专题4-Linux内核访问外设IO--动态映射(ioremap)和静态映射(map_desc)

一、Linux内核访问外设I/O资源的方式

我们知道默认外设I/O资源是不在Linux内核空间中的（如sram或硬件接口寄存器等），若需要访问该外设I/O资源，必须先将其地址映射到内核空间中来，然后才能在内核空间中访问它。Linux内核访问外设I/O内存资源的方式有两种：动态映射(ioremap)和静态映射(map_desc)。

1.1、动态映射(ioremap)方式

动态映射方式是大家使用了比较多的，也比较简单。即直接通过内核提供的ioremap函数动态创建一段外设I/O内存资源到内核虚拟地址的映射表，从而可以在内核空间中访问这段I/O资源。ioremap宏定义在asm/io.h内：

```
#define ioremap(cookie,size)        __ioremap(cookie,size,0)
```

__ioremap函数原型为(arm/mm/ioremap.c)：

```
void __iomem * __ioremap(unsigned long phys_addr, size_t size, unsigned long flags);
```

phys_addr：要映射的起始的IO地址

size：要映射的空间的大小

flags：要映射的IO空间和权限有关的标志

该函数返回映射后的内核虚拟地址(3G-4G). 接着便可以通过读写该返回的内核虚拟地址去访问之这段I/O内存资源。

举一个简单的例子: (取自s3c2410的iis音频驱动)

比如我们要访问s3c2410平台上的I2S寄存器, 查看datasheet 知道IIS物理地址为0x55000000，我们把它定义为宏S3C2410_PA_IIS，如下：

```
#define S3C2410_PA_IIS   (0x55000000)
```

若要在内核空间(iis驱动)中访问这段I/O寄存器(IIS)资源需要先建立到内核地址空间的映射:

```
our_card->regs = ioremap(S3C2410_PA_IIS, 0x100);
if (our_card->regs == NULL) {
        err = -ENXIO;
        goto exit_err;
}
```

创建好了之后，我们就可以通过readl(our_card->regs)或writel(value, our_card->regs)等IO接口函数去访问它。

1.2、静态映射(map_desc)方式

下面重点介绍静态映射方式即通过map_desc结构体静态创建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。创建好了静态映射表，在内核或驱动中访问该I/O资源时则无需再进行ioreamp动态映射，可以直接通过映射后的I/O虚拟地址去访问它。

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结构体的成员包含了体系架构相关部分的几个最重要的初始化函数，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：

```
struct machine_desc {
    /*
     * Note! The first four elements are used
     * by assembler code in head-armv.S
     */
    unsigned int        nr;       /* architecture number    */
    unsigned int        phys_io;    /* start of physical io    */
    unsigned int        io_pg_offst;    /* byte offset for io
                         * page tabe entry    */
    const char      *name;      /* architecture name    */
    unsigned long       boot_params;    /* tagged list       */
    unsigned int        video_start;    /* start of video RAM    */
    unsigned int        video_end;    /* end of video RAM    */
    unsigned int        reserve_lp0 :1;    /* never has lp0    */
    unsigned int        reserve_lp1 :1;    /* never has lp1    */
    unsigned int        reserve_lp2 :1;    /* never has lp2    */
    unsigned int        soft_reboot :1;    /* soft reboot       */
    void            (*fixup)(struct machine_desc *,
                 struct tag *, char **,
                 struct meminfo *);
    void            (*map_io)(void);/* IO mapping function    */
    void            (*init_irq)(void);
    struct sys_timer    *timer;       /* system tick timer    */
    void            (*init_machine)(void);
};
```

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。

注：MACHINE_START的使用及各个成员函数的调用过程请参考:

专题5-从MACHINE_START开始

用户可以在定义Machine_desc结构体时指定Map_io的接口函数，这里以s3c2410平台为例。

s3c2410 machine_desc结构体定义如下：

```
/* arch/arm/mach-s3c2410/Mach-smdk2410.c */
MACHINE_START(SMDK2410, "SMDK2410") /* @TODO: request a new identifier and switch
            * to SMDK2410 */
   /* Maintainer: Jonas Dietsche */
   .phys_io    = S3C2410_PA_UART,
   .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
   .boot_params    = S3C2410_SDRAM_PA + 0x100,
   .map_io       = smdk2410_map_io,
   .init_irq    = s3c24xx_init_irq,
   .init_machine    = smdk2410_init,
   .timer       = &s3c24xx_timer,
MACHINE_END
```

如上,map_io被初始化为smdk2410_map_io。smdk2410_map_io即我们自己定义的创建静态I/O映射表的函数。在Porting内核到新开发板时，这个函数需要我们自己实现。

