- Linux内核启动分析和移植（一）

主要内容

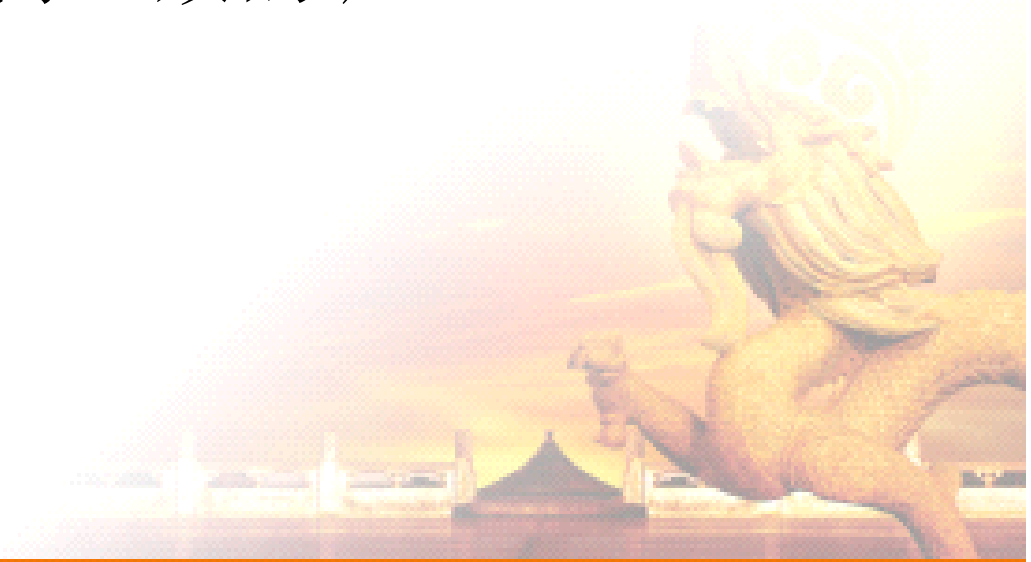
- Linux内核源码结构以及Makefile分析
- 内核的KConfig分析
- Linux内核的.config文件
- Linux内核的移植步骤和原理
- 理解Linux FraemBuffer驱动原理

配置内核的几种方法

- Linux发展到现在已经对非常的硬件设备提供了支持,整个Linux内核中80%都是各种硬件驱动,我们在编译Linux内核时就必须参照自己的硬件设备在内核中选择相应的驱动或者叫配置内核
  - 利用pach命令使用补丁文件对内核进行打补丁操作
  - make menuconfig手工进行配置
  - 先使用跟硬件最为接近的配置文件，然后在其基础上修改
  - 使用厂家提供的配置文件

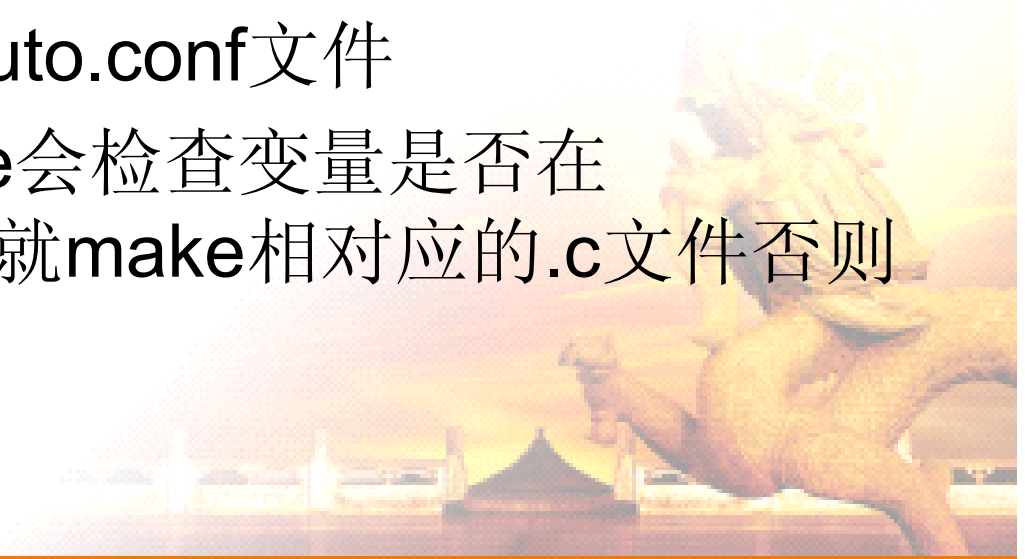
.config配置文件

- 在编译Linux内核之前必须通过适当的配置才能正确使用
- 所有的设置好的配置都必须保存到Linux内核根目录下名为.config文件中
- 那么配置文件是如何生效的呢？



