PCI Subsystem -2- (Core)

2018-10-05 (http://jake.dothome.co.kr/pci-2/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v4.14>

PCI Subsystem Initialization

pci_driver_init()

drivers/pci/pci-driver.c

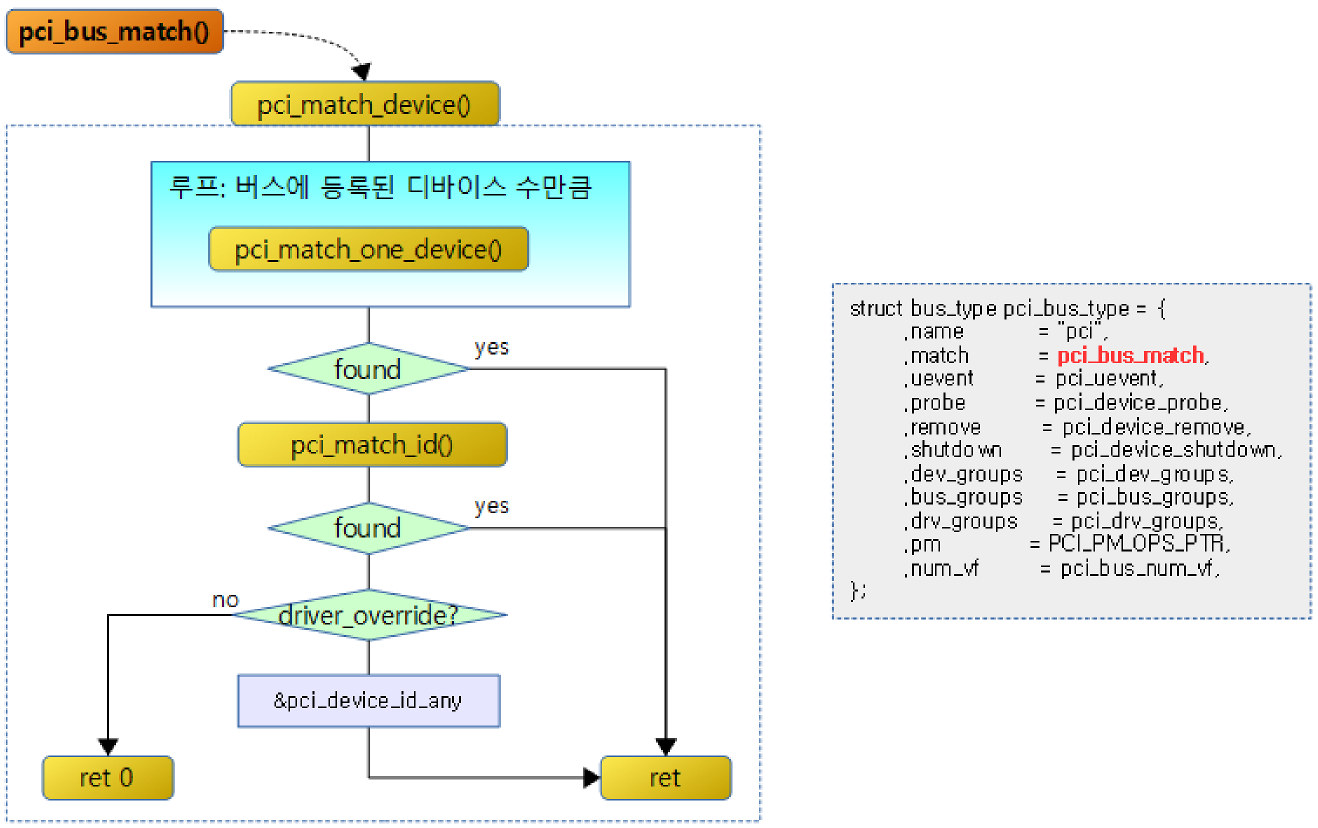
```
static int __init pci_driver_init(void)
{
    return bus_register(&pci_bus_type);
}
postcore_initcall(pci_driver_init);
```

Register the PCI bus type below.

drivers/pci/pci-driver.c

```
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_groups = pci_dev_groups,
    .bus_groups = pci_bus_groups,
    .drv_groups = pci_drv_groups,
    .pm     = PCI_PM_OPS_PTR,
    .num_vf     = pci_bus_num_vf,
};
EXPORT_SYMBOL(pci_bus_type);
```

PCI Device & Driver Match



```
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_groups     = pci_dev_groups,
        .bus_groups     = pci_bus_groups,
        .drv_groups     = pci_drv_groups,
        .pm             = PCI_PM_OPS_PTR,
        .num_vf         = pci_bus_num_vf,
};
```

(http://jake.dothome.co.kr/wp-content/uploads/2018/09/pci_bus_match-1.png)

pci_bus_match()

drivers/pci/pci-driver.c

```
/**
 * pci_bus_match - Tell if a PCI device structure has a matching PCI device id structure
 * @dev: the PCI device structure to match against
 * @drv: the device driver to search for matching PCI device id structures
 *
 * Used by a driver to check whether a PCI device present in the
 * system is in its list of supported devices. Returns the matching
 * pci_device_id structure or %NULL if there is no match.
 */
static int pci_bus_match(struct device *dev, struct device_driver *drv)
{
    struct pci_dev *pci_dev = to_pci_dev(dev);
    struct pci_driver *pci_drv;
    const struct pci_device_id *found_id;

    if (!pci_dev->match_driver)
        return 0;

    pci_drv = to_pci_driver(drv);
    found_id = pci_match_device(pci_drv, pci_dev);
    if (found_id)
        return 1;

    return 0;
}
```

When a device or driver is added to the PCI bus, it is a function to check whether they match each other.

pci_match_device()

drivers/pci/pci-driver.c

```
01 /**
02  * pci_match_device - Tell if a PCI device structure has a matching PCI device id structure
03  * @drv: the PCI driver to match against
04  * @dev: the PCI device structure to match against
05  *
06  * Used by a driver to check whether a PCI device present in the
07  * system is in its list of supported devices.  Returns the matching
08  * pci_device_id structure or %NULL if there is no match.
09  */
10 static const struct pci_device_id *pci_match_device(struct pci_driver *drv,
11                                                     struct pci_dev *dev)
12 {
13     struct pci_dynid *dynid;
14     const struct pci_device_id *found_id = NULL;
15 
16     /* When driver_override is set, only bind to the matching driver */
17     if (dev->driver_override && strcmp(dev->driver_override, drv->name))
18         return NULL;
19 
20     /* Look at the dynamic ids first, before the static ones */
21     spin_lock(&drv->dynids.lock);
22     list_for_each_entry(dynid, &drv->dynids.list, node) {
23         if (pci_match_one_device(&dynid->id, dev)) {
24             found_id = &dynid->id;
25             break;
26         }
27     }
28     spin_unlock(&drv->dynids.lock);
29 
30     if (!found_id)
31         found_id = pci_match_id(drv->id_table, dev);
32 
33     /* driver_override will always match, send a dummy id */
34     if (!found_id && dev->driver_override)
35         found_id = &pci_device_id_any;
36 
37     return found_id;
38 }
```

Matches PCI devices and drivers registered on the PCI bus with each other and returns the matched pci_device_id pointers. If they are not matched, it returns null.

- This is the case when the driver name to be used for the device is specified in code lines 17~18. If the device is not overdriven, it returns null.
- In code lines 21~28, check the match with the devices registered in the dynids list first.
- If it is not matched in code lines 30~31, check the match at the id_table specified in the driver.
- If lines 34~35 of the code are still unmatched and there is a driver override setting, it returns pci_device_id_any.

Match Priority

1. Dynamic PCI Devices

2. Static PCI devices with id_table
3. If overdrive is specified

If the PCI device is set to overdrive by specifying the driver name, only that driver can be matched.

pci_match_one_device()

drivers/pci/pci.h

```c
/**
 * pci_match_one_device - Tell if a PCI device structure has a matching
 *                        PCI device id structure
 * @id: single PCI device id structure to match
 * @dev: the PCI device structure to match against
 *
 * Returns the matching pci_device_id structure or %NULL if there is no match.
 */
static inline const struct pci_device_id *
pci_match_one_device(const struct pci_device_id *id, const struct pci_dev *dev)
{
    if ((id->vendor == PCI_ANY_ID || id->vendor == dev->vendor) &&
        (id->device == PCI_ANY_ID || id->device == dev->device) &&
        (id->subvendor == PCI_ANY_ID || id->subvendor == dev->subsystem_vendor) &&
        (id->subdevice == PCI_ANY_ID || id->subdevice == dev->subsystem_device) &&
        !((id->class ^ dev->class) & id->class_mask))
        return id;
    return NULL;
}
```

If all of the conditions for the next match are satisfied, it returns pci_device_id as-is, and if it is not matched, it returns null.

- True if the vendor name, device name, subvendor name, and sub-device name are specified, or if the PCI_ANY_ID is listed.
- The class should be the same.

