嵌入式 Linux 操作系统 第六讲 Linux 内核与驱动

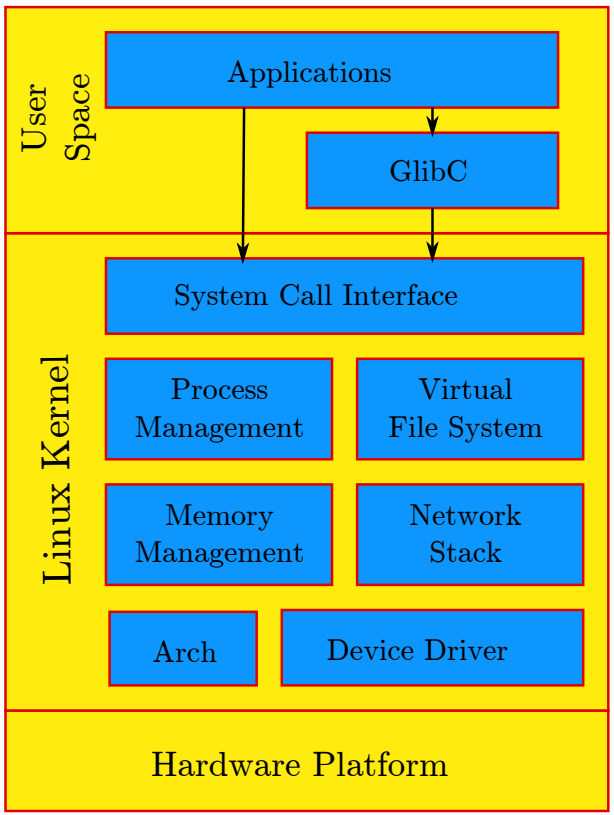
杨延军
yangyj.ee@gmail.com

北京大学

2016 年

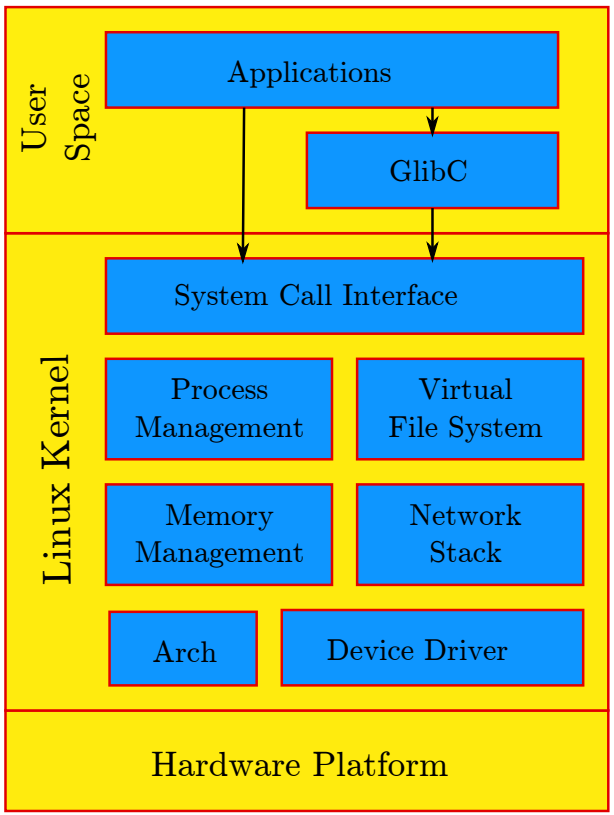
主要内容

Linux 内核的基本功能



- 硬件的设备驱动
- 虚拟内存的管理和分配
- 多任务的管理和调度
- 网络协议和安全
- 文件与用户权限的管理
- 提供给程序员的统一编程界面

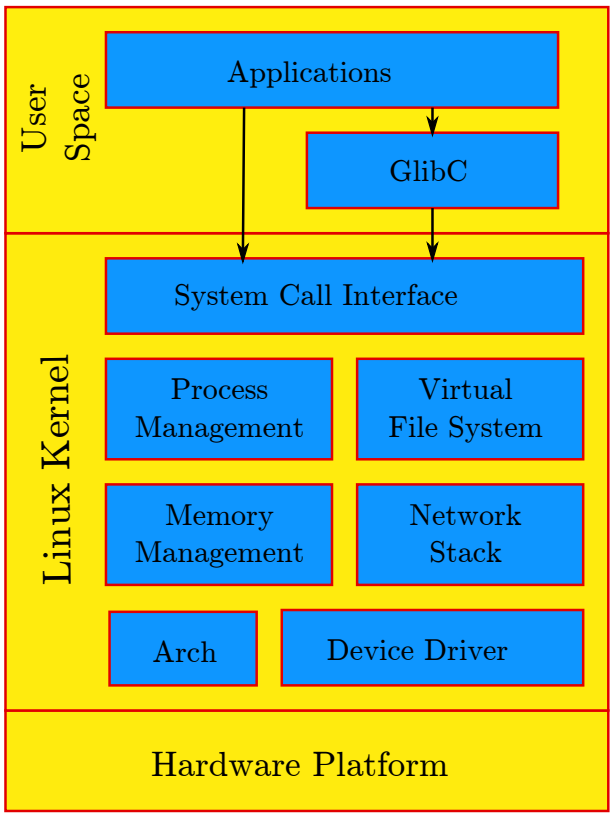
Linux 内核的基本功能



内存管理

- 虚拟地址：三（32 位系统）级页表
- Buddy 内存分配系统，避免外碎片
- Slab 算法，减少内碎片，提高内存使用效率
- 0xc0000000：内核空间与用户空间的分界

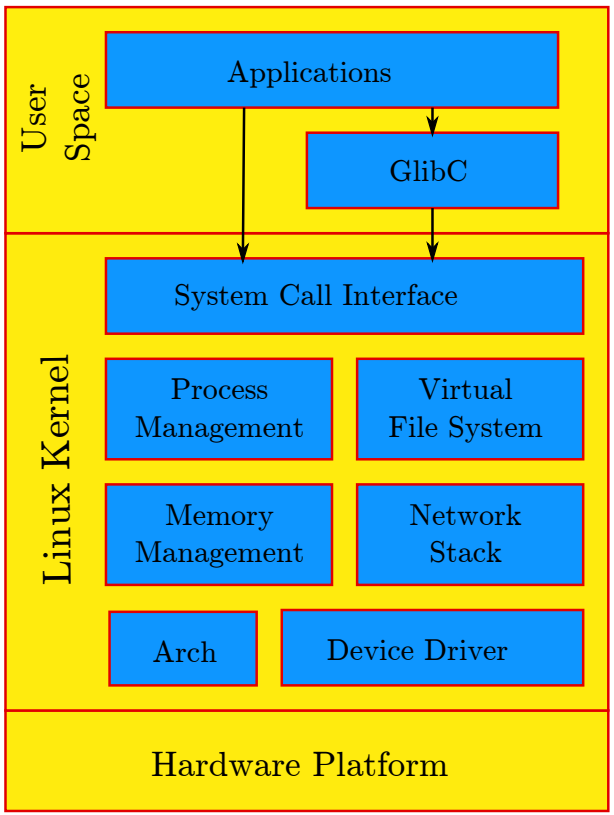
Linux 内核的基本功能



进程管理

- 调度策略：动态优先级，抢占式调度
- O(1) 的进程调度算法
- 使用 fork 类系统调用建立进程
- 进程建立开销低

Linux 内核的基本功能



设备管理

- 设备文件和虚拟文件系统
- 设备类型：字符设备、块设备、网络设备
- 设备命名系统
  - 主设备号和次设备号
  - Devfs
  - udev

Linux 内核版本

- 源码下载地址 http://kernel.org/
- 版本号：a.b.c(.x)
  - a.b：主版本号。其中如果 b 是奇数，表示开发版，否则是稳定版
  - c：发行号，x：2.6 内核引入
  - http://wiki.kernelnewbiew.org/LinuxChanges 介绍了不同版本之间的主要区别
  - rc：测试发行
- 版本 2.6.35.11，2.6.39，3.0.1，3.2

