“菜鸟玩转嵌入式”视频培训讲座
— 专题篇

主办：上海申嵌信息科技有限公司
承办：嵌入式家园
协办：上海嵌入式家园-开发板商城
广州友善之臂计算机科技有限公司
主讲：贺光辉（嵌入式系统工程师）

嵌入式家园 www.embedclub.com

上海嵌入式家园-开发板商城 http://embedclub.taobao.com/

上章回顾

- Linux设备驱动的简介，以及分类
  - 字符设备
  - 块设备
  - 网络接口
- 模块的应用
  - 如何编写模块
  - 模块相关的宏
  - 模块和应用程序的区别
  - 编译和装载内核模块

第二章

字符设备驱动程序

主讲：贺光辉

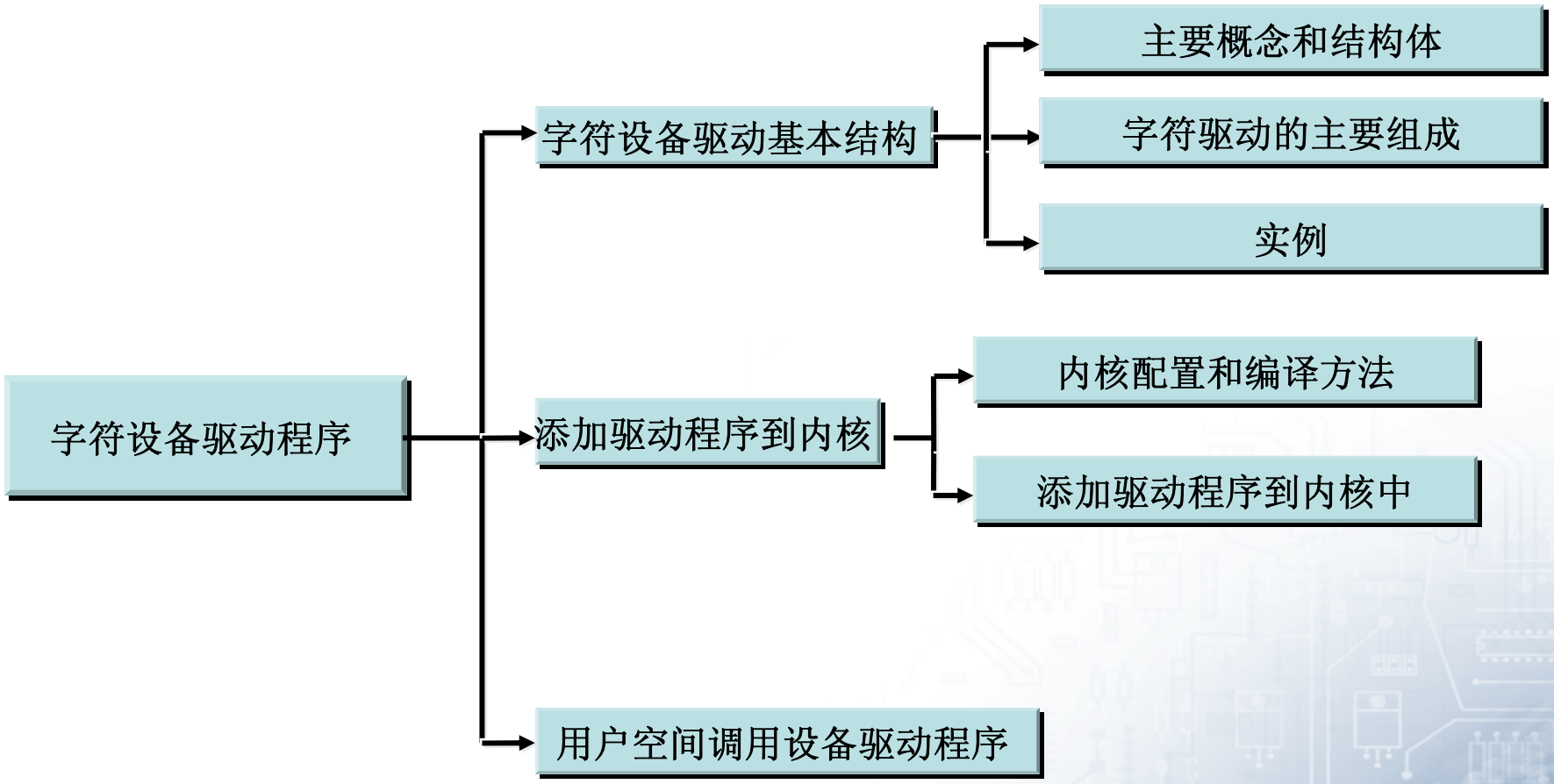
预习检查

- 字符设备有哪两个设备号？
- 字符设备驱动程序的基本成员函数有哪些？

本章目标

- 掌握字符设备驱动程序的基本结构和开发方法
- 掌握用户空间调用设备驱动程序的方法

本章结构



- 字符设备开发的基本步骤
  - 确定主设备号和次设备号
  - 实现字符驱动程序
    - 实现file_operations结构体
    - 实现初始化函数，注册字符设备
    - 实现销毁函数，释放字符设备
    - 实现字符设备其他基本成员函数
  - 创建设备文件节点