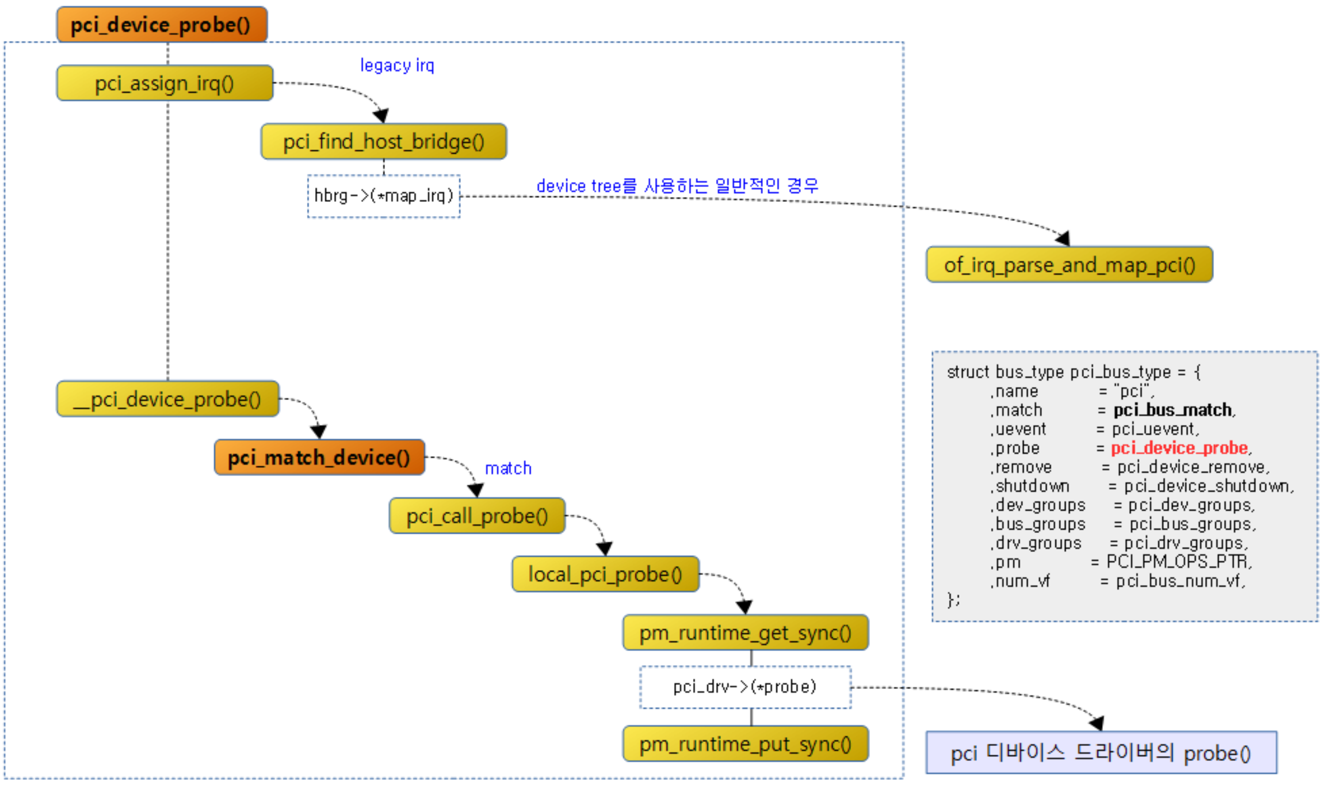
pci_match_id()

drivers/pci/pci-driver.c

```c
/**
 * pci_match_id - See if a pci device matches a given pci_id table
 * @ids: array of PCI device id structures to search in
 * @dev: the PCI device structure to match against.
 *
 * Used by a driver to check whether a PCI device present in the
 * system is in its list of supported devices.  Returns the matching
 * pci_device_id structure or %NULL if there is no match.
 *
 * Deprecated, don't use this as it will not catch any dynamic ids
 * that a driver might want to check for.
 */
const struct pci_device_id *pci_match_id(const struct pci_device_id *ids,
                                         struct pci_dev *dev)
{
    if (ids) {
        while (ids->vendor || ids->subvendor || ids->class_mask) {
            if (pci_match_one_device(ids, dev))
                return ids;
            ids++;
        }
    }
    return NULL;
}
EXPORT_SYMBOL(pci_match_id);
```

If any of the designated pci_device_id vendor, subvendor, or class mask are specified, the match will be checked. If no match is resolved, it returns null.

PCI Driver Probe



```
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_groups     = pci_dev_groups,
        .bus_groups     = pci_bus_groups,
        .drv_groups     = pci_drv_groups,
        .pm             = PCI_PM_OPS_PTR,
        .num_vf         = pci_bus_num_vf,
};
```

(http://jake.dothome.co.kr/wp-content/uploads/2018/09/pci_device_probe-1a.png)

pci_device_probe()

drivers/pci/pci-driver.c

```
static int pci_device_probe(struct device *dev)
{
    int error;
    struct pci_dev *pci_dev = to_pci_dev(dev);
    struct pci_driver *drv = to_pci_driver(dev->driver);

    pci_assign_irq(pci_dev);

    error = pcibios_alloc_irq(pci_dev);
    if (error < 0)
        return error;

    pci_dev_get(pci_dev);
    if (pci_device_can_probe(pci_dev)) {
        error = __pci_device_probe(drv, pci_dev);
        if (error) {
```

```
17                  pcibios_free_irq(pci_dev);
18                  pci_dev_put(pci_dev);
19              }
20          }
21
22          return error;
23      }
```

Set the legacy interrupt of the PCI device and proceed with the probe.

- In line 7 of the code, the legacy interrupt to be used by the PCI device is assigned by reading the PCI configuration information.
- In code lines 9~11, the interrupt to be used by the pci device is obtained from the PC using ACPI and assigned.
- In code lines 13~20, increment the PCI device's reference counter by 1 and proceed with the probe.

pci_device_can_probe()

drivers/pci/pci-driver.c

```
#ifdef CONFIG_PCI_IOV
static inline bool pci_device_can_probe(struct pci_dev *pdev)
{
        return (!pdev->is_virtfn || pdev->physfn->sriov->drivers_autoprobe);
}
#else
static inline bool pci_device_can_probe(struct pci_dev *pdev)
{
        return true;
}
#endif
```

The pci_device_can_probe() function is always 1 if PCI IO Virtualization is not enabled.

- If PCI IO Virtualization is enabled, it returns 1 only if the drivers_autoprobe is set.

__pci_device_probe()

drivers/pci/pci-driver.c

```
/**
 * __pci_device_probe - check if a driver wants to claim a specific PCI device
 * @drv: driver to call to check if it wants the PCI device
 * @pci_dev: PCI device being probed
 *
 * returns 0 on success, else error.
 * side-effect: pci_dev->driver is set to drv when drv claims pci_dev.
 */
static int __pci_device_probe(struct pci_driver *drv, struct pci_dev *pci_dev)
{
    const struct pci_device_id *id;
    int error = 0;

    if (!pci_dev->driver && drv->probe) {
        error = -ENODEV;

        id = pci_match_device(drv, pci_dev);
        if (id)
            error = pci_call_probe(drv, pci_dev, id);
    }
    return error;
}
```

If a driver has not yet been assigned to the PCI device and the driver's probe hook function exists, the match ID is obtained and the probe is performed.

- If a driver is already specified, it returns success (0) without proceeding with the match or probe process.

pci_call_probe()

drivers/pci/pci-driver.c

```
01 static int pci_call_probe(struct pci_driver *drv, struct pci_dev *dev,
02                const struct pci_device_id *id)
03 {
04     int error, node, cpu;
05     struct drv_dev_and_id ddi = { drv, dev, id };
06 
07     /*
08      * Execute driver initialization on node where the device is
09      * attached.  This way the driver likely allocates its local memory
10      * on the right node.
11      */
12     node = dev_to_node(&dev->dev);
13     dev->is_probed = 1;
14 
15     cpu_hotplug_disable();
16 
17     /*
18      * Prevent nesting work_on_cpu() for the case where a Virtual Functi
on
19      * device is probed from work_on_cpu() of the Physical device.
20      */
21     if (node < 0 || node >= MAX_NUMNODES || !node_online(node) ||
22         pci_physfn_is_probed(dev))
23         cpu = nr_cpu_ids;
24     else
25         cpu = cpumask_any_and(cpumask_of_node(node), cpu_online_mask);
26 
27     if (cpu < nr_cpu_ids)
28         error = work_on_cpu(cpu, local_pci_probe, &ddi);
29     else
30         error = local_pci_probe(&ddi);
31 
32     dev->is_probed = 0;
33     cpu_hotplug_enable();
34     return error;
35 }
```

Use the CPU's workqueue on the node where the PCI device is located to invoke the PCI driver's probe.

- On line 12 of the code, find the node number that corresponds to the PCI device.
- In line 13 of the code, indicate that the PCI device is in the process of being probeed.
- If another CPU hotplug is being performed on line 15, wait for it to complete before acquiring the lock.

- In line 21~25 of the code, find the CPU number of one of the online CPUs running on the node. If PCI IO Virtualization is working, use the current CPU.
- On lines 27~30 of the code, call the local_pci_probe function from the workqueue on the CPU.
- On line 32 of the code, the PCI device's probe progress is complete, so clear it.
- Disable the CPU hotplug lock in code line 33.