内核配置流程

- 在linux目录中执行 make s3c2410_defconfig,会在根目录下生成.config配置文件信息
- 当执行make zImage时,系统会根据.config内容创建两个文件include/config/auto.conf和include/generated/autoconf.h
- 顶层makefile会包含auto.conf文件
- 各个子目录的makefile会检查变量是否在auto.conf中定义,那么就make相对应的.c文件否则就不编译。



Makefile分析

- linux内核Makefile分类

名称	描述
顶层MakeFile	是所有Makefile的核心
.config	配置文件,在配置内核时发生。所有Makefile都根据.config来决定使用哪些文件
script/Makefile.*	Makefile共用的通用规则、脚本
KBuild Makefile	各级子目录下的Makefile，相对简单,被上一层Makefile调用来编译当前目录下的文件
arcg/$(ARCH)/Makefile	对应体系结构的Makefile,用来决定哪些跟体系结构有关文件参与内核生成

head.s文件

- head.s是linux内核启动时首先执行该文件中的汇编代码(arch/arm/kernel/head.s)
  - 处理从UBoot传来的机器码和参数地址信息

head.s

- 首先把cpu设置为SVC模式并关闭中断
- 通过mrc指令获取CPU ID ,然后调用lookup_process_typ获取处理器类型与CPUID比较
- 调用lookup_processor_type判断UBoot传来的机器码是否和内核机器码相符合
- 最后启用MMU开启页表进行虚拟地址和链接地址映射

```
ENTRY(stext)
                setmode   PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9 @ ensure svc mode
                                                                @ and irqs disabled
                mrc       p15, 0, r9, c0, c0                    @ get processor id
                bl        __lookup_processor_type                       @ r5=procinfo r9=cpuid
                movs      r10, r5                                       @ invalid processor
(r5=0)?
                beq       __error_p                             @ yes, error 'p'
                bl        __lookup_machine_type                         @ r5=machinfo
                movs      r8, r5                                        @ invalid machine
(r5=0)?
                beq       __error_a                             @ yes, error 'a'
                bl        __vet_atags
                bl        __create_page_tables
```

比较机器码

- Linux通过调用__lookup_machine_type这个汇编函数从struct machine_desc结构体中获取事先设置的机器码信息,然后和UBoot传来的机器码进行比较判断,如果相等那么就启动Linux否则进入死loop
  - 那么Linux内核从哪个位置去获取machine_desc这个结构体的内容呢，因为现在代码还在汇编阶段运行。

__lookup_machine_type函数

```
__lookup_machine_type:
                adr     r3, 4b
                ldmia   r3, {r4, r5, r6}
                sub     r3, r3, r4                      @ get offset between
virt&phys
                add     r5, r5, r3                      @ convert virt addresses to
                add     r6, r6, r3                      @ physical address space
1:              ldr     r3, [r5, #MACHINFO_TYPE]        @ get machine type
                teq     r3, r1                                  @ matches loader
number?
                beq     2f                                      @ found
                add     r5, r5, #SIZEOF_MACHINE_DESC    @ next
machine_desc
                cmp     r5, r6
                blo     1b
                mov     r5, #0                                  @ unknown machine
2:              mov     pc, lr
ENDPROC(__lookup_machine_type)
```

单板信息的保存

```
struct machine_desc {
        unsigned int            nr;             /* architecture
number   */
        unsigned int            phys_io; /* start of physical io          */
        unsigned int            io_pg_offst;    /* byte offset for io
                                                 * page tabe entry  */
        const char          *name;          /* architecture name            */
        unsigned long       boot_params;    /* tagged list                  */
        unsigned int        video_start;    /* start of video RAM           */
        unsigned int        video_end;      /* end of video RAM             */
        unsigned int        reserve_lp0 :1; /* never has lp0      */
        unsigned int        reserve_lp1 :1; /* never has lp1      */
        unsigned int        reserve_lp2 :1; /* never has lp2      */
        unsigned int        soft_reboot :1; /* soft reboot                  */
        void                (*fixup)(struct machine_desc *,
                                     struct tag *, char **,
                                     struct meminfo *);
        void                (*map_io)(void);/* IO mapping function          */
        void                (*init_irq)(void);
        struct sys_timer    *timer;         /* system tick timer            */
        void                (*init_machine)(void);
};
```

修改内核机器码

- 在UBoot源码中传给Linux内核的机器码为1999,因此我们也必须修改内核机器码为1999.
- 内核的机器码登记信息被保存在linux2.6.33/arch/arm/tools/mach-types文件中，修改s3c2440机器码为1999

