Introduction to PCI devices and functions used by PCI drivers

转载 _kerneler Posted at 2021-01-04 09:16:39 2341 Favorite 12 copyright

Category column: Hda driver

Hda driver The column includes this content

1. Introduction to PCI devices

1.
```
PCI总线的特点:
```

(1) The speed is fast, the clock frequency is increased to 33M, and there is room for further increasing the clock frequency to 66MHZ and the bus bandwidth to 64 bits. (2) For address allocation and setting, the system software class automatically sets, and each peripheral tells the system how many storage intervals and I/O address intervals the peripheral has, the size of each interval, and the local address through some means. After the system software knows how many peripherals and various storage spaces there are, it will uniformly assign physical addresses to the peripherals. (3) For bus competition, an arbiter is equipped on the PCI bus. When encountering a conflict, the arbitrator will choose one of them to temporarily become the current master device, while the others can only wait. At the same time, considering such efficiency issues, the PCI bus outlines a buffer for writing, so writing will be faster than reading. (4) For bus expansion, PCI bus introduces HOST-PCI bridge (North bridge), PCI-PCI bridge, PCI-ISA bridge (South bridge). The CPU and the memory are connected through the system bus, the north bridge connects the memory controller and the main PCI bus, the south bridge connects the main PCI bus and the ISA bus, and the PCI-PCI bridge connects the main PCI and the secondary PCI bus.

2.
```
PCI设备概述
```

Each PCI device has many registers for address configuration. During initialization, the bus address of the device must be configured through these registers. Once the configuration is completed, the CPU can access various resources of the device. The PCI standard stipulates that the configuration register set of each device can have a maximum of 256 consecutive byte spaces. The first 64 bytes are called headers, which are divided into type 0 (PCI device) and type 1 (PCI bridge) headers. The first 16 bytes are the type, model and manufacturer of the device. In addition to the function of address configuration, these header registers can also enable the CPU to detect the existence of corresponding devices, so that the user does not need to tell the system which devices are there, but the CPU automatically scans and detects them through a process called enumeration All devices attached to the PCI bus.

The configuration register group of the device uses the same address, which is distinguished by the PCI bridge on the bus adding other conditions when accessing. For the I386 processor, there are two 32-bit registers, 0XCF8 is the address register, and 0XCFC is the data register. The content written in the address register includes the bus number, device number, and function number. Logical address (XX:YY.Z), XX represents the PCI bus number, up to 256 buses. YY represents the PCI device number, up to 32 devices. Z represents the PCI device function number, up to 8 functions.

3.
```
查询PCI总线和设备的命令
```

View the tree diagram of PCI bus and PCI devices lspci –t

Check the configuration area lspci –x, note that the PCI register is in little endian format

4.
```
PCI总线架构
```

All root buses are linked in the pci_root_buses list. The Pci_bus ->device linked list links all devices under this bus. The pci_bus->children linked list is linked to its underlying bus. For pci_dev, pci_dev->bus points to the pci_bus to which it belongs. Pci_dev->bus_list is linked to the device linked list of the bus to which it belongs. Also, all pci devices are linked in the pci_device linked list.

two. PCI driver

1.
```
PCI寻找空间
```

PCI devices include several addressing spaces: configuration space, I/O port space, and memory space.

1.1 PCI configuration space:

Functions provided by the kernel for the driver:

pci_read_config_byte/word/dword(struct pci_dev *pdev, int offset, int *value)

pci_write_config_byte/word/dword(struct pci_dev *pdev, int offset, int *value)

The offset of the configuration space is defined in include/linux/pci_regs.h

1.2 PCI I/O and memory space:

Get the base address of the I/O area from the corresponding register in the configuration area:

pci_resource_start(struct pci_dev *dev, int bar) The range of Bar value is 0-5.

Get the memory area length of the I/O area from the corresponding register in the configuration area:

pci_resource_length(struct pci_dev *dev, int bar) The range of Bar value is 0-5.