内核补丁

- 升级内核采用的补丁
  - 从 2.6.10 内核到 2.6.11 内核使用 patch-2.6.11.tar.gz 作为补丁
- 特定的内核版本补丁
  - 个人推出的补丁（mm，ac）
  - 特定体系的补丁（at91）
- Linux 补丁的应用次序

应用内核补丁实例

- 按照正确的顺序应用补丁

```
cd linux-2.6.10/
bzcat ../patch-2.6.11.bz2 | patch -p1
bzcat ../patch-2.6.11.12.bz2 | patch -p1
cd ..
mv linux-2.6.10 linux-2.6.11.12
```

补丁实例

```
diff -Nru a/Makefile b/Makefile
--- a/Makefile  2005-03-04 09:27:15 -08:00
+++ b/Makefile  2005-03-04 09:27:15 -08:00
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
-EXTRAVERSION =
+EXTRAVERSION = .12
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

内核源码结构

- init 初始化代码
- kernel 核心代码
- arch 平台相关代码（arm，mips 等）
- include 内核头文件
- drivers 驱动程序
- fs 文件系统代码
- mm 内存管理代码
- net 网络协议相关代码（非驱动）
- Documentation 内核文档资料

编译 Linux 内核

- make menuconfig 内核配置
  - config，xconfig，oldconfig，defconfig
  - 配置文件保存为.config 文件
- make dep 生成内核依赖文件目录（2.4）
- make zImage 生成内核映象
  - make bzImage
  - make modules; make modules_install
  - 内核保存位置：arch/arm/boot/zImage
- make clean/mrproper 清理代码

内核配置

- 内核剪裁系统
  - 内核功能单元可以配置成模块或者直接配置到内核中
- 配置选项
  - < > 表示没有编译到内核
  - <M> 表示作为模块
  - <*> 表示静态链接到内核
  - help 按钮可以作为最直接帮助信息来源

编译生成的文件

- vmlinux 未压缩的 Linux 内核
- zImage zlib 压缩的内核映象
- bzImage (bz 代表 big zImage)
- System.map 内核符号表
- 内核模块安装位置 /lib/modules/<version>/
- 内核模块依赖关系 modules.dep
- 内核模块符号表 modules.symbols

内核安装

主机系统

- 拷贝 bzImage 到 /boot 目录
- 修改启动管理器的配置
- (重新安装启动管理器)

嵌入式系统

- 把 zImage 烧入到 Flash 中
- 把 zImage 拷贝到 SD 卡中
- 调试时可以把 zImage/uImage 放到主机的 tftp 服务器上

主要内容

主要内容

驱动程序分类

- 字符设备驱动
- 块设备驱动
- 网络接口设备
- 其他类别
  - USB
  - pcmcia
- 内核驱动模型

设备文件

- 保存在 /dev 目录下

```
brw-rw---- 1 root disk 8, 1 Feb 24   sda1
```

- 主设备号与从设备号
- udev 动态建立设备文件的策略

内核驱动编程的基本原则

- 尽量不使用浮点数
- 一般定义符号为 static
- 不能使用用户空间的函数
- 尽量节省资源、减少运行时间
- 小心操作内核的数据结构
- 一般不要引入使用策略
- 注意并发问题
- Documentation/CodingStyle

主要内容

最简单的驱动示例

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

static int __init hello_init(void)
{
        printk(KERN_ALERT "Hello, world\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, cruel world\n");
}
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("Greeting module");

module_init(hello_init);
module_exit(hello_exit);
```

编译模块

2.6 版本以上的内核可以使用如下 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o
else
    KERNELDIR = /home/embedded/linux-2.6.33
    PWD := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
        rm -rf *.o *.ko *.mod.c .tmp_versions .depend .*.cmd core
endif
```

模块操作

- insmod 加载模块
- rmmod 卸载模块，模块名称作为参数
- lsmod 列出已经加载的模块
- modprobe 智能模块加载
- 模块参数 insmod newvalue=0x4000000

主要内容

字符设备驱动的基本框架

初始化

- 在内核中注册设备号，静态/动态
- 初始化 cdev 结构体
- 在内核中注册设备

注销

- 设备 cdev 结构体的注销
- 设备号的注销

设备编号

- 设备号是一个 dev_t 结构，一般由下面的宏生成

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

- 注册设备号