什么是主设备号/次设备号

- 主设备号是内核识别一个设备的标识。
  - 整数(占12bits)，范围从0到4095，通常使用1到255
- 次设备号由内核使用，用于正确确定设备文件所指的设备。
  - 整数(占20bits)，范围从0到1048575，一般使用0到255

设备编号的内部表达

- dev_t类型(32位）：
  - 用来保存设备编号(包括主设备号(12位)和次设备号(20位))
- 从dev_t获得主设备号和次设备号：
  - MAJOR(dev_t);
  - MINOR(dev_t);
- 将主设备号和次设备号转换成dev_t类型：
  - MKDEV(int major, int minor);

分配主设备号

- 手工分配主设备号：找一个内核没有使用的主设备号来使用。

```
#include <linux/fs.h>

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

要分配的设备编号范围的起始值，次设备号经常为0

所请求的连续设备编号的个数

和该编号范围关联的设备名称

2-1 字符设备驱动程序基本结构

- 动态分配设备号

输出的设备号

```
#include <linux/fs.h>
int  alloc_chrdev_region(dev_t *dev，  unsigned int firstminor，
                         unsigned int count，  char *name)；
```

要使用的被请求的
第一个次设备号

- 释放设备号

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

通常在模块的清除函数中调用。

- 实现字符驱动程序
  - cdev 结构体

```
struct cdev
{
        struct kobject kobj;            /* 内嵌的kobject 对象 */
        struct module *owner;           /*所属模块*/
        struct file_operations *ops;    /*文件操作结构体*/
        struct list_head list;
        dev_t dev;                      /*设备号*/
        unsigned int count;
};
```

操作cdev的函数

```
void cdev_init( struct cdev *cdev, struc t file_operations * fops);

struct cdev *cdev_alloc(void) ;

int cdev_add(struct cdev *dev, dev_t num, unsigr

void cdev_del(struct cdev *cc
```

用于初始化已分配的cdev结构，并建立cdev和ons之间的连接

函数用于动态申请一个cdev 内存

分别向系统删除一个cdev，完成字符设备的注销，通常在模块的卸载函数

分别向系统添加一个cdev，完成字符设备的注册,通常在模块加载函数中调用

file_operations 结构体

- 字符驱动和内核的接口：
  - 在include/linux/fs.h定义
- 字符驱动只要实现一个file_operations结构体
- 当注册到内核中，内核就有了操作此设备的能力。

2-1 字符设备驱动程序基本结构

- **file_operations的主要成员：**
  - **struct module *owner:** 指向模块自身：THIS_MODULE
  - **open**：打开设备
  - **release**：关闭设备
  - **read**：从设备上读数据
  - **write**：向设备上写数据
  - **ioctl**：I/O控制函数
  - **llseek**：定位当前读/写位置指针
  - **mmap**：映射设备空间到进程的地址空间

2-1 字符设备驱动程序基本结构

- **file 结构体**
  - **file结构：**
    - file_operations结构相关的一个结构体。
    - 描述一个正在打开的设备文件。
  - 成员：
    - **loff_t f_pos:**
      - 当前读/写位置
    - **unsigned int f_flags**
      - 标识文件打开时，是否可读或可写
      - O_RDONLY
      - O_NONBLOCK
    - **struct file_operations *f_op**
      - 文件相关的操作，指向所实现的struct file_operations
    - **void *private_data:**
      - 私有数据指针。驱动程序可以将这个字段用于任何目的或者忽略这个字段。

inode 结构体

- 内核用inode结构在内部表示文件
- inode与file的区别
  - file表示打开的文件描述符
  - 多个表示打开的文件描述符的file结构，可以指向单个inode结构。

2-1 字符设备驱动程序基本结构

- **inode结构中的两个主要字段：**
  - **dev_t i_rdev；**
    - 对表示设备文件的inode结构，该字段包含了真正的设备编号。
  - **struct cdev *i_cdev；**
    - struct cdev是表示字符设备的内核的内部结构。
    - 当inode指向一个字符设备文件时，该字段包含了指向struct cdev结构的指针
- **从一个inode中获得主设备号和次设备号：**

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

2-1 字符设备驱动程序基本结构

- **注册设备**,在模块或驱动初始化时调用
  - Linux-2.4 及之前