(注：这个函数通常情况下可以实现得很简单，只要直接调用iotable_init创建映射表就行了，我们的板子内核就是。不过s3c2410平台这个函数实现得稍微有点复杂，主要是因为它将要创建IO映射表的资源分为了三个部分(smdk2410_iodesc, s3c_iodesc以及s3c2410_iodesc)在不同阶段分别创建。这里我们取其中一个部分进行分析，不影响对整个概念的理解。)

S3c2410平台的smdk2410_map_io函数最终会调用到s3c2410_map_io函数。

流程如下：s3c2410_map_io -> s3c24xx_init_io -> s3c2410_map_io

下面分析一下s3c2410_map_io函数：

```
void __init s3c2410_map_io(struct map_desc *mach_desc, int mach_size)
{
   /* register our io-tables */
   iotable_init(s3c2410_iodesc, ARRAY_SIZE(s3c2410_iodesc));
   ……
}
```

iotable_init内核提供，定义如下：

```
/*
* 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);
}
```

由上知道，s3c2410_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类型的结构体，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结构体定义如下：

```
/* include/asm-arm/mach/map.h */
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。

这样的话我们就知道了创建I/O映射表的大致流程为：只要定义相应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(即上面s3c2410_map_io函数内的s3c2410_iodesc)。

```
/* arch/arm/mach-s3c2410/s3c2410.c */
static struct map_desc s3c2410_iodesc[] __initdata = {
   IODESC_ENT(USBHOST),
   IODESC_ENT(CLKPWR),
   IODESC_ENT(LCD),
   IODESC_ENT(TIMER),
   IODESC_ENT(ADC),
   IODESC_ENT(WATCHDOG),
};
```

IODESC_ENT宏如下：

```
#define IODESC_ENT(x) { (unsigned long)S3C24XX_VA_##x, __phys_to_pfn(S3C24XX_PA_##x), S3C24XX_SZ_##x,
```