```
int register_chrdev_region(dev_t first,
    unsigned int count, char *name);
```

- 注销设备号

```
void unregister_chrdev_region(dev_t first,
    unsigned int count);
```

cdev 结构体

- cdev 结构体的定义

```
#include <linux/cdev.h>
static struct cdev *acme_cdev;
```

- 初始化 cdev 结构体

```
acme_cdev = cdev_alloc();
acme_cdev->ops = &acme_fops;
acme_cdev->owner = THIS_MODULE;
```

cdev 结构注册

- 注册

```
int cdev_add(
    struct cdev *p, /* cdev结构 */
    dev_t dev,      /* 起始设备号 */
    unsigned count); /* 设备个数 */
```

- 注销

```
void cdev_del(struct cdev *p);
```

file_operations 结构体 (简化)

```
struct file_operations {
    struct module *owner;     // 文件的属主
    loff_t (*llseek) // 修改文件的当前读写位置
    ssize_t (*read)  // 文件读操作
    ssize_t (*write) // 文件写操作
    unsigned int (*poll) // 查询文件读写状态
    long (*unlocked_ioctl)    // 执行设备特定命令
    int (*mmap)      // 映射设备内存到进程地址空间
    int (*open)      // 打开文件
    int (*flush)     // 清缓冲区
    int (*release)   // 关闭文件
    int (*fsync)     // 刷新待处理的数据
    ......
```

读写操作

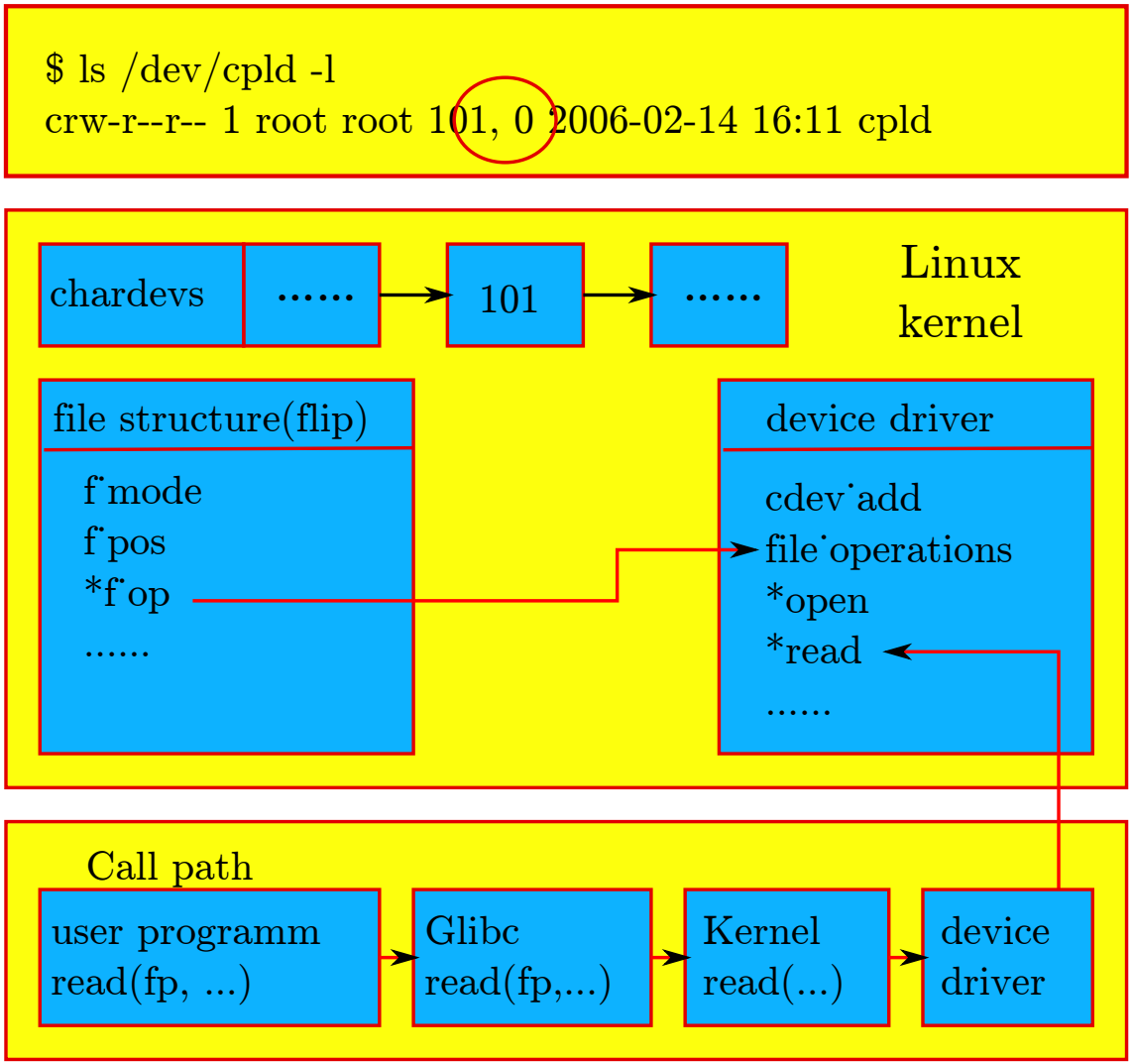
- 用户空间读取设备文件