内核添加yaffs2文件系统支持

- 在Linux2.6.33内核中系统并不支持yaffs2文件系统,因此必须对内核进行修改后加以支持
  - 下载yaffs2文件系统源代码
  - 分析yaffs2系统补丁shell脚本
  - 执行yaffs2系统的patch-ker.sh脚本对内核打补丁

yaffs2补丁脚本

```
VERSION=0
PATCHLEVEL=0
SUBLEVEL=0
COPYORLINK=$1
LINUXDIR=$2
usage () {
        echo "usage:  $0  c/l kernelpath"
        echo " if c/l is c, then copy. If l then link"
        exit 1
}

。。。。。。。。。。。。。。

YAFFSDIR=$LINUXDIR/fs/yaffs2

if [ -e $YAFFSDIR ]
then
  echo "$YAFFSDIR exists, not patching"
else
  mkdir $LINUXDIR/fs/yaffs2
  $CPY  $PWD/Makefile.kernel $LINUXDIR/fs/yaffs2/Makefile
  $CPY $PWD/Kconfig $LINUXDIR/fs/yaffs2
  $CPY $PWD/*.c $PWD/*.h  $LINUXDIR/fs/yaffs2
fi
```

修改makefile

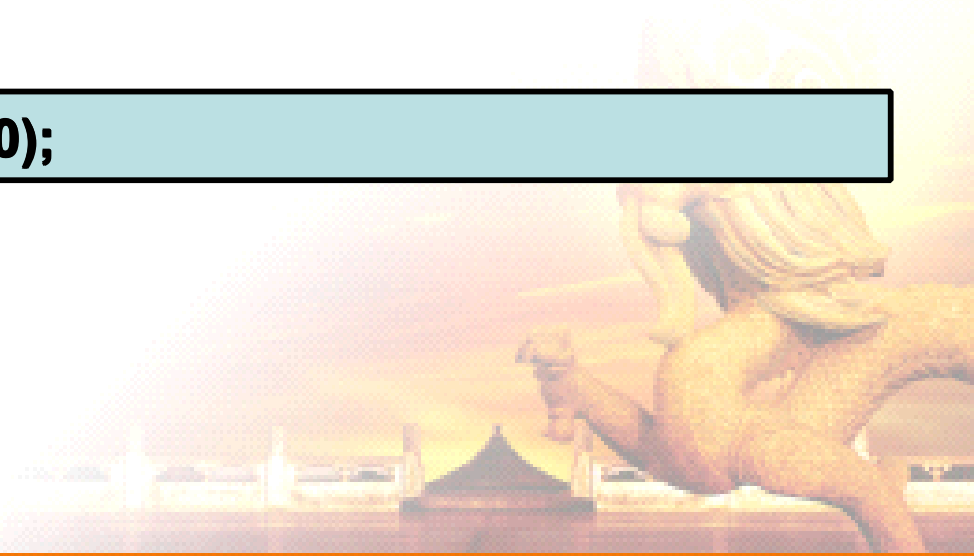
- 由于我们编译的是arm linux因此必须对顶层makefile所使用的编译器和编译架构进行设置

```
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH            ?= arm
CROSS_COMPILE   ?= arm-linux-

# Architecture as present in compile.h
UTS_MACHINE     := $(ARCH)
SRCARCH         := $(ARCH)

# Additional ARCH settings for x86
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif
# Additional ARCH settings for sparc
ifeq ($(ARCH),sparc64)
       SRCARCH := sparc
endif
```

修改外部时钟晶振

- 开发板上电后CPU的时钟频率并不会在正常值内而是利用板上的外街晶振频率决定s3c2440外接晶振12M
- 修改s3c24xx_init_clocks函数参数改为12M该函数位于(arch/arm/mach-s3c2440/mach-smdk2440.c)

```
3c24xx_init_clocks(12000000);
```