Get the relevant flags of the memory in the I/O area from the corresponding register in the configuration area:

pci_resource_flags(struct pci_dev *dev, int bar) Bar value ranges from 0-5.

Apply for I/O port:

request_mem_region(io_base, length, name)

read and write:

inb() inw() inl() outb() outw() outl()

2. Devices supported by the PCI bus

The PCI driver registers its supported manufacturer ID, device ID and device class code with the PCI subsystem. Using this database, after the inserted card is identified through the configuration space, the PCI subsystem binds the inserted card with the corresponding driver.

PCI device list

struct pci_device_id {
__u32 vendor, device; /* Vendor and device ID or PCI_ANY_ID*/

```
__u32 subvendor, subdevice; /* Subsystem ID's or PCI_ANY_ID */

__u32 class, class_mask;      /* (class,subclass,prog-if) triplet */

kernel_ulong_t driver_data;    /* Data private to the driver */
```

};

```
注意：如果可以处理任何情况，可将相应的寄存器设置为PCI_ANY_ID。
```

3. PCI drives other APIs

Get driver private data: pci_get_drvdata();

Enable PCI devices: pci_enable_device()

Bus master DMA mode setting: pci_set_master()

1. pci driver registration

pci_register_driver(struct pci_driver *drv)

static struct pci_driver hamachi_driver = {
.name = DRV_NAME,
.id_table = hamachi_pci_tbl,

.probe = hamachi_init_one,
.remove = __devexit_p(hamachi_remove_one),
};

2. Private data

pci_set_drvdata set driver private data

pci_get_drvdata Get driver private data

3. PCI configuration space

pci_read_config_byte/word/dword(struct pci_dev *pdev, int offset, int *value)

pci_write_config_byte/word/dword(struct pci_dev *pdev, int offset, int *value)

4. PCI I/O and memory space

pci_resource_start(struct pci_dev *dev, int bar) Bar value ranges from 0-5; get the base address of the I/O area from the corresponding register in the configuration area:

pci_resource_length(struct pci_dev *dev, int bar) Bar value ranges from 0-5; get the memory area length of the I/O area from the corresponding register in the configuration area

pci_resource_flags(struct pci_dev *dev, int bar) Bar value ranges from 0-5; get the relevant flags of the memory in the I/O area from the corresponding register in the configuration area

request_mem_region(io_base, length, name) apply for I/O port

release_mem_region(io_base, length, name) release I/O port
inb() inw() inl() outb() outw() outl() read and write I/O port

pci_enable_device enables device I/O
and memory resources, allocates insufficient resources, and wakes up a suspended device if necessary. Note that this operation may fail.

pci_set_master Set the device to work in bus master mode

ioremap_nocache allows the CPU to access the device's memory

pci_dma_supported

The pci_set_dma_mask() and dma_set_mask() helper functions are used to check whether the bus can receive a bus address of a given size (mask), and if so, notify the bus layer that a given peripheral will use a bus address of that size.

pci_alloc_consistent returns the virtual address of the consistent dma-mapped buffer

upci_free_consistent release consistent dma buffer mapping

pci_save_state configuration space (including PCI, PCI-X, PCI-E) is stored in pci_dev

pci_restore_state restores the state of the configuration space from the value saved in pci_dev

atomic operation

Atomic_read(v) returns the value of an atomic variable

atomic_set(v,i) sets the value of an atomic variable

Atomic_add(int i, atomic_t *v) value of atomic increment count

static inline int atomic_add_return(int i, atomic_t *v) just returns the latest value of variable v

