努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处
http://blog.csdn.net/woshixingaaa/archive/2011/05/15/6421954.aspx
首先介绍一下 I/O 端口和 I/O 内存。
1. I/O 端口：当一个寄存器或内存位于 I/O 空间时，称其为 I/O 端口。
2. I/O 内存：当一个寄存器或内存位于内存空间时，称其为 I/O 内存。
再来看一下 I/O 寄存器和常规内存的区别：I/O 寄存器具有边际效应（side effect），而内存操作则没有,内存写操作的唯一结果就是在指定位置存贮一个数值；内存读操作则仅仅是返回指定位置最后一次写入的数值。何为边际效应呢？就是读取某个地址时可能导致该地址内容发生变化。比如很多设备的中断状态寄存器只要一读取，便自动清零。
现在来看一看如何在 Linux 驱动程序中使用 I/O 端口和 I/O 内存。
使用 I/O 端口的步骤：
1. 申请
2. 访问
3. 释放
申请 I/O 端口：
在尚未对这些端口进行申请之前我们不应该对这些端口进行操作的。内核为我们提供了一个注册用的接口：

```
1. #include <linux/ioport.h>
2. struct resource *request_region(unsigned long first, unsigned long n, const char *name);
```

这个函数告诉内核，我们要使用起始于 first 的 n 个端口，参数 name 应该是设备的名称。如果分配成功，则返回非 NULL。如果 request_region 返回 NULL，那么我们就不能使用这些期望的端口。
访问 I/O 端口：
访问 I/O 端口时，多数硬件都会把 8 位，16 位和 32 位的端口区分开。因此，C 语言程序中必须调用不同的函数来访问大小不同的端口。

```
1. unsigned inb(unsigned port);
2. void outb(unsigned char byte, unsigned port);
```

字节（8 位宽度）读写端口。

```
1. unsigned inw(unsigned port);
2. void outw(unsigned short word, unsigned port);
```

读写 16 位端口。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

读写 32 位端口。

释放 I/O 端口：

如果不在使用某组 I/O 端口（可能在卸载模块时），则应该使用下面的函数将这些端口返回给系统：

```
void release_region(unsigned long start, unsigned long n);
```

使用 I/O 内存的步骤：

1. 申请
2. 映射
3. 访问
4. 释放

根据计算机平台和所使用总线的不同，I/O 内存可能是，也可能不是通过页表访问的。如果访问是经由页表进行的，内核必须首先安排物理地址使其对设备驱动程序可见（这通常意味着在进行任何 I/O 之前必须先调用 ioremap）。如果访问无需页表，那么 I/O 内存区域就非常类似于 I/O 端口，可以使用适当形式的函数读写它们。

I/O 内存申请：

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
```

该函数从 start 开始分配 len 字节长的内存区域。如果成功，返回非 NULL 指针；否则返回 NULL 值。

I/O 内存映射：

```
#include<asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
```

把物理地址转化成虚拟地址，返回值是虚拟地址。

```
void *iounmap(void *addr);
```

解除映射。

I/O 内存访问：

从内存中读取：

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

其中 addr 应该是从 ioremap 获得的地址。
还有一组写入 I/O 内存类似函数：

```
1. void iowrite8(u8 value, void *addr);
2. void iowrite16(u16 value, void *addr);
3. void iowrite32(u32 value, void *addr);
```

I/O 内存释放：

```
1. void release_mem_region(unsigned long start, unsigned long len);
```

像 I/O 内存一样使用 I/O 端口：

```
1. void *ioport_map(unsigned long port, unsigned int count);
```

该函数重新映射 count 个 I/O 端口，使其看起来像 I/O 内存。此后，驱动程序可在该函数返回的地址上使用 ioread8 及其同类的函数。
当不再需要这种映射时，需要调用下面的函数来撤销：

```
1. void ioport_unmap(void *addr);
```

上边的是基础知识。
在我们的开发板上内存映射分 3 个层次（下边的所有内核代码使用的是 linux3.6.30.4）：
1.开发板的层次
如：声卡，网卡和开发板相关的部分
2.最小系统的层次
系统必须的几个，如 GPIO,IRQ,MEMCTRL,UART。
3.其他系统的层次
不影响开机的部分，如 USB,LCD,ADC。
开发板 mapio 的初始化：
smdk2440_map_io 函数中会调用：