MTD分区修改

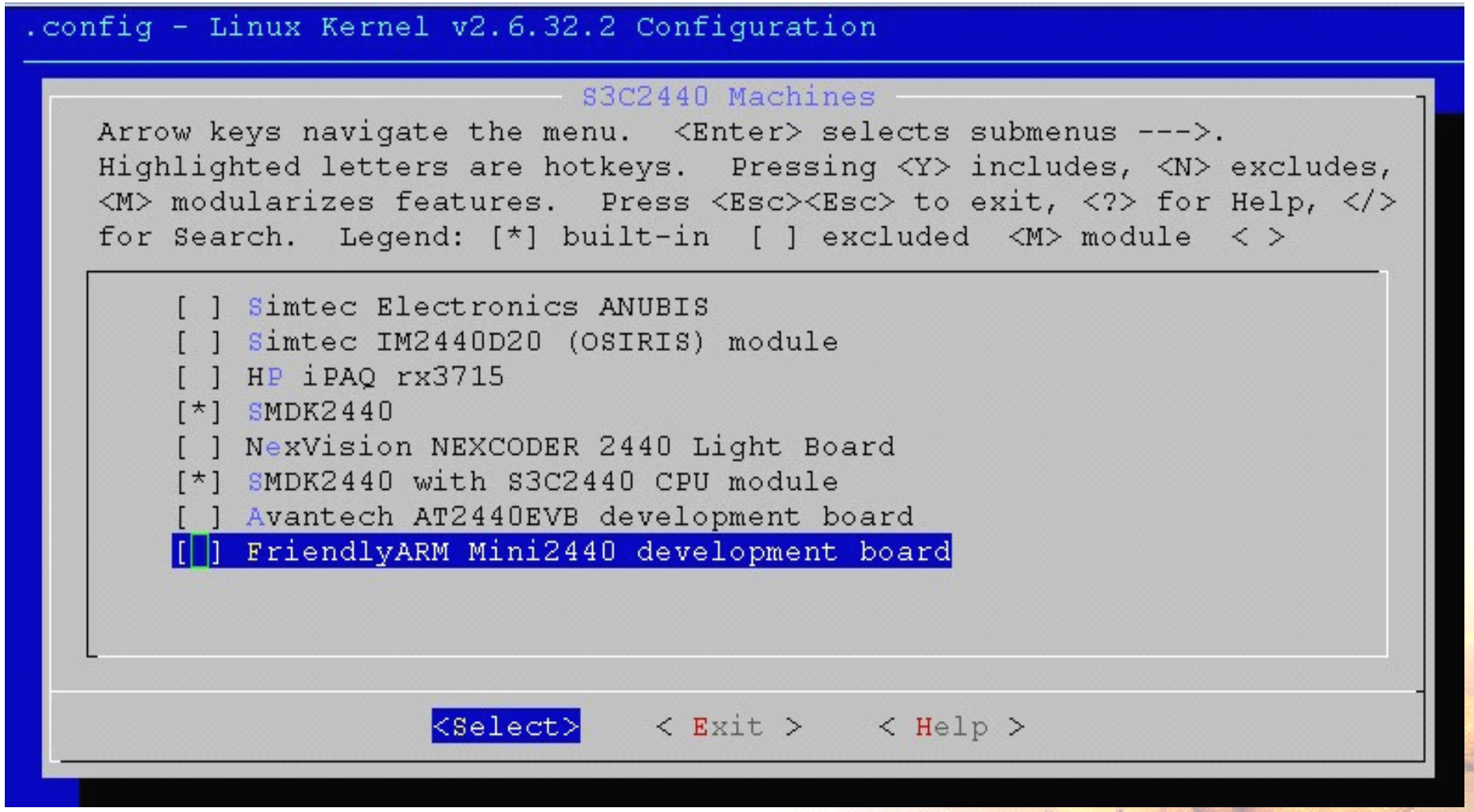
- arm-linux内核启动后加载完毕所有的设备驱动程序之后,最后就要读取文件系统内容,那么它是如何知道文件系统位于存储器哪个位置呢？
  - Linux内核代码中提供了对MTD设备驱动的支持
  - MTD(memory technology device内存技术设备)是用于访问memory设备（ROM、flash）的Linux的子系统。MTD的主要目的是为了使新的memory设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。MTD的所有源代码在/drivers/mtd子目录下
  - 在Linux中用static struct mtd_partition这个结构体来描述NANDFLASH的分区情况(位于linux/arch/arm/plat-s3c24xx/common-smdk.c文件中)

MTD分区信息

static struct mtd_partition smdk_default_nand_part[]的结构体，将内容修改为：

```
[0] = {
        .name       = "boot",
        .size       = 0x00020000,
        .offset = 0
},
        [1] = {
        .name       = "bootParam",
        .size       = 0x00060000,
        .offset = 0x00020000,
},
[2] = {
        .name       = "Kernel",
        .size       = 0x00300000,
        .offset = 0x00500000,
},
[3] = {
        .name       = "fs_yaffs",
        .size       = 0x03c00000,
        .offset = 0x00800000,
},
[4] = {
        .name       = "WINCE",
        .size       = 0x03b80000,
        .offset = 0x04480000,
}
```

menuconfig


.config - Linux Kernel v2.6.32.2 Configuration
S3C2440 Machines
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >
[ ] Simtec Electronics ANUBIS
[ ] Simtec IM2440D20 (OSIRIS) module
[ ] HP iPAQ rx3715
[*] SMDK2440
[ ] NexVision NEXCODER 2440 Light Board
[*] SMDK2440 with S3C2440 CPU module
[ ] Avantech AT2440EVB development board
[ ] FriendlyARM Mini2440 development board
<Select>
< Exit >
< Help >