Atomic_sub(int i, atomic_t *v) value of atomic countdown

static inline int atomic_sub_return(int i, atomic_t *v) just returns the latest value of variable v

atomic_cmpxchg(atomic_t *ptr, int old, int new) Compare old and the value in the atomic variable ptr, if they are equal, then assign the new value to the atomic variable. returns the value in the old atomic variable ptr

atomic_clear_mask atomic clear mask

atomic_inc(v) increments the value of the atomic variable by one atomic_inc_return(v) Same as above, except return the latest value of variable v

atomic_dec(v) subtracts one from the value of the atomic variable

atomic_dec_return(v) Same as above, except return the latest value of variable v

atomic_sub_and_test(i, v) Subtract i from an atomic variable v, and judge whether the latest value of variable v is equal to 0

atomic_add_negative(i,v) Add i to an atomic variable v, and determine whether the latest value of variable v is negative

static inline int atomic_add_unless(atomic_t *v, int a, int u) As long as the atomic variable v is not equal to u, then perform the operation of adding a to the atomic variable v. If v is not equal to u, return a non-zero value, otherwise return a 0 value

char device registration

1.int register_chrdev_region(dev_t from, unsigned count, const char *name)

```
 为一个字符驱动获取一个或多个设备编号来使用。用于分配指定的设备编号范围。如果申请的设备编号范围跨越了主设备号，它会把分配范围内的编号按主设
```

int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)

```
用于动态申请设备编号范围，这个函数好像并没有检查范围过大的情况，不过动态分配总是找个空的散列桶，所以问题也不大。通过指针参数返回实际获得的起
```

void unregister_chrdev_region(dev_t from, unsigned count)

void cdev_init(struct cdev *cdev, const struct file_operations *fops) cdev static memory definition initialization

struct cdev *cdev_alloc(void) cdev dynamic memory definition initialization

int cdev_add(struct cdev *p, dev_t dev, unsigned count) After initializing cdev, add it to the system

void cdev_del(struct cdev *p) releases the memory occupied by cdev

2.misc register cdev

static struct file_operations led_fops;

static struct miscdevice up4412_led_dev = {
.minor = MISC_DYNAMIC_MINOR,
.name = “abc”,
.fops = &led_fops,
};

Register (return 0 success):
ret = misc_register(&up4412_led_dev);
Unregister:
misc_deregister(&up4412_led_dev);

Apply for memory

get_zeroed_page(unsigned int flags);
returns a pointer to a new page and fills the page with zeros.
__get_free_page(unsigned int flags);

Similar to get_zeroed_page, but does not clear the page.
__get_free_pages(unsigned int flags, unsigned int order);

Allocates and returns a pointer to the first byte of a memory region, which may be several (physically contiguous) pages long but is not zeroed out.

kmalloc() allocates contiguous physical addresses for small memory allocations

void *vmalloc(unsigned long size) allocates a continuous memory area in the virtual memory space (the virtual space is continuous but the physical space is not necessarily continuous)

void vfree(void *addr)

__pa(address) converts a virtual address to a physical address __va(addrress) converts physical address to virtual address

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

```
void free_page(unsigned long addr)
void free_pages(unsigned long addr, unsigned long order)
```

kernel linked list

struct list_head my_head;

(1) Initialize
INIT_LIST_HEAD(&my_head);
each list_head must be initialized before joining the linked list

(2) Add
list_add(new, &my_head);
list_add_tail(new, &my_head);
Add new members to the linked list

(3) delete
list_del(new);

(4) Find the pointer of the external data structure through the pointer of list_head and
use the container_of macro

```
struct b {
long test;
char ch;
struct list_head list;
…
};
```

```
void my_test(struct list_head *entry)
{
//Find the external structure b through entry
struct b *tmp = container_of(entry, struct b, list);
printk("%d:%c\n", tmp->test , tmp->ch);
…
}
```

(5) Traversal of the linked list

```
struct list_head *pos;
struct b *tmp;
list_for_each(pos, &my_head) {
tmp = container_of(pos, struct b, list);
…
}
```

list_for_each_entry(...);

```
struct b *tmp;
list_for_each_entry(tmp, &my_head, list) {
printk("%d\n", tmp->test);
…
}
```