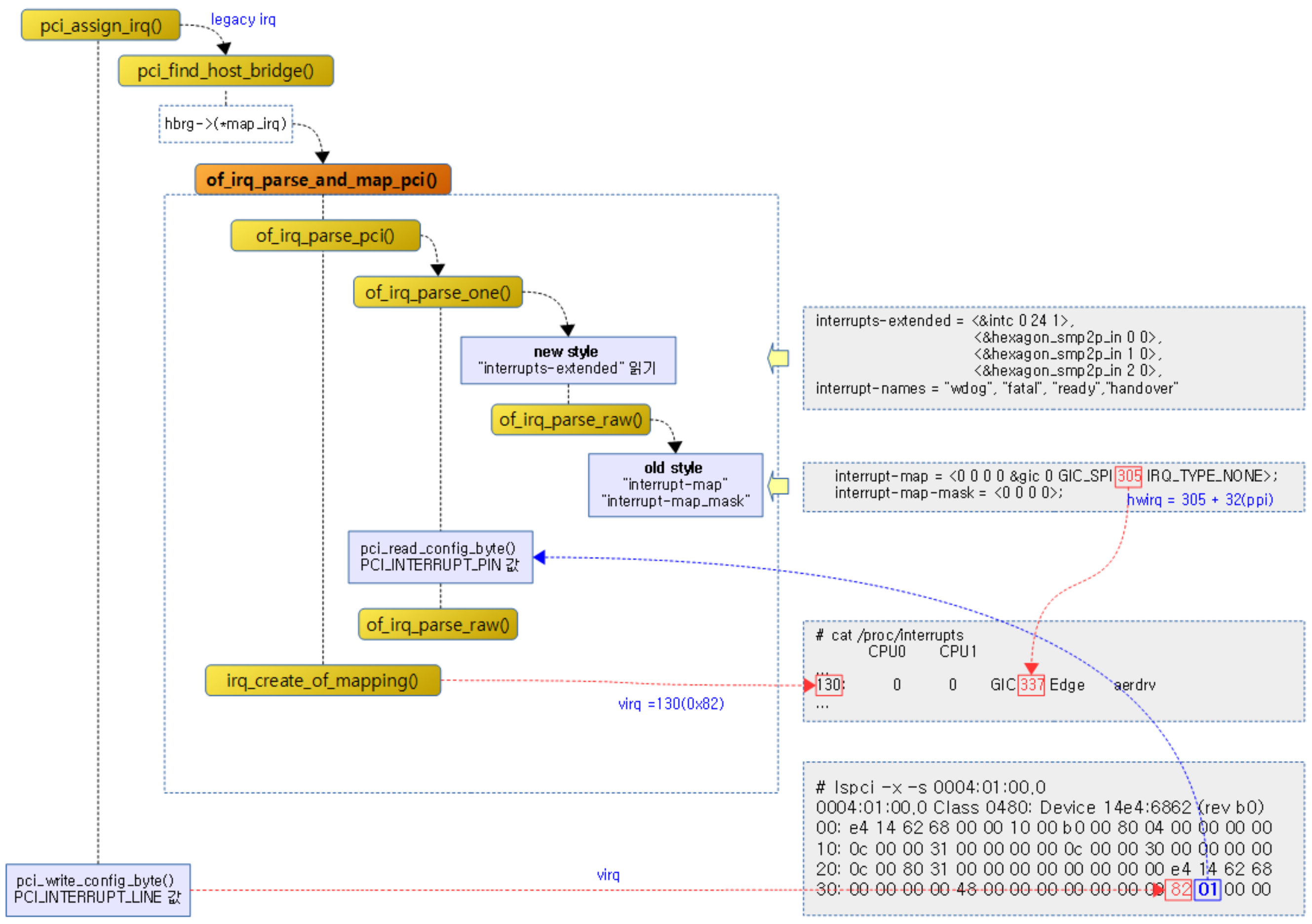
local_pci_probe()

drivers/pci/pci-driver.c

```
01 static long local_pci_probe(void *_ddi)
02 {
03         struct drv_dev_and_id *ddi = _ddi;
04         struct pci_dev *pci_dev = ddi->dev;
05         struct pci_driver *pci_drv = ddi->drv;
06         struct device *dev = &pci_dev->dev;
07         int rc;
08 
09         /*
10          * Unbound PCI devices are always put in D0, regardless of
11          * runtime PM status.  During probe, the device is set to
12          * active and the usage count is incremented.  If the driver
13          * supports runtime PM, it should call pm_runtime_put_noidle(),
14          * or any other runtime PM helper function decrementing the usage
15          * count, in its probe routine and pm_runtime_get_noresume() in
16          * its remove routine.
17          */
18         pm_runtime_get_sync(dev);
19         pci_dev->driver = pci_drv;
20         rc = pci_drv->probe(pci_dev, ddi->id);
21         if (!rc)
22                 return rc;
23         if (rc < 0) {
24                 pci_dev->driver = NULL;
25                 pm_runtime_put_sync(dev);
26                 return rc;
27         }
28         /*
29          * Probe function should return < 0 for failure, 0 for success
30          * Treat values > 0 as success, but warn.
31          */
32         dev_warn(dev, "Driver probe function unexpectedly returned %d\n", rc);
33         return 0;
34 }
```

After the PCI device recovers from sleep, increment the PM reference counter. It then calls the PCI driver's probe function.

Legacy IRQ Assignment



```
interrupts-extended = <&intc 0 24 1>,
                      <&hexagon_smp2p_in 0 0>,
                      <&hexagon_smp2p_in 1 0>,
                      <&hexagon_smp2p_in 2 0>,
interrupt-names = "wdog", "fatal", "ready","handover"
```

```
interrupt-map = <0 0 0 0 &gic 0 GIC_SPI 305 IRQ_TYPE_NONE>;
interrupt-map-mask = <0 0 0 0>;
```

```
# cat /proc/interrupts
          CPU0     CPU1
...
130:        0        0     GIC 337 Edge     aerdrv
...
```

```
# lspci -x -s 0004:01:00.0
0004:01:00.0 Class 0480: Device 14e4:6862 (rev b0)
00: e4 14 62 68 00 00 10 00 b0 00 80 04 00 00 00 00
10: 0c 00 00 31 00 00 00 00 0c 00 00 30 00 00 00 00
20: 0c 00 80 31 00 00 00 00 00 00 00 00 e4 14 62 68
30: 00 00 00 00 48 00 00 00 00 00 00 00 82 01 00 00
```

(http://jake.dothome.co.kr/wp-content/uploads/2018/09/pci_assign_irq-1.png)

pci_assign_irq()

drivers/pci/setup-irq.c

```
void pci_assign_irq(struct pci_dev *dev)
{
        u8 pin;
        u8 slot = -1;
        int irq = 0;
        struct pci_host_bridge *hbrg = pci_find_host_bridge(dev->bus);

        if (!(hbrg->map_irq)) {
                dev_dbg(&dev->dev, "runtime IRQ mapping not provided by arch\n");
                return;
        }

        /* If this device is not on the primary bus, we need to figure out
           which interrupt pin it will come in on.   We know which slot it
           will come in on 'cos that slot is where the bridge is.   Each
           time the interrupt line passes through a PCI-PCI bridge we must
           apply the swizzle function.  */

        pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
        /* Cope with illegal. */
        if (pin > 4)
                pin = 1;

        if (pin) {
                /* Follow the chain of bridges, swizzling as we go.  */
26                 if (hbrg->swizzle_irq)
27                         slot = (*(hbrg->swizzle_irq))(dev, &pin);
28 
29                 /*
30                  * If a swizzling function is not used map_irq must
31                  * ignore slot
32                  */
33                 irq = (*(hbrg->map_irq))(dev, slot, pin);
34                 if (irq == -1)
35                         irq = 0;
36         }
37         dev->irq = irq;
38 
39         dev_dbg(&dev->dev, "assign IRQ: got %d\n", dev->irq);
40 
41         /* Always tell the device, so the driver knows what is
42            the real IRQ to use; the device does not use it. */
43         pci_write_config_byte(dev, PCI_INTERRUPT_LINE, irq);
44 }
```

Assign legacy interrupts to be used by PCI devices by reading the PCI configuration information. The detailed operation is as follows.

- In line 6 of the code, we obtain the host bridge for the PCI bus.
- In code lines 19~22, read the interrupt pin (1~4) from the PCI device's default configuration header information. However, PCI devices that do not use interrupts will read 0.
  - PCI_INTERRUPT_PIN(0x3d)
- In lines 26~27 of code, we get the slot number via the (*swizzle_irq) hook function implemented in the Noise Bridge.
- For the pin number read from lines 33~35, pass the slot and pin arguments via the (*map_irq) hook function implemented in the host bridge, read the hwirq from the ACPI or device tree, and know the virq mapped to it.
- In line 37~39 of the code, set it to dev->irq and print a log message.
- On line 43 of the code, record the interrupt number on the interrupt line of the PCI device's default configuratio header information.
  - PCI_INTERRUPT_LINE(0x3c)

PCI Driver Registration

pci_register_driver()

include/linux/pci.h

```
/*
 * pci_register_driver must be a macro so that KBUILD_MODNAME can be exp
anded
 */
#define pci_register_driver(driver)     \
    __pci_register_driver(driver, THIS_MODULE, KBUILD_MODNAME)
```

PCI drivers can be registered using the macro function above.

__pci_register_driver()

drivers/pci/pci-driver.c

```
/**
 * __pci_register_driver - register a new pci driver
 * @drv: the driver structure to register
 * @owner: owner module of drv
 * @mod_name: module name string
 *
 * Adds the driver structure to the list of registered drivers.
 * Returns a negative value on error, otherwise 0.
 * If no error occurred, the driver remains registered even if
 * no device was claimed during registration.
 */
int __pci_register_driver(struct pci_driver *drv, struct module *owner,
                          const char *mod_name)
{
        /* initialize common driver fields */
        drv->driver.name = drv->name;
        drv->driver.bus = &pci_bus_type;
        drv->driver.owner = owner;
        drv->driver.mod_name = mod_name;
        drv->driver.groups = drv->groups;

        spin_lock_init(&drv->dynids.lock);
        INIT_LIST_HEAD(&drv->dynids.list);

        /* register with core */
        return driver_register(&drv->driver);
}
EXPORT_SYMBOL(__pci_register_driver);
```

Register the requested PCI driver on the PCI bus.

- Use Dynids to dynamically manage the identity of PCI devices.

pci_driver Struct

This is the structure used to register PCI drivers.

