Linux 的驱动机制及实现

——by GeYingjun

1 Linux 驱动与模块

1.1 Linux 的模块

驱动程序为了与外设硬件进行交互，必须调用 Linux 内核提供的内存管理、文件系统等多种功能函数。而用户态的程序只能通过系统调用来使用有限的内核功能，因此驱动程序在 linux 系统中被安排在内核态运行，也即可以访问所有的内核功能。

但是为了把内核通用功能与这些驱动程序区分开来，Linux 通常要求驱动程序作为一个模块(module)在内核空间运行，并且由内核进行管理。当然，实际上模块并非只能用来编写驱动程序，模块可以作为任何一个独立的功能块，操作内核函数并通过特定接口向应用程序提供服务。

1.2 模块的编写要求

模块的代码中，必须加入以下头文件，才能正确地引用模块机制的若干函数：

```
#include <linux/module.h>
#include<linux/init.h>
```

当然，如果编写的模块是驱动程序，则还需要加入与驱动程序相关的内存管理、设备注册、中断管理等头文件。

模块与通常的应用程序不同，并没有一个 main 函数，而是一些相互作用的函数的集合。当然，这些集合中必须有一个首先调用的初始化函数，以及一个退出函数。

初始化函数：在模块加载时被调用，负责申请并初始化模块运行时必须的数据结构，此后这些结构有可能在模块存续期间一直存在。

退出函数：在模块卸载时被调用，卸载后模块内任何数据结构都不该继续存在，因此退出函数必须仔细的把模块运行带来的所有数据结构释放掉，否则将会造成内存泄露。

为了使内核知道模块中初始化函数和退出函数的函数名，必须用以下两个宏来定义：

```
module_init(s3c_ts_init);
module_exit(s3c_ts_exit);
```

上例中 s3c_ts_init 被定义为模块的初始化函数，s3c_ts_exit 被定义为模块的退出函数。该例取自 S3C6410 平台的触摸屏驱动。

1.3 模块的编译

模块可以包含在内核之中，也可以在内核之外。包含在内核之中，Linux 启动时会初始化模块（即调用模块的初始化函数）；在内核之外的模块，可以在内核启动之后，通过外部

命令 insmod 来将模块加载进内核，此时才调用模块初始化函数。两种方法对于模块的功能没有任何差别，差别主要在模块的初始化时机，以及内核本身的大小。

外部模块的编译通常用如下 makefile：

```
obj-m := module.o
module-objs := file1.o file2.o
make –C  ~/kernel-2.6  M='pwd' modules
```

例子中模块的名字为 module.o，由两个源文件(file1.c file2.c)生成，make 命令行指定了内核文件所在的位置，即内核源码目录。

以上方法编译的模块，通常用于开发阶段的调试或者 PC 平台的驱动程序。而在外设数量有限的嵌入式平台上，往往将驱动模块作为 Linux 内核内部模块编译。将内核与驱动生成的单一镜像向外发布。

内核内部（驱动）模块的编译：

1、 通过 make menuconfig 等工具对驱动模块进行配置，可选择为内部模块或者外部模块。配置工具根据配置选项，将生成一系列 CONFIG_变量，变量的值为 y（内部模块）或者 m（外部模块），或者 n（不编译）。例如将 S3C 平台的 touchscreen driver 配置为内部模块，将生成 CONFIG_TOUCHSCREEN_S3C 变量，值为 y。
2、 对应目录下的 Makefile 将根据这种变量，把对应的.o 文件加入$(obj-y)列表中。
obj-$(CONFIG_TOUCHSCREEN_S3C) += s3c-ts.o
3、 在最顶层的 Makefile 中，所有的$(obj-y)内的.o 文件都将被编译，并被链接成内核镜像。需要注意的是，除了驱动模块，$(obj-y)也会包括其他很多配置的内核自身功能。因此$(obj-y)并不区分要编译的.o 文件是内核功能还是驱动模块。

内部驱动模块是本文讨论的重点，此后所有的分析均基于内部驱动模块的机制。

1.4 内部驱动模块的加载

上文说明，编译到 Linux 内核内部的驱动模块，将在 Linux 启动时自动加载（调用初始化函数），本节将详细说明 Linux 何时加载这些驱动模块，以及各模块加载的顺序是如何确定的。

1.4.1 module_init 的定义

要说明何时被加载，就要从 module_init 的定义说起。在/include/linux/Init.h 中有如下声明：

```
#define __define_initcall(level,fn,id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" level ".init"))) = fn
/*
 * Early initcalls run before initializing SMP.
 *
 * Only for built-in code, not modules.
 */
```

```
#define early_initcall(fn)              __define_initcall("early",fn,early)
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)               __define_initcall("0",fn,0)
#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)             __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)        __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                 __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)             __define_initcall("6",fn,6)
#define device_initcall_sync(fn)        __define_initcall("6s",fn,6s)
#define late_initcall(fn)               __define_initcall("7",fn,7)
#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
#define __initcall(fn)  device_initcall(fn)
#define __exitcall(fn) \
        static exitcall_t __exitcall_##fn __exit_call = fn
#define module_init(x)  __initcall(x);
#define module_exit(x)  __exitcall(x);
```

这一段声明的前提条件是#ifndef MODULE，也就是说如果驱动没有定义为外部模块的情况下才有效。从声明中可以看出，module_init(x)最终被定义为__define_initcall("6",x,6)，该宏定义会将函数名 x 添加入.initcall" level ".init 段，也就是.initcall6.init 段。

在 Linux 的链接脚本 vmlinux.lds 中对上述段有如下布局：

```
__early_begin = .;
    *(.early_param.init)
  __early_end = .;
__initcall_start = .;
    *(.initcallearly.init) __early_initcall_end = .; *(.initcall0.init) *(.initcall0s.init) *(.initcall1.init) *(.initcall1s.init)
*(.initcall2.init) *(.initcall2s.init) *(.initcall3.init) *(.initcall3s.init) *(.initcall4.init) *(.initcall4s.init) *(.initcall5.init)
*(.initcall5s.init) *(.initcallrootfs.init) *(.initcall6.init) *(.initcall6s.init) *(.initcall7.init) *(.initcall7s.init)
  __initcall_end = .;
```

可见，所有 module_init()定义的函数名，都被放入了__initcall_start 与__initcall_end 之间的表中，level 越低，排在越靠前的位置。

1.4.2 Linux 初始化模块的过程

Linux 的初始化函数为 start_kernel()，函数内最后一个步骤调用 rest_init。内部模块就是在这个函数执行时被初始化的。调用路径为：

rest_init()---------->kernel_thread(kernel_init,...)---------->
do_basic_setup()---------->do_initcalls()

```
static void __init do_initcalls(void)
{
    initcall_t *call;
    for (call = __early_initcall_end; call < __initcall_end; call++)
        do_one_initcall(*call);
    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_work();
}
```

函数 do_initcalls()查找位于__early_initcall 和__initcall_end 之间的表内所有的函数，并用 do_one_initcall()来执行该函数。因此包括驱动在内的所有内部模块都会在这一步被加载，由于驱动使用 module_init(x)来定义，level 为 6，因此会有其他模块先于驱动模块被加载。而所有的驱动模块之间的先后顺序则是由 Makefile 将.o 加入$(obj-y)的顺序决定的。因此在多层次驱动模块加载的场合下，需要通过不同层次的 Makefile 来安排先后顺序。

2 Linux 系统中的设备与驱动

2.1 设备与驱动的关系

Linux 系统为了能操作和管理各种外设，将外设抽象成了设备(device)，而将对外设进行的各种操作逻辑包装进了驱动(driver)模型。

在 Linux 的设备模型中，所有的设备都被连接在某个总线(bus)上。总线的概念显然是从 X86 架构上继承而来的，X86 架构的 CPU 通过南桥芯片扩展出各种总线，外设必须插在某个总线上才能与 CPU 相连接。然而在基于 ARM 的嵌入式 CPU 上，通常把很多外设控制器都集成在 CPU 内，而且通常也不具备 PCI 等复杂总线控制器。为了使得嵌入式平台能够在形式上与 X86 相似，普遍采用一种成为“platform”的总线来表示集成到 CPU 平台内的设备。

设备与驱动，这两个概念对 CPU 操作外设都至关重要，但是设备本身在 Linux 内仅是划分出了操作该外设时所必须要保留的一些资源（如内存映射区域、IRQ 号等），同时指定了设备的名字。几乎所有的对外设操作的逻辑都在驱动模型内，而驱动与设备的关联则是通过二者所具有相同的名字实现的。

以下仍然以触摸屏设备和驱动说明二者关系：

S3C6410 平台触摸屏设备的定义：

/arch/arm/plat-s3c/Dev-ts.c 中

```
struct platform_device s3c_device_ts = {
    .name           = "s3c-ts",
```

```
        .id             = -1,
        .num_resources  = ARRAY_SIZE(s3c_ts_resource),
        .resource       = s3c_ts_resource,
};
static struct resource s3c_ts_resource[] = {
        [0] = {
                .start = S3C_PA_ADC,
                .end   = S3C_PA_ADC + SZ_4K - 1,
                .flags = IORESOURCE_MEM,
        },
        [1] = {
                .start = IRQ_PENDN,
                .end   = IRQ_PENDN,
                .flags = IORESOURCE_IRQ,
        },
        [2] = {
                .start = IRQ_ADC,
                .end   = IRQ_ADC,
                .flags = IORESOURCE_IRQ,
        }
};
```

触摸屏设备定义为 platform_device 类型的结构体，.name 为“s3c-ts”，它包括的设备资源为 s3c_ts_resource[]数组内定义的 3 个 resource 结构体。resource 结构体内的成员含义及作用将在后文设备注册中介绍。

S3C6410 平台触摸屏驱动的定义为：

/drivers/input/touchscreen/S3c-ts.c 中

```
static struct platform_driver s3c_ts_driver = {
        .probe          = s3c_ts_probe,
        .remove         = s3c_ts_remove,
        .suspend        = s3c_ts_suspend,
        .resume         = s3c_ts_resume,
        .driver         = {
            .owner      = THIS_MODULE,
            .name       = "s3c-ts",
        },
};
```

该结构体为 platform_driver 类型，其中的 driver 成员的.name 设定为“s3c-ts”，与上面的设备名字一致。

2.2 设备注册

设备包含了操作外设所需的资源，通过注册可以让 linux 内核对这些资源进行保留和初始化，也同时由内核创建一系列数据结构用来管理这个设备。

Linux 中对 platform 的设备使用 platform_device_register 进行注册，该函数具体分析如下：

```
int platform_device_register(struct platform_device *pdev)
{
    device_initialize(&pdev->dev);
    return platform_device_add(pdev);
}
```

首先调用 device_initialize()对 platform_device 中的 dev 成员进行初始化，该函数是 Linux 内部创建一系列内核数据的过程，对任何总线类型（不仅是 platform 总线）的设备注册都需要这一步，完成这一步之后 Linux 可以对所有总线类型的设备进行统一管理。

与总线类型相关的注册则是通过调用 platform_device_add()来完成。

platform_device_add()中除了基本数据结构的初始化，最重要的是以下两个函数调用：

```
insert_resource()
device_add(&pdev->dev)
```

insert_resource()将设备资源中 IORESOURCE_MEM 或 IORESOURCE_IO 类型的 resource 定义的范围进行保留，以避免多个设备共用相同的资源。

device_add()将该设备加入对应的总线之上，并且探测是否在该总线上已经注册了某个驱动程序能够驱动该设备。如果找到符合要求的驱动，则调用驱动程序的探测功能对设备进行初始化。对设备的注册操作涉及大量的 Linux 内部 kobject 等核心数据结构的操作，后文将会对这些核心数据结构进行详细分析。

设备注册之后，将会在文件系统的/dev 中对应的总线下(如/dev/platform/)出现该设备的描述文件以及一系列属性等。关于这些文件和属性是如何产生的也将在后文介绍 Linux 内部关于驱动的核心数据时详细说明。

2.3 驱动注册

Linux 中的设备需要注册以保留外设所需的资源以及创建与设备管理相关的内核数据结构。同样的，驱动程序也需要通过注册来让 Linux 系统对其进行管理。

Linux 内通过 platform_driver_register 函数注册 platform 设备的驱动程序。该函数

1、 将 platform driver.drive 的 bus 设定为 platform_bus_type

2、 将 platform driver 的.probe，.remove 等成员函数复制到.driver 成员内对应的指针

3、 然后调用 driver_register(&drv->driver);将 platform_drvier 的.driver 成员注册进 linux。

drvier_register()函数为该 driver 创建一系列用于管理的数据结构，并且查找 bus 上的所有已注册设备，如果有符合该驱动(名字相同)的设备，则调用 driver.probe 函数来完成一些设备相关的初始化工作（如 IRQ 申请等）。

2.4 设备与驱动的注册顺序

从上面两节可以看出，Linux 平等对待设备和驱动，添加设备时会查找是否有注册过的驱动可以使用；添加驱动时查找是否有符合的设备可以被初始化。但是作为 Linux 系统启动过程，必定是先注册其中一种，再注册另一种。下面的分析将可以看出，Linux 启动过程是先注册各种设备，而后再注册对应的各种驱动程序的。