If the purpose of traversing the linked list is to release the linked list, it is recommended to use:
list_for_each_entry_safe(…);

```
struct b *tmp1, *tmp2;
list_for_each_entry_safe(tmp1, tmp2, &my_head, list) {
list_del(&tmp1->list);
kfree(tmp1);
…;
}
```

access register

Map physical address to virtual address

```
static void __iomem *vir_base;
vir_base = ioremap(GPIO_BASE, GPIO_SIZE);
if (!vir_base) {
printk("Cannot ioremap\n");
return -EIO;
}
```

To access registers, generally use the base address plus offset mode

```
/* 8-bit register*/
char value;
value = readb(vir_base + offset);
writeb(value, (vir_base+offset));

/* 16-bit register*/
short value;
value = readw(vir_base+offset);
writew(value, (vir_base+offset));

/* 32-bit register*/
int value;
value = readl(vir_base+offset);
writel(value, (vir_base+offset));

/* 64-bit register*/
u64 value;
value = readq(vir_base+offset);
writeq(value, (vir_base+offset));

//If the register is no longer accessed, it should be unmapped
iounmap(vir_base);
```

Use of gpio library

Get the GPIO number in <mach/gpio.h> and assign it to gpio_num

Apply to the gpio library to use gpio
ret = gpio_request(gpio_num, “myio”)

Configure io
s3c_gpio_cfgpin(gpio_num, S3C_GPIO_OUTPUT)
can also be configured as S3C_GPIO_INPUT and S3C_GPIO_SFN(x)
The above macros are defined in <plat/gpio-cfg.h>

Set gpio output 0 or 1
gpio_set_value(gpio_num, 0|1)

Get the value of gpio output (0 or 1)
int ret = gpio_get_value(gpio_num)

Release gpio
gpio_free(gpio_num)

Buzzer driver

Before using pwm library to control GPIO, first configure GPIO as PWM output:
int gpio_num = EXYNOS4_GPD0(0)
gpio_request(…)
s3c_gpio_cfgpin(gpio_num, S3C_GPIO_SFN(2))

struct pwm_device *dev;

Apply for a pwm channel (according to the 4412 manual, the id is from 0 to 3)
dev = pwm_request(pwd_id, “xxx”)

Release pwm channel
pwm_free(dev)

Configure the duty cycle and frequency of pwm. The time is in ns.
pwm_config (dev, the ns number of high level in one cycle, the ns number of the whole cycle)
Do not use floating point numbers in the kernel. If the duty cycle is 47%, then calculate The ns number of high level can be ns number of one cycle/100*47

enable pwm
pwm_enable(dev)

turn off pwm
pwm_disable(dev)

kernel spin lock

(1) Only one person can enter the critical section
(2) The waiting person is busy waiting (only SMP will be really busy waiting)
(3) The person holding the lock cannot sleep

Initialize before use

spinlock_t mylock;

spin_lock_init(&mylock);

Ordinary unlocking
spin_lock(&mylock);
critical section
spin_unlock(&mylock);

If the critical section may be interrupted by the interrupt processing function and affect the variables to be protected, the interrupt should be closed while locking
unsigned long flags;
spin_lock_irqsave(&mylock, flags); lock and close the interrupt at the same time, and the current value of CPSR Store in flags
critical section
spin_unlock_irqrestore(&mylock, flags); turn on interrupt when unlocking, and restore the value of flags to CPSR

mutex

The characteristics of mutex: (the waiting time for the lock is usually at the ms level)
(1) a person in the critical section
(2) sleep, etc.
(3) sleep when holding the lock (must ensure that it can be woken up)

Initialize before use

struct mutex mylock;

mutex_init(&mylock);

Lock and unlock
mutex_lock(&mylock);
ret = mutex_lock_interruptible(&mylock);
if (ret)
return -ERESTARTSYS;
critical section
mutex_unlock(&mylock);

semaphore (semaphore)

In the current kernel, semaphore is basically not used to protect the critical section. If some resources in the kernel limit the number of visitors (for example, only 3 people are allowed to visit at the same time), semaphore can be used for protection at this time.