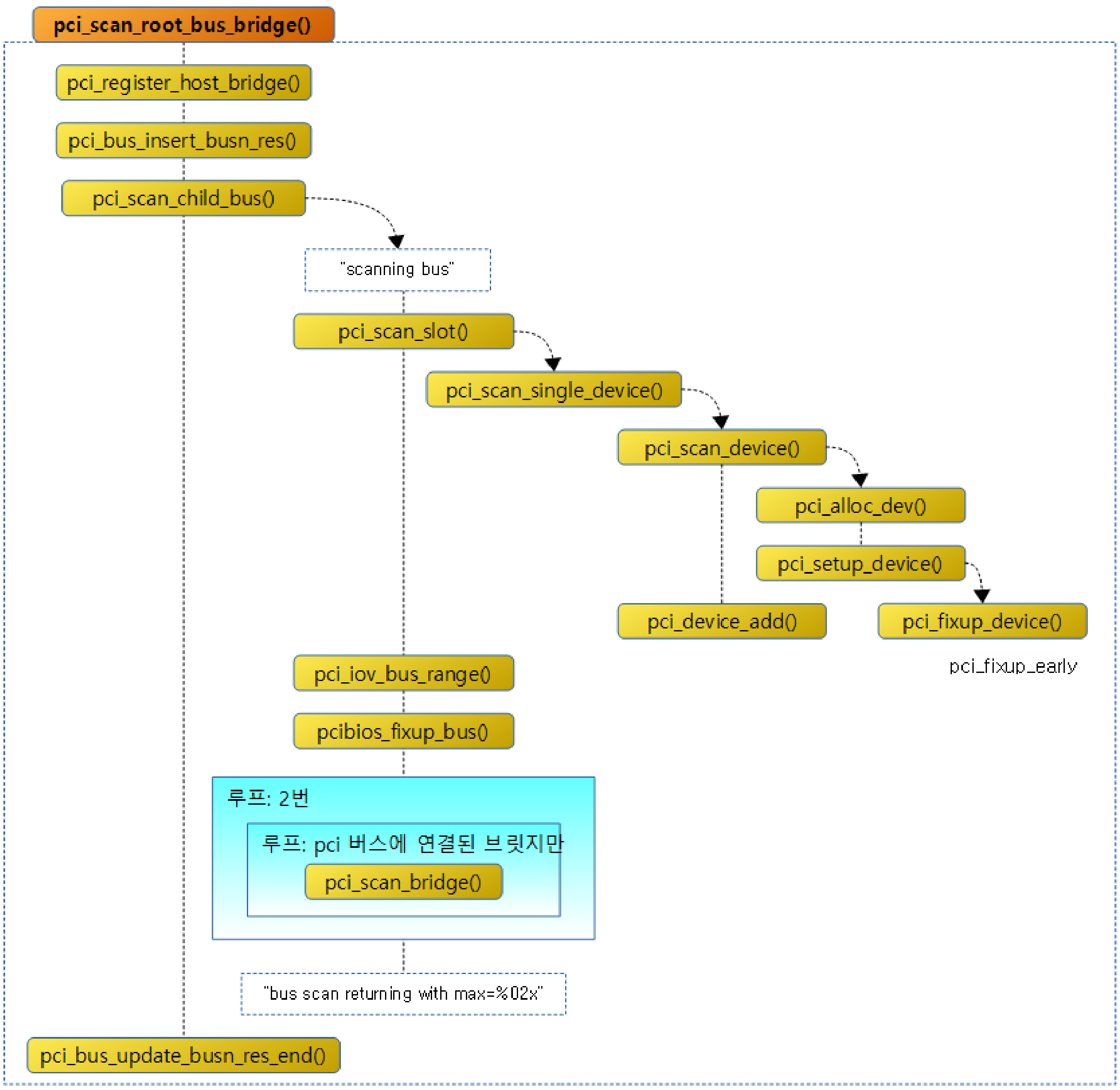
include/linux/pci.h

```
struct pci_driver {
    struct list_head node;
    const char *name;
    const struct pci_device_id *id_table;   /* must be non-NULL for probe to be called */
    int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);   /* New device inserted */
    void (*remove) (struct pci_dev *dev);   /* Device removed (NULL if not a hot-plug capable driver) */
    int  (*suspend) (struct pci_dev *dev, pm_message_t state);  /* Device suspended */
    int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);
    int  (*resume_early) (struct pci_dev *dev);
    int  (*resume) (struct pci_dev *dev);                   /* Device woken up */
    void (*shutdown) (struct pci_dev *dev);
    int (*sriov_configure) (struct pci_dev *dev, int num_vfs); /* PF pdev */
    const struct pci_error_handlers *err_handler;
    const struct attribute_group **groups;
    struct device_driver    driver;
    struct pci_dynids dynids;
};
```

- *name

  - Driver Name
- *id_table
  - Specifies the ID table of the PCI devices that you want to match and use.
- (*probe)
  - When a new device is added, it specifies a probe hook function.
- (*remove)
  - Specifies the hook function to call when the device is removed.
- (*suspend),
- (*suspend_late),
- (*resume_early),
- (*resume),
- (*shutdown)
  - These are power-saving hook functions and are used for legacy PCI driver compatibility.
  - Recent interfaces use driver->pm, which is embedded in the PCI driver.
- (*sriov_configure)
  - This is a hook function for setting up single-root I/O virtualization that allows you to specify the number of virtual functions (VFs) to be operated.
- *err_handler
  - Specifies a pci_error_handlers structure containing callbacks to handle PCI bus error events.
- **groups
  - Specifies an array of attribute groups for the driver.
- driver
  - It is an embedded driver.
- dynids
  - This is a list to manage dynamic IDs.

PCI Device Scan



(http://jake.dothome.co.kr/wp-content/uploads/2018/09/pci_scan_root_bus_bridge-1.png)

Route Bus Bridge Scan

pci_scan_root_bus_bridge()

drivers/pci/probe.c

```
int pci_scan_root_bus_bridge(struct pci_host_bridge *bridge)
{
        struct resource_entry *window;
        bool found = false;
        struct pci_bus *b;
        int max, bus, ret;

        if (!bridge)
                return -EINVAL;

        resource_list_for_each_entry(window, &bridge->windows)
                if (window->res->flags & IORESOURCE_BUS) {
                        found = true;
                        break;
15                 }
16 
17         ret = pci_register_host_bridge(bridge);
18         if (ret < 0)
19                 return ret;
20 
21         b = bridge->bus;
22         bus = bridge->busnr;
23 
24         if (!found) {
25                 dev_info(&b->dev,
26                  "No busn resource found for root bus, will use [bus %02x-ff]\n",
27                         bus);
28                 pci_bus_insert_busn_res(b, bus, 255);
29         }
30 
31         max = pci_scan_child_bus(b);
32 
33         if (!found)
34                 pci_bus_update_busn_res_end(b, max);
35 
36         return 0;
37 }
38 EXPORT_SYMBOL(pci_scan_root_bus_bridge);
```

Scan the Route Bus Bridge.

- If the bridge is null in line 8~9, it returns a -EINVAL error.
- In lines 11~15 of the code, only if there is a bus resource among the platform resources, set the found value to true and leave the loop.
- Register the host bridge in lines 17~19 of code.
- If no bus resource is found in code lines 21~29, create and add a bus resource to use the bus number from the bus number to 0xff of the requested bus as an argument.
  - If this is the first time a host bridge is registered, the resource will be registered on bus numbers 0 through 255.
- Scan the child bus at code line 31.
- If the first bus resource was not found in code lines 33~34 and it was created and added, the bus resource is updated up to the bus number of the child bus.

Child Bus Scan

pci_scan_child_bus()

drivers/pci/probe.c

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

        dev_dbg(&bus->dev, "scanning bus\n");

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

        /* Reserve buses for SR-IOV capability. */
        max += pci_iov_bus_range(bus);
14
15          /*
16           * After performing arch-dependent fixup of the bus, look behind
17           * all PCI-to-PCI bridges on this bus.
18           */
19          if (!bus->is_added) {
20                  dev_dbg(&bus->dev, "fixups for bus\n");
21                  pcibios_fixup_bus(bus);
22                  bus->is_added = 1;
23          }
24
25          for (pass = 0; pass < 2; pass++)
26                  list_for_each_entry(dev, &bus->devices, bus_list) {
27                          if (pci_is_bridge(dev))
28                                  max = pci_scan_bridge(bus, dev, max, pass);
29                  }
30
31          /*
32           * Make sure a hotplug bridge has at least the minimum requested
33           * number of buses.
34           */
35          if (bus->self && bus->self->is_hotplug_bridge && pci_hotplug_bus_size) {
36                  if (max - bus->busn_res.start < pci_hotplug_bus_size - 1)
37                          max = bus->busn_res.start + pci_hotplug_bus_size - 1;
38
39                  /* Do not allocate more buses than we have room left */
40                  if (max > bus->busn_res.end)
41                          max = bus->busn_res.end;
42          }
43
44          /*
45           * We've scanned the bus and so we know all about what's on
46           * the other side of any bridges that may be on this bus plus
47           * any devices.
48           *
49           * Return how far we've got finding sub-buses.
50           */
51          dev_dbg(&bus->dev, "bus scan returning with max=%02x\n", max);
52          return max;
53  }
54  EXPORT_SYMBOL_GPL(pci_scan_child_bus);
```

Scan the child PCI bus.

- Scan the entire slot from code lines 9~10.
- In line 13 of code, the max value is skipped by that amount in order to reserve the bus number used by the virtual function.
- If the bus is added for the first time on lines 19~23 of code, perform a PCIBOS fixup.
  - Each architecture has its own code.
    - For arm, use the arm/kernel/bios32.c – pcibios_fixup_bus() function.
    - In the case of arm64, it does not perform the fixup code.
- On code lines 25~29, scan all PCI bridge devices currently registered on the bus twice.
- Leave the bus number for the hot plug bus on code lines 35~42.

PCI Slot Scan

pci_scan_slot()

drivers/pci/probe.c