```
1. static struct map_desc smdk2440_iodesc[] __initdata = {
2.     /* ISA IO Space map (memory space selected by A24) */
3.     {
4.         .virtual    = (u32)S3C24XX_VA_ISA_WORD,
5.         .pfn        = __phys_to_pfn(S3C2410_CS2),
6.         .length     = 0x10000,
7.         .type       = MT_DEVICE,
8.     }, {
9.         .virtual    = (u32)S3C24XX_VA_ISA_WORD + 0x10000,
10.        .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
11.        .length     = SZ_4M,
12.        .type       = MT_DEVICE,
```

```
    }, {
        .virtual    = (u32)S3C24XX_VA_ISA_BYTE,
        .pfn        = __phys_to_pfn(S3C2410_CS2),
        .length     = 0x10000,
        .type       = MT_DEVICE,
    }, {
        .virtual    = (u32)S3C24XX_VA_ISA_BYTE + 0x10000,
        .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
        .length     = SZ_4M,
        .type       = MT_DEVICE,
    },{
        .virtual    = (u32)S3C2410_ADDR(0x07600000),
        .pfn        = __phys_to_pfn(0x40000000),
        .length     =SZ_4K,
        .type       = MT_DEVICE,
    }
};
```

最小系统 IO 初始化：

s3c24xx_init_io 函数会调用：

```
iotable_init(s3c_iodesc,ARRAY_SIZE(s3c_iodesc));
/* minimal IO mapping */
static struct map_desc s3c_iodesc[] __initdata = {
    IODESC_ENT(GPIO),
    IODESC_ENT(IRQ),
    IODESC_ENT(MEMCTRL),
    IODESC_ENT(UART)
};
```

这个部分是系统启动必须的映射。后续会调用（cpu->map_io）(mach_desc,size)；来完成其他映射。这个函数会调用：

```
Iotable_init(s3c2440_iodesc,ARRAY_SIZE(s3c2440_iodesc));
/* Initial IO mappings */
static struct map_desc s3c2410_iodesc[] __initdata = {
    IODESC_ENT(CLKPWR),
    IODESC_ENT(TIMER),
    IODESC_ENT(WATCHDOG),
};
```

所以，如果新添加一个驱动，首先要看是否完成了 IO 映射，如果没有的话，就在开发板部分加入。Linux 内核访问外设 I/O 资源的方式有两种：动态映射和静态映射(map_desc)。

动态映射方式上边已经讲述，这里着重讲述静态映射——通过 map_desc 结构静态创建 I/O 资源映射表。内核提供了在系统启动时通过 map_desc 结构体静态创建 I/O 资源到内核地址空间的线性映射表（即 page table）的方式。这种映射表

是一种一一映射的关系。程序员可以自己定义该 I/O 内存资源映射后的虚拟地址。创建好静态映射表，在内核或驱动中访问该 I/O 资源时则无需再进行 ioremap 动态映射，可以直接通过映射后的 I/O 虚拟地址去访问。下面详细分析这种机制的原理并举例说明如何通过这种静态映射的方式访问外设 I/O 内存资源。

内核提供了一个重要的结构体 struct machine_desc，这个结构体在内核移植中起到相当重要的作用，内核通过 machine_desc 结构体来控制系统体系架构相关部分的初始化。machine_desc 结构体包含了体系结构相关部分的几个重要成员的初始化函数，包括 map_io,init_irq,init_machine 以及 phys_io，timer 成员等。machine_desc 结构体定义如下：

```
1. struct machine_desc {
2.    /*
3.     * Note! The first four elements are used
4.     * by assembler code in head.S, head-common.S
5.     */
6.    unsigned int        nr;     /* architecture number  */
7.    unsigned int        phys_io;    /* start of physical io */
8.    unsigned int        io_pg_offst;    /* byte offset for io
9.                         * page tabe entry  */
10.   const char      *name;      /* architecture name    */
11.   unsigned long       boot_params;    /* tagged list      */
12.   unsigned int        video_start;    /* start of video RAM   */
13.   unsigned int        video_end;  /* end of video RAM */
14.   unsigned int        reserve_lp0 :1; /* never has lp0    */
15.   unsigned int        reserve_lp1 :1; /* never has lp1    */
16.   unsigned int        reserve_lp2 :1; /* never has lp2    */
17.   unsigned int        soft_reboot :1; /* soft reboot      */
18.   void            (*fixup)(struct machine_desc *,
19.                    struct tag *, char **,
20.                    struct meminfo *);
21.   void            (*map_io)(void);/* IO mapping function  */
22.   void            (*init_irq)(void);
23.   struct sys_timer    *timer;     /* system tick timer    */
24.   void            (*init_machine)(void);
25.};
```

这里的 map_io 成员即内核提供给用户的创建外设 I/O 资源到内核虚拟地址静态映射表的接口函数。map_io 成员函数会在系统初始化过程中被调用，流程如下：start_kernel->setup_arch()->paging_init()->devicemaps_init()中被调用。

machine_desc 结构体通过 MACHINE_START 宏来初始化。