```
int register_chrdev(unsigned int major,
                    const char *name,
                    struct file_operations *fops)
```

如何操作字符设备的接口

  - Linux-2.6

```
void cdev_init( struct cdev *dev, struc t file_operations *fops);

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

- 注销设备:在模块卸载时调用
  - Linux-2.4及之前

```
int unregister_chrdev(unsigned int major,
                      const char *name);
```

  - Linux-2.6

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

2-1 字符设备驱动程序基本结构


```
/ /设备驱动模块加载函数
static int __init xxx_init(void)
 {
    ...
    cdev_init(&xxx_dev.cdev, &xxx_fops); / /初始化cdev
    xxx_dev.cdev.owner = THIS_MODULE;
    / /获取字符设备号
    if (xxx_major)
    {
            register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
    }
    else
    {
            alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
    }
    ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1 ) ; / /注册设备
    ...
}
```

2-1 字符设备驱动程序基本结构


```
/*设备驱动模块卸载函数*/
 static void __exit xxx_exit(void)
 {
    unregister_chrdev_region(xxx_dev_no, 1); / /释放占用的设备号

    cdev_del(&xxx_dev.cdev); / /注销设备
    ...
 }
```

- 打开

```
int xxx_open(struct inode *inode, struct file *filp) ;
```

  - 模块使用计数加1
  - 识别次设备号
  - 硬件操作：
    - 检查设备相关错误（诸如设备未就绪或类似的硬件问题）；
    - 如果设备是首次打开，则对其初始化；
    - 如果有中断操作，申请中断处理程序；

- 关闭

```
int xxx_release(struct inode *inode, struct file *filp) ;
```

  - 模块使用计数减1
  - 释放由open分配的，保存在filp>private_data里的所有内容。
  - 硬件操作:
    - 如果申请了中断，则释放中断处理程序。
    - 在最后一次关闭操作时关闭设备。

2-1 字符设备驱动程序基本结构

- **read/write**

指向用户空间的缓冲区，这个缓冲区或者保存将写入的数据，或者是一个存放新读入数据的空缓冲区。

```
ssize_t read(struct file *filp, char __user *buff, size_t  count, loff_t *offp);

ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
```

用户在文件中存取操作的位置

2-1 字符设备驱动程序基本结构

- 用户空间和内核空间之间的数据拷贝过程，
  - 不能简单的用指针操作或者memcpy来进行数据拷贝
    - 用户空间的数据是可以被换出的，会产生一个页面失效异常。
    - 用户空间的地址无法在内核空间中使用。
- 用户空间和内核空间之间进行数据拷贝的函数：

```
unsigned long copy_from_user(void *to,  const void __user *from,
                                        unsigned long count) ;
unsigned long copy_to_user(void __user *to,  const void *from,
                                      unsigned long count );
```

  - 如果要复制的内存是简单类型，如char、int、long 等，
    - put_user()和get_user()

2-1 字符设备驱动程序基本结构

- 读设备模板

```
ssize_t xxx_read(struct file *filp, char __user *buf,
                                 size_t count ,loff_t*f_pos)
{
    ...
    copy_to_user(buf, ..., ... );
    ...
}
```

- 写设备模板

```
ssize_t xxx_write(struct file *fil p, const char __user *buf ,
                                 size_t count ,loff_t *f_pos)
{
    ...
    copy_from_user(..., buf, ... );
    ...
}
```

- **ioctl函数**
  - 为设备驱动程序执行“命令”提供了一个特有的入口点。
  - 用来设置或者读取设备的属性信息。

```
int    ioctl (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg);
```

事先定义的IO控制命令 代码

arg为对应于cmd命令的参数

2-1 字符设备驱动程序基本结构

- cmd 参数的定义
  - 不推荐用0x1，0x2，0x3之类的值
  - Linux对ioctl()的cmd参数有特殊的定义

设备类型（type）	序列号（number）	方向（direction）	数据尺寸（size）
8bit	8bit	2bit	13/14bit

  - 构造命令编号的宏：
    - _IO(type，nr)用于构造无参数的命令编号；
    - _IOR(type，nr，datatype)用于构造从驱动程序中读取数据的命令编号；
    - _IOW(type，nr，datatype)用于写入数据的命令；
    - _IOWR(type，nr，datatype)用于双向传输。
      - type和number位字段通过参数传入，而size位字段通过对datatype参数取sizeof获得。

