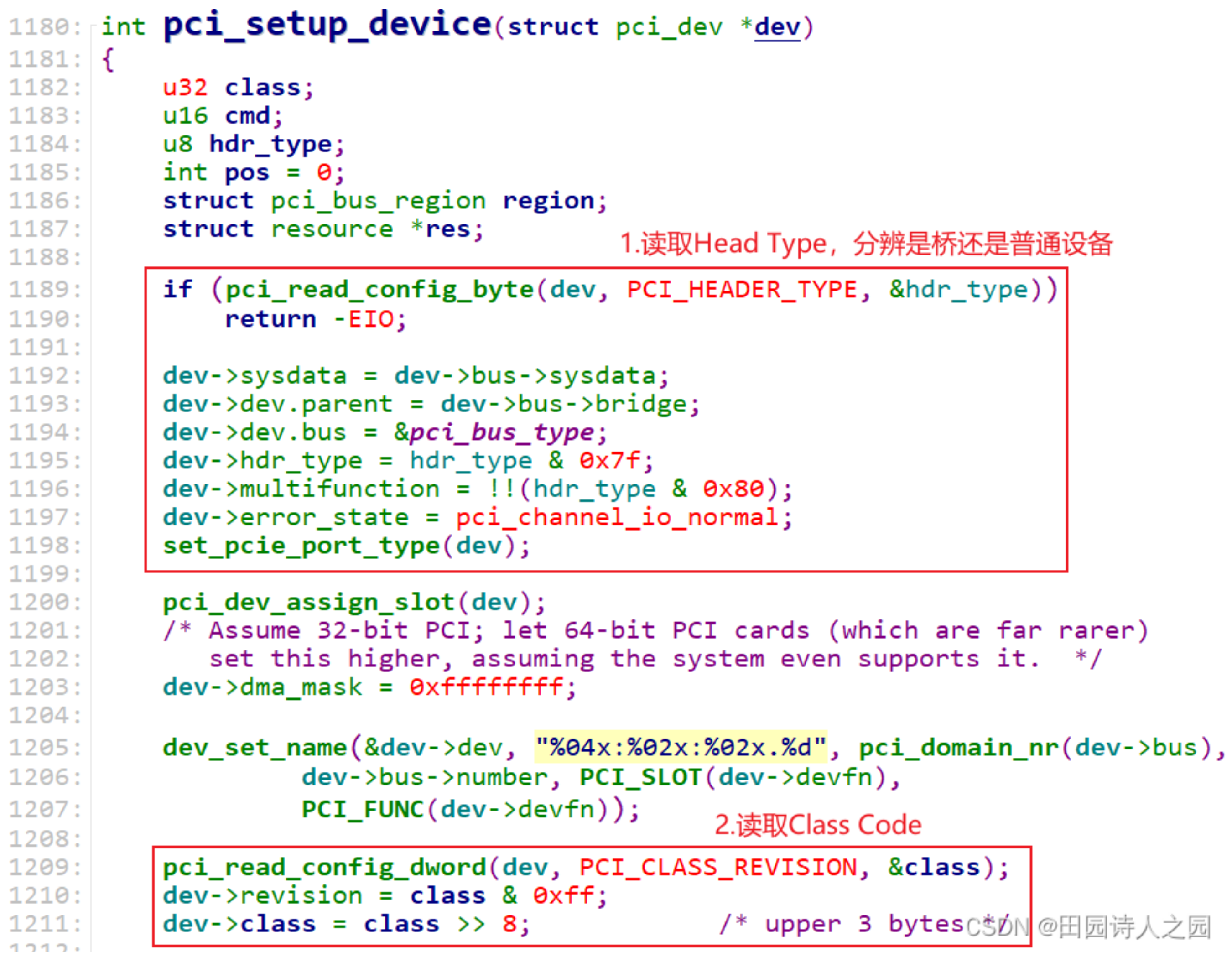
```
rockchip_pcie_probe
    bus = pci_scan_root_bus(&pdev->dev, 0, &rockchip_pcie_ops, rockchip, &res);
        pci_scan_root_bus_msi
            pci_scan_child_bus
                pci_scan_slot
                    dev = pci_scan_single_device(bus, devfn);
                        dev = pci_scan_device(bus, 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);
```