```
MT_DEVICE }
```
展开后等价于：
```
static struct map_desc s3c2410_iodesc[] __initdata = {
  {
    .virtual   =    (unsigned long)S3C24XX_VA_ LCD),
    .pfn       =    __phys_to_pfn(S3C24XX_PA_ LCD),
    .length    =   S3C24XX_SZ_ LCD,
    .type    =    MT_DEVICE
  },
  ……
};
```
S3C24XX_PA_ LCD和S3C24XX_VA_ LCD为定义在map.h内的LCD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_ LCD，pfn成员值通过__phys_to_pfn内核函数计算，只需要传递给它该I/O资源的物理地址就行。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_ LCD，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。
S3C24XX_VA_ LCD以及S3C24XX_PA_ LCD定义如下：
```
/* include/asm-arm/arch-s3c2410/map.h */
/* LCD controller */
#define S3C24XX_VA_LCD        S3C2410_ADDR(0x00600000)   //LCD映射后的虚拟地址
#define S3C2410_PA_LCD        (0x4D000000)    //LCD寄存器物理地址
#define S3C24XX_SZ_LCD        SZ_1M       //LCD寄存器大小
```
S3C2410_ADDR 定义如下：
```
#define S3C2410_ADDR(x)      ((void __iomem *)0xF0000000 + (x))
```
这里就是一种线性偏移关系，即s3c2410创建的I/O静态映射表会被映射到0xF0000000之后。(这个线性偏移值可以改，也可以你自己在virtual成员里手动定义一个值，只要不和其他IO资源映射地址冲突,但最好是在0XF0000000之后。)
(注：其实这里S3C2410_ADDR的线性偏移只是s3c2410平台的一种做法，很多其他ARM平台采用了通用的IO_ADDRESS宏来计算物理地址到虚拟地址之前的偏移。)
IO_ADDRESS宏定义如下：
```
/* include/asm/arch-versatile/hardware.h */
/* macro to get at IO space when running virtually */
#define IO_ADDRESS(x)         (((x) & 0x0fffffff) + (((x) >> 4) & 0x0f000000) + 0xf0000000) )
```
s3c2410_iodesc这个映射表建立成功后，我们在内核中便可以直接通过S3C24XX_VA_ LCD访问LCD的寄存器资源。
如：S3c2410 lcd驱动的probe函数内
```
/* Stop the video and unset ENVID if set */
info->regs.lcdcon1 &= ~S3C2410_LCDCON1_ENVID;
lcdcon1 = readl(S3C2410_LCDCON1); //read映射后的寄存器虚拟地址
writel(lcdcon1 & ~S3C2410_LCDCON1_ENVID, S3C2410_LCDCON1); //write映射后的虚拟地址
```
S3C2410_LCDCON1寄存器地址为相对于S3C24XX_VA_LCD偏移的一个地址，定义如下：
```
/* include/asm/arch-s3c2410/regs-lcd.h */
#define S3C2410_LCDREG(x) ((x) + S3C24XX_VA_LCD)
/* LCD control registers */
#define S3C2410_LCDCON1       S3C2410_LCDREG(0x00)
```
到此，我们知道了通过map_desc结构体创建I/O内存资源静态映射表的原理了。总结一下发现其实过程很简单，一通过定义map_desc结构体创建静态映射表，二在内核中通过创建映射后虚拟地址访问该IO资源。

二、I/O静态映射方式应用实例

I/O静态映射方式通常是用在寄存器资源的映射上，这样在编写内核代码或驱动时就不需要再进行ioremap，直接使用映射后的内核虚拟地址访问。同样的IO资源只需要在内核初始化过程中映射一次，以后就可以一直使用。

寄存器资源映射的例子上面讲原理时已经介绍得很清楚了，这里我举一个SRAM的实例介绍如何应用这种I/O静态映射方式。当然原理和操作过程同寄存器资源是一样的，可以把SRAM看成是大号的I/O寄存器资源。
比如我的板子在0x30000000位置有一块64KB大小的SRAM。我们现在需要通过静态映射的方式去访问该SRAM。我们要做的事内容包括修改kernel代码，添加SRAM资源相应的map_desc结构，创建sram到内核地址空间的静态映射表。写一个Sram Module,在Sram Module 内直接通过静态映射后的内核虚拟地址访问该sram。

第一步：创建SRAM静态映射表
在我板子的map_des结构体数组(xxx_io_desc）内添加SRAM资源相应的map_desc。如下：
```
static struct map_desc xxx_io_desc[] __initdata = {
  ……
  {
    .virtual   = IO_ADDRESS(XXX _UART2_BASE),
    .pfn       = __phys_to_pfn(XXX _UART2_BASE),
    .length      = SZ_4K,
    .type      = MT_DEVICE
  },{
    .virtual   = IO_ADDRESS(XXX_SRAM_BASE),
    .pfn       = __phys_to_pfn(XXX_SRAM_BASE),
    .length      = SZ_4K,
    .type      = MT_DEVICE
  },
};
```

宏XXX_SRAM_BASE为我板子上SRAM的物理地址,定义为0x30000000。我的kernel是通过IO_ADDRESS的方式计算内核虚拟地址的，这点和之前介绍的S3c2410有点不一样。不过原理都是相同的，为一个线性偏移, 范围在0xF0000000之后。

第二步：写个SRAM Module,在Module中通过映射后的虚拟地址直接访问该SRAM资源
SRAM Module代码如下：

```
/* Sram Testing Module */
……
static void sram_test(void)
{
   void * sram_p;
   char str[] = "Hello,sram!\n";

   sram_p = (void *)IO_ADDRESS (XXX_SRAM_BASE); /* 通过IO_ADDRESS宏得到SRAM映射后的虚拟地址 */
   memcpy(sram_p, str, sizeof(str));    //将 str字符数组拷贝到sram内
   printk(sram_p);
   printk("\n");
}
static int __init sram_init(void)
{
   struct resource * ret;

   printk("Request SRAM mem region .....\n");
   ret = request_mem_region(SRAM_BASE, SRAM_SIZE, "SRAM Region");

   if (ret ==NULL) {
      printk("Request SRAM mem region failed!\n");
      return -1;
   }

   sram_test();
   return 0;
}
static void __exit sram_exit(void)
{
   release_mem_region(SRAM_BASE, SRAM_SIZE);

   printk("Release SRAM mem region success!\n");
   printk("SRAM is closed\n");
}
module_init(sram_init);
module_exit(sram_exit);
```

在开发板上运行结果如下：

```
/ # insmod bin/sram.ko
Request SRAM mem region .....
Hello,sram!      &szlig; 这句即打印的SRAM内的字符串
/ # rmmod sram
Release SRAM mem region success!
SRAM is close
```

实验发现可以通过映射后的地址正常访问SRAM。
最后，这里举SRAM作为例子的还有一个原因是通过静态映射方式访问SRAM的话，我们可以预先知道SRAM映射后的内核虚拟地址（通过IOADDRESS计算）。这样的话就可以尝试在SRAM上做点文章。比如写个内存分配的MODULE管理SRAM或者其他方式，将一些critical的数据放在SRAM内运行，这样可以提高一些复杂程序的运行效率(SRAM速度比SDRAM快多了)，比如音视频的编解码过程中用到的较大的buffer等。