注册设备：在平台相关的代码中，smdk6410_machine_init()函数内调用了 platform_add_devices(smdk6410_devices, ARRAY_SIZE(smdk6410_devices));
来注册 smdk6410_devices[]数组内包含的所有设备结构体。该数组定义为：

```
static struct platform_device *smdk6410_devices[] __initdata = {
#ifdef CONFIG_FB_S3C_V2
        &s3c_device_fb,
#endif
#ifdef CONFIG_SMDK6410_SD_CH0
        &s3c_device_hsmmc0,
#endif
#ifdef CONFIG_SMDK6410_SD_CH1
        &s3c_device_hsmmc1,
#endif
#ifdef CONFIG_SMDK6410_SD_CH2
        &s3c_device_hsmmc2,
#endif
        &s3c_device_wdt,
        &s3c_device_rtc,
        &s3c_device_i2c0,
//      &s3c_device_i2c1,
#if defined(CONFIG_SPI_CNTRLR_0)
        &s3c_device_spi0,
#endif
#if defined(CONFIG_SPI_CNTRLR_1)
        &s3c_device_spi1,
#endif
        &s3c_device_keypad,
        &s3c_device_ts,
        &s3c_device_smc911x,
        &s3c_device_dm9ks,
        &s3c_device_lcd,
        &s3c_device_vpp,
        &s3c_device_mfc,
        &s3c_device_tvenc,
        &s3c_device_tvscaler,
        &s3c_device_rotator,
        &s3c_device_jpeg,
        &s3c_device_nand,
```

```
        &s3c_device_onenand,
        &s3c_device_usb,
        &s3c_device_usbgadget,
        &s3c_device_usb_otghcd,
        &s3c_device_fimc0,
        &s3c_device_fimc1,
        &s3c_device_g2d,
        &s3c_device_g3d,
        &s3c_device_rp,
#ifdef CONFIG_S3C64XX_ADC
        &s3c_device_adc,
#endif
#ifdef CONFIG_HAVE_PWM
        &s3c_device_timer[0],
        &s3c_device_timer[1],
#endif
        &s3c_device_aes,
        &s3c_device_power_supply,
        &s3c_device_gpio_button,
#ifdef CONFIG_ANDROID_PMEM
            &android_pmem_device,
//            &android_pmem_adsp_device,
#endif
};
static struct i2c_board_info i2c_devs0[] __initdata = {
        { I2C_BOARD_INFO("24c08", 0x50), },
/*      { I2C_BOARD_INFO("WM8580", 0x1b), },        */
};
static struct i2c_board_info i2c_devs1[] __initdata = {
        { I2C_BOARD_INFO("24c128", 0x57), },  /* Samsung S524AD0XD1 */
        { I2C_BOARD_INFO("WM8580", 0x1b), },
};
static struct s3c_ts_mach_info s3c_ts_platform __initdata = {
        .delay                  = 10000,
        .presc                  = 49,
        .oversampling_shift     = 2,
        .resol_bit              = 12,
        .s3c_adc_con            = ADC_TYPE_2,
};
```

上文例中的触摸屏设备 s3c_device_ts 也在其中。

smdk6410_machine_init()在 setup_arch 时，被指定为平台相关的.init_machine 例程。该例程被调用的情况如下：

```
        static int __init customize_machine(void)
        {
```

```
        /* customizes platform devices, or adds new ones */
        if (init_machine)
                init_machine();
        return 0;
}
arch_initcall(customize_machine);
```

arch_initcall 的定义参见 1.4.1 节，#define arch_initcall(fn) __define_initcall("3",fn,3) 为 level=3 的 initcall 函数。

注册驱动：各设备的驱动一般都是由驱动模块的初始化函数完成，如触摸屏驱动的初始化函数：

```
static int __init s3c_ts_init(void)
{
        printk(banner);
        return platform_driver_register(&s3c_ts_driver);
}
```

根据 1.4.2 节的分析，注册驱动采用 level=6 的 initcall 机制，优先度低于 arch_initcall（level=3），因此 Linux 在启动过程的后期，先调用 smdk6410_machine_init()注册各个设备，然后在加载各驱动模块的时候再对驱动程序逐一注册。

3 Linux 驱动相关的内核模型

Linux 在运行期间，为了管理设备及驱动程序，建立了一系列数据结构（也可以称为对象,objects），这些数据结构相互构成一个多层次、相互关联的复杂体系。这些体系不但从逻辑上保持着内核对设备驱动的管理，同时也通过文件系统中一个特殊的子系统/sysfs 向用户空间展示体系的结构，并且提供用户程序访问的入口。

本节首先介绍/sysfs 文件子系统中的信息及含义，然后介绍隐藏在这些信息背后的内核模型以及驱动管理的内部机制。

3.1 Sysfs 文件系统

sysfs 文件系统是特殊文件系统之一，所谓特殊文件系统是指它不同于常规的基于磁盘或其他外存储器的文件系统，sysfs 文件系统中的文件、文件夹等都仅存在于内存中，是 linux 在运行时根据系统中注册的设备、驱动信息而动态建立起来的，而不是从某个外存储器中读取的。系统关闭后，该文件系统就不存在了，与之类似的还有 proc 文件系统等。

特殊文件系统采用了与通常文件系统完全一致的方式将信息呈现给用户空间，sysfs 目录内也有文件以及子目录，子目录内也同样可以有文件和子目录。

3.1.1 Sysfs 目录结构

Sysfs 的根目录通常是/sys，进入该目录，可以看到这一层内仅包含了若干子目录，根据平台的不同，子目录的数量会略有差别，但是通常都包括如下几个子目录：

block/　　所有的块设备，包括外存储器设备、ramdisk 等。

bus/　　内有各种总线的子目录，将各设备及驱动都按照总线类型存放在对应子目录内

class/　　按照功能将设备及驱动放在对应子目录内，如 input 设备、net 设备等。

devices/　　将所有的设备分类放在该目录的子目录内。

等。可以看出，同一个设备或者驱动的信息，将会出现在多个目录内，为了避免重复创建这些信息，使用了符号链接技术，该技术可以使得多个目录位置最终都指向同一个实际位置。

3.1.2 Sysfs 中的内容

从文件系统的角度来看，sysfs 内包括的是文件或者目录（除去符号链接），每个目录都被认为是一个入口，这个入口对应 Linux 驱动管理体系内一个对象，而一个文件通常对应一个与该对象相关联的“属性”。

用户程序可以打开目录，最终访问其中的文件，实际上也就是读/写内核对象关联的属性，Linux 正是通过这个途径使得用户程序与内核空间在驱动管理功能上进行交互。

3.2 Linux 设备模型的核心数据结构

Linux 管理设备和驱动，向用户空间提供 sysfs 访问入口，都是使用了 kobject, kset, kobj_type 等几个核心数据结构来实现的。

3.2.1 Kobject 结构体

kobject 结构是描述设备、驱动等对象的核心结构，该结构中的定义如下：

```
struct kobject {
        const char          *name;
        struct list_head    entry;
        struct kobject              *parent;
        struct kset         *kset;
        struct kobj_type    *ktype;
        struct sysfs_dirent     *sd;
        struct kref         kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
```

};

可以看到，该结构内没有任何涉及设备或者驱动程序的成员，因此为了描述一个有实际功能的设备或程序对象，必须要构造一个更大的结构体，结构体中包含 kobject。Kobject 结构的作用实际上仅是为了让内核可以将多个设备、驱动对象连接成层次复杂的体系，另一个功能则是对这些对象的引用进行计数管理。如果以面向对象的概念来理解这种关系，kobject 可以认为是基类，而设备、驱动等则是扩展了基类的子类。Kobject 的功能更像是设备、驱动对象中的链接点，这些链接点被 linux 内核用来构成了一个层次体系，而内核并不关心这些对象中的其他功能。

以下对 kobject 内的几个重要成员进行说明：

name：一个描述 kobject 的字符串指针，实际上无论 kobject 扩展成什么样的结构，扩展后的机构体对象的名字，通常最终也是设定在其包含的 kobject 的 name 成员中。这个字符串也通常就是我们在 sysfs 中看到的设备或驱动的名字。

parent：向上层 kobject 的指针，以此构成层次体系。由于系统内使用的对象都是扩展了 kobject 之后的结构，因此 parent 所指向的 kobject 通常也是包含在另一个结构之中的。

kset：指向一个 kset 结构的指针，kset 结构将在下文中介绍。

ktype：指向与该 kobject 关联的 kobj_type 的指针，kobj_type 也将在下文中介绍。

sd：指向与该 kobject 关联的 sysfs 中的一个目录。

3.2.2 kset 结构体

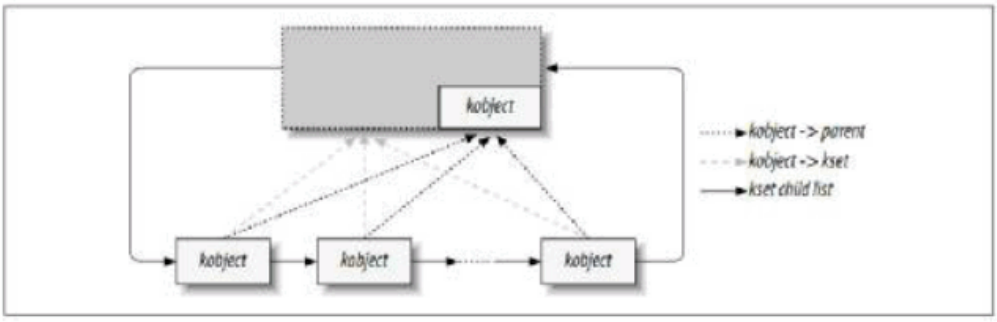
kset 象是同一类型的 kobject 的集合。当然 kobject 都是一样的，所谓的同一类型，是指扩展了 kobject 之后的结构属于同一类型。比如系统内注册了鼠标、键盘两种设备，都属于输入类型的设备，那就会有一个对应输入设备类型的 kset，里面集合了鼠标和键盘的设备结构中嵌入的 kobject。

kset 的定义如下：

```
struct kset {
struct list_head list;
spinlock_t list_lock;
struct kobject kobj;
struct kset_uevent_ops *uevent_ops;
};
```

可以看出，kset 本身内部也包含自己的一个 kobject，因此 kset 的名字最终也就是它包含的 kobject 的 name 成员。list 则是一个链表，将 kset 包含的 kobject 集合链接起来，kset 中还有一个 uevent_ops 指针，用于指向处理设备热插拔时的事件处理函数。提到热插拔，可以想象到任何一个总线驱动结构，必定包含 kset 结构体在内。如果总线支持热插拔的话，就需要指定 kset 中的 uevent_ops 指针，而实际情况也正是这样。

下面这张图比较明确的展示了常见的 kobject 和 kset 的相互关系：



如图，上面一个 kset 中包含了一个自己 kobject，kset 通过链表将若干个 kobject 链接起来，这些链接的 kobject 则把 parent 指向 kset 内的 kobject，而把 kset 指针指向链接他们的那个 kset 结构体。

需要再次强调，实际系统中不会具有单纯的 kobject，kset 等结构，必定是嵌入到其他更高级的结构内。这些更高级的结构通过 kobject,kset 相互连接起来。

3.2.3 kobj_type 结构体

kobject 结构体包含一个指向 kobj_type 结构的指针，kobj_type 负责与之相关联的 kobject 的一些操作。其定义如下：

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

release 操作的对象是指向该 kobj_type 的 kobject 结构，当对 kobject 的引用减为 0（初始化的 kobject 引用为 1）时，将会调用 release 方法释放 kobject。为什么不将 release 方法放入 kobject 内部呢？这是为了对同一类型的 kobject 对象只需要定义一个 kobj_type 来指定同一种 release 方法；同时如果 kobject 属于某个 kset，那么 kobject 可以将自己的 kobj_type 设为 NULL，转而使用 kset（自己的 kobject）的 kobj_type。这样一来，使得 kset 可以在很大程度上对集合内所有的 kobject 进行统一管理。

3.3 Linux 中的设备、驱动相关结构

上一节中介绍了 Linux 内核中的几个核心数据结构，这些数据结构主要用于 Linux 内部管理，而并不涉及具体的设备、驱动相关的特性和细节。本节将介绍 Linux 中用于描述实际设备或者驱动的具体数据结构，可以看到这些结构是如何将上节介绍的 kobject，kset，kobj_type 等结构嵌入在自身中的。

3.3.1 device 结构体

device 结构体是 linux 中描述一个设备的基本数据类型，同样的，这个结构体也并不能完全描述各种各样的设备，一个真实设备的描述结构常常是扩展了 device 结构体之后得到的，但是有必要首先了解 device 结构体的定义：

```
struct device {
        struct klist            klist_children;
        struct klist_node       knode_parent;   /* node in sibling list */
        struct klist_node       knode_driver;
        struct klist_node       knode_bus;
        struct device           *parent;
        struct kobject kobj;
        char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
        const char              *init_name; /* initial name of the device */
        struct device_type      *type;
        unsigned                uevent_suppress:1;
        struct semaphore        sem;    /* semaphore to synchronize calls to
                                         * its driver.
                                         */
        struct bus_type *bus;           /* type of bus device is on */
        struct device_driver *driver;   /* which driver has allocated this
                                           device */
        void            *driver_data;   /* data private to the driver */
        void            *platform_data; /* Platform specific data, device
                                           core doesn't touch it */
        struct dev_pm_info      power;
        u64             *dma_mask;      /* dma mask (if dma'able device) */
        u64             coherent_dma_mask;/* Like dma_mask, but for
                                             alloc_coherent mappings as
                                             not all hardware supports
                                             64 bit addresses for consistent
                                             allocations such descriptors. */
        struct device_dma_parameters *dma_parms;
        struct list_head        dma_pools;      /* dma pools (if dma'ble) */
        struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                                             override */
        /* arch specific additions */
        struct dev_archdata     archdata;
        spinlock_t              devres_lock;
        struct list_head        devres_head;
        struct klist_node       knode_class;
        struct class            *class;
        dev_t                   devt;   /* dev_t, creates the sysfs "dev" */
        struct attribute_group **groups;        /* optional groups */
```

```
    void  (*release)(struct device *dev);
};
```

kobj：结构体内包含的 kobject 结构体变量

driver：指向驱动本设备的 device_driver 结构体的指针

class：指向该设备从属的一类，如鼠标设备指向 input 类

还有很多其他成员在此不一一说明，在后文分析设备注册过程的时候，将会看到这些成员发挥的作用。

一个完整的描述设备的结构体，例如 platform_device 包含了上述的 device 结构体，并加上了一些必须的数据成员。定义如下：

```
struct platform_device {
    const char  * name;
    int         id;
    struct device     dev;
    u32         num_resources;
    struct resource   * resource;
};
```