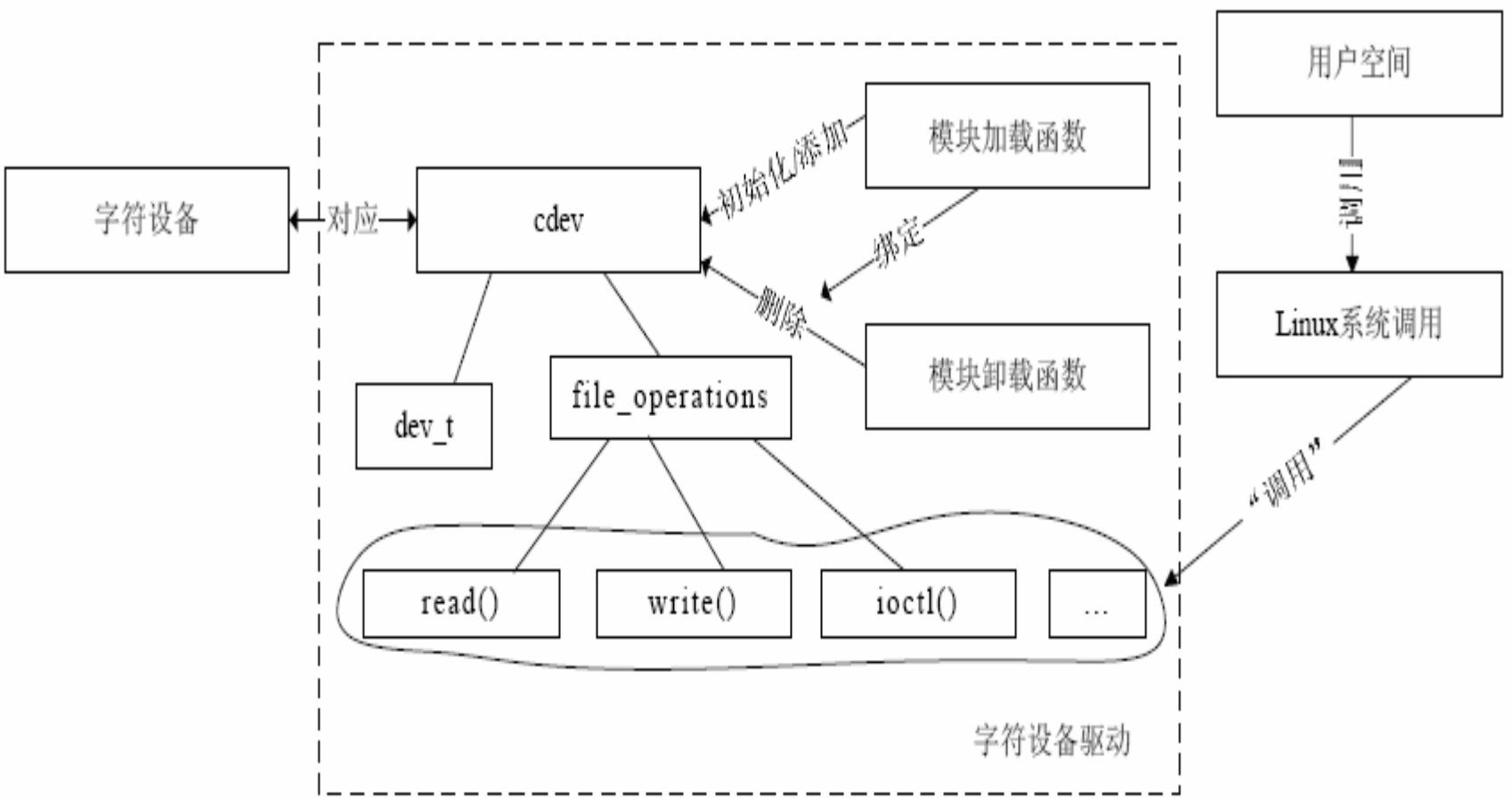
2-1 字符设备驱动程序基本结构

- **ioctl函数模板**

```
int xxx_ioctl( struct inode *inode, struct f ile *filp, unsigned int cmd,
                unsigned long arg)
{
    ...
    switch (cmd)
    {
        case XXX_CMD1:
                ...
                break;
        case XXX_CMD2:
                ...
                break;
        default:  ///*不能支持的命令 */
                return - EINVAL;
    }
    return 0;
}
```

阶段总结

- 本节介绍了字符设备驱动结构



2-2 内核重要头文件目录

- **Linux-2.6.29内核重要头文件目录：**
  - linux-2.6.29/arch/arm/mach-s3c2410/include/mach/regs-gpio.h
  - linux-2.6.29/arch/arm/mach-s3c2410/include/mach/hardware.h
  - linux-2.6.29/arch/arm/include/asm/io.h