Initialize semaphores according to resource constraints

struct semaphore mysem;

sema_init(&mysem, 3);

down(&mysem);
ret = down_interruptible(&mysem);
…//Code for accessing restricted resources
up(&mysem);

kernel queue

struct workqueue_struct *create_workqueue(const char *name) is used to create a workqueue queue and create a kernel thread for each CPU in the system

struct workqueue_struct *create_singlethread_workqueue(const char *name) creates workqueue, only one kernel thread is created

void destroy_workqueue(struct workqueue_struct *wq) releases the workqueue queue

int schedule_work(struct work_struct *work) schedules and executes a specific task, and the executed task will be linked to the workqueue provided by the Linux system

int schedule_delayed_work(struct delayed_work *work, unsigned long delay) delays for a certain period of time to execute a specific task, the function is similar to schedule_work, with an additional delay time

int queue_work(struct workqueue_struct *wq, struct work_struct *work) schedules and executes tasks in a specified workqueue

int queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *work, unsigned long delay) Delayed scheduling and execution of tasks in a specified workqueue, the function is similar to queue_work

int cancel_delayed_work(struct delayed_work *work) cancels the work before it is executed

void flush_workqueue(struct workqueue_struct *wq);

void flush_scheduled_work(void) generally continues to call flush_delayed_work after calling cancel_delayed_work. This is used to wait until the executing queue is finished. In fact, the latter is to solve the deadlock problem when canceling.

To use a work queue, the first thing to do is create some work that needs to be deferred

DECLARE_WORK(name, void (*func) (void *), void *data) creates a work_struct structure named name, the function to be executed is func, and the parameter is data

DECLARE_DELAYED_WORK(name, func);

INIT_WORK(structwork_struct *work, woid(*func) (void *), void *data) creates a work by pointer at runtime

PREPARE_WORK(struct work_struct work, work_func_t func);
INIT_DELAYED_WORK(struct delayed_work work, work_func_t func);
PREPARE_DELAYED_WORK(struct delayed_work work, work_func_t func);

wait_queue_head_t mywait;

Before using the queue head, initialize
init_waitqueue_head(&mywait);

Go to sleep
wait_event(mywait, dev->wp!=dev->buf_size);
ret = wait_event_interruptible(mywait, dev-wp!=dev->buf_size);

Wake up the sleeping process in the waiting queue

wake_up(&mywait);

wake_up_interruptible(&mywait);

ndelay(10); //delay 10ns
udelay(20); //delay 20us
mdelay(30); //delay 30ms

struct timeval tval;
struct timespec tspec;

Call the kernel function to get the absolute time
do_gettimeofday(&tval);
getnstimeofday(&tspec);

timer

Declare timer
struct timer_list mytimer;

The execution function of the timer, when the timer expires, it will be executed once by the hardware timer interrupt
static void my_timer_func(unsigned long data)
{
...//no sleep
}

Initialize the timer
setup_timer(&mytimer, my_timer_func, data);
the parameter passed in when the timer is initialized is the pointer of timer_list; the execution function; the parameter passed to the execution function

Start the timer
mod_timer(&mytimer, jiffies+HZ);
Once the timer is started, it will be added to a linked list of timer_list, and it will be executed once it expires.
The person who starts the timer and the person who executes it are not the same. Even if the initiator exits, the timer still executes.

Delete timer
del_timer(&mytimer);
If the module needs to be rmmod, all unexecuted timers must be deleted before uninstalling.

kernel interrupt

Interrupt numbers are defined in <mach/irqs.h>, you can use two different methods to find the interrupt number:

(1) Peripherals inside the chip
First, specify the name of the device, and then use the name matching to find the corresponding interrupt number in irqs.h; for
example, the interrupt number corresponding to the watchdog device is IRQ_WDT, and the corresponding interrupt number of the rtc hardware is IRQ_RTC_ALARAM/ IRQ_RTC_TIC

(2) Devices connected to the outside of the chip
Since the interrupt pins of the device are connected to GPIO, you can use the GPIO number to find the interrupt number.
Interrupt number = gpio_to_irq(GPIO number)

When the driver designs the interrupt processing function, the requirements to be followed are:
(1) Nestable and non-reentrant
(2) Cannot sleep
(3) If the hardware has an interrupt status register, the software should be responsible for clearing the interrupt flag. Generally speaking, if the flag bit is not cleared, the device cannot generate an interrupt again
kzalloc(size, GFP_KERNEL); //may sleep
kzalloc(size, GFP_ATOMIC); //will not sleep

1.