3.3.2 device_driver 结构体

与 device 结构类似，device_driver 结构是一个驱动设备的基本结构，完整的驱动描述通常是扩展该结构而得到的。device_driver 结构体的定义如下：

```
struct device_driver {
    const char        *name;
    struct bus_type         *bus;
    struct module           *owner;
    const char              *mod_name;     /* used for built-in modules */
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    struct attribute_group **groups;
    struct pm_ops *pm;
    struct driver_private *p;
};
```

成员中似乎看不到 kobject 类型的变量，但是 driver_private 的定义内有 kobject：

```
struct driver_private {
    struct kobject kobj;
    struct klist klist_devices;
    struct klist_node knode_bus;
    struct module_kobject *mkobj;
    struct device_driver *driver;
};
```

与 device 用来管理设备的信息不同，device_driver 更多的功能是负责对设备的操作，因

此结构体内包含了很多函数指针。同样的，描述一个具体的驱动结构，需要在 device_driver 的基础上增加一些数据成员以及操作指针成员。如 platform_driver 定义如下：

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*suspend_late)(struct platform_device *, pm_message_t state);
        int (*resume_early)(struct platform_device *);
        int (*resume)(struct platform_device *);
        struct pm_ext_ops *pm;
        struct device_driver driver;
};
```

可以看到，platform_driver 中包含的函数指针与 device_driver 中的完全一致。实际上，linux 在管理时并不去调用 platform_driver 中的那些函数，而是仅使用 device_driver 结构内的函数。platform_driver 内的那些函数在驱动初始化时进行赋值，驱动注册后，在需要的情况下，linux 将会通过 devicer_driver 找到上层的 platform_driver 中的这些函数并进行调用。

3.3.3 bus_type 结构体

除了普通设备，Linux 定义了总线的概念。从本质上来说总线也是一种设备，也需要驱动程序，但是对于总线的处理，linux 引入了一种特殊的结构 bus_type。可能有多个设备连接在同一个总线上，这种关系类似于多个 kobject 属于一个 kset，而 bus_type 正是扩展了 kset 来管理多个 kobject，从而管理相应的设备和驱动的。

bus_type 的定义如下：

```
struct bus_type {
        const char          *name;
        struct bus_attribute    *bus_attrs;
        struct device_attribute*dev_attrs;
        struct driver_attribute *drv_attrs;
        int (*match)(struct device *dev, struct device_driver *drv);
        int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);
        int (*suspend)(struct device *dev, pm_message_t state);
        int (*suspend_late)(struct device *dev, pm_message_t state);
        int (*resume_early)(struct device *dev);
        int (*resume)(struct device *dev);
        struct pm_ext_ops *pm;
        struct bus_type_private *p;
};
```

其中的 bus_type_private 结构的定义为：

```
struct bus_type_private {
```

```
    struct kset subsys;
    struct kset *drivers_kset;
    struct kset *devices_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;
};
```

其中

subsys 被称为子系统，其通常显示于 sysfs 的顶层目录内。

bus_notifier 是总线发生改变，如总线加入、卸载时可能会调用到的例程，用来通知内核进行必要的操作。

drivers_autoprobe 用来指示该总线是否支持自动探测功能，默认 1 表示支持，0 表示不支持。

3.4 结构体注册过程

上文中已经介绍了 linux 管理设备、驱动所用到的核心数据结构，以及描述具体设备和驱动的数据结构。这些结构体都是通过注册(register)操作将自身加入到 linux 内核的管理逻辑中的。也就是说，通过注册，Linux 内核会把这些结构体连接到内部的复杂体系中，并且在用户程序需要时，在这个体系中查找到对应的结构和操作例程进行服务。

3.4.1 注册 device

设备连接在总线上，因此注册设备之前通常需要注册好对应的总线(即便只是 platform 总线)，而总线也是一种特殊设备，因此注册总线的过程也是从注册总线对应的设备开始的。所以首先来分析注册设备(device)。

1、 单纯 device 结构的注册

无论多么复杂的设备结构体，最终都可以通过调用 device_register()来注册，将其内部的 device 结构连接到 linux 管理体系中。

```
int device_register(struct device *dev)
{
  device_initialize(dev);
  return device_add(dev);
  }
```

该函数执行了两个步骤，首先初始化 device，然后加入到合适的链表中。下面具体分析这两个函数。

```
void device_initialize(struct device *dev)
    {
      dev->kobj.kset = devices_kset;
      kobject_init(&dev->kobj, &device_ktype);
```

```
        klist_init(&dev->klist_children, klist_children_get,
                   klist_children_put);
        INIT_LIST_HEAD(&dev->dma_pools);
        init_MUTEX(&dev->sem);
        spin_lock_init(&dev->devres_lock);
        INIT_LIST_HEAD(&dev->devres_head);
        device_init_wakeup(dev, 0);
        device_pm_init(dev);
        set_dev_node(dev, -1);
    }
```

1) 将 device 内的 kobject 中的 kset 设定为全局的 device_kset，这个变量在启动时初始化，对应 sysfs 内的 devices 目录。
2) 初始化 device 内的 kobject，并把 kobject 的 ktype 设定为全局的 device_ktype
3) 初始化 device 内的一些链表和互斥访问控制变量
4) 如果需要，设定电源管理方面的变量
5) 如果在非连续内存的平台上，设定相关的 node=-1。

这一步主要是进行了一些初始化设定，并没有将 kobject 连接到 linux 内部。

```
int device_add(struct device *dev)
{
    struct device *parent = NULL;
    struct class_interface *class_intf;
    int error = -EINVAL;
    dev = get_device(dev);                          1)
    if (!dev)
        goto done;
    /* Temporarily support init_name if it is set.
     * It will override bus_id for now */
    if (dev->init_name)
        dev_set_name(dev, "%s", dev->init_name);    2)
    if (!strlen(dev->bus_id))
        goto done;
    pr_debug("device: '%s': %s\n", dev->bus_id, __func__);
    parent = get_device(dev->parent);               3)
    setup_parent(dev, parent);
    /* use parent numa_node */
    if (parent)
        set_dev_node(dev, dev_to_node(parent));
    /* first, register with generic layer. */
    error = kobject_add(&dev->kobj, dev->kobj.parent, "%s", dev->bus_id); 4)
    if (error)
        goto Error;
    /* notify platform of device entry */
    if (platform_notify)
```

```
        platform_notify(dev);
/* notify clients of device entry (new way) */
if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                     BUS_NOTIFY_ADD_DEVICE, dev);          5)
error = device_create_file(dev, &uevent_attr);                             6)
if (error)
        goto attrError;
if (MAJOR(dev->devt)) {                                                    7)
        error = device_create_file(dev, &devt_attr);
        if (error)
                goto ueventattrError;
        error = device_create_sys_dev_entry(dev);
        if (error)
                goto devtattrError;
}
error = device_add_class_symlinks(dev);                                    8)
if (error)
        goto SymlinkError;
error = device_add_attrs(dev);
if (error)
        goto AttrsError;
error = bus_add_device(dev);
if (error)
        goto BusError;
error = dpm_sysfs_add(dev);
if (error)
        goto DPMError;
device_pm_add(dev);
kobject_uevent(&dev->kobj, KOBJ_ADD);                                      9)
bus_attach_device(dev);                                                    10)
if (parent)
        klist_add_tail(&dev->knode_parent, &parent->klist_children);
if (dev->class) {
        mutex_lock(&dev->class->p->class_mutex);
        /* tie the class to the device */
        klist_add_tail(&dev->knode_class,
                       &dev->class->p->class_devices);
        /* notify any interfaces that the device is here */
        list_for_each_entry(class_intf,
                            &dev->class->p->class_interfaces, node)
                if (class_intf->add_dev)
                        class_intf->add_dev(dev, class_intf);
        mutex_unlock(&dev->class->p->class_mutex);
```

```
        }
done:
        put_device(dev);                                        11)
        return error;
 DPMError:
        bus_remove_device(dev);
 BusError:
        if (dev->bus)
                blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                             BUS_NOTIFY_DEL_DEVICE, dev);
        device_remove_attrs(dev);
 AttrsError:
        device_remove_class_symlinks(dev);
 SymlinkError:
        if (MAJOR(dev->devt))
                device_remove_sys_dev_entry(dev);
 devtattrError:
        if (MAJOR(dev->devt))
                device_remove_file(dev, &devt_attr);
 ueventattrError:
        device_remove_file(dev, &uevent_attr);
 attrError:
        kobject_uevent(&dev->kobj, KOBJ_REMOVE);
        kobject_del(&dev->kobj);
 Error:
        cleanup_device_parent(dev);
        if (parent)
                put_device(parent);
        goto done;
}
```

下面对重要的步骤进行分析：

1) 增加一次对 device 内 kobject 的引用
2) 如果 device 内指定了 init_name，则将其复制到 device->bus_id[]字符数组内
3) 增加对 device->parent 的 kobject 的引用，然后根据 device 是否属于某个 class，或者其 parent 是否属于某个 class，找到合适的 parent 指定为 device 内 kobject 的 parent。
4) 将 device 的 bus_id[]内保存的名字复制到 kobject 的 name 中，并且将 kobject 添加到 linux 管理体系中。具体说来，如果 kobject 设定了 kset，则将其 parent 设定为 kset，并将 kobject 加入到这个 kset 的链表中。同时，回在 sysfs 中 kset 或者 parent 的目录下建立一个 kobject 对应的目录，如果 kset 和 parent 都为 NULL，就在 sysfs 根目录下建立一个 kobject 对应的目录（也叫入口）。
5) 如果 device->bus 定义了，则会调用通知例程，告诉内核该 bus 上增加了一个新设备。
6) 在设备目录下建立由 uevent 属性文件，用以支持热插拔。
7) 如果定义了主设备号，则在设备目录下建立设备号属性文件；并且根据设备所属的 class，在对应 class 目录下建立符号链接，链接到该设备号属性文件。

8) 根据 device 所属的 bus、class 在设备目录下建立 bus 和 class 要求的属性文件。
9) 通知内核加入了这个 kobject，如果有必要内核会调用相关的处理例程通知用户空间。
10) 将设备加入到 bus 上，如果设定了 autoprobe，则自动探测是否有合适的驱动可以匹配设备。
11) 减少一次对 device 内 kobject 的引用，完成注册

2、 platform_device 的注册

platform_device 的注册与一般 device 的注册类似，只是在注册过程中，对 bus 指定为 platform_bus_type。

```
int platform_device_register(struct platform_device *pdev)
{
    device_initialize(&pdev->dev);
    return platform_device_add(pdev);
}
```

第一步初始化与 device_register 完全一致，第二步 platform_device_add 函数如下：

```
int platform_device_add(struct platform_device *pdev)
{
    int i, ret = 0;
    if (!pdev)
        return -EINVAL;
    if (!pdev->dev.parent)
        pdev->dev.parent = &platform_bus;
    pdev->dev.bus = &platform_bus_type;
    if (pdev->id != -1)
        snprintf(pdev->dev.bus_id, BUS_ID_SIZE, "%s.%d", pdev->name,
             pdev->id);
    else
        strlcpy(pdev->dev.bus_id, pdev->name, BUS_ID_SIZE);
    for (i = 0; i < pdev->num_resources; i++) {
        struct resource *p, *r = &pdev->resource[i];
        if (r->name == NULL)
            r->name = pdev->dev.bus_id;
        p = r->parent;
        if (!p) {
            if (resource_type(r) == IORESOURCE_MEM)
                p = &iomem_resource;
            else if (resource_type(r) == IORESOURCE_IO)
                p = &ioport_resource;
        }
        if (p && insert_resource(p, r)) {
            printk(KERN_ERR
                "%s: failed to claim resource %d\n",
                pdev->dev.bus_id, i);
            ret = -EBUSY;
```

```
                goto failed;
            }
        }
        pr_debug("Registering platform device '%s'. Parent at %s\n",
             pdev->dev.bus_id, pdev->dev.parent->bus_id);
        ret = device_add(&pdev->dev);
        if (ret == 0)
            return ret;
 failed:
        while (--i >= 0) {
            struct resource *r = &pdev->resource[i];
            unsigned long type = resource_type(r);
            if (type == IORESOURCE_MEM || type == IORESOURCE_IO)
                release_resource(r);
        }
        return ret;
}
```

可见，最终仍然是调用了 device_add，而在之前则是指定了 platform_device 内部 device 的 parent 和 bus，另外就是调用 insert_resource 为设备设置了所需要的内存映射等。

3.4.2 注册 device_driver

本节讨论如何将驱动程序注册进内核的管理体系。

1、 单纯 device_driver 的注册

通过调用 driver_register()函数来完成 device_driver 的注册工作，定义如下：

```
int driver_register(struct device_driver *drv)
{
        int ret;
        struct device_driver *other;
        if ((drv->bus->probe && drv->probe) ||
            (drv->bus->remove && drv->remove) ||
            (drv->bus->shutdown && drv->shutdown))
             printk(KERN_WARNING "Driver '%s' needs updating - please use "
                    "bus_type methods\n", drv->name);
        other = driver_find(drv->name, drv->bus);
        if (other) {
            put_driver(other);
            printk(KERN_ERR "Error: Driver '%s' is already registered, "
                    "aborting...\n", drv->name);
            return -EEXIST;
        }
        ret = bus_add_driver(drv);
        if (ret)
```

```
        return ret;
    ret = driver_add_groups(drv, drv->groups);
    if (ret)
        bus_remove_driver(drv);
    return ret;
}
```

1) 如果 driver 本身以及 driver->bus 都拥有 probe、remove、shutdown 等操作函数，则打印出警告信息，要求使用 bus 的操作函数来代替 driver 本身的操作。
2) 搜索 driver->bus 上对应的 kset，看是否已经有 kobject 的 name 与 driver 同名，如果有的话，表明已经注册过，不能再次注册 device_driver。
3) 调用 bus_add_driver 函数，新增 driver 对应的 kobject 并添加进 bus 对应的 kset 内，然后对 bus 上所有注册的 device 进行探测(match)，如果有 device 与该 driver 匹配，则调用 driver 的 probe 函数。另外也会在 sysfs 对应目录下建立该 driver 对应的目录以及相关的属性文件。
4) 为新注册的 device_driver 增加 sysfs 内的属性组。

2、platform_driver_register

上文介绍了通用数据结构 device_driver 注册的过程，系统中实际使用的驱动结构通常是扩展该结构而来，在调用 driver_register()之前还需要进行必要的初始化。例如：