In the `pci_scan_device` function, it will try to read the VID and PID first, and then continue to call if it succeeds `pci_setup_device` :



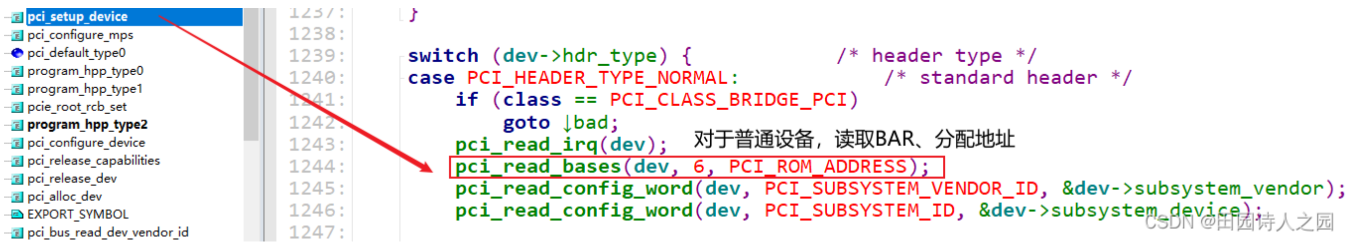
```
static struct pci_dev *pci_scan_device(struct pci_bus *bus, int devfn)
{
    struct pci_dev *dev;
    u32 l;

    if (!pci_bus_read_dev_vendor_id(bus, devfn, &l, 60*1000))
        return NULL;

    dev = pci_alloc_dev(bus);
    if (!dev)
        return NULL;

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

    pci_set_of_node(dev);

    if (pci_setup_device(dev)) {
        pci_bus_put(dev->bus);
        kfree(dev);
        return NULL;
    }

    return dev;
} « end pci_scan_device »
```

Internally `pci_setup_device` , other information continues to be read:



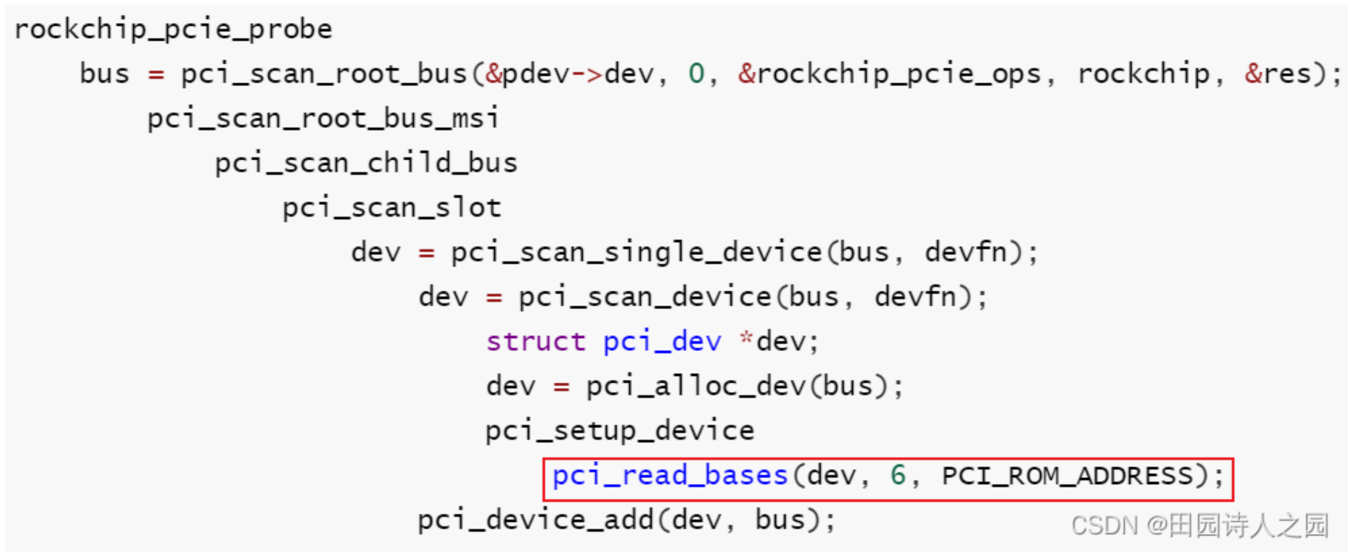
```
int pci_setup_device(struct pci_dev *dev)
{
    u32 class;
    u16 cmd;
    u8 hdr_type;
    int pos = 0;
    struct pci_bus_region region;
    struct resource *res;

    if (pci_read_config_byte(dev, PCI_HEADER_TYPE, &hdr_type))
        return -EIO;

    dev->sysdata = dev->bus->sysdata;
    dev->dev.parent = dev->bus->bridge;
    dev->dev.bus = &pci_bus_type;
    dev->hdr_type = hdr_type & 0x7f;
    dev->multifunction = !!(hdr_type & 0x80);
    dev->error_state = pci_channel_io_normal;
    set_pcie_port_type(dev);

    pci_dev_assign_slot(dev);
    /* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
       set this higher, assuming the system even supports it.  */
    dev->dma_mask = 0xffffffff;

    dev_set_name(&dev->dev, "%04x:%02x:%02x.%d", pci_domain_nr(dev->bus),
             dev->bus->number, PCI_SLOT(dev->devfn),
             PCI_FUNC(dev->devfn));

    pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
    dev->revision = class & 0xff;
    dev->class = class >> 8;             /* upper 3 bytes */
```

2.2.3 Read BAR



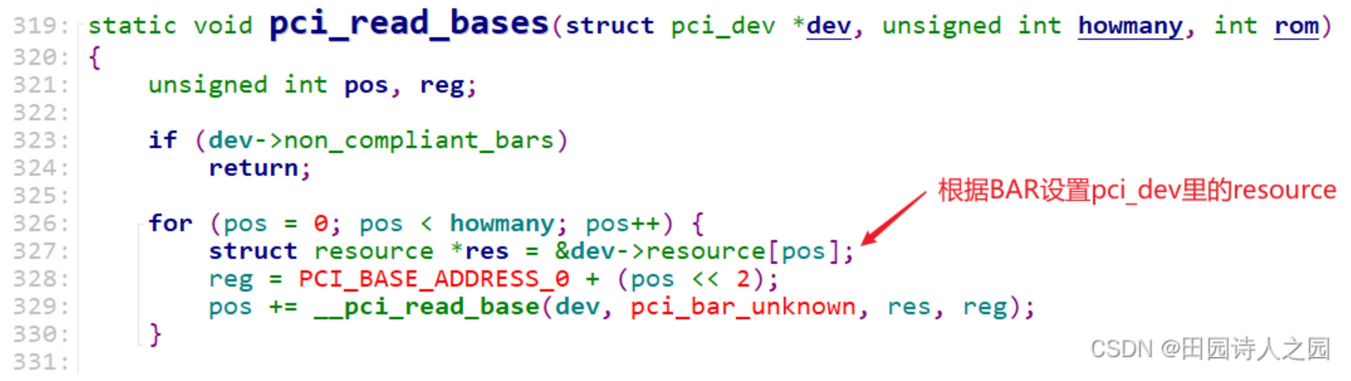
```
pci_setup_device
pci_configure_mps
pci_default_type0
program_hpp_type0
program_hpp_type1
pcie_root_rcb_set
program_hpp_type2
pci_configure_device
pci_release_capabilities
pci_release_dev
pci_alloc_dev
EXPORT_SYMBOL
pci_bus_read_dev_vendor_id
```

```
}

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);  对于普通设备，读取BAR、分配地址
    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);
```



```
rockchip_pcie_probe
    bus = 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);
```