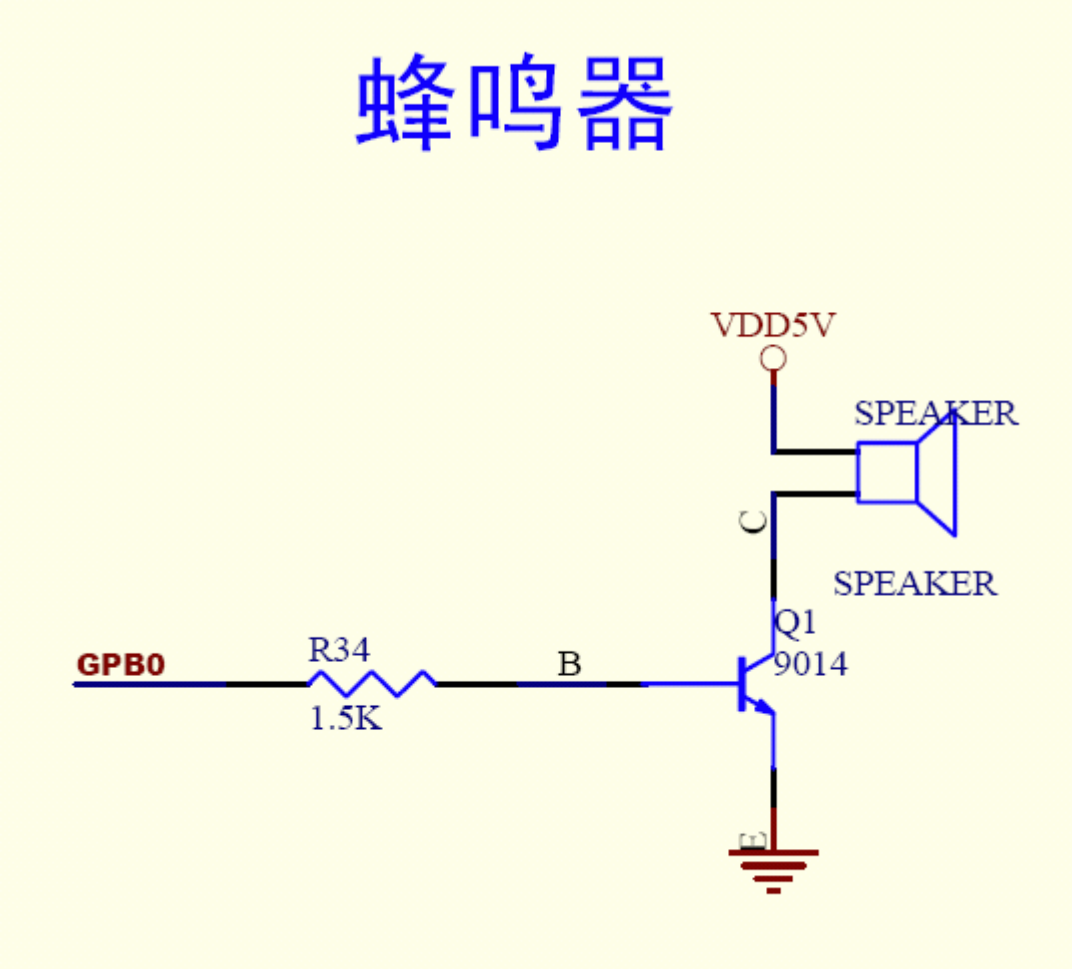
```
asm   --- linux-2.6.29/arch/arm/include/asm
mach --- linux-2.6.29/arch/arm/mach-s3c2410/include/mach
plat   --- linux-2.6.29/arch/arm/plat-s3c24xx/include/plat
           linux-2.6.29/arch/arm/plat/include/plat
```

2-2 内核重要头文件目录

- **Linux-2.6.32.2内核重要头文件目录：**
  - linux-2.6.32.2/arch/arm/mach-s3c2410/include/mach/regs-gpio.h
  - linux-2.6.32.2/arch/arm/mach-s3c2410/include/mach/gpio-nrs.h
  - linux-2.6. 32.2/arch/arm/plat-s3c24xx/gpio.c
  - linux-2.6. 32.2/include/linux/asm-generic/io.h
  - linux-2.6. 32.2/include/linux/wait.h

```
asm   ---  linux-2.6.32.2/include/linux/asm-generic
mach ---  linux-2.6.32.2/arch/arm/mach-s3c2410/include/mach
plat   --- linux-2.6.32.2/arch/arm/plat-s3c24xx/include/plat
           linux-2.6.32.2/arch/arm/plat-s3c/include/plat
```

- BEEP驱动程序实例讲解



2-4 用户空间调用设备驱动程序

- 创建设备节点
  - $mknod /dev/node_name c major minor
- 示例代码

```
int main(void)
{
        int dev_fd;
        char read_buf[10];

        dev_fd = open("/dev/node_name",O_RDWR | O_NONBLOCK);
        if ( dev_fd == -1 ) {
                printf("Cann't open file /dev/ node_name \n");
                exit(1);
        }
        read(dev_fd, read_buf, 5);
        ioctl (dev_fd, XXX_IOCTL_CMD,0);
        close(dev_fd);
        return 0;
}
```