```
int platform_driver_register(struct platform_driver *drv)
{
    drv->driver.bus = &platform_bus_type;
    if (drv->probe)
        drv->driver.probe = platform_drv_probe;
    if (drv->remove)
        drv->driver.remove = platform_drv_remove;
    if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;
    if (drv->suspend)
        drv->driver.suspend = platform_drv_suspend;
    if (drv->resume)
        drv->driver.resume = platform_drv_resume;
    if (drv->pm)
        drv->driver.pm = &drv->pm->base;
    return driver_register(&drv->driver);
}
```

首先将预定义的 platform_bus_type 设定为内部 driver 的 bus，然后设定内部 driver 的各种操作函数。深入查看这些操作函数，实际上都是直接调用了 driver 的上层结构，也就是 platform_driver 的对应操作函数。

3.4.3 注册 bus_type

上两节中都提到，设备注册时会探测驱动；驱动注册时会探测设备，这种探测行为一般

都是有总线完成的。具体说来，都是调用了注册设备/驱动所属总线的 match 函数。由此可知，在注册设备或者驱动之前，应该先完成总线的注册工作。总线注册的函数如下：

```
int bus_register(struct bus_type * bus)
{
	int retval;
	struct bus_type_private *priv;
	priv = kzalloc(sizeof(struct bus_type_private), GFP_KERNEL);
	if (!priv)
		return -ENOMEM;
	priv->bus = bus;
	bus->p = priv;
	BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);
	retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
	if (retval)
		goto out;
	priv->subsys.kobj.kset = bus_kset;
	priv->subsys.kobj.ktype = &bus_ktype;
	priv->drivers_autoprobe = 1;
	retval = kset_register(&priv->subsys);
	if (retval)
		goto out;
	retval = bus_create_file(bus, &bus_attr_uevent);
	if (retval)
		goto bus_uevent_fail;
	priv->devices_kset = kset_create_and_add("devices", NULL,
						 &priv->subsys.kobj);
	if (!priv->devices_kset) {
		retval = -ENOMEM;
		goto bus_devices_fail;
	}
	priv->drivers_kset = kset_create_and_add("drivers", NULL,
						 &priv->subsys.kobj);
	if (!priv->drivers_kset) {
		retval = -ENOMEM;
		goto bus_drivers_fail;
	}
	klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
	klist_init(&priv->klist_drivers, NULL, NULL);
	retval = add_probe_files(bus);
	if (retval)
		goto bus_probe_files_fail;
	retval = bus_add_attrs(bus);
	if (retval)
		goto bus_attrs_fail;
```

```
        pr_debug("bus: '%s': registered\n", bus->name);
        return 0;
bus_attrs_fail:
        remove_probe_files(bus);
bus_probe_files_fail:
        kset_unregister(bus->p->drivers_kset);
bus_drivers_fail:
        kset_unregister(bus->p->devices_kset);
bus_devices_fail:
        bus_remove_file(bus, &bus_attr_uevent);
bus_uevent_fail:
        kset_unregister(&bus->p->subsys);
        kfree(bus->p);
out:
        return retval;
}
```

总线注册过程看似复杂，实际上仅是完成了如下流程：

1) 为 bus 初始化一个 bus_type_private 结构。
2) 设置这个 private 结构内相关的 kobject 的名字
3) 将 private 的 kobject 设定到全局的 bus_kset 中，这样就可以在/sys/bus/中看到该总线的信息了。另外在总线的目录中创建属性文件。
4) 针对 private 创建两个 kset，一个是“devices”，一个是“drivers”，连接到 private 的两个 kset 指针上（devices_kset 和 drivers_kset）。这样也就能在 bus 的目录内看到两个目录，分别是 devices 目录和 drivers 目录。正如上文所述，注册设备或者驱动时，会把 kobject 加入到这两个 kset 中的一个，并把设备或驱动的目录显示在这两个目录中。
5) 如果定义了支持热插拔，会在该 bus 目录下创建热插拔属性文件。
6) 创建 bus 的其它属性文件。

3.5 与设备、驱动相关的 Linux 启动流程

通过前文可知，系统注册的第一个设备应该是总线设备，然后注册总线，之后就可以在该总线上先后注册设备和驱动。本节就具体说明 Linux 中哪些函数实现了这个流程。

参见前文 1.4.2 节模块初始化过程中，会调用 do_basic_setup()，该函数实际上完成了内核包含的整个设备/驱动管理体系的建立过程。

```
static void __init do_basic_setup(void)
{
        rcu_init_sched(); /* needed by module_init stage. */
        init_workqueues();
        usermodehelper_init();
        driver_init();
        init_irq_proc();
        do_initcalls();
}
```

红色标示的函数即建立管理体系。从 1.4.2 内容可知，do_initcalls()中先后把所有平台上的设备和驱动都注册进入管理体系，本节重点说明 driver_init 如何初始化该体系的。

```
void __init driver_init(void)
{
	/* These are the core pieces */
	devices_init();
	buses_init();
	classes_init();
	firmware_init();
	hypervisor_init();
	/* These are also core pieces, but must come after the
	 * core core pieces.
	 */
	platform_bus_init();
	system_bus_init();
	cpu_dev_init();
	memory_dev_init();
}
```

3.5.1 devices_init()

```
int __init devices_init(void)
{
	devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);
	if (!devices_kset)
		return -ENOMEM;
	dev_kobj = kobject_create_and_add("dev", NULL);
	if (!dev_kobj)
		goto dev_kobj_err;
	sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);
	if (!sysfs_dev_block_kobj)
		goto block_kobj_err;
	sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);
	if (!sysfs_dev_char_kobj)
		goto char_kobj_err;
	return 0;
 char_kobj_err:
	kobject_put(sysfs_dev_block_kobj);
 block_kobj_err:
	kobject_put(dev_kobj);
 dev_kobj_err:
	kset_unregister(devices_kset);
	return -ENOMEM;
}
```

代码中红色标示的变量都是全局变量，有 kset 类型的，也有 kobject 类型的。
devices_kset 建立一个顶层 kset，并且在 sysfs 根目录下建立对应的目录（devices 目录）。
dev_kobj 建立一个 kobject，因为没有 parenet 和 kset，也在 sysfs 根目录内建立一个“dev”目录。
sysfs_dev_block_kobj 和 sysfs_dev_char_kobj 分别在 dev_kobj 下层建立两个 kobject，parent 为 dev_kobj，因此在/sys/dev/目录下会建立“block”，“char”两个目录。

3.5.2 bus_init()

该函数与 devices_kset 类似，建立了一个顶层的 kset（bus_kset），并在/sys 根目录下建立“bus”目录。

3.5.3 classes_init()

该函数与 devices_kset 类似，建立了一个顶层的 kset(class_kset)，并在/sys 根目录下建立“class”目录。
firmware_init()，hypervisor_init()两个函数执行的功能也完全类似。

3.5.4 platform_bus_init()

该函数注册 platform_bus。

```
int __init platform_bus_init(void)
{
        int error;
        error = device_register(&platform_bus);
        if (error)
                return error;
        error =  bus_register(&platform_bus_type);
        if (error)
                device_unregister(&platform_bus);
        return error;
}
```

由于前面已经注册好了顶层的 devices_kset、bus_kset 等，因此该函数将在对应的 kset 下依次注册总线设备、注册总线。关于 platform_bus 的注册前文中已有详细说明。

3.5.5 system_bus_init()

在 devices_kset 下创建并注册一个新的 system_kset。

3.5.6 cpu_dev_init()

在上面建立的 system_kset 之下创建 cpu_kset，并且创建 cpu 的属性文件。
memory_dev_init()如果定义了支持内存热插拔，则会在 system_kset 下建立 memory_kset，并建立相关属性文件。

3.5.7 小结

从上面的流程可以看出，管理体系的建立过程，实际上是从建立若干个顶层的 kset、kobject 开始，而后在这些顶层结构之下创建新的 kset 或者 kobject。创建这些 kset、kobject 的同时，会在 sysfs 内对应的位置创建目录或者文件。这样可以进一步证实，sysfs 并非存储在外部存储器上，而是有内核在内存中创建并管理的。

上述的 driver_init()各个步骤完成后，就已经创建好了顶层的各种 kset，也已经注册了 platform_bus 等几个下层 kset 和 kobject。之后的 do_initcalls()就可以在这个现有的基础上继续加入 kobject 或者 kset，并且在对应目录下创建相应的子目录或者文件。

4 Linux 内几种驱动模型

上文中经常提到的 platform 总线上的驱动，通常是与 Linux 本身有着密切关系，是系统正常工作必须的设备驱动，比如用作输入的触摸屏、键盘，以及用于显示的 LCD、2D 加速器等。这些驱动通常由系统内部的功能使用；应用程序一般不直接使用这些设备，而是通过使用内核提供的功能接口间接使用这些设备。

然而，也有一些设备，它们与内核的功能联系并不紧密，更多的是为了服务于应用程序。应用程序使用这些设备时，就像对待一个文件那样进行操作。为此，linux 中提供了几种便于使用的驱动模型，本章将进行介绍。

4.1 字符设备

字符设备驱动程序适合于大多数简单的硬件设备，而且结构简单，易于理解。在开发外设驱动时，开发人员比较容易就能将设备抽象成一个字符设备模型。

4.1.1 字符设备结构体

为了将一个设备作为字符设备注册进入 linux 管理体系，必须借助字符设备结构体，其定义为：

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
```

```
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

结构体内包含了必须的 kobject 成员，一个设备号 dev_t(包含了主设备号与次设备号)，以及非常重要的 file_operations 指针。

每个注册的字符设备都会在/sys/dev/char/目录内生成一个符号链接，指向/sys/devices/内对应的一个设备。而符号链接的名字就是 dev_t 对应的主设备号与次设备号的组合。通常一个物理设备都会使用不同于其他物理设备的主设备号，而针对同一个物理设备可以虚拟出多个设备，这些设备使用不同的次设备号加以区分，但是这并不是强制要求。

应用程序访问字符设备，可以直接访问/sys/下对应的代表设备的文件，而应用程序最终使用的是与文件操作有关的系统调用，因此可以理解字符设备里 file_operations 的必要性了。

4.1.2 设备号

设备号在 linux 内用 dev_t 类型来保存，它是一个 32 位数，其中 12 位表示主设备号，20 位表示次设备号。主、次设备号的关系已经在上节说明。要获得 dev_t 中的主、次设备号，必须用下列宏：

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
```

要将主、次设备号生成 dev_t 类型，则使用：

```
MKDEV(int major, int minor)
```

并非每种设备模型内都包含设备号，但是为了管理使用同一种模型的多个设备，往往使用设备号来区分。比如使用字符设备模型的所有设备，都会使用设备号来标示。

字符设备注册时，通常动态的申请设备号，而很少在代码中指定设备号。

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

dev 是返回参数，指向分配后的设备号。baseminor 是起始的次设备号，count 是分配的次设备号个数，name 是分配的设备号对应的名字。

4.1.3 字符设备的注册

上节从系统中获得了分配的设备号，之后就要利用这个设备号把设备注册进 Linux 管理体系。注册分为两步，初始化和添加设备，使用如下两个函数：

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
    memset(cdev, 0, sizeof *cdev);
    INIT_LIST_HEAD(&cdev->list);
    kobject_init(&cdev->kobj, &ktype_cdev_default);
    cdev->ops = fops;
}

int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

```
{
    p->dev = dev;
    p->count = count;
    return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

初始化函数把输入的 file_operations 传递给字符设备结构，file_operations 定义如下：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned
long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
};
```

该结构是由文件系统中对文件的所有操作函数指针组成的，正如前文所述，字符设备的使用形式与文件一致，所以一般字符设备的 file_operations 里面至少要实现 open、release、read、write 等几个基本操作。而功能较为复杂的字符设备，通常还需要实现 ioctl 等操作。

4.2 块设备驱动程序

块设备是一种比较复杂的设备模型，设计这种模型的重要目的之一就是用于管理磁盘或者类似于磁盘的设备。这类设备的共同特点是读、写操作通常都是连续的一段数据，而且这

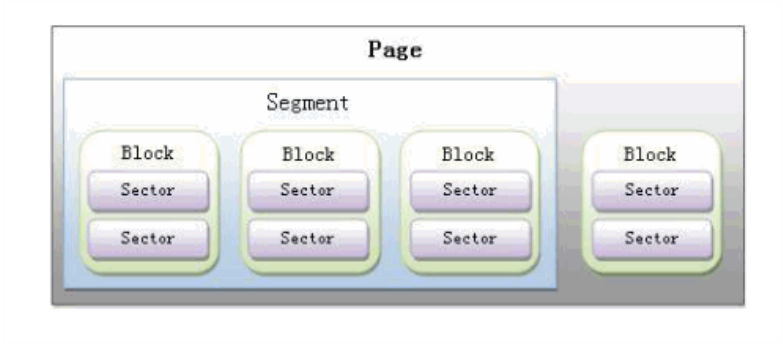
些数据可以在设备上通过指定位置进行操作。举例作为对比，鼠标或者键盘是典型的字符设备，内核获得这些设备传来的数据都是按照产生的顺序而得到的；而磁盘上数据的读写，则必须要指定数据所在磁盘上的具体位置。为了描述这种位置信息有以下几个块设备中的基本概念：