Determine the interrupt number
#define KEY_IRQ gpio_to_irq(gpio number);

Interrupt handling function
static irqreturn_t key_service(int irq, void *dev_id)
{
...
return IRQ_HANDLED or IRQ_NONE;
}

To register an interrupt handler, the return value must be checked

u32 flags = IRQF_TRIGGER_FALLING
| IRQF_TRIGGER_RISING;

ret = request_irq(KEY_IRQ, /* interrupt number /
key_service, / interrupt processing function /
flags, / interrupt flag /
"xxx", / interrupt processing function name /
dev_id);
/ The last parameter dev_id is passed to the interrupt processing function Parameters are generally set as pointers to private structures, which cannot be NULL */
Actually, if it is a non-shared interrupt, dev_id can be NULL

Logout interrupt processing function
free_irq(irq, dev_id) dev_id must be consistent with the last parameter in request_irq.

Artificially close (mask)/open an interrupt:
disable_irq(int irq);
enable_irq(int irq);
The above two functions support nesting, that is, if you call disable_irq 3 times, you need to enable_irq 3 times to actually use Can be interrupted
Make sure to call disable_irq first, then call enable_irq;

If you want to shield the interrupt of the entire cpu, you can use:
local_irq_disable();
local_irq_enable();
actually set the I position of the CPSR register to 1 or clear to 0

2.

interrupt bottom half

Before entering the interrupt processing function, the interrupt will be disabled by default. For some interrupts that require quick response or high data throughput, it is necessary to consider dividing the work of the interrupt handler into two parts, which are called the upper half and the lower half of the interrupt respectively.
There are many ways to implement the lower part, including softirq, tasklet and work queue. For drivers, only tasklets and work queues are used

Turn on or off the lower half of the cpu:
local_bh_enable();
local_bh_disable();

tasklet

(1) Execute immediately after the first half is executed, but all interrupts are turned on at this time;
(2) The kernel is still in the interrupt context when the tasklet is executed, so it cannot sleep;
(3) The execution function of the tasklet will not be reentrant;
(4) If scheduling occurs again during the execution of the tasklet, the second scheduling is invalid;

Declare the tasklet structure
struct tasklet_struct mytask;

Tasklet execution function
void bo_service(unsigned long data)
{
}

The execution function of the upper part
irqreturn_t up_service(int irq, void *dev_id)
{
//First complete the important work such as interacting with the hardware
//trigger the tasklet lower part
tasklet_schedule(&mytask);
or tasklet_hi_schedule(&mytask);
...
}

Initialize tasklet
tasklet_init(&mytask, bo_service, (unsigned long)dev);

work queue

(1) Postpone to the process context execution, at this time all interrupts are turned on;
(2) When executing work, it is in the process context, so it can sleep;
(3) The execution function of work will not be reentrant;
(4) If it is in the work Scheduling occurs again during the execution of , and the second scheduling is invalid;

macro definition

The cmd of ioctl() has a size of 32 bits and is divided into 4 domains:
_IOC_DIR

The 2 bits of bit31~bit30 are "differential reading and writing" area, which is used to distinguish whether it is a read command or a write command.

_IOC_SIZE

The 14 bits of bit29~bit15 are the "data size" area, indicating the memory size transferred by the arg variable in ioctl().

_IOC_TYPE

The 8 bits of bit20~bit08 are the "magic number" (also called "magic number") area, and this value is used to distinguish it from the ioctl commands of other device drivers.

_IOC_NR()

The 8 bits of bit07~bit00 are the "differential serial number" area, which is the command sequence number for distinguishing commands.