2-4 用户空间调用设备驱动程序

- 编写Makefile

```
KERNELDIR ?=/home/student/linux-2.6.32.2/include
all: test

test : test.c
        arm-linux-gcc -I$(KERNELDIR) -o $@ $^
clean :
        rm –rf test
```

2-4 编写BEEP测试程序

- 编写BEEP测试程序beep_test.c，编译生成模块文件：beep_test.ko
- 上电开发板，运行zImage内核
- 下载模块文件至开发板
- 动态装载模块文件
  - insmod beep_test.ko
- 查看BEEP驱动系统自动分配的主设备号

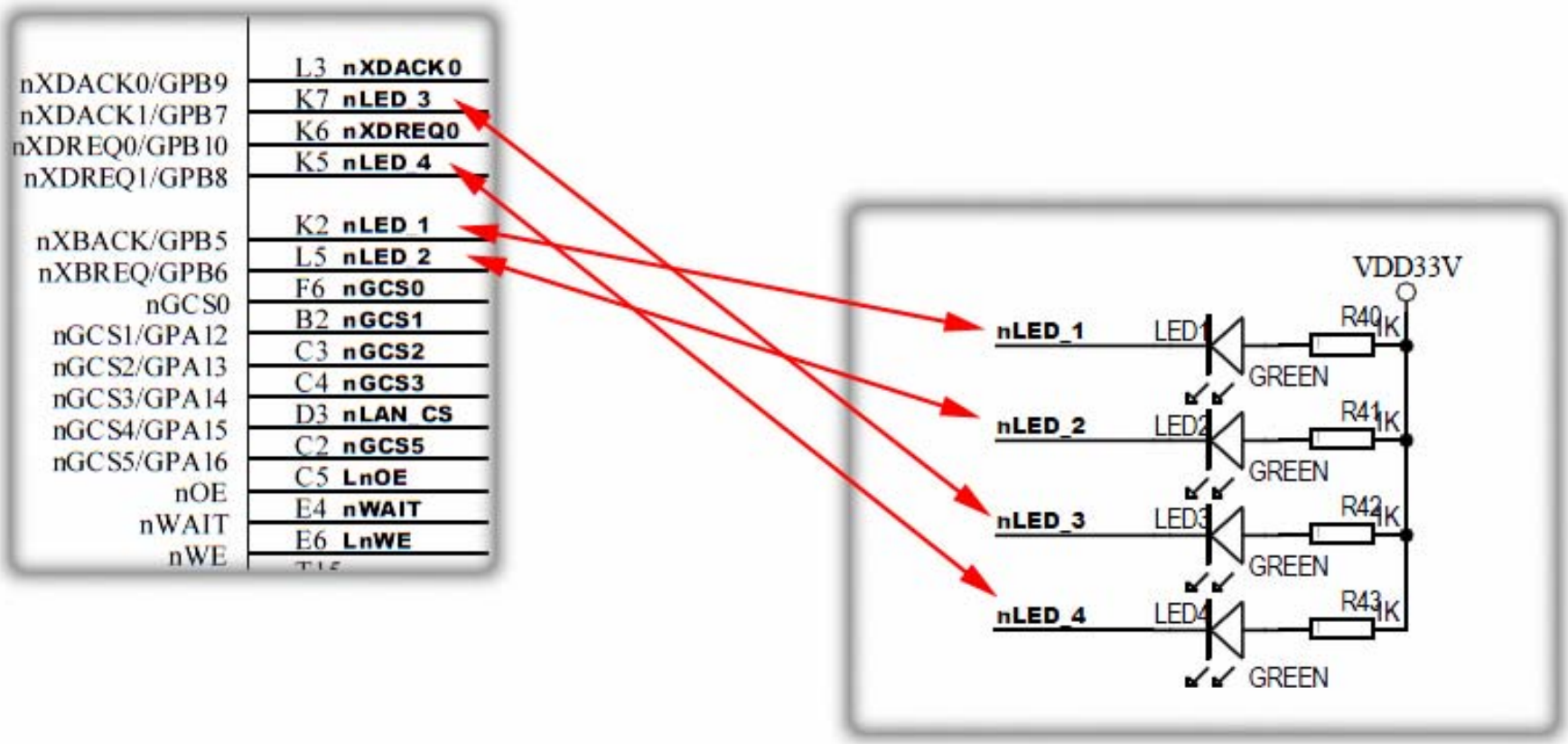
  cat /proc/devices | grep beep
- 制作BEEP对应的设备文件节点
  - mknod /dev/beep c major 0
- 将编译生成的beep_test，下载到开发板，测试BEEP驱动：