```
MACHINE_START 定义在 arch/arm/include/asm/mach/arch.h 中
/*
 * Set of macros to define architecture features.  This is built into
 * a table by the linker.
 */
#define MACHINE_START(_type,_name)          /
static const struct machine_desc __mach_desc_##_type    /
 __used                          /
 __attribute__((__section__(".arch.info.init"))) = {    /
  .nr     = MACH_TYPE_##_type,        /
  .name       = _name,
#define MACHINE_END             /
};
```

用户可以在定义 machine_desc 结构时指定 map_io 的接口函数。s3c2410 machine_desc 结构体如下定义：

```
MACHINE_START(S3C2440, "SMDK2440")
    /* Maintainer: Ben Dooks <ben@fluff.org> */
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C2410_SDRAM_PA + 0x100,
    .init_irq   = s3c24xx_init_irq,
    .map_io     = smdk2440_map_io,
    .init_machine   = smdk2440_machine_init,
    .timer      = &s3c24xx_timer,
MACHINE_END
```

展开后的结果是：

```
static const struct machine_desc __mach_desc_SMDK2410
 __attribute_used__
 __attribute__((__section__(".arch.info.init"))) = {
 .nr = MACH_TYPE_SMDK2410, /* architecture number */
 .name = "SMDK2410", /* architecture name */
 /* Maintainer: Jonas Dietsche */
 .phys_io = S3C2410_PA_UART, /* start of physical io */
 .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
 .boot_params = S3C2410_SDRAM_PA + 0x100, /* tagged list */
 .map_io = smdk2410_map_io, /* IO mapping function */
 .init_irq = s3c24xx_init_irq,
 .init_machine = smdk_machine_init,
 .timer = &s3c24xx_timer,
}
```

下边是 smdk2440_map_io 函数：

```
static void __init smdk2440_map_io(void)
{
```

```
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    s3c24xx_init_clocks(12000000);
    s3c24xx_init_uarts(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}
```

它调用了 s3c24xx_init_io 函数：

```
void __init s3c24xx_init_io(struct map_desc *mach_desc, int size)
{
    unsigned long idcode = 0x0;
    /* initialise the io descriptors we need for initialisation */
    iotable_init(mach_desc, size);
    iotable_init(s3c_iodesc, ARRAY_SIZE(s3c_iodesc));
    if (cpu_architecture() >= CPU_ARCH_ARMv5) {
        idcode = s3c24xx_read_idcode_v5();
    } else {
        idcode = s3c24xx_read_idcode_v4();
    }
    arm_pm_restart = s3c24xx_pm_restart;
    s3c_init_cpu(idcode, cpu_ids, ARRAY_SIZE(cpu_ids));
}
/*
* Create the architecture specific mappings
*/
void __init iotable_init(struct map_desc *io_desc, int nr)
{
    int i;
    for (i = 0; i < nr; i++)
        create_mapping(io_desc + i);
}
```

```
/*
 * Create the architecture specific mappings
 */
void __init iotable_init(struct map_desc *io_desc, int nr)
{
    int i;
    for (i = 0; i < nr; i++)
        create_mapping(io_desc + i);
}
```

所以，smdk2410_map_io 最终调用 iotable_init 建立映射表。

iotable_init 函数的参数有两个：一个是 map_desc 结构体，另一个是该结构体的数量 nr。这里最关键的就是 struct map_desc，map_desc 结构的定义如下：

map_desc 定义在 arch/arm/include/asm/mach/map.h 中，

```
struct map_desc {
    unsigned long virtual;   /*映射后的虚拟地址*/
    unsigned long pfn;     /*I/O 资源物理地址所在的页帧号*/
    unsigned long length;   /*I/O 资源长度*/
    unsigned int type;      /*I/O 资源类型*/
};
```

create_mapping 函数就是通过 map_desc 提供的信息创建线性映射表的。

这样的话我们就可以知道了创建 I/O 映射表的大致流程为：只要定义相应的 I/O 资源的 map_desc 结构体，并将该结构体传给 iotable_init 函数执行，就可以创建相应的 I/O 资源到内核虚拟地址空间的映射表了。我们来看看 s3c2410 是怎么定义 map_desc 结构体的。