`pci_read_bases` Function code analysis:



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

    if (dev->non_compliant_bars)
        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);
    }
```

`pci_read_bases` The function will be called again `__pci_read_base` , `__pci_read_base` just read the BAR and calculate the size of the space you want to apply for:

- 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;

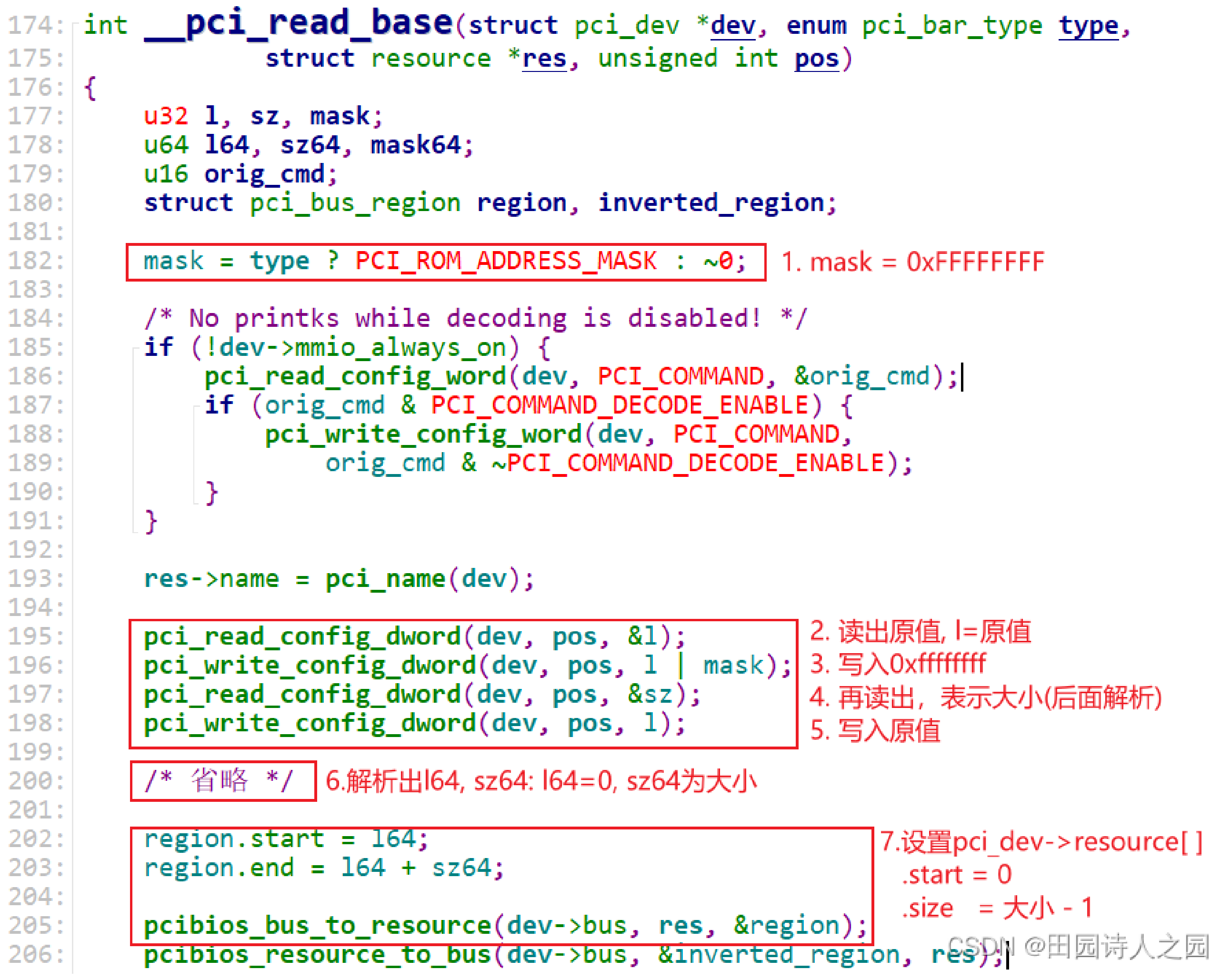
Posting what I said earlier will help you understand the code:

How does BAR indicate how much space it wants to apply for? Take a 32-bit address as an example:

- Software writes 0xFFFFFFFF to BAR
- Software to read BAR
- The value read is assumed to be 0xFFF0,000?, and the lowest 4 bits are ignored, and the result is: 0xFFF0,0000
    - This means that the "Base Address" that can be written in BAR is only the highest 12 bits
    - It also means that the lowest 20 bits can be changed, so the size of this space is 2^20=1M Byte

Below is `__pci_read_bases` the code analysis of the function.

- Get the size (original data, further analysis is required): For example, in the following code, sz is assigned a value of 0xFFF0,000?, which requires further analysis