```
ssize_t (*read) (
    struct file *, /* 文件描述符 */
    char *,        /* 用户空间缓冲区 */
    size_t,        /* 缓冲区大小 */
    loff_t *);     /* 文件偏移 */
```

- 用户空间写设备文件

```
ssize_t (*write) ( struct file *,
    const char *, /* 用户空间缓冲区 */
size_t, loff_t *);
```

字符设备驱动的工作原理



主要内容

用户空间数据访问

- 把用户数据拷贝到内核 copy_from_user()
- 把内核数据拷贝给用户空间 copy_to_user()

访问物理地址

- 由于内核同样采用虚拟地址，所以对物理地址的访问必须重新映射

```
cpld_base = ioremap(0x08000000, 0x01000);
*((unsigned long *)(cpld_base + 0x28))
        = DLED_VALUE;
```

- 启动时定义的页表

arch/arm/mach-pxa/xxxx.c 文件中

```
static struct map_desc xxxx_io_desc[]
    __initdata = {
```

内存申请

- kmalloc 函数
  - 申请到的内存并不会初始化
  - 物理上连续内存
  - 实际申请的内存会比需要的多一点，而且总是 2 的整数次幂。
- kfree 释放申请的内存
- 获取多页内存

阻塞 IO 的处理

- 为什么采用阻塞方式？
- 内核中的等待队列
- 涉及到的内核函数：

```
down_interruptible()
wait_event_interruptible()
wake_up_interruptible()
```

中断的处理

- 申请中断号 - request_irq()
- 释放中断号 - free_irq()
- 中断处理时间要尽量短
- 顶半部和低半部
  tasklet 和 workqueue

```
DECLARE_TASKLET (module_tasklet,   /* name */
                 module_do_tasklet, /* function */
                 0);                /* data */
tasklet_schedule(&module_do_tasklet);
```

显示内核消息

- printk 函数与 printf 类似，不支持浮点数
- 日志记录八个级别
- 可以写入/proc/sys/kernel/printk 文件来改变
- dmesg 命令
- 内核参数 console
- 循环缓冲区

kgdb

- 类似 gdb 的调试工具，是内核的一部分
- 通过串口和调试机相连
- 调试机运行 gdb 连接被调试机，可以执行常见的各种调试手段。

主要内容

dled 驱动分析 I

```
#include <linux/module.h>   /* Specifically, a module */
#include <linux/init.h>
#include <linux/fs.h>       /* The character device
            * definitions are here */
#include <linux/cdev.h>

#include "cpld.h"

#define CPLD_BASE 0xfee70010

// static unsigned long cpld_virtual;

static struct cdev *my_cdev;

static int cpld_open (struct inode *inode, struct file *filp)
{
  return 0;
}
```

dled 驱动分析 II