```
01 /**
02  * pci_scan_slot - scan a PCI slot on a bus for devices.
03  * @bus: PCI bus to scan
04  * @devfn: slot number to scan (must have zero function.)
05  *
06  * Scan a PCI slot on the specified PCI bus for devices, adding
07  * discovered devices to the @bus->devices list.  New devices
08  * will not have is_added set.
09  *
10  * Returns the number of new devices found.
11  */
12 int pci_scan_slot(struct pci_bus *bus, int devfn)
13 {
14         unsigned fn, nr = 0;
15         struct pci_dev *dev;
16 
17         if (only_one_child(bus) && (devfn > 0))
18                 return 0; /* Already scanned the entire slot */
19 
20         dev = pci_scan_single_device(bus, devfn);
21         if (!dev)
22                 return 0;
23         if (!dev->is_added)
24                 nr++;
25 
26         for (fn = next_fn(bus, dev, 0); fn > 0; fn = next_fn(bus, dev, fn)) {
27                 dev = pci_scan_single_device(bus, devfn + fn);
28                 if (dev) {
29                         if (!dev->is_added)
30                                 nr++;
31                         dev->multifunction = 1;
32                 }
33         }
34 
35         /* only one slot has pcie device */
36         if (bus->self && nr)
37                 pcie_aspm_init_link_state(bus->self);
38 
39         return nr;
40 }
41 EXPORT_SYMBOL(pci_scan_slot);
```

Scan the PCI slot.

- In code lines 17~18, if the device is already connected to the PCIe port, or if it is prevented from scanning, it returns 0.
- Scan one device from code lines 20~22. If no devices are scanned, it returns 0.
- In line 23~24 of the code, increment the NR unless the device has already been added.
- In code lines 26~33, scan one device for the following functions: If the device is added, set the multifunction setting.
- In line 36~37 of the code, if the device is added to the PCIe root port device, initialize the link state.

Single Device Scan

pci_scan_single_device()

drivers/pci/probe.c

```
01 struct pci_dev *pci_scan_single_device(struct pci_bus *bus, int devfn)
02 {
03         struct pci_dev *dev;
04
05         dev = pci_get_slot(bus, devfn);
06         if (dev) {
07                 pci_dev_put(dev);
08                 return dev;
09         }
10
11         dev = pci_scan_device(bus, devfn);
12         if (!dev)
13                 return NULL;
14
15         pci_device_add(dev, bus);
16
17         return dev;
18 }
19 EXPORT_SYMBOL(pci_scan_single_device);
```

Scan a device and add it as a PCI device.

- In line 5~9 of the code, import the device that corresponds to devfn among the devices registered in the bus.
- Scan the devfn device in line 11~13 of the code.
- In line 15 of the code, add the device that was successfully scanned as a PCI device.

PCI Device Scan

pci_scan_device()

drivers/pci/probe.c

```
/*
 * Read the config data for a PCI device, sanity-check it
 * and fill in the dev structure...
 */
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);
25                  kfree(dev);
26                  return NULL;
27          }
28
29          return dev;
30  }
```

Scan the PCI device corresponding to one devfn and set it up. If it fails, it returns null.

- In line 10~11 of the code, the device ID and vendor ID of the PCI device of the requested bus device function number are read within a maximum of 60 seconds.
- Assign a PCI device in code lines 13~15. Then, substitute pci_dev_type for the type and increase the bus reference counter by 1.
- In lines 17~19 of the code, specify the devfn for the PCI device and assign the device ID to the 16-bit vendor you know.
- In line 21 of the code, search for the node for that device in the child node of the host controller's device tree and assign it to the of_node.
  - The bits[15:8] value of the "reg" attribute value is the device (slot) and function number.
- Read the configuration space header information from lines 23~27 to set up the PCI device.

pci_bus_read_dev_vendor_id()

drivers/pci/probe.c

```
01  bool pci_bus_read_dev_vendor_id(struct pci_bus *bus, int devfn, u32 *l,
02                                  int timeout)
03  {
04          if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, l))
05                  return false;
06
07          /* some broken boards return 0 or ~0 if a slot is empty: */
08          if (*l == 0xffffffff || *l == 0x00000000 ||
09              *l == 0x0000ffff || *l == 0xffff0000)
10                  return false;
11
12          if (pci_bus_crs_vendor_id(*l))
13                  return pci_bus_wait_crs(bus, devfn, l, timeout);
14
15          return true;
16  }
17  EXPORT_SYMBOL(pci_bus_read_dev_vendor_id);
```

It reads the device ID and vendor ID from the configuration space of the requested bus device function, prints them to argument L, and returns true.

- Read the vendor ID from the configuration space of the bus device function requested in line 4~5 of the code and find out it in argument l. If it can't be read, it returns false.
- Returns false if the slot is empty on lines 8~10 of the code.
- There may be a delay when reading the vendor ID from code lines 12~13. In such a case, it will try to read again with a delay of 1 second to 2 times within the timeout range.
  - 참고: PCI: Add pci_bus_crs_vendor_id() to detect CRS response data (https://github.com/torvalds/linux/commit/62bc6a6f7468bc6d6cb39177504e79df401aea76)

PCI Device Setup

pci_setup_device()

drivers/pci/probe.c -1/2-

```
/**
 * pci_setup_device - fill in class and map information of a device
 * @dev: the device structure to fill
 *
 * Initialize the device structure with information about the device's
 * vendor,class,memory and IO-space addresses,IRQ lines etc.
 * Called at initialisation of the PCI subsystem and by CardBus services.
 * Returns 0 on success and negative if unknown type of device (not normal,
 * bridge or CardBus).
 */
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 */

        dev_printk(KERN_DEBUG, &dev->dev, "[%04x:%04x] type %02x class %#08x\n",
                   dev->vendor, dev->device, dev->hdr_type, dev->class);

        /* need to have dev->class ready */
        dev->cfg_size = pci_cfg_space_size(dev);

        /* need to have dev->cfg_size ready */
        set_pcie_thunderbolt(dev);

        /* "Unknown power state" */
        dev->current_state = PCI_UNKNOWN;

        /* Early fixups, before probing the BARs */
        pci_fixup_device(pci_fixup_early, dev);
        /* device class may be changed after fixup */
```

```
59          class = dev->class >> 8;
60
61          if (dev->non_compliant_bars) {
62                  pci_read_config_word(dev, PCI_COMMAND, &cmd);
63                  if (cmd & (PCI_COMMAND_IO | PCI_COMMAND_MEMORY)) {
64                          dev_info(&dev->dev, "device has non-compliant BA
    Rs; disabling IO/MEM decoding\n");
65                          cmd &= ~PCI_COMMAND_IO;
66                          cmd &= ~PCI_COMMAND_MEMORY;
67                          pci_write_config_word(dev, PCI_COMMAND, cmd);
68                  }
69          }
70
71          dev->broken_intx_masking = pci_intx_mask_broken(dev);
```

Read the configuration space header information to configure pci_dev information.

- In line 20~21 of the code, read the header type from the PCI configuration space of the device.
- Fill in the information of the PCI device in lines 23~28 of the code.
  - If the header type bit15 is 1, it is an mmultifunction device.
- Set the PCIe device information in line 29 of the code.
  - Read PCIe capability in Configuration Space to learn pcie_cap, pcie_flags_reg, pcie_mpss, and has_secondary_link information.
- On line 31 of the code, specify the slot number in the bus.
- Line 34 of code allows for both 32bit PCI DMA zones.
- Specify the device name in line 36~38 of the code.
  - "<4 digit domain number>: < 2 digit bus number>: < digit slot number>.<2 digit function number>"
- In line 40~45 of the code, read the double word (4 byte) value corresponding to class_revision from the configuration space, assign the bottom 8 bits to the revision value of the device, and assign the remaining value to the class. It then debugs this information.
- In line 48 of the code, we find out the size of the configaration space.
  - PCI: Standard Size 256
  - PCIe: 256 for default size, 4096 for extended information
  - pcix: Supports extended information for basic sizes 256, 266M, or 533Mhz, allowing the 4096
- In line 51 of the code, if the vendor in the PCI_EXT_CAP_ID_VNDR cap is Intel and the vendor header ID is Thunderbolt, set the is_thunderbolt to 1.
- On line 55 of the code, set the initial PM state to unknown.
- If there is a PCI fixup hook to run at boot time on code lines 58~60, do it.
  - DECLARE_PCI_FIXUP_EARLY() macro action.
  - Create a pci_fixup struct and store it in the .pci_fixup_early section.
  - Certain architectures and driver routines are present.
    - It doesn't exist in the arm64 architecture, but in the arm architecture there are fixup_early routines for three devices from PCI_VENDOR_ID_PLX vendors.
    - drivers/pci/quirks.c has fixup_early routines for specific vendors and devices.
      - e.g. PCI_VENDOR_ID_BROADCOM, 0x16cd and 0x16f0 devices, call the quirk_paxc_bridge() hook function.
- This code only applies when using fake BAR in the host/pcie-tango.c driver on code lines 62~70. Omission.

- On line 72 of the code, the PCI command register is tested with INTX_DISABLE (bit10) bit history to see if it is immutable.

drivers/pci/probe.c -2/2-