1、扇区(Sectors)：任何块设备硬件对数据处理的基本单位。通常，1 个扇区的大小为 512byte。

2、块(Blocks)：由 Linux 制定对内核或文件系统等数据处理的基本单位。通常，1 个块由 1 个或多个扇区组成。

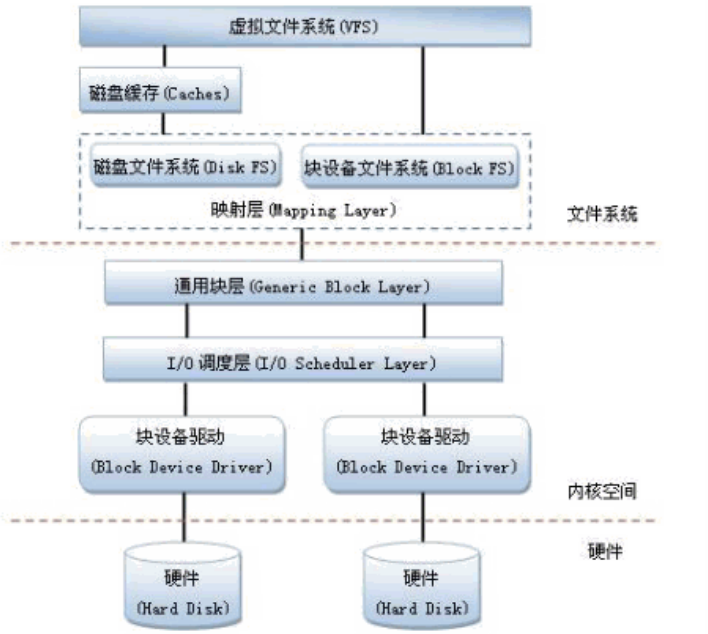
3、段(Segments)：由若干个相邻的块组成。是 Linux 内存管理机制中一个内存页或者内存页的一部分。

页、段、块、扇区之间的关系图如下：



4.2.1 块设备驱动的角色

由于块设备模型被设计用于管理磁盘类型的核心设备，这些设备通常承载着文件系统等重要数据，因此块设备与 linux 内核的联系非常紧密。下图展示了块设备驱动在 linux 系统中的作用和角色。



4.2.2 块设备的注册

块设备驱动程序通过调用 register_blkdev()进行注册。该函数的原型为：

```
int register_blkdev(unsigned int major, const char *name)
```

在新的内核中，这个函数仅是将设备名 name 与指定的 major 相关联。

4.2.3 块设备的操作方式

在 Linux 中，驱动对块设备的输入或输出(I/O)操作，都会向块设备发出一个请求，在驱动中用 request 结构体描述。但对于一些磁盘设备而言请求的速度很慢，这时候内核就提供一种队列的机制把这些 I/O 请求添加到队列中(即：请求队列)，在驱动中用 request_queue 结构体描述。在向块设备提交这些请求前内核会先执行请求的合并和排序预操作，以提高访问的效率，然后再由内核中的 I/O 调度程序子系统(即：上图中的 I/O 调度层)来负责提交 I/O 请求，I/O 调度程序将磁盘资源分配给系统中所有挂起的块 I/O 请求，其工作是管理块设备的请求队列，决定队列中的请求的排列顺序以及什么时候派发请求到设备。

其次，块设备驱动又是怎样维持一个 I/O 请求在上层文件系统与底层物理磁盘之间的关系呢？这就是上图中通用块层(Generic Block Layer)要做的事情了。在通用块层中，通常用一个 bio 结构体来对应一个 I/O 请求，它代表了正在活动的以段(Segment)链表形式组织的块 IO 操作，对于它所需要的所有段又用 bio_vec 结构体表示。

再次，块设备驱动又是怎样对底层物理磁盘进行访问的呢？上面讲的都是对上层的访问对上层的关系。Linux 提供了一个 gendisk 数据结构体，用来表示一个独立的磁盘设备或分区。在 gendisk 中有一个类似字符设备中 file_operations 的硬件操作结构指针，它就是 block_device_operations 结构体。

4.2.4 块设备初始化分析

上节简单叙述了块设备的操作方式，本节通过对一个具体块设备的初始化过程的分析，来详细描述其中涉及的结构体，并且说明结构体之间如何相互联系。

Ram backed block device（Ramdisk）设备

Ramdisk 设备是用一块内存来模拟磁盘设备，可供 linux 在必要时利用这种设备存取数据或者挂载为文件系统。本节具体分析这种设备(brd_device)的初始化过程：

brd 设备的初始化过程与其他设备相似，都是由 module_init()定义，在 do_initcall()的时候被执行。执行初始化的具体函数为：

```
static int __init brd_init(void)
{
        int i, nr;
        unsigned long range;
        struct brd_device *brd, *next;
        part_shift = 0;
        if (max_part > 0)
                part_shift = fls(max_part);
        if (rd_nr > 1UL << (MINORBITS - part_shift))
```

```
        return -EINVAL;
    if (rd_nr) {
        nr = rd_nr;
        range = rd_nr;
    } else {
        nr = CONFIG_BLK_DEV_RAM_COUNT;
        range = 1UL << (MINORBITS - part_shift);
    }
    if (register_blkdev(RAMDISK_MAJOR, "ramdisk"))
        return -EIO;
    for (i = 0; i < nr; i++) {
        brd = brd_alloc(i);
        if (!brd)
            goto out_free;
        list_add_tail(&brd->brd_list, &brd_devices);
    }
    /* point of no return */
    list_for_each_entry(brd, &brd_devices, brd_list)
        add_disk(brd->brd_disk);
    blk_register_region(MKDEV(RAMDISK_MAJOR, 0), range,
                        THIS_MODULE, brd_probe, NULL, NULL);
    printk(KERN_INFO "brd: module loaded\n");
    return 0;
out_free:
    list_for_each_entry_safe(brd, next, &brd_devices, brd_list) {
        list_del(&brd->brd_list);
        brd_free(brd);
    }
    unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
    return -ENOMEM;
}
```

首先获取 Ramdisk 的 number 参数。如果使用 lsmod 载入模块的话，则从命令行获取，如果是作为内核内部模块直接在启动时加载(我们正是使用这种方式)，则将其设定为 nr = CONFIG_BLK_DEV_RAM_COUNT=16;

下一步，即调用 register_blkdev(RAMDISK_MAJOR, "ramdisk")注册块设备，也即将“ramdisk”与 RAMDISK_MAJOR(=1)关联起来。启动之后，所有 ramdisk 设备对应的块设备主设备号为 1。

接着，根据最初获得的 nr 值，初始化 nr(16)个 brd_device，调用函数为 brd_alloc()

brd_alloc()分析：

该函数的实现如下：

```
static struct brd_device *brd_alloc(int i)
{
    struct brd_device *brd;
    struct gendisk *disk;
    brd = kzalloc(sizeof(*brd), GFP_KERNEL);
```

```
        if (!brd)
                goto out;
        brd->brd_number         = i;
        spin_lock_init(&brd->brd_lock);
        INIT_RADIX_TREE(&brd->brd_pages, GFP_ATOMIC);
        brd->brd_queue = blk_alloc_queue(GFP_KERNEL);
        if (!brd->brd_queue)
                goto out_free_dev;
        blk_queue_make_request(brd->brd_queue, brd_make_request);
        blk_queue_max_sectors(brd->brd_queue, 1024);
        blk_queue_bounce_limit(brd->brd_queue, BLK_BOUNCE_ANY);
        disk = brd->brd_disk = alloc_disk(1 << part_shift);
        if (!disk)
                goto out_free_queue;
        disk->major             = RAMDISK_MAJOR;
        disk->first_minor       = i << part_shift;
        disk->fops              = &brd_fops;
        disk->private_data      = brd;
        disk->queue             = brd->brd_queue;
        disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
        sprintf(disk->disk_name, "ram%d", i);
        set_capacity(disk, rd_size * 2);
        return brd;
out_free_queue:
        blk_cleanup_queue(brd->brd_queue);
out_free_dev:
        kfree(brd);
out:
        return NULL;
}
```

首先要介绍两个重要的结构体：brd_device 和 gendisk，前者是描述 brd 设备的结构，后者则是 linux 内用来将块设备与磁盘概念相关联起来的重要结构。

```
struct brd_device {
        int             brd_number;
        int             brd_refcnt;
        loff_t          brd_offset;
        loff_t          brd_sizelimit;
        unsigned        brd_blocksize;
        struct request_queue    *brd_queue;
        struct gendisk          *brd_disk;
        struct list_head        brd_list;
        spinlock_t              brd_lock;
        struct radix_tree_root  brd_pages;
};
```

```
struct gendisk {
        int major;                  /* major number of driver */
        int first_minor;
        int minors;                             /* maximum number of minors, =1 for
                                                 * disks that can't be partitioned. */
        char disk_name[DISK_NAME_LEN];      /* name of major driver */
        struct disk_part_tbl *part_tbl;
        struct hd_struct part0;
        struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        int flags;
        struct device *driverfs_dev;   // FIXME: remove
        struct kobject *slave_dir;
        struct timer_rand_state *random;
        atomic_t sync_io;               /* RAID */
        struct work_struct async_notify;
#ifdef  CONFIG_BLK_DEV_INTEGRITY
        struct blk_integrity *integrity;
#endif
        int node_id;
};
```

1) 分配一个 brd_device，用 brd 指针指向该结构。设定结构内的 brd_number，初始化结构内的 spin_lock、radix_tree_root 成员。
2) 分配并初始化一个 request_queue 结构，然后连接到 brd->brd_queue 指针上。
3) 调用 blk_queue_make_request(brd->brd_queue, brd_make_request)进一步初始化这个 request_queue 的成员，将 requst_queue->make_request_fn 成员指定为 brd_make_request 函数。
4) 设定 request_queue 的最大 sectors，dma 相关的参数。
5) 调用 alloc_disk()分配并初始化一个 gendisk 结构，并连接到 brd->brd_disk 指针上。初始化过程中特别初始化了一个 device 结构 (disk)->part0.__dev，这个是 3.3.1 节中介绍的 linux 驱动核心数据结构，通过这个方法将 brd_device 纳入 linux 设备管理体系。
6) 继续设定这个 gendisk 的成员，值得一提的是以下几个：
   disk->fops = &brd_fops 指定 disk 的 block_device_operations
   disk->queue = brd->brd_queue; 指定 disk->queue
   disk->private_data = brd;
   sprintf(disk->disk_name, "ram%d", i)设定 disk 的名字，即 ram0,ram1...等。

brd_alloc()函数基本功能就是分配并初始化了一个 brd_device 结构，包括初始化里面的一个 gendisk 结构。

brd_alloc()函数返回后，调用 list_add_tail(&brd->brd_list, &brd_devices);将刚才初始化的 brd_device 连接到 brd_devices 链表头上。

完成了 nr 个 brd_device 的初始化之后，调用 add_disk(brd->brd_disk);将每一个 brd_devices 链表上的 brd_device 内的 gendisk 添加进系统。

void add_disk(struct gendisk *disk)分析

```
void add_disk(struct gendisk *disk)
{
        struct backing_dev_info *bdi;
        dev_t devt;
        int retval;
        WARN_ON(disk->minors && !(disk->major || disk->first_minor));
        WARN_ON(!disk->minors && !(disk->flags & GENHD_FL_EXT_DEVT));
        disk->flags |= GENHD_FL_UP;
        retval = blk_alloc_devt(&disk->part0, &devt);
        if (retval) {
                WARN_ON(1);
                return;
        }
        disk_to_dev(disk)->devt = devt;
        /* ->major and ->first_minor aren't supposed to be
         * dereferenced from here on, but set them just in case.
         */
        disk->major = MAJOR(devt);
        disk->first_minor = MINOR(devt);
        blk_register_region(disk_devt(disk), disk->minors, NULL, exact_match, exact_lock, disk);
        register_disk(disk);
        blk_register_queue(disk);
        bdi = &disk->queue->backing_dev_info;
        bdi_register_dev(bdi, disk_devt(disk));
        retval = sysfs_create_link(&disk_to_dev(disk)->kobj, &bdi->dev->kobj, "bdi");
        WARN_ON(retval);
}
```

先看一下涉及的一个新结构体，backing_dev_info 结构体

```
struct backing_dev_info {
        unsigned long ra_pages;      /* max readahead in PAGE_CACHE_SIZE units */
        unsigned long state;    /* Always use atomic bitops on this */
        unsigned int capabilities; /* Device capabilities */
        congested_fn *congested_fn; /* Function pointer if device is md/dm */
        void *congested_data; /* Pointer to aux data for congested func */
        void (*unplug_io_fn)(struct backing_dev_info *, struct page *);
        void *unplug_io_data;
        struct percpu_counter bdi_stat[NR_BDI_STAT_ITEMS];
        struct prop_local_percpu completions;
        int dirty_exceeded;
        unsigned int min_ratio;
        unsigned int max_ratio, max_prop_frac;
        struct device *dev;
};
```

1) 通过 disk->part0 得到设备号 dev_t，并设定至 disk->part0.__dev(device 结构)内
2) blk_register_region(disk_devt(disk), disk->minors, NULL, exact_match, exact_lock, disk);注册块设备的设备号范围，使得设备号能够被 bdev_map(kobj_map 类型)指针所引用和管理。
3) 调用 register_disk()注册磁盘。注册磁盘的过程建立了与磁盘对应的 block_device 结构等。
4) blk_register_queue(disk);注册 disk 的 request_queue，同前面 brd_device 的 request_queue。
5) bdi_register_dev(bdi, disk_devt(disk))注册对应的 backing_dev_info，创建（主设备号：次设备号形式的）设备文件链接。

完成了 brd_devices 上所有 brd->disk 的 add_disk 操作之后，会调用 blk_register_region (MKDEV(RAMDISK_MAJOR, 0), range,THIS_MODULE,brd_probe, NULL, NULL);来设定 ramdisk 的 probe 操作为 brd_probe。该函数基本上重复 brd_init 中的分配 brd，初始化 disk 等操作。因此初始化完成后，如果需要进一步添加 ramdisk 设备，可以通过回调这个函数实现。