```
static int cpld_release (struct inode *inode, struct file *filp)
{
  return 0;
}

static int cpld_ioctl(struct inode *inode, struct file *file,
    unsigned int ioctl_num, unsigned long ioctl_param)
{
  switch (ioctl_num) {
    case IOCTL_SET_VALUE:
      *(unsigned char *)(CPLD_BASE) = ioctl_param & 0xff;
      break;
  }
  return 0;
}

struct file_operations cpld_ops = {
  .owner    = THIS_MODULE,
  .ioctl    = cpld_ioctl,    /* ioctl */
  .open     = cpld_open,
```

dled 驱动分析 III

```
  .release = cpld_release  /* a.k.a. close */
};

static int __devinit cpld_init(void)
{
  int result;
  dev_t id;

  id = MKDEV(MAJOR_NUM, 0);
  result = register_chrdev_region(id, 1, DEVICE_NAME);
  if (result < 0) {
    printk("cpld: unable to get a major %d. :(\n", MAJOR_NUM);
    return result;
  }
  my_cdev = cdev_alloc();
  my_cdev->ops = &cpld_ops;
  my_cdev->owner = THIS_MODULE;
  result = cdev_add(my_cdev, id, 1);
  if (result < 0) {
    printk("cpld: unable to add the device.\n");
```

dled 驱动分析 IV

```
    unregister_chrdev_region(MKDEV(MAJOR_NUM, 0), 1);
    return result;
  }
  return 0;
}

static void __devexit cpld_cleanup(void)
{
  cdev_del(my_cdev);
  unregister_chrdev_region(MKDEV(MAJOR_NUM, 0), 1);
}
module_init(cpld_init);
module_exit(cpld_cleanup);
MODULE_LICENSE("GPL");
```

主要内容

pci 驱动分析 I

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Basic Driver for DE2i_150");
MODULE_AUTHOR("Pluto Yang"); /* based on Patrick Schaumont's work */

//-- Hardware Handles

static void __iomem *hexport;    // handle to 32-bit output PIO
static void __iomem *inport;    // handle to 18-bit input PIO
static void __iomem *key;
static void __iomem *ctrlmem;
static u8 myirq;
```

pci 驱动分析 II

```
//-- Char Driver Interface
static int MAJOR_NUMBER = 91;
const int KEY_IRQ_MASK = 1UL << 1;

static int char_device_open(struct inode *, struct file *);
static int char_device_release(struct inode *, struct file *);
static ssize_t char_device_read(struct file *, char *, size_t, loff_t *);
static ssize_t char_device_write(struct file *, const char *, size_t, loff_t *);

static struct file_operations file_opts = {
  .read = char_device_read,
  .open = char_device_open,
  .write = char_device_write,
  .release = char_device_release
};

struct pio_reg {
  u32   data;
  u32   direction;
```

pci 驱动分析 III

```
  u32   interruptmask;
  u32  edgecapture;
};

static int is_my_isr()
{
  return (ioread32(ctrlmem + 0x40) & KEY_IRQ_MASK);
}

static void set_pio_irq(bool enable)
{
  u32 val;
  struct pio_reg *reg = key;
  val = ioread32(ctrlmem + 0x50);
  if (enable) {
    val |= (KEY_IRQ_MASK);
    reg->interruptmask |= 0xf;
  }
  else {
    val &= ~KEY_IRQ_MASK;
```

pci 驱动分析 IV

```
    reg->interruptmask &= ~0xf;
  }
  iowrite32(val, ctrlmem + 0x50);
}

static irqreturn_t button_isr(int irq, void *arg)
{
  struct pio_reg *reg = key;
  if (!is_my_isr()) return IRQ_NONE;
  reg->edgecapture = 0xff; // clear edgecapture bits;
  printk(KERN_INFO "Key pressed: 0x%x.\n", *(int*)key);
  return IRQ_HANDLED;
}

static int char_device_open(struct inode *inodep, struct file *filep)
{
  int retval;
 retval = request_irq(myirq, button_isr, IRQF_SHARED, "altera_basic", &myirq);
  if (retval) {
   printk(KERN_ALERT "altera_driver: device open failed(%d), irq is %d.\n", retval
```