```
01         switch (dev->hdr_type) {                    /* header type */
02         case PCI_HEADER_TYPE_NORMAL:                /* standard header */
03                 if (class == PCI_CLASS_BRIDGE_PCI)
04                         goto bad;
05                 pci_read_irq(dev);
06                 pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
07                 pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
08                 pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);
09 
10                 /*
11                  * Do the ugly legacy mode stuff here rather than broken chip
12                  * quirk code. Legacy mode ATA controllers have fixed
13                  * addresses. These are not always echoed in BAR0-3, and
14                  * BAR0-3 in a few cases contain junk!
15                  */
16                 if (class == PCI_CLASS_STORAGE_IDE) {
17                         u8 progif;
18                         pci_read_config_byte(dev, PCI_CLASS_PROG, &progif);
19                         if ((progif & 1) == 0) {
20                                 region.start = 0x1F0;
21                                 region.end = 0x1F7;
22                                 res = &dev->resource[0];
23                                 res->flags = LEGACY_IO_RESOURCE;
24                                 pcibios_bus_to_resource(dev->bus, res, &region);
25                                 dev_info(&dev->dev, "legacy IDE quirk: reg 0x10: %pR\n",
26                                         res);
27                                 region.start = 0x3F6;
28                                 region.end = 0x3F6;
29                                 res = &dev->resource[1];
30                                 res->flags = LEGACY_IO_RESOURCE;
31                                 pcibios_bus_to_resource(dev->bus, res, &region);
32                                 dev_info(&dev->dev, "legacy IDE quirk: reg 0x14: %pR\n",
33                                         res);
34                         }
35                         if ((progif & 4) == 0) {
36                                 region.start = 0x170;
37                                 region.end = 0x177;
38                                 res = &dev->resource[2];
39                                 res->flags = LEGACY_IO_RESOURCE;
40                                 pcibios_bus_to_resource(dev->bus, res, &region);
41                                 dev_info(&dev->dev, "legacy IDE quirk: reg 0x18: %pR\n",
42                                         res);
43                                 region.start = 0x376;
44                                 region.end = 0x376;
45                                 res = &dev->resource[3];
46                                 res->flags = LEGACY_IO_RESOURCE;
47                                 pcibios_bus_to_resource(dev->bus, res, &region);
48                                  dev_info(&dev->dev, "legacy IDE quirk: reg 0x1c: %pR\n",
49                                           res);
50                          }
51                  }
52                  break;
53 
54          case PCI_HEADER_TYPE_BRIDGE:                /* bridge header */
55                  if (class != PCI_CLASS_BRIDGE_PCI)
56                          goto bad;
57                  /* The PCI-to-PCI bridge spec requires that subtractive
58                     decoding (i.e. transparent) bridge must have programming
59                     interface code of 0x01. */
60                  pci_read_irq(dev);
61                  dev->transparent = ((dev->class & 0xff) == 1);
62                  pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
63                  set_pcie_hotplug_bridge(dev);
64                  pos = pci_find_capability(dev, PCI_CAP_ID_SSVID);
65                  if (pos) {
66                          pci_read_config_word(dev, pos + PCI_SSVID_VENDOR_ID, &dev->subsystem_vendor);
67                          pci_read_config_word(dev, pos + PCI_SSVID_DEVICE_ID, &dev->subsystem_device);
68                  }
69                  break;
70 
71          case PCI_HEADER_TYPE_CARDBUS:               /* CardBus bridge header */
72                  if (class != PCI_CLASS_BRIDGE_CARDBUS)
73                          goto bad;
74                  pci_read_irq(dev);
75                  pci_read_bases(dev, 1, 0);
76                  pci_read_config_word(dev, PCI_CB_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
77                  pci_read_config_word(dev, PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
78                  break;
79 
80          default:                                    /* unknown header */
81                  dev_err(&dev->dev, "unknown header type %02x, ignoring device\n",
82                          dev->hdr_type);
83                  return -EIO;
84 
85          bad:
86                  dev_err(&dev->dev, "ignoring class %#08x (doesn't match header type %02x)\n",
87                          dev->class, dev->hdr_type);
88                  dev->class = PCI_CLASS_NOT_DEFINED << 8;
89          }
90 
91          /* We found a fine healthy device, go go go... */
92          return 0;
93  }
```

- In code lines 1~8, the header type of the PCI device is Normal Device.
    - Read the irq line and (0~5) irq number,