  ./beep_test

阶段总结

- 手动创建设备节点方法
- 用户层如何调用驱动程序
- 编写用户程序的Makefile

- 任务一、蜂鸣器驱动程序编写与测试
- 任务二、LED驱动程序编写与测试



- **实现LED驱动测试案例：**
  - led_test on　//对应四个LED全亮
  - led_test off　// 对应四个LED全灭
  - led_test run　// 运行跑马灯实验
  - led_test shine //4个LED灯全灭、全亮交替闪烁

  - led_test 1 on　//对应LED1点亮
  - led_test 1 off　// 对应LED1熄灭
  - ...
  - led_test 4 on　//对应LED4点亮
  - led_test 4 off　// 对应LED4熄灭

2-5 添加驱动程序到内核

- 添加驱动程序到内核源代码位置
- 配置内核
- 编译内核
- 下载运行测试

2-5-1 添加驱动程序到内核

添加驱动程序到内核

- Linux 2.6内核的配置系统由以下3个部分组成。
  - **Makefile**：分布在Linux内核源代码中的Makefile
    - 定义Linux内核的编译规则
  - **配置文件(Kconfig)**：给用户提供配置选择的功能。
  - **配置工具**：
    - 包括配置命令解释器(对配置脚本中使用的配置命令进行解释)
    - 配置用户界面(提供字符界面和图形界面)。
    - 这些配置工具都是使用脚本语言编写的，如Tcl/TK、Perl等。

- 在Linux内核中增加程序需要完成以下3项工作。
  - 将编写的源代码复制到Linux内核源代码的相应目录。
  - 在目录的Kconfig文件中增加新源代码对应项目的编译配置选项。
  - 在目录的Makefile文件中增加对新源代码的编译条目。

2-5-1 添加驱动程序到内核

- 添加BEEP驱动程序到内核：
  - 方法一：在linux-2.6.32.2/drivers/char/下直接添加beep_drv.c源程序
  - 方法二：在linux-2.6.32.2/drivers/char/下添加beep驱动目录

2-5-1 添加BEEP驱动程序到内核—方法一

- 添加BEEP设备的内核配置选项，打开drivers/char/Kconfig文件，添加如下红色条目内容：

```
config BEEP_MINI2440
        tristate “BEEP Driver Support for Mini2440 BEEP Test"
        depends on MACH_MINI2440
        default y if MACH_MINI2440
        help
          This option enables support for BEEP connected to GPIO lines
          on Mini2440 boards.
config MINI2440_ADC
        bool "ADC driver for FriendlyARM Mini2440 development boards"
        depends on MACH_MINI2440
        default y if MACH_MINI2440
        help
          this is ADC driver for FriendlyARM Mini2440 development
    boards
          Notes: the touch-screen-driver required this option
```

2-5-1 添加BEEP驱动程序到内核 —方法一

- 根据该驱动的配置定义，把对应的驱动目标文件加入内核中，打开linux-2.6.32.2/drivers/char/Makefile文件，添加如下红色部分内容：

```
obj-$(CONFIG_LEDS_MINI2440)    += mini2440_leds.o
obj-$(CONFIG_MINI2440_ADC)     += mini2440_adc.o
obj-$(CONFIG_BEEP_MINI2440)    += beep_drv.o
```

- 在内核 linux-2.6.32.2/drivers/char目录下，新建BEEP驱动文件beep_drv.c

2-5-1 添加BEEP驱动程序到内核—方法二

- 实例：在内核源代码drivers/char/目录下新增BEEP驱动

BEEP driver的树形目录：

```
|--beep
    |--beep_drv.c
    |--Kconfig
    |--Makefile
```

步骤：

1、在drivers/char/路径下新建beep目录

2、在beep目录下添加beep_drv.c文件

3、在beep目录下创建Kconfig和Makefile

4、修改新增目录父目录的Kconfig和Makefile，以便新增的Kconfig和Makefile能够被引用

2-5-1 添加BEEP驱动程序到内核—方法二

- **步骤：**

1、在drivers/char路径下新建beep目录

2、添加beep_drv.c驱动源文件

3、为新增目录创建Kconfig和Makefile

```
config BEEP_MINI2440
        tristate “BEEP Driver Support for Mini2440 BEEP Test"
        depends on MACH_MINI2440
        default y if MACH_MINI2440
        help
          This option enables support for BEEP connected to GPIO lines
          on Mini2440 boards.
```

4、为新增目录创建Makefile

```
obj-$(CONFIG_BEEP_MINI2440)      += beep_drv.o
```

2-5-1 添加BEEP驱动程序到内核—方法二

- 修改新增目录的父目录的Kconfig和Makefile
  - 在drivers/char/Kconfig中加入：source “drivers/char/beep/Kconfig”

```
source “drivers/char/beep/Kconfig”
```