4.3 MTD(Memory Technology Device)模型

Linux 系统越来越多的运用于便携式设备，这些设备的一个重要特点是通常不采用传统磁盘作为存储介质，而采用 nand Flash 取代。为了更高效的使用 nand flash 存储器，linux 引进了 MTD 设备模型。MTD 模型更适合纯 Nand Flash/Nor Flash 介质，而由 Nand Flash 加上控制器生产出来的 MMC、U 盘等设备更适合用块设备模型。

因为 MTD 是一个更适合于 Android 平台的设备模型，本节详细讨论 S3C6410 平台上该模型的实现和使用。

4.3.1 MTD 概览

MTD 子系统有如下接口：
MTD 字符设备，通常通过/dev/mtd0, /dev/mtd1 等来使用，这类字符设备提供了访问原始闪存的 IO 接口。它们提供了大量的 ioctl 调用用来擦除擦除块，标记擦除块坏或者检测擦除块是否坏掉，获取 MTD 设备的信息等。
sysfs 接口提供了系统内 MTD 设备的全部信息。这个接口可以方便的扩展并且鼓励开发者尽可能使用 sysfs 接口替代旧式 ioctl 或者/proc/mtd 接口。
/proc/mtd 接口提供了通用的 MTD 信息。这是一个遗留接口，sysfs 提供了更多信息。
MTD 子系统支持带硬件或软件 ECC 的裸 NAND 闪存，OneNAND 闪存，CFI NOR 闪存及其它闪存类型
此外，MTD 支持遗留的 FTL/NFTL 翻译层，M-Systems' DiskOnChip 2000 和 Millennium 芯片，以及 PCMCIA 闪存。

与 MTD 设备相关的几个结构体：

```
struct mtd_info {
    u_char type;
    u_int32_t flags;
    u_int32_t size;     // Total size of the MTD
    u_int32_t erasesize;
    u_int32_t writesize;
    u_int32_t oobsize;     // Amount of OOB data per block (e.g. 16)
    u_int32_t oobavail;    // Available OOB bytes per block
```

```
    // Kernel-only stuff starts here.
    const char *name;
    int index;
    struct nand_ecclayout *ecclayout;
    int numeraseregions;
    struct mtd_erase_region_info *eraseregions;
    int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
    int (*point) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, void **virt, resource_size_t *phys);
    void (*unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
    int (*read) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*write) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const u_char *buf);
    int (*panic_write) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const u_char *buf);
    int (*read_oob) (struct mtd_info *mtd, loff_t from, struct mtd_oob_ops *ops);
    int (*write_oob) (struct mtd_info *mtd, loff_t to, struct mtd_oob_ops *ops);
    int (*get_fact_prot_info) (struct mtd_info *mtd, struct otp_info *buf, size_t len);
    int (*read_fact_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*get_user_prot_info) (struct mtd_info *mtd, struct otp_info *buf, size_t len);
    int (*read_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*write_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*lock_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len);
    int (*writev) (struct mtd_info *mtd, const struct kvec *vecs, unsigned long count, loff_t to, size_t *retlen);
    void (*sync) (struct mtd_info *mtd);
    int (*lock) (struct mtd_info *mtd, loff_t ofs, size_t len);
    int (*unlock) (struct mtd_info *mtd, loff_t ofs, size_t len);
    /* Power Management functions */
    int (*suspend) (struct mtd_info *mtd);
    void (*resume) (struct mtd_info *mtd);
    /* Bad block management functions */
    int (*block_isbad) (struct mtd_info *mtd, loff_t ofs);
    int (*block_markbad) (struct mtd_info *mtd, loff_t ofs);
    struct notifier_block reboot_notifier;  /* default mode before reboot */
    struct mtd_ecc_stats ecc_stats;
    /* Subpage shift (NAND) */
    int subpage_sft;
    void *priv;
    struct module *owner;
    int usecount;
    int (*get_device) (struct mtd_info *mtd);
    void (*put_device) (struct mtd_info *mtd);
};
```

mtd_info 结构包含了 mtd 设备完整的描述，包括各种信息以及指向操作函数的指针。mtd 设备在 Linux 的设备管理体系中属于一个中间层次，它的上层通过 mtdchar 和 mtdblock 两种模型建立 linux 的字符设备或者块设备，而下层则调用 Nand Flash 等具体设备的驱动程序来完成操作。

```
struct mtd_notifier {
    void (*add)(struct mtd_info *mtd);
    void (*remove)(struct mtd_info *mtd);
    struct list_head list;
};
```

mtd_notifier 结构用于在 mtd 设备层上构建字符或者块设备时指定设备添加/删除时的通知操作函数。

4.3.2 mtd 层的初始化

mtd 是操作 Nand Flash/Nor Flash 的中间层设备，在 Nand Flash 设备的初始化过程中，会初始化 mtd 层的设备。从 nand Flash 设备初始化入手分析 mtd 的初始化过程。

s3c_nand_probe()

在 2.4 节中提到的设备注册过程中，S3C6410 平台包含了很多 platform 总线上的设备，s3c_nand_device 也是其中之一。在初始化时，这些设备将会注册到 linux 内部，并且会调用相关驱动中的 probe 函数完成驱动程序的初始化。s3c_nand_probe()函数负责完成 s3c_nand_device 对应的驱动程序初始化过程，其中有以下几个步骤涉及到了 mtd 层的初始化。

```
s3c_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip), GFP_KERNEL);
nand = (struct nand_chip *) (&s3c_mtd[1]);
memset((char *) s3c_mtd, 0, sizeof(struct mtd_info));
memset((char *) nand, 0, sizeof(struct nand_chip));
s3c_mtd->priv = nand;
nand_scan(s3c_mtd, 1)
add_mtd_partitions(s3c_mtd, partition_info, plat_info->mtd_part_nr);
```

s3c_mtd 是一个指向 mtd_info 结构体的指针，这个结构中的 priv（内部专用数据指针）指向一个 nand_chip 类型的结构体，这个 nand_chip 类型描述了 mtd 下层的 nand flash 相关的操作和信息。

nand_scan(s3c_mtd, 1)

该函数通过调用指定的操作函数，对系统连接的 Nand Flash 进行扫描，确认是否是预期的 nand flash 类型、ID 等，还要设定 mtd_info 中的总 size 信息。

add_mtd_partitions()

该函数向系统注册 mtd 设备的分区信息。

```
add_mtd_partitions(s3c_mtd, partition_info, plat_info->mtd_part_nr);
```

其中 partition_info 参数由 s3c_nand_device 提供，这个信息在早期的平台初始化过程中指定为：s3c_device_nand.dev.platform_data = &s3c_nand_mtd_part_info;

```
struct s3c_nand_mtd_info s3c_nand_mtd_part_info = {
.chip_nr = 1,
.mtd_part_nr = ARRAY_SIZE(s3c_partition_info),
.partition = s3c_partition_info,
};
```

```
struct mtd_partition s3c_partition_info[] = {
        {
                .name           = "cramfs",
                .offset         = (4* SZ_1M),
                .size       = (4* SZ_1M),
        },
         {
                .name           = "ubifs",
                .offset         = MTDPART_OFS_APPEND,
                .size       = MTDPART_SIZ_FULL,
        }
};
```

Nand Flash 分为两个 mtd 分区，"cramfs"和"ubifs"，并且分别指定了在 nand flash 上的起始位置及 size。

add_mtd_partitions 函数对每个分区调用 add_one_partition()进行注册。注册过程创建一个 mtd_part 的结构体，连接到全局的 mtd_partitions 链表头上，并且设定 mtd_part 的内部信息及操作函数，最后调用 add_mtd_device()将 mtd_part 内的 mtd_info 进行注册。

```
struct mtd_part {
    struct mtd_info mtd;
    struct mtd_info * master;
    u_int32_t offset;
    int index;
    struct list_head list;
    int registered;
};
```

mtd_part 结构基本上是将初始化后 s3c_mtd 中的某些信息复制出来，而其 master 指针则指向 s3c_mtd。

add_mtd_device()除了进行检查之外，主要功能是回调 mtd_notifiers 链表上的所有通知操作函数中的 add()。mtd_notifiers 链表上的 notifier 是由 mtdchar 设备和 mtd_blkdev 设备调用 register_mtd_user 来加入的。

mtdchar 字符设备：

```
register_mtd_user(&notifier);
static struct mtd_notifier notifier = {
    .add  = mtd_notify_add,
    .remove    = mtd_notify_remove,
};
static void mtd_notify_add(struct mtd_info* mtd)
{
  if (!mtd)
      return;
  device_create(mtd_class, NULL, MKDEV(MTD_CHAR_MAJOR, mtd->index* 2), NULL, "mtd%d", mtd->index);
  device_create(mtd_class, NULL, MKDEV(MTD_CHAR_MAJOR, mtd->index* 2+1),NULL,"mtd%dro", mtd->index);
}
```

可见，添加 2 个 mtd 分区时，会添加名为 mtd0,mtd0ro,mtd1,mtd1ro 的字符设备，它们的主

设备号都是 MTD_CHAR_MAJOR=90，次设备号为 index*2 和 index*2+1，奇数为 ro（只读）设备。添加成功后，可以通过这些字符设备节点访问 mtd 设备。

4.3.3 mtdblock 设备初始化

mtdblock 设备是在 mtd 层之上建立的块设备，目的是为了让 linux 能够像访问普通磁盘那样访问基于 nand flash 的 mtd 设备。

mtd_blktrans_ops 结构体：

```
struct mtd_blktrans_ops{
    char *name;
    int major;
    int part_bits;
    int blksize;
    int blkshift;
    /* Access functions */
    int (*readsect)(struct mtd_blktrans_dev *dev, unsigned long block, char *buffer);
    int (*writesect)(struct mtd_blktrans_dev *dev, unsigned long block, char *buffer);
    int (*discard)(struct mtd_blktrans_dev *dev, unsigned long block, unsigned nr_blocks);
    /* Block layer ioctls */
    int (*getgeo)(struct mtd_blktrans_dev *dev, struct hd_geometry *geo);
    int (*flush)(struct mtd_blktrans_dev *dev);
    /* Called with mtd_table_mutex held; no race with add/remove */
    int (*open)(struct mtd_blktrans_dev *dev);
    int (*release)(struct mtd_blktrans_dev *dev);
    void (*add_mtd)(struct mtd_blktrans_ops *tr, struct mtd_info *mtd);
    void (*remove_dev)(struct mtd_blktrans_dev *dev);
    struct list_head devs;
    struct list_head list;
    struct module *owner;
    struct mtd_blkcore_priv *blkcore_priv;
};
```

mtdblock 块结构有时也称为块转换层，mtd_blktrans_ops 用来定义这个转换所需的操作和信息。linux 代码中定义的具体结构成员为：

```
static struct mtd_blktrans_ops mtdblock_tr = {
    .name       = "mtdblock",
    .major      = 31,
    .part_bits  = 0,
    .blksize    = 512,
    .open       = mtdblock_open,
    .flush      = mtdblock_flush,
    .release    = mtdblock_release,
    .readsect   = mtdblock_readsect,
    .writesect  = mtdblock_writesect,
    .add_mtd    = mtdblock_add_mtd,
```

```
    .remove_dev     = mtdblock_remove_dev,
    .owner          = THIS_MODULE,
};
```

mtdblock 设备初始化过程其实就是注册上述 mtd_blktrans_ops 的过程。

```
    register_mtd_blktrans(&mtdblock_tr);
```

该函数的主要流程如下：

```
int register_mtd_blktrans(struct mtd_blktrans_ops *tr)
{
    if (!blktrans_notifier.list.next)
        register_mtd_user(&blktrans_notifier);
    tr->blkcore_priv = kzalloc(sizeof(*tr->blkcore_priv), GFP_KERNEL);
    ret = register_blkdev(tr->major, tr->name);
    tr->blkcore_priv->rq = blk_init_queue(mtd_blktrans_request, &tr->blkcore_priv->queue_lock);
    tr->blkcore_priv->rq->queuedata = tr;
    tr->blkcore_priv->thread = kthread_run(mtd_blktrans_thread, tr, "%sd", tr->name);
    list_add(&tr->list, &blktrans_majors);
}
```

1) 注册 blktrans_notifier，在后期添加 mtd 分区时，会调用其指定的 add 函数添加块设备。
2) 调用 register_blkdev()注册块设备。
3) 调用 blk_init_queue，指定 mtdblock 设备接收到 IO request 后调用 mtd_blktrans_request 函数处理，该函数仅是唤醒下面将要介绍到的一个内核线程 mtd_blktrans_thread。
4) 创建 mtdblock 的处理线程 mtd_blktrans_thread。
5) 将 mtdblock_tr 连接到 blktrans_majors 链表中。

   第 1 步注册的 notifier 操作，在添加 mtd 分区时回调 add 函数。该 add 函数则会进一步回调第 5 步注册的 mtdblock_tr 的 add_mtd 函数，即 mtdblock_add_mtd()函数。该函数创建了一个 mtd_blktrans_dev 类型的结构，并基于这个结构完成块设备的注册，如添加磁盘、添加块设备节点等工作。得到的块设备节点名称用 mtdblock0,mtdblock1 等表示。

4.3.4 mtd 设备的动态运行

在 mtd 设备之上建立的 mtdblock 设备被作为一个常规的块设备使用，系统对块设备（磁盘）的访问通过虚拟文件系统(VFS)进行，从用户程序的 open 调用开始，到最终 Nand Flash 的具体操作，需要经过多层的转换。详细的实现细节，参见虚拟文件系统的说明文档。