```
int __pci_read_base(struct pci_dev *dev, enum pci_bar_type type,
            struct resource *res, unsigned int pos)
{
    u32 l, sz, mask;
    u64 l64, sz64, mask64;
    u16 orig_cmd;
    struct pci_bus_region region, inverted_region;

    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);

    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);

    /* 省略 */

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

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

2.2.4 Allocating Address Space

The function call in this part of the code is very deep, we can grasp two problems:

- Where is the address space allocated from?
  - In the device tree, the corresponding relationship between CPU address and PCI address is specified, which are recorded in pci_bus as "resources"
  - When reading BAR, it records the size of the space it wants to apply for in pci_dev->resource[]
- The allocated base address, to be written into BAR

The code calling relationship is as follows:

- Sort the resources to be applied for according to the alignment requirements, and then call assign_requested_resources_sorted, the code is as follows:

```

/* 把要申请的资源，按照对齐要求排序
 * 然后调用assign_requested_resources_sorted
 */

rockchip_pcie_probe
        pci_bus_size_bridges(bus);
        pci_bus_assign_resources(bus);
                __pci_bus_assign_resources
            pbus_assign_resources_sorted
                /* pci_dev->resource[]里记录有想申请的资源的大小,
                 * 把这些资源按对齐的要求排序
                 * 比如资源A要求1K地址对齐，资源B要求32地址对齐
                 * 那么资源A排在资源B前面，优先分配资源A
                 */
                list_for_each_entry(dev, &bus->devices, bus_list)
                    __dev_sort_resources(dev, &head);
```

```
                            // 分配资源
                            __assign_resources_sorted
                      assign_requested_resources_sorted(head, &local_fail_head);
```

- `assign_requested_resources_sorted` function does two things
    - allocate address space
    - Write the PCI address corresponding to this space into the BAR of the PCIe device
    - code show as below:

```
assign_requested_resources_sorted(head, &local_fail_head);
    pci_assign_resource
        ret = _pci_assign_resource(dev, resno, size, align);
                // 分配地址空间
            __pci_assign_resource
                pci_bus_alloc_resource
                    pci_bus_alloc_from_region
                        /* Ok, try it out.. */
                        ret = allocate_resource(r, res, size, ...);
                            err = find_resource(root, new, size,...);
                                __find_resource

                                    // 从资源链表中分配地址空间
                                    // 设置pci_dev->resource[]
                                    new->start = alloc.start;
                                    new->end = alloc.end;
            // 把对应的PCI地址写入BAR
            pci_update_resource(dev, resno);
                pci_std_update_resource
                    /* 把CPU地址转换为PCI地址: PCI地址 = CPU地址 - offset
                     * 写入BAR
                     */
                    pcibios_resource_to_bus(dev->bus, &region, res);
                    new = region.start;
                    reg = PCI_BASE_ADDRESS_0 + 4 * resno;
                    pci_write_config_dword(dev, reg, new);
```