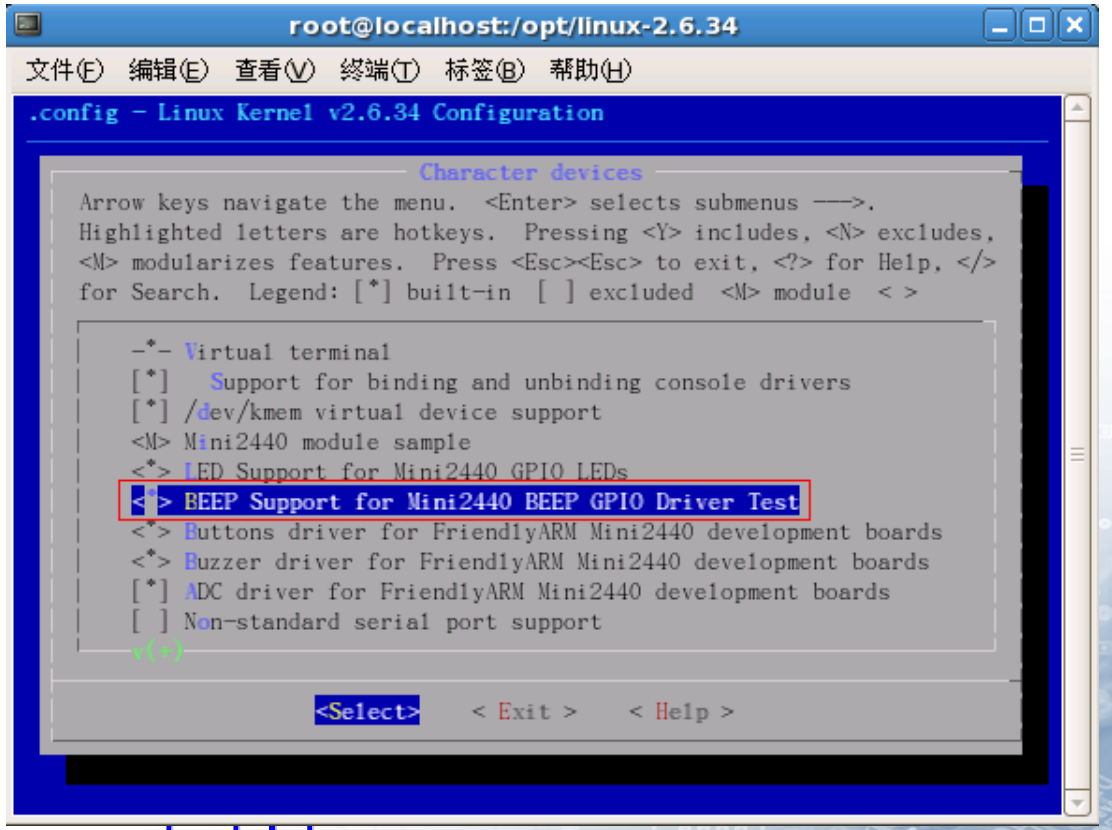
  - 在drivers/char/Makefile中加入：obj-$(CONFIG_BEEP_MINI2440) += beep/

```
obj-$(CONFIG_LEDS_MINI2440)    += mini2440_leds.o
obj-$(CONFIG_MINI2440_ADC)     += mini2440_adc.o
obj-$(CONFIG_BEEP_MINI2440)    += beep/
```

2-5-2 配置编译新内核

- 在内核源代码目录下执行：make menuconfig重新配置内核，依次选择进入如下子菜单项：

Device Drivers --->
Character devices --->



- 设置BEEP配置选项

2-5-2 配置编译新内核

- 进入Linux内核根目录重新编译内核:

```
cd /home/student/linux-2.6.32.2
make zImage
```

- 进入arch/arm/boot，将新生成的zImage下载到开发板

2-5-3 编写BEEP测试程序

- 编写BEEP测试程序beep_test.c，完成对BEEP的调试
- 上电开发板，运行新的zImage内核
- 查看BEEP驱动系统自动分配的主设备号

  cat /proc/devices | grep beep

- 制作BEEP对应的设备文件节点

  mknod /dev/beep c major 0

- 将编译生成的beep_test，下载到开发板，测试BEEP驱动：

  ./beep_test

阶段总结

- 配置和编译Linux内核的方法
- 如何将驱动程序加入到内核中---Makefile & Kconfig

实验

- 任务一、蜂鸣器驱动程序添加到内核实验
- 任务二、LED驱动程序添加到内核实验

2-6 案例分析：memdev虚拟内存设备驱动

- memdev虚拟内存字符设备: 在驱动中分配一片指定大小的内存空间，作为虚拟字符设备。并在驱动中提供针对该片内存的读写、控制和定位函数seek，以供用户空间的进程能通过Linux系统调用访问这片内存。

- 代码分析讲解

- 任务一、memdev驱动程序编写与测试

本章总结