4.4 sysdev 模型

sysdev 是一种非常简单的设备驱动模型，该模型基本上仅是在 linux 的核心数据结构 kobj、kset 等之上进行了少量扩展。这种模型非常适用于描述平台相关的硬件功能，比如 DMA、IRQ 等。通常这些硬件功能并不独立使用，而是提供接口给其他设备(字符设备/块设备等)调用，因此这种 sysdev 一般不会在 sysfs 中被用户层访问。下面是几个 sysdev 的数据结构：

```
struct sysdev_class {
	const char *name;
	struct list_head	drivers;

	/* Default operations for these types of devices */
	int	(*shutdown)(struct sys_device *);
	int	(*suspend)(struct sys_device *, pm_message_t state);
	int	(*resume)(struct sys_device *);
	struct kset		kset;
};
struct sys_device {
	u32		id;
	struct sysdev_class	* cls;
	struct kobject		kobj;
};
struct sysdev_driver {
	struct list_head	entry;
	int	(*add)(struct sys_device *);
	int	(*remove)(struct sys_device *);
	int	(*shutdown)(struct sys_device *);
	int	(*suspend)(struct sys_device *, pm_message_t state);
	int	(*resume)(struct sys_device *);
};
```

由上面的结构体定义可以看出，sysdev 相关的设备/驱动结构体非常简单，甚至 sysdev_driver 结构中都没有包含自己的 kobj，只是简单的指定了一系列操作指针，从这个角度来看，sysdev_driver 并不能算一个驱动体系内的实体。实际上，整个体系内的 sysdev 实体只有一个 sys_device，各 sysdev_driver 注册了各自的操作指针，而这些操作实际上并不会直接作用于 sysdev_device，而是操作了各自相关的其它结构体。这一点在下文介绍的 DMA 设备驱动中会得到验证。

4.4.1 sysdev 设备/驱动注册过程

与 platform 设备的注册过程类似，在 linux 启动初期，setup_arch()中就先向系统添加了 sysdev_device，具体的调用关系如下：

```
setup_arch()---------------->paging_init()------------------------->devicemaps_init()------------------>
smdk6410_map_io()------------------------------>s3c64xx_init_io()------------------>s3c_init_cpu---------
------>s3c6410_init()
int __init s3c6410_init(void)
{
	printk("S3C6410: Initialising architecture\n");

	return sysdev_register(&s3c6410_sysdev);
}
```

```
struct sysdev_class s3c6410_sysclass = {
	.name		= "s3c6410-core",
};

static struct sys_device s3c6410_sysdev = {
	.cls	= &s3c6410_sysclass,
};
static struct sysfs_ops sysfs_ops = {
	.show		= sysdev_show,
	.store= sysdev_store,
};

static struct kobj_type ktype_sysdev = {
	.sysfs_ops	= &sysfs_ops,
};
```

sysdev_register(&s3c6410_sysdev)的功能非常简单，将 s3c6410_sysdev 实体加入 Linux 管理体系，指定其对应的 ktype 为 ktype_sysdev，名字为"s3c6410-core"，kset 为 s3c6410_sysclass。可以说，除了将 kobj 连接到管理体系的必要步骤外，仅是增加了名字和属性的操作方法。sysdev 本身并没有任何额外的数据信息和操作方法。

在启动后期，do_initcalls 中，会首先调用到 core_initcall()指定的 s3c6410_core_init 函数。core_initcall 指定的 level 是 1，会先于其它 level 的调用(参见 1.4.2)。

```
static int __init s3c6410_core_init(void)
{
	return sysdev_class_register(&s3c6410_sysclass);
}
int sysdev_class_register(struct sysdev_class * cls)
{
	pr_debug("Registering sysdev class '%s'\n", cls->name);

	INIT_LIST_HEAD(&cls->drivers);
	memset(&cls->kset.kobj, 0x00, sizeof(struct kobject));
	cls->kset.kobj.parent = &system_kset->kobj;
	cls->kset.kobj.ktype = &ktype_sysdev_class;
	cls->kset.kobj.kset = system_kset;
	kobject_set_name(&cls->kset.kobj, cls->name);
	return kset_register(&cls->kset);
}
```

这一步是将 s3c6410_sysclass 对应的 kset 注册到管理体系内。

接下来，将会使用 arch_initcall()指定 level 为 3 的初始化函数。与 sysdev 相关的几个具体设备，如 DMA、IRQ 等都是在这个 level 被初始化的。正是因为这样，到后面 module_init (level=6)指定的各设备驱动初始化的时候，才能使用 DMA、IRQ 等驱动提供的功能函数。

4.4.2 DMA 设备的注册

本节将看一个具体的 sysdev 设备的注册过程。

```
static int __init s3c6410_dma_init(void)
{
    return sysdev_driver_register(&s3c6410_sysclass, &s3c6410_dma_driver);
}
arch_initcall(s3c6410_dma_init);
static struct sysdev_driver s3c6410_dma_driver = {
    .add  = s3c6410_dma_add,
};
```

正如上节所提到的，arch_initcall()指定了 s3c6410_dma_init 的 level=3，而该函数实际上调用了 sysdev_driver_register(&s3c6410_sysclass, &s3c6410_dma_driver)进行注册。

```
int sysdev_driver_register(struct sysdev_class *cls, struct sysdev_driver *drv)
{
    int err = 0;
    if (!cls) {
        WARN(1, KERN_WARNING "sysdev: invalid class passed to sysdev_driver_register!\n");
        return -EINVAL;
    }
    /* Check whether this driver has already been added to a class. */
    if (drv->entry.next && !list_empty(&drv->entry))
        WARN(1, KERN_WARNING "sysdev: class %s: driver (%p) has already"
             " been registered to a class, something is wrong, but will forge on!\n", cls->name, drv);
    mutex_lock(&sysdev_drivers_lock);
    if (cls && kset_get(&cls->kset)) {
        list_add_tail(&drv->entry, &cls->drivers);
        /* If devices of this class already exist, tell the driver */
        if (drv->add) {
            struct sys_device *dev;
            list_for_each_entry(dev, &cls->kset.list, kobj.entry)
                drv->add(dev);
        }
    } else {
        err = -EINVAL;
        WARN(1, KERN_ERR "%s: invalid device class\n", __func__);
    }
    mutex_unlock(&sysdev_drivers_lock);
    return err;
}
```

sysdev_driver 的注册过程形式上与一般的 device_driver 注册类似，不同的是 device_driver 发现符合条件的 device 之后，会调用 probe 函数，而 sysdev_driver 则调用 add 函数。DMA 设备驱动对应的 add 函数为：

```
static int __init s3c6410_dma_add(struct sys_device *sysdev)
```

```
{
    s3c_dma_init(S3C_DMA_CHANNELS, IRQ_DMA0, 0x20);
    return s3c_dma_init_map(&s3c6410_dma_sel);
}
```

正如前面提到的，add 函数的入口参数是 sys_device 指针，但是实际上在 add 函数内不会使用该结构，因此系统内只需要一个 sys_device，就能够注册多个 sysdev_driver，而这些 sysdev_driver 实际上与 sys_device 并没有什么直接联系，仅为了确保系统内已经注册了 sysdev 相关的结构。

4.4.3 s3c6410 平台的 DMA 设备初始化

s3c6410 平台的 DMA 设备初始化过程，由上节提到的 add 函数完成，分两个步骤：

s3c_dma_init 和 s3c_dma_init_map

首先分析第一步

s3c_dma_init(S3C_DMA_CHANNELS, IRQ_DMA0, 0x20);

```
int __init s3c_dma_init(unsigned int channels, unsigned int irq, unsigned int stride)
{
    struct s3c2410_dma_chan *cp;
    s3c_dma_controller_t *dconp;
    int channel, controller;
    int ret;
    dma_channels = channels;
    ret = sysdev_class_register(&dma_sysclass);
    dma_kmem = kmem_cache_create("dma_desc", sizeof(struct s3c_dma_buf), 0,
                                 SLAB_HWCACHE_ALIGN, (void *)s3c_dma_cache_ctor);
    for (controller = 0; controller < S3C_DMA_CONTROLLERS; controller++) {
        dconp = &s3c_dma_cntlrs[controller];
        memset(dconp, 0, sizeof(s3c_dma_controller_t));
        if(controller < 2) {
            dma_base = ioremap((S3C_PA_DMA + (controller * 0x100000)), 0x200);
            if (dma_base == NULL) {
                printk(KERN_ERR "DMA failed to ioremap register block\n");
                return -ENOMEM;
            }
            /* dma controller's irqs are in order.. */
            dconp->irq = controller + IRQ_DMA0;
        }
        else {
            dma_base = ioremap(((S3C_PA_DMA + 0x8B00000) + ((controller%2) * 0x100000)), 0x200);
            dconp->irq = (controller%2) + IRQ_SDMA0;
        }
        dconp->number = controller;
        dconp->regs = dma_base;
    }
```

```
    for (channel = 0; channel < channels; channel++) {
        controller = channel / S3C_CHANNELS_PER_DMA;
        cp = &s3c_dma_chans[channel];
        memset(cp, 0, sizeof(struct s3c2410_dma_chan));
        cp->dma_con = &s3c_dma_cntlrs[controller];
        cp->index = channel;
        cp->number = channel%S3C_CHANNELS_PER_DMA;
        cp->irq = s3c_dma_cntlrs[controller].irq;
        cp->regs = s3c_dma_cntlrs[controller].regs + ((channel%S3C_CHANNELS_PER_DMA) * stride) + 0x100;
        cp->stats = &cp->stats_store;
        cp->stats_store.timeout_shortest = LONG_MAX;
        cp->load_timeout = 1 << 18;
        cp->dev.cls = &dma_sysclass;
        cp->dev.id = channel;
    }
    return 0;
}
```

文中只显示了重要的部分代码。这一步实际上初始化了两个全局数组，并且注册了一个 sysdev_class。

注册的 sysdev_class 如下：

```
struct sysdev_class dma_sysclass = {
    .name = "s3c64xx-dma",
    .suspend = s3c_dma_suspend,
    .resume = s3c_dma_resume,
};
```

初始化的两个全局数组分别是：

```
    struct s3c2410_dma_chan    s3c_dma_chans[S3C_DMA_CHANNELS];
    s3c_dma_controller_t       s3c_dma_cntlrs[S3C_DMA_CONTROLLERS];
```

s3c6410 平台的 DMA 控制器包含四个独立的 controller，每个 controller 具备 8 个通道，可以分别执行 DMA 操作，整个 DMA 控制器具有 32 个 DMA 通道。

s3c_dma_chans 是包含了 32 个描述 DMA 通道的结构体数组；s3c_dma_cntlrs 则是包含了 4 个描述 controller 的结构体数组。

描述通道和 controller 的结构体定义分别如下：

```
struct s3c2410_dma_chan {
    /* channel state flags and information */
    unsigned char       number;         /* number of this dma channel */
    unsigned char       in_use;         /* channel allocated */
    unsigned char       irq_claimed;    /* irq claimed for channel */
    unsigned char       irq_enabled;    /* irq enabled for channel */
    unsigned char       xfer_unit;      /* size of an transfer */
    /* channel state */
    enum s3c_dma_state      state;
    enum s3c_dma_loadst     load_state;
```

```
    struct s3c2410_dma_client *client;
    /* channel configuration */
    enum s3c2410_dmasrc    source;
    unsigned long          dev_addr;
    unsigned long          load_timeout;
    unsigned int           flags;          /* channel flags */
    struct s3c_dma_map     *map;           /* channel hw maps */
    /* channel's hardware position and configuration */
    void __iomem           *regs;          /* channels registers */
    void __iomem           *addr_reg;      /* data address register */
    unsigned int           irq;        /* channel irq */
    unsigned long          dcon;           /* default value of DCON */
    /* driver handles */
    s3c2410_dma_cbfn_t     callback_fn;  /* buffer done callback */
    s3c2410_dma_opfn_t     op_fn;          /* channel op callback */
    /* stats gathering */
    struct s3c_dma_stats *stats;
    struct s3c_dma_stats   stats_store;
    /* buffer list and information */
    struct s3c_dma_buf     *curr;          /* current dma buffer */
    struct s3c_dma_buf     *next;          /* next buffer to load */
    struct s3c_dma_buf     *end;           /* end of queue */
    /* system device */
    struct sys_device    dev;
    unsigned int                index;              /* channel index */
    unsigned int                config_flags;           /* channel flags */
    unsigned int                control_flags;           /* channel flags */
    s3c_dma_controller_t    *dma_con;
};
struct s3c_dma_controller {
    /* channel state flags and information */
    unsigned char               number;         /* number of this dma channel */
    unsigned char               in_use;         /* channel allocated and how many channel are
used */
    unsigned char               irq_claimed; /* irq claimed for channel */
    unsigned char               irq_enabled; /* irq enabled for channel */
    unsigned char               xfer_unit;     /* size of an transfer */
    /* channel state */
    enum s3c_dma_state      state;
    enum s3c_dma_loadst     load_state;
    struct s3c2410_dma_client    *client;
    /* channel configuration */
    unsigned long               dev_addr;
    unsigned long               load_timeout;
```

```
    unsigned int            flags;          /* channel flags */
    /* channel's hardware position and configuration */
    void __iomem            *regs;          /* channels registers */
    void __iomem            *addr_reg;      /* data address register */
    unsigned int            irq;            /* channel irq */
    unsigned long           dcon;           /* default value of DCON */

};
```

4.5 网络设备

网络设备是除了字符设备、块设备之外，Linux 的另一种重要设备模型。网络设备与其它设备的重要区别体现在他们使用了 net_device 等结构体描述设备。这些结构体中除了包含用于网络传输所必需的信息之外，还包括发送和接收队列，这些队列由内核进行管理，提供给更高层次的网络组件使用。

linux 的网络设备模型，可以用于各种不同网络机制的硬件，如 ATM、令牌环网络、以太网等。linux 系统将不同机制的网络硬件设备，抽象成统一的模型，使用同样的接口进行操作，甚至连打印机这样的设备也可以抽象成网络设备。本节将以 DM9000 这款以太网芯片为例，分析网络设备如何注册、初始化，以及怎样提供操作接口给上层的。