    - BAR0~5 address and BAR6-ROM address.
    - Read the subsystem vendor ID and device ID.
- Omit the code for IDE type storage in code lines 16~52
- In code lines 54~69, the header type of the PCI device is Bridge.
    - Read the irq line and (0~5) irq number,

    - BAR0~1 address and BAR6-ROM address. Read whether hot-plug is supported and apply it to the is_hotplug_bridge.
    - If a subsystem vendor CAP exists, it reads the subsystem vendor ID and device ID for it.
- In code lines 71~78, the header type of the PCI device is Cardbus.
    - Read the irq line and (0~5) irq number,
    - BAR0 address and BAR6-ROM address. Read whether hot-plug is supported and apply it to the is_hotplug_bridge.
    - If a subsystem vendor CAP exists, it reads the subsystem vendor ID and device ID for it.

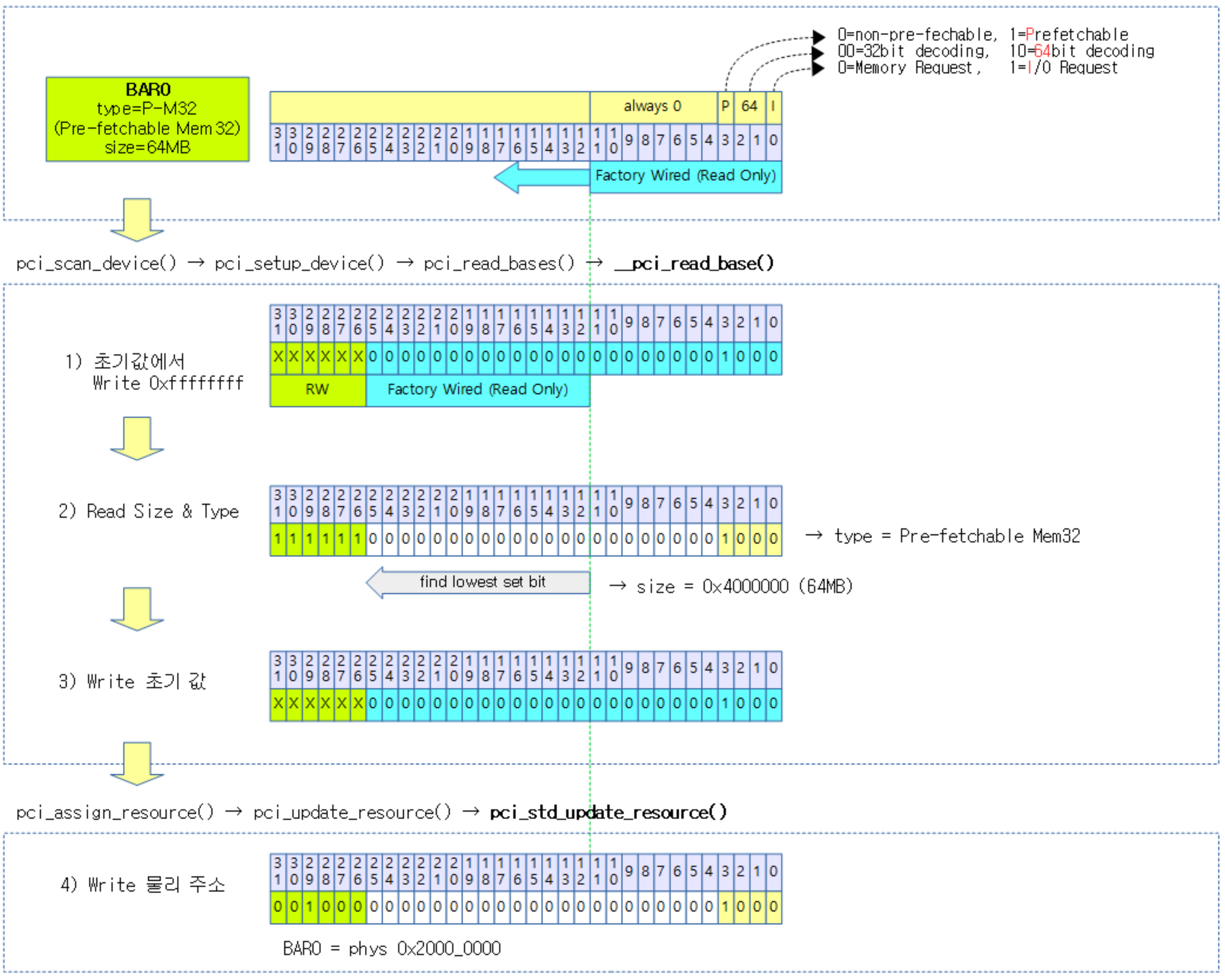
BAR(Base Address Register) Setup

When a PCI host controller detects a PCI device by scanning the bus, it sets up the PCI device, which reads the PCI device's BAR registers, reads the size and type of IO or memory the PCI device has, and records it to the physical memory address of the host CPU.

Mem32 Type BAR

The BAR value of a PCI device records a portion of the BAR register at the factory and provides it as Read Only.

- The notation of Size is determined by recording all the bits in the BAR register as 1 and then determining the size by the position value of the lowest bit set to 4, excluding type 1 bits.
    - Below we found 26 in bit1, so we can figure out that it is 0x400_0000 (64MB) in size.
- Finally, it is completed by recording the address of the deployed resource, starting with the physical address of the host CPU of the memory type specified in ranges in the device tree.
    - 예) ranges = <0x82000000 0 0 0 0x20000000 0 0x400_0000>;
        - Type: Pre-fetchable Mem32
        - PCI Address: 0x0
        - Host CPU Physical Address: 0x20000000
        - Size: 0x4000000 (64MB)

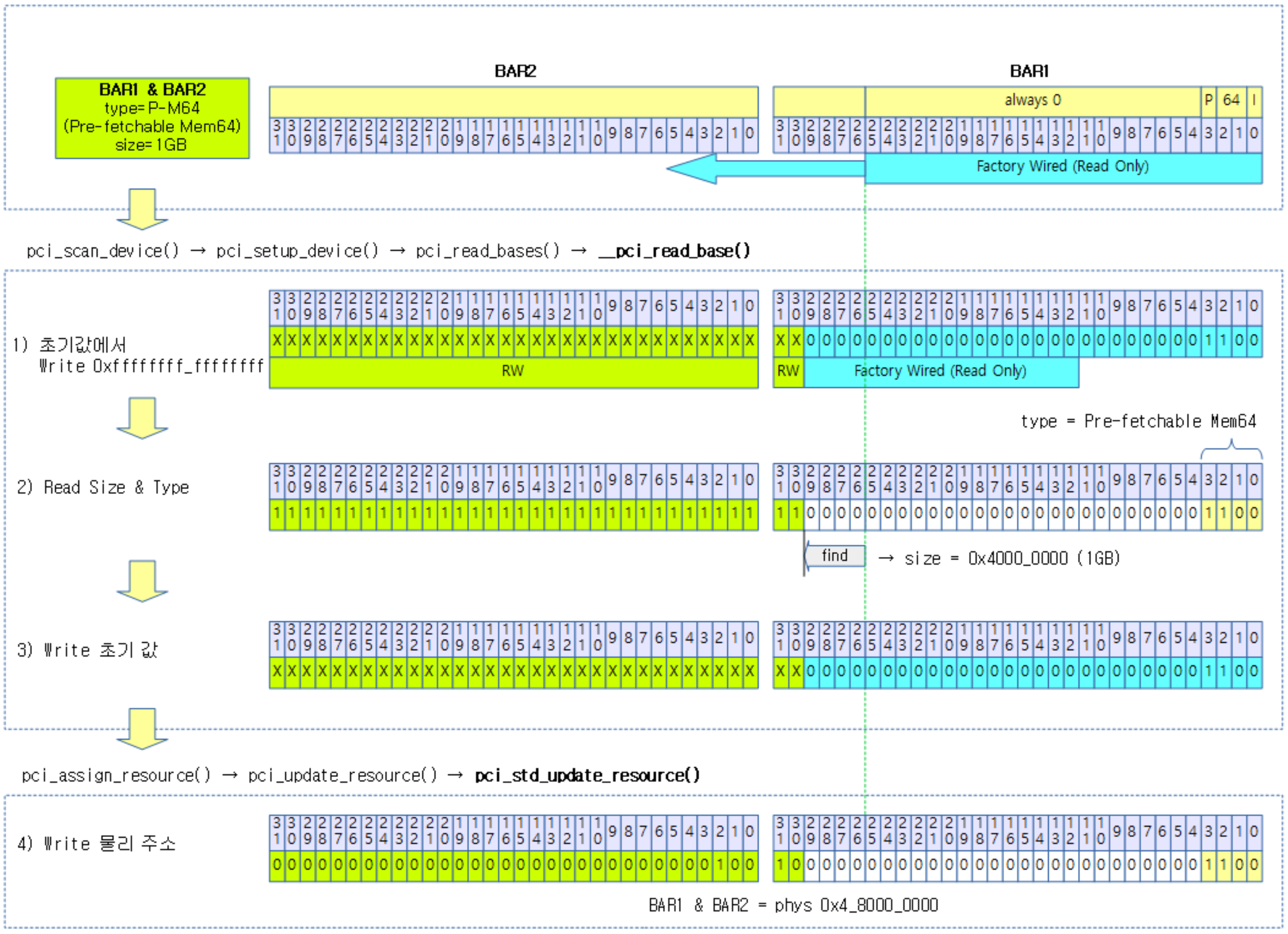

BAR0
type=P-M32
(Pre-fetchable Mem 32)
size=64MB
always 0
P
64
I
0=non-pre-fechable, 1=Prefetchable
00=32bit decoding, 10=64bit decoding
0=Memory Request, 1=I/O Request
Factory Wired (Read Only)
pci_scan_device() → pci_setup_device() → pci_read_bases() → __pci_read_base()
1) 초기값에서 Write 0xffffffff
RW
Factory Wired (Read Only)
2) Read Size & Type
→ type = Pre-fetchable Mem32
find lowest set bit
→ size = 0x4000000 (64MB)
3) Write 초기 값
pci_assign_resource() → pci_update_resource() → pci_std_update_resource()
4) Write 물리 주소
BAR0 = phys 0x2000_0000

(http://jake.dothome.co.kr/wp-content/uploads/2018/10/pci-54.png)

Mem64 Type BAR

The following illustration shows two BARs being used to get the address of 2-bit memory.

- The notation of Size is determined by recording all bits in the BAR1 and BAR2 registers as 1, and then determining the size by the position value of the lowest bit set to 4, except for type 1 bits.
  - Below we found 30 on bit1, so we can figure out that the size is 0x4000_0000 (1GB).
- Finally, it is completed by recording the address of the deployed resource, starting with the physical address of the host CPU of the memory type specified in ranges in the device tree.
  - 예) ranges = <0x83000000 0 0 4 0x80000000 0 0x4000_0000>;
    - Type: Pre-fetchable Mem64
    - PCI Address: 0x0
    - Host CPU Physical Address: 0x4_80000000
    - Size: 0x40000000 (1GB)

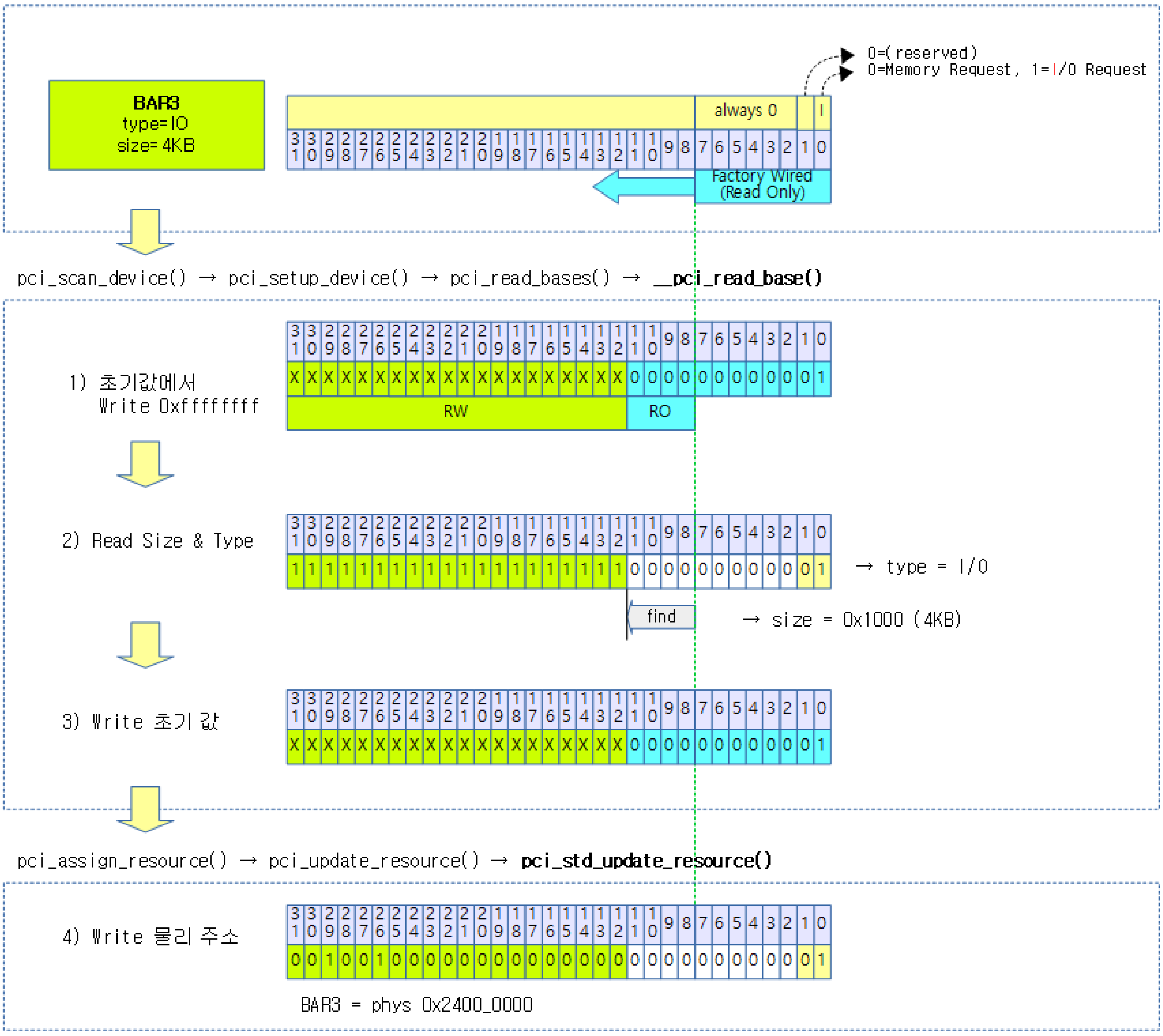

BAR2
BAR1
BAR1 & BAR2
type=P-M64
(Pre-fetchable Mem64)
size=1GB
always 0
P
64
I
Factory Wired (Read Only)
pci_scan_device() → pci_setup_device() → pci_read_bases() → __pci_read_base()
1) 초기값에서
Write 0xffffffff_ffffffff
RW
RW
Factory Wired (Read Only)
type = Pre-fetchable Mem64
2) Read Size & Type
find
→ size = 0x4000_0000 (1GB)
3) Write 초기 값
pci_assign_resource() → pci_update_resource() → pci_std_update_resource()
4) Write 물리 주소
BAR1 & BAR2 = phys 0x4_8000_0000

(http://jake.dothome.co.kr/wp-content/uploads/2018/10/pci-55a.png)

I/O Type BAR

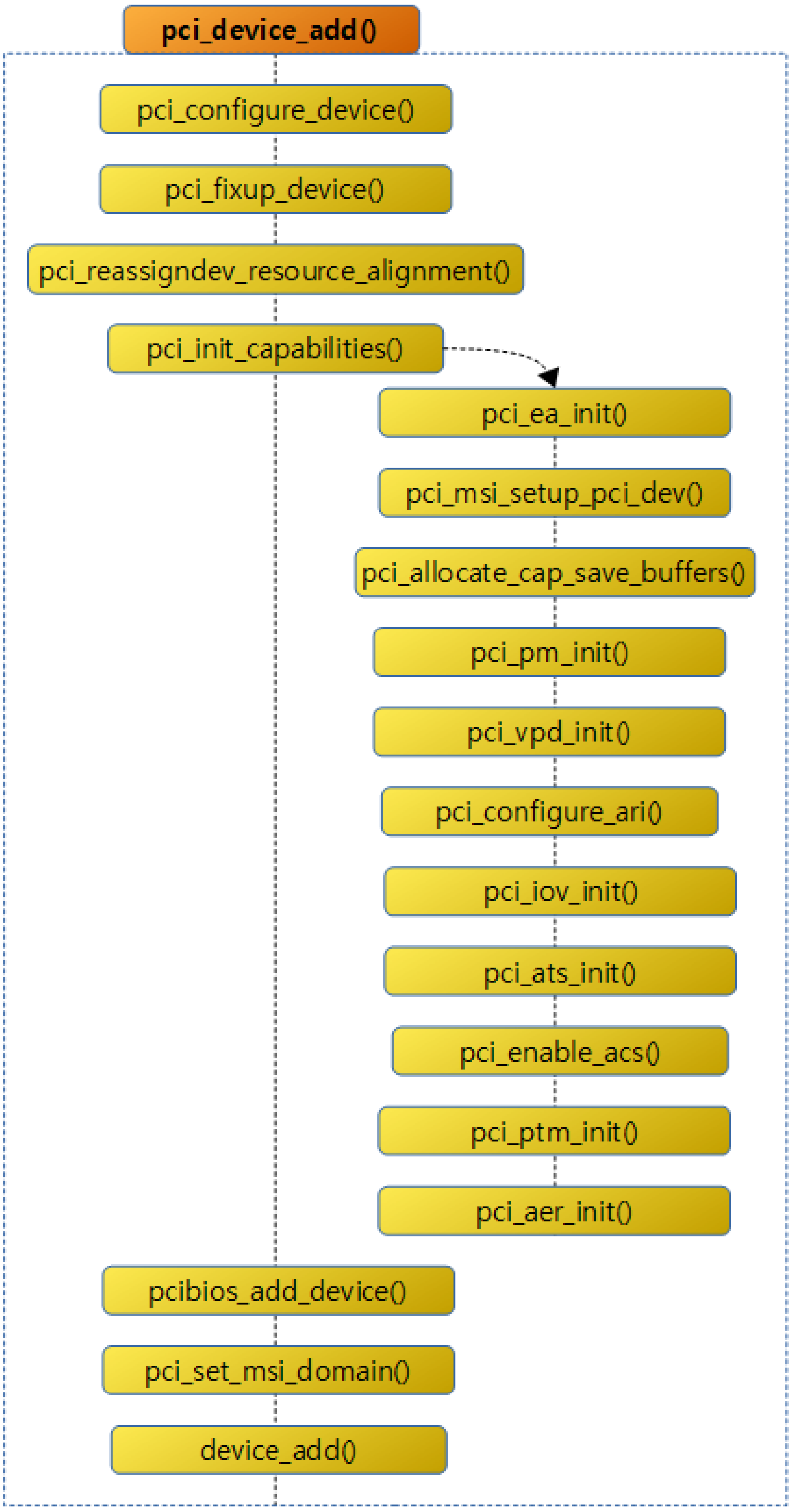
The following figure shows BAR3 representing the IO address.

- The notation of Size is determined by recording all the bits in the BAR3 register as 1 and then using the position value of the lowest bit set to 2, except for type 1 bits.
  - Since 12 is found in bit1 below, we can figure out that the size is 0x1000 (4KB).
- Finally, it is completed by recording the address of the deployed resource, starting with the physical address of the host CPU of the IO type specified in ranges in the device tree.
  - 예) ranges = <0x81000000 0 0 0 0x24000000 0 0x1000>;
    - Type: I/O
    - PCI Address: 0x0
    - Host CPU Physical Address: 0x24000000
    - Size: 0x1000 (4KB)


BAR3
type=IO
size=4KB
always 0
I
0=(reserved)
0=Memory Request, 1=I/O Request
Factory Wired (Read Only)
pci_scan_device() → pci_setup_device() → pci_read_bases() → __pci_read_base()
1) 초기값에서 Write 0xffffffff
RW
RO
2) Read Size & Type
→ type = I/O
find
→ size = 0x1000 (4KB)
3) Write 초기 값
pci_assign_resource() → pci_update_resource() → pci_std_update_resource()
4) Write 물리 주소
BAR3 = phys 0x2400_0000

(http://jake.dothome.co.kr/wp-content/uploads/2018/10/pci-56a.png)

Set up and add PCI devices



(http://jake.dothome.co.kr/wp-content/uploads/2018/09/pci_device_add-1a.png)

pci_device_add()

drivers/pci/probe.c