```
/* Initial IO mappings */
static struct map_desc s3c2410_iodesc[] __initdata = {
    IODESC_ENT(CLKPWR),
    IODESC_ENT(TIMER),
    IODESC_ENT(WATCHDOG),
};
```

IODESC_ENT 定义在 arch/arm/plat-s3c/include/plat/cpu.h 中，展开后等价于：

```
static struct map_desc s3c2410_iodesc[] __initdata = {
    {
        .virtual    =    (unsigned long)S3C24XX_VA_ TIMER),
        .pfn        =    __phys_to_pfn(S3C24XX_PA_ TIMER),
        .length     =    S3C24XX_SZ_ TIMER,
        .type     =    MT_DEVICE
    },
    ……
};
```

这里 S3C24XX_PA_TIMER 和 S3C24XX_VA_TIMER 为定义在 arch/arm/plat-s3c24xx/include/plat/map.h 内 TIMER 寄存器的物理地址和虚拟地址。在这里 map_desc 结构体的 virtual 成员被初始化为 S3C24XX_VA_TIMER，pfn 成员值通过__phys_to_pfn 内核函数计算，只要传递给它该 I/O 的物理地址就可行了。Length 为映射资源的大小。MT_DEVICE 为 I/O 类型，通常定义为 MT_DEVICE。这里最重要的即 virtual 成员的值 S3C24XX_VA_ TIMER，这个值即该 I/O 资源映射后的内核虚拟地址，创建映射表成功后，便可以在内核或驱动中直接通过该虚拟地址访问这个 I/O 资源。

```
/* Timers */
#define S3C24XX_VA_TIMER   S3C_VA_TIMER
#define S3C2410_PA_TIMER   (0x51000000)
#define S3C24XX_SZ_TIMER   SZ_1M
#define S3C_VA_TIMER    S3C_ADDR(0x00300000)    /* timer block */
```

```
#define S3C_ADDR_BASE   (0xF4000000)
#ifndef __ASSEMBLY__
#define S3C_ADDR(x) ((void __iomem __force *)S3C_ADDR_BASE + (x))
#else
#define S3C_ADDR(x) (S3C_ADDR_BASE + (x))
#endif
```

在 ARM9 中，如果从 nand 启动，sram 被映射到 bank0,在启动后这个 sram 可以用作其他用途。下边是测试程序 sram.c：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/ioport.h>
#include <asm/mach/map.h>
#include <mach/hardware.h>
void sram_test(void){
    void *test;
    char sram[] = "my_iomap_test success";
    test = (void*)S3C2410_ADDR(0x07600000);
    memcpy(test,sram,sizeof(sram));
    printk(test);
    printk("/n");
}
static int __init my_iomap_init(void){
    struct resource *ret;
    printk("my_iomap_test init/n");
    ret = request_mem_region(0x0000000,0x1000,"sram");
    if(ret == NULL){
        printk("io memory request fail!/n");
        return -1;
    }
    sram_test();
    return 0;
}
static void __exit my_iomap_exit(void){
    printk("my_iomap_test exit/n");
    release_mem_region(0x00000000,0x1000);
}
module_init(my_iomap_init);
module_exit(my_iomap_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("liwanpeng");
```

在内核中修改：

```
static struct map_desc smdk2440_iodesc[] __initdata = {
    /* ISA IO Space map (memory space selected by A24) */

    {
        .virtual    = (u32)S3C24XX_VA_ISA_WORD,
```

```
        .pfn        = __phys_to_pfn(S3C2410_CS2),
        .length     = 0x10000,
        .type       = MT_DEVICE,
    }, {
        .virtual    = (u32)S3C24XX_VA_ISA_WORD + 0x10000,
        .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
        .length     = SZ_4M,
        .type       = MT_DEVICE,
    }, {
        .virtual    = (u32)S3C24XX_VA_ISA_BYTE,
        .pfn        = __phys_to_pfn(S3C2410_CS2),
        .length     = 0x10000,
        .type       = MT_DEVICE,
    }, {
        .virtual    = (u32)S3C24XX_VA_ISA_BYTE + 0x10000,
        .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
        .length     = SZ_4M,
        .type       = MT_DEVICE,
    },{
        .virtual    = (u32)S3C2410_ADDR(0x07600000),
        .pfn        = __phys_to_pfn(0x00000000),
        .length     =SZ_4K,
        .type       = MT_DEVICE,
    }
};
```

测试结果：

```
[root@LWP usb]# insmod sram.ko
my_iomap_test init
my_iomap_test success
```