4.5.1 相关的数据结构

net_device

net_device 是网络设备模型最核心的结构，正是通过该结构，将不同的硬件设备抽象为统一的网络设备并提供操作接口的。

net_device 结构成员众多，这里并不列出所有成员，而是将成员分成以下几大类，由此可以理解 net_device 结构和作用。

1) 全局信息。包含了设备名字、状态、linux 的 device 结构体指针，指向所有网络设备的链表指针等。
2) 硬件信息，包含了相关设备的底层硬件信息。如共享内存的信息，设备寄存器的基地址，irq 信息、DMA 信息等。
3) 接口信息，与具体某个网络机制相关的信息。如数据包的大小、数据包头部的信息、网络地址等。这些信息通常由网络机制相关的 setup 函数设定，如果使用以太网接口，则 ether_setup()负责设定为符合以太网机制的值。
4) 设备方法，即网络设备的操作函数，如 open,stop,hard_start_xmit,do_ioctl 等。
5) 工具成员，包含一些用于查看设备的信息，其中有一个 void 指针 priv 指向设备特有的数据内容，如网卡信息等。

board_info

该结构是硬件特有的结构，不同的底层硬件可能会使用不同的结构。下面是 DM9000 网卡芯片使用的 board_info 结构体的定义。

```
typedef struct board_info {
    void __iomem *io_addr;      /* Register I/O base address */
    void __iomem *io_data;      /* Data I/O address */
```

```
    u16 irq;            /* IRQ */
    u16 tx_pkt_cnt;
    u16 queue_pkt_len;
    u16 queue_start_addr;
    u16 dbug_cnt;
    u8 io_mode;             /* 0:word, 2:byte */
    u8 phy_addr;
    void (*inblk)(void __iomem *port, void *data, int length);
    void (*outblk)(void __iomem *port, void *data, int length);
    void (*dumpblk)(void __iomem *port, int length);
    struct resource   *addr_res;    /* resources found */
    struct resource *data_res;
    struct resource   *addr_req;    /* resources requested */
    struct resource *data_req;
    struct resource *irq_res;
    struct timer_list timer;
    unsigned char srom[128];
    spinlock_t lock;
    struct mii_if_info mii;
    u32 msg_enable;
} board_info_t;
```

DM9000 的驱动程序中，net_device 结构包含的 priv（设备特有数据指针）即指向一个 board_info 的数据结构体。

4.5.2 DM9000 驱动注册

DM9000 设备是网卡设备对应的具体硬件，它的总线设定为 platform 总线，因此作为一个常规的 platform_device，其注册的流程与其他 platform_device 注册流程一致：首先注册 platform 设备，后期注册 platform 驱动。注册驱动的时候，调用 probe 函数进行初始化处理。与网络设备相关的注册和初始化工作，也都在 probe 函数内完成。

```
static int   dm9000_probe(struct platform_device *pdev)
{
    struct dm9000_plat_data *pdata = pdev->dev.platform_data;
    struct board_info *db; /* Point a board information structure */
    struct net_device *ndev;
    ...
    /* Init network device */
    ndev = alloc_etherdev(sizeof (struct board_info));
    /* setup board info structure */
    db = (struct board_info *) ndev->priv;
    memset(db, 0, sizeof (*db));
    spin_lock_init(&db->lock);
    if (pdev->num_resources < 2) {
        ret = -ENODEV;
```

```
                goto out;
        } else if (pdev->num_resources == 2) {
                base = ioremap(pdev->resource[0].start, pdev->resource[0].start - pdev->resource[0].start + 1);
                if (!request_mem_region(base, 4, ndev->name)) {
                        ret = -EBUSY;
                        goto out;
                }
                ndev->base_addr = base;
                ndev->irq = pdev->resource[1].start;
                db->io_addr = (void __iomem *)base;
                db->io_data = (void __iomem *)(base + 4);
                /* ensure at least we have a default set of IO routines */
                dm9000_set_io(db, 2);
        } else {
                ...
        }
        /* check to see if anything is being over-ridden */
        if (pdata != NULL) {
                if (pdata->flags & DM9000_PLATF_8BITONLY)
                        dm9000_set_io(db, 1);
                ....
        }
        dm9000_reset(db);
        /* try two times, DM9000 sometimes gets the first read wrong */
        for (i = 0; i < 2; i++) {
                id_val  = ior(db, DM9000_VIDL);
                ...
                if (id_val == DM9000_ID)
                        break;
                printk("%s: read wrong id 0x%08x\n", CARDNAME, id_val);
        }
        if (id_val != DM9000_ID) {
                ...
        }
        /* driver system function */
        ether_setup(ndev);
        ndev->open              = &dm9000_open;
        ndev->hard_start_xmit   = &dm9000_start_xmit;
        ndev->tx_timeout        = &dm9000_timeout;
        ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
        ndev->stop          = &dm9000_stop;
        ndev->set_multicast_list = &dm9000_hash_table;
#ifdef CONFIG_NET_POLL_CONTROLLER
        ndev->poll_controller   = &dm9000_poll_controller;
```

```
#endif
        db->msg_enable       = NETIF_MSG_LINK;
        ...
        db->mii.dev          = ndev;
        db->mii.mdio_read    = dm9000_phy_read;
        db->mii.mdio_write   = dm9000_phy_write;
        /* Read SROM content */
        for (i = 0; i < 64; i++)
                ((u16 *) db->srom)[i] = read_srom_word(db, i);
        unsigned char    enet_addr[6]={0x00, 0x22, 0x12, 0x34, 0x56, 0x90};
        for (i=0; i < 6; i++) {
                ndev->dev_addr[i] = enet_addr[i];
        }
        platform_set_drvdata(pdev, ndev);
        ret = register_netdev(ndev);
        return 0;
out:
        ...
        return ret;
}
```

上文省略了很多具体代码，但是保留了主要流程。注册过程中用到两个关键变量，ndev 指向一个 net_device 结构，db 指向一个 board_info 结构，ndev->priv=db。

1) 调用 alloc_etherdev 分配一个 net_device 结构体，同时也分配了一个 board_info 结构体。分配之后，按照以太网的要求(因为调用的是 alloc_etherdev)对数据结构进行初始化。其中 net_device 的名字为"eth%d"形式，发送/接受队列数量都是 1 个。
2) 根据 platform_device 中定义的 resource，给 DM9000 分配资源，并且据此设定 ndev，db 中的成员，如寄存器地址、irq 号码等。
3) 根据需要，设定 DM9000 的工作模式，然后 reset DM9000 芯片，读出芯片 ID，检验是否是 DM9000 的 ID。
4) 调用 ether_setup(ndev)初始化 ndev 的某些成员。

```
void ether_setup(struct net_device * dev)
        {
         dev->header_ops             = &eth_header_ops;
         dev->change_mtu             = eth_change_mtu;
         dev->set_mac_address        = eth_mac_addr;
         dev->validate_addr      = eth_validate_addr;
         dev->type           = ARPHRD_ETHER;
         dev->hard_header_len        = ETH_HLEN;
         dev->mtu            = ETH_DATA_LEN;
         dev->addr_len           = ETH_ALEN;
         dev->tx_queue_len       = 1000;     /* Ethernet wants good queues */
         dev->flags          = IFF_BROADCAST| IFF_MULTICAST;
         memset(dev->broadcast, 0xFF, ETH_ALEN);
        }
```

```
const struct header_ops eth_header_ops ____cacheline_aligned = {
        .create         = eth_header,
        .parse          = eth_header_parse,
        .rebuild    = eth_rebuild_header,
        .cache          = eth_header_cache,
        .cache_update   = eth_header_cache_update,
};
```

ndev 的相关成员都初始化成以太网相关的操作函数。

5) 设定 ndev 底层操作相关的函数，如 open，hard_start_xmit 等。
6) 设定 db 的底层信息和相关操作。
7) 设定 ndev->dev_addr[]网络地址，在以太网体系内即 MAC 地址。
8) 最后调用 register_netdev(ndev)将初始化好的 ndev 注册进入 linux 管理体系。

4.5.3 register_netdev 函数分析

register_netdev(struct net_device *dev)函数将初始化之后的 net_device 结构注册到 linux 管理体系中，此后就可以使用这个网络设备了。

该函数首先分配网络设备具体的名字，上节中提到，DM9000 的设备名字被指定为"eth%d"形式；到了这里，会具体的分配一个编号，使得 ndev 的名字变成"eth0"之类的。分配编号的原则是在该 net_device 所属的网络命名空间(net 结构)中，根据先后顺序依次分配编号。有的系统可能包含多个网络命名空间，也可能只有一个(被称为 init_net)。

分配了具体的名字后，会调用 register_netdevice(dev)完成剩下的注册工作。

```
int register_netdevice(struct net_device *dev)
{
        struct hlist_head *head;
        struct hlist_node *p;
        int ret;
        struct net *net;
        might_sleep();
        /* When net_device's are persistent, this will be fatal. */
        net = dev_net(dev);
        spin_lock_init(&dev->addr_list_lock);
        if (dev->init) {
                ret = dev->init(dev);
                if (ret) {
                        if (ret > 0)
                                ret = -EIO;
                        goto out;
                }
        }
        dev->ifindex = dev_new_index(net);
        if (dev->iflink == -1)
                dev->iflink = dev->ifindex;
        /* Check for existence of name */
```

```
        head = dev_name_hash(net, dev->name);
        hlist_for_each(p, head) {
                struct net_device *d
                        = hlist_entry(p, struct net_device, name_hlist);
                if (!strncmp(d->name, dev->name, IFNAMSIZ)) {
                        ret = -EEXIST;
                        goto err_uninit;
                }
        }
                if ((dev->features & NETIF_F_HW_CSUM) &&
               (dev->features & (NETIF_F_IP_CSUM|NETIF_F_IPV6_CSUM))) {
                printk(KERN_NOTICE "%s: mixed HW and IP checksum settings.\n",
                        dev->name);
                dev->features &= ~(NETIF_F_IP_CSUM|NETIF_F_IPV6_CSUM);
        }
        if ((dev->features & NETIF_F_NO_CSUM) &&
              (dev->features & (NETIF_F_HW_CSUM|NETIF_F_IP_CSUM|NETIF_F_IPV6_CSUM))) {
                printk(KERN_NOTICE "%s: mixed no checksumming and other settings.\n",
                        dev->name);
                dev->features &= ~(NETIF_F_IP_CSUM|NETIF_F_IPV6_CSUM|NETIF_F_HW_CSUM);
        }
        dev->features = netdev_fix_features(dev->features, dev->name);
        /* Enable software GSO if SG is supported. */
        if (dev->features & NETIF_F_SG)
                dev->features |= NETIF_F_GSO;
        netdev_initialize_kobject(dev);
        ret = netdev_register_kobject(dev);
        if (ret)
                goto err_uninit;
        dev->reg_state = NETREG_REGISTERED;
        set_bit(__LINK_STATE_PRESENT, &dev->state);
        dev_init_scheduler(dev);
        dev_hold(dev);
        list_netdevice(dev);
        /* Notify protocols, that a new device appeared. */
        ret = call_netdevice_notifiers(NETDEV_REGISTER, dev);
        ret = notifier_to_errno(ret);
        if (ret) {
                rollback_registered(dev);
                dev->reg_state = NETREG_UNREGISTERED;
        }
out:
        return ret;
err_uninit:
```

```
        if (dev->uninit)
            dev->uninit(dev);
        goto out;
}
```

上文中省略了很多代码，其主要工作是：

1) 初始化 ndev 中的 lock 等结构
2) 如果 ndev->init()存在，则调用它
3) 设定 ndev 的 ifindex,iflink 成员
4) 将 ndev 的名字存放到网络命名空间 net 的 hash 表中
5) 设定 ndev 的 features 成员
6) 调用 netdev_initialize_kobject，实际上是调用了 device_initialize 初始化了 ndev->dev 结构。
7) 调用 netdev_register_kobject 将初始化了的 dev 结构添加到管理体系，同时设定了 ndev->sysfs_groups，即与网络设备相关的属性等。
8) 调用 dev_init_scheduler 将 ndev 的 tx/rx 队列初始化，主要是指定队列操作函数为：

```
struct Qdisc noop_qdisc = {
        .enqueue =      noop_enqueue,
        .dequeue =      noop_dequeue,
        .flags          =   TCQ_F_BUILTIN,
        .ops        =   &noop_qdisc_ops,
        .list       =   LIST_HEAD_INIT(noop_qdisc.list),
        .requeue.lock =     __SPIN_LOCK_UNLOCKED(noop_qdisc.q.lock),
        .q.lock         =   __SPIN_LOCK_UNLOCKED(noop_qdisc.q.lock),
        .dev_queue  =   &noop_netdev_queue,
};
```

9) 调用 call_netdevice_notifiers 通知所有连接在 netdev_chain 上的 notifier 函数进行回调，通知的事件参数为 NETDEV_REGISTER。这些 notifier 函数有些是 fs_initcall 指定的网络协议初始化过程中注册到 netdev_chain 上的，有的则是其它 module_init 时注册的。都是通过调用 register_netdevice_notifier 完成注册 notifier 的。

5 总结

本文叙述了 linux 内核与设备驱动相关的数据结构，如何将各种设备/驱动纳入统一的管理体系，并且具体分析了几种常见的驱动模型。需要注意的是，虽然 linux 管理设备和驱动都使用了 kobject 等核心结构，从概念上看，“设备”必定通过某种“总线”与系统相连，而驱动则与这些“总线”相关联来控制某个“设备”。而文中描述的几种驱动模型，与具体的总线无关，仅是根据设备提供服务的方式进行分类，为应用程序访问（其实是系统调用）提供了某几类接口而已。具体某个硬件设备到底用哪种驱动模型，可以根据需要选择。

本文将设备/驱动这一层次单独拿出来进行分析，主要集中在设备/驱动如何初始化以及注册进入 linux 的管理体系上。而关于运行中如何使用驱动程序来操作设备，描述的较少，这是因为通常驱动程序并非由内核直接使用，而往往是由用户程序发出系统调用之后才使用到驱动程序的。系统调用的过程中，涉及到 linux 的另一个非常重要且复杂的概念——虚拟文件系统(VFS)，正是通过这个系统，linux 实现了“一切都是文件”的管理方式。在 VFS 的分析文档中，将会再次涉及到一些设备驱动的分析，并且会详细分析驱动程序是如何运行的。