```
void pci_device_add(struct pci_dev *dev, struct pci_bus *bus)
{
        int ret;

        pci_configure_device(dev);

07          device_initialize(&dev->dev);
08          dev->dev.release = pci_release_dev;
09
10          set_dev_node(&dev->dev, pcibus_to_node(bus));
11          dev->dev.dma_mask = &dev->dma_mask;
12          dev->dev.dma_parms = &dev->dma_parms;
13          dev->dev.coherent_dma_mask = 0xffffffffull;
14
15          pci_set_dma_max_seg_size(dev, 65536);
16          pci_set_dma_seg_boundary(dev, 0xffffffff);
17
18          /* Fix up broken headers */
19          pci_fixup_device(pci_fixup_header, dev);
20
21          /* moved out from quirk header fixup code */
22          pci_reassigndev_resource_alignment(dev);
23
24          /* Clear the state_saved flag. */
25          dev->state_saved = false;
26
27          /* Initialize various capabilities */
28          pci_init_capabilities(dev);
29
30          /*
31           * Add the device to our list of discovered devices
32           * and the bus list for fixup functions, etc.
33           */
34          down_write(&pci_bus_sem);
35          list_add_tail(&dev->bus_list, &bus->devices);
36          up_write(&pci_bus_sem);
37
38          ret = pcibios_add_device(dev);
39          WARN_ON(ret < 0);
40
41          /* Setup MSI irq domain */
42          pci_set_msi_domain(dev);
43
44          /* Notifier could use PCI capabilities */
45          dev->match_driver = false;
46          ret = device_add(&dev->dev);
47          WARN_ON(ret < 0);
48  }
```

- In line 5 of the code, set the MPS, extended tag, relaxed ordering, and hot plug parameter information of the PCI device.
- In line 6 of code, substitute the pci_release_dev() function on the (*release) hook of the PCI device.
- In code lines 8~14, set the device node with the node number used by the bus, and set the device's DMA-related settings to the maximum.
- If you have a pci_fixup_header fuku for a specific device in line 17 of the code, do this.
  - DECLARE_PCI_FIXUP_HEADER() macro action.
  - Create a pci_fixup struct and store it in the .pci_fixup_header section.
- If you specified the resource_alignment bus attribute in line 20 of the code, perform resource alignment. (PowerPC only)
  - resource_alignment=
    - [<order of align>@] [<domain>:]<bus>:<slot>.<func>
    - [:noresize] [; ...]
- On line 23 of code, let pm state save or not false.
- Initialize the various caps in line 26 of code.

- Add the device to the bus in line 32~34 of code.
- In line 36 of the code, the device is mapped according to the PCIBOS settings.
    - This is only if you have a BIOS such as x86, s390, powerpc, or sparc.
    - arm, arm64 are not applicable.
- On line 40 of the code, specify the IRQ domain for the MSI.
- Add the device in lines 43~44 but don't do the match action.

pci_configure_device()

drivers/pci/probe.c

```
01 static void pci_configure_device(struct pci_dev *dev)
02 {
03         struct hotplug_params hpp;
04         int ret;
05 
06         pci_configure_mps(dev);
07         pci_configure_extended_tags(dev, NULL);
08         pci_configure_relaxed_ordering(dev);
09 
10         memset(&hpp, 0, sizeof(hpp));
11         ret = pci_get_hp_params(dev, &hpp);
12         if (ret)
13                 return;
14 
15         program_hpp_type2(dev, hpp.t2);
16         program_hpp_type1(dev, hpp.t1);
17         program_hpp_type0(dev, hpp.t0);
18 }
```

Set the MPS, extended tag, relaxed ordering, hot plug parameter information of the PCI device, etc.

- In line 6 of the code, set the mps of the PCI device to the Maximum Payload Size (mps) value of the PCI Upstream bridge.
    - If the MPS of the PCI device and the PCI upstream bridge are the same, no changes are required.
    - You can use one of the following values: 128, 256, 512, 1024, 2048, or 4096.
    - If the PCI bus is not already configured, it can also be handled later.
- If the extended tag of the PCI host is set in line 7 of the code, it also sets the PCI device.
    - If the Extended Tag bit is set in the PCIe EXP_DEVCAP, the device's settings are changed depending on whether the host can handle the Extended Tag.
    - The message "disabling Extended Tags" or "enabling Extended Tags" is displayed.
- In line 8 of the code, if the PCI root port does not allow relaxed ordering, the PCI device's settings are also cleared.
    - If the PCI_EXP_DEVCTL_RELAX_EN bit of the PCI_EXP_DEVCTL register is set, and the root port has a PCI_DEV_FLAGS_NO_RELAXED_ORDERING flag set, the PCI_EXP_DEVCTL_RELAX_EN of the PCI_EXP_DEVCTL register of the PCI device is cleared.
    - "Disable Relaxed Ordering because the Root Port didn't support it" message is printed.
- In code lines 10~13, when using ACPI, get the PCI Hot Plug parameter and set it on the PCI device.

consultation

- PCI Subsystem -1- (Basic) (http://jake.dothome.co.kr/pci-1) | 문c
- PCI Subsystem -2- (Core) | Question C – Current Article
- PCI Subsystem -3- (Host Controller) (http://jake.dothome.co.kr/pci-3) | 문c

---

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

< Today's ('18.09.05) Traffic Exceeded (http://jake.dothome.co.kr/atraffic-1/)

PCI Subsystem -3- (Host Controller) > (http://jake.dothome.co.kr/pci-3/)

Munc Blog (2015 ~ 2023)