pci 驱动分析 V

```
    return retval;
  }
  set_pio_irq(true);
  return 0;
}

static int char_device_release(struct inode *inodep, struct file *filep)
{
  set_pio_irq(false);
  free_irq(myirq, &myirq);
  printk(KERN_ALERT "altera_driver: device closed.\n");
  return 0;
}

static ssize_t char_device_read(struct file *filep, char *buf, size_t len,
        loff_t * off)
{
  int switches;
  int count = len;
  //  printk(KERN_ALERT "altera_driver: read %d bytes\n", len);
```

pci 驱动分析 VI

```
  while (count > 0) {
    switches = ioread32(inport);
    put_user(switches, (int *)buf);
    count -= 4;
    buf += 4;
  }
  return len;
}

static ssize_t char_device_write(struct file *filep, const char *buf,
         size_t len, loff_t * off)
{
  char *ptr = (char *)buf;
  int b = 0;
  //  printk(KERN_ALERT "altera_driver: write %d bytes\n", len);
  while (b < len) {
    unsigned int k;
    get_user(k, (int *)ptr);
    ptr += 4;
    b += 4;
```

pci 驱动分析 VII

```
    iowrite32(k, hexport);
  }
  return len;
}

//-- PCI Device Interface

static struct pci_device_id pci_ids[] = {
  {PCI_DEVICE(0x1172, 0x0004),},
  {0,}
};

MODULE_DEVICE_TABLE(pci, pci_ids);

static int pci_probe(struct pci_dev *dev, const struct pci_device_id *id);
static void pci_remove(struct pci_dev *dev);

static struct pci_driver pci_driver = {
  .name = "altera_basic",
  .id_table = pci_ids,
```

pci 驱动分析 VIII

```
  .probe = pci_probe,
  .remove = pci_remove,
};

static int pci_probe(struct pci_dev *dev, const struct pci_device_id *id)
{
  int retval;
  unsigned long resource;
  u8 revision;

  retval = pci_enable_device(dev);

  pci_read_config_byte(dev, PCI_REVISION_ID, &revision);
  if (revision != 0x01) {
    printk(KERN_ALERT "altera_driver: cannot find pci device\n");
    return -ENODEV;
  }

  myirq = dev->irq;
```

pci 驱动分析 IX

```
  resource = pci_resource_start(dev, 0);
  printk(KERN_ALERT "altera_driver: Resource start at bar 0: 0x%lx\n",
    resource);

  inport = ioremap(resource + 0xC020, 0x10);
  key = ioremap(resource + 0xC030, 0x10);
  hexport = ioremap(resource + 0xC040, 0x10);
  ctrlmem = ioremap(resource + 0x8000, 0x4000);

  return 0;
}

static void pci_remove(struct pci_dev *dev)
{
  iounmap(hexport);
  iounmap(inport);
  iounmap(key);
  iounmap(ctrlmem);
}
```

pci 驱动分析 X

```
//-- Global module registration
static int __init altera_driver_init(void)
{
  int t = register_chrdev(MAJOR_NUMBER, "de2i150_altera", &file_opts);
  if (t < 0) {
    printk(KERN_ALERT "altera_driver: cannot register chrdev.\n");
    return t;
  }
  t = pci_register_driver(&pci_driver);

  if (t < 0) {
    printk(KERN_ALERT "altera_driver: cannot register pci.\n");
    unregister_chrdev(MAJOR_NUMBER, "de2i150_altera");
  }
  else
    printk(KERN_INFO
      "altera_driver: char+pci drivers registered.\n");

  return t;
}
```

pci 驱动分析 XI

```
static void __exit altera_driver_exit(void)
{
  printk(KERN_ALERT "Goodbye from de2i150_altera.\n");

  unregister_chrdev(MAJOR_NUMBER, "de2i150_altera");
  pci_unregister_driver(&pci_driver);
}

module_init(altera_driver_init);
module_exit(altera_driver_exit);
```

参考书籍

- 《现代操作系统》Andrew S. Tannenbaum
- 内核文档 Documentation 目录
- 《Linux Device Driver》3rd
- 《Understanding Linux Kernel》3nd
- 《Linux Kernel Development》3nd
- 内核代码