LCD驱动移植

- 移植Linux内核移植LCD驱动是大家最为关心的,要想正确移植LCD驱动并理解移植的原理就必须搞清楚下面几个问题
  - 理解Linux下设备驱动的结构
  - 理解Linux下FraemBuffer驱动实现原理
  - 理解要移植的LCD芯片手册参数含义

platform设备

- platform 可以理解成一种设备类型，就像字符设备、块设备和网络设备一样，而LCD 就属于这种设备。
- 为了向内核添加一个platform 设备，程序员应该填写两个数据结构platform_device 和platform_driver，这两个数据结构的定义都可以在include/linux/platform_device.h 文件中找到。

platform_device

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

- 该结构中有一个name变量。内核必须为设备取个名字,该名字为后面的设备驱动提供身份识别。
- dev结构体描述了该设备的具体信息
- num_resources用来描述该设备共使用了多少资源
- resource具体用来描述设备使用的资源信息,这些资源包括设备使用中断号,内存地址等等

device_driver结构体

```
truct device_driver {
    const char        * name;
    struct bus_type        * bus;
    struct kobject        kobj;
    struct klist        klist_devices;
    struct klist_node    knode_bus;
    struct module        * owner;
    const char        * mod_name;  /* used for built-in modules */
    struct module_kobject    * mkobj;
    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);
};
```

- 需要注意这两个变量：name和owner。那么的作用主要是为了和相关的platform_device关联起来，owner的作用是说明模块的所有者，驱动程序中一般初始化为THIS_MODULE。

LCD设备的注册

- 搞清楚Linux内核中设备驱动的数据结构后，系统如何感知设备的存在呢，这就需要通过内核提供的函数把设备添加进内核中。
- 设备添加进内核函数
  - int platform_add_devices(struct platform_device **devs, int num)

Linux设备注册流程

```
static struct platform_device *smdk2440_devices[] __initdata = {
        &s3c_device_usb,
        &s3c_device_lcd,
        &s3c_device_wdt,
        &s3c_device_i2c0,
        &s3c_device_iis,
};
static void __init smdk2440_machine_init(void)
{
        s3c24xx_fb_set_platdata(&smdk2440_fb_info);
        s3c_i2c0_set_platdata(NULL);

        platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
        smdk_machine_init();
}
MACHINE_START(S3C2440, "SMDK2440")
        /* Maintainer: Ben Dooks <ben@fluff.org> */
        .phys_io    = S3C2410_PA_UART,
        .io_pg_offst        = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
        .boot_params        = S3C2410_SDRAM_PA + 0x100,

        .init_irq     = s3c24xx_init_irq,
        .map_io             = smdk2440_map_io,
        .init_machine       = smdk2440_machine_init,
        .timer              = &s3c24xx_timer,
MACHINE_END
```

framebuffer驱动

- 帧缓冲(Framebuffer)是 Linux 系统提供的用户与LCD等显示设备的接口， 它把显示缓冲区抽象化以此来屏蔽不同显示设备硬件的底层差异，这样上层应用程序通过直接对显示缓冲区的读写操作，即可控制LCD屏幕的输出用户可以不必关系物理显存的具体位置、换页机制等细节信息，这些都由帧缓冲设备驱动程序来完成。应用程序只要在显示缓冲区中与LCD显示点对应的区域写入颜色值，相应的颜色即可显示在LCD屏幕上。 显示缓冲区的大小由屏幕分辨率和显示颜色数决定。帧缓冲设备是标准的字符设备，采用“文件层—驱动层”的接口方式， 主设备号为29，对应的设备文件为/dev/fb*
- 当设备已经被linux内核感知后，剩下的就是编写相应的设备驱动,framebuffer的驱动linux/driver/video/s3c2410fb.c中

platform_driver结构体

```
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 device_driver driver;
} ;
```

- 由上可见，它包含了设备操作的几个功能函数，同样重要的是，它还包含了一个device_driver结构。刚才提到了驱动程序中需要初始化这个变量。下面看一下这个变量的定义，位于include/linux/device.h中

framebuffer驱动实现

```
static struct platform_driver s3c2412fb_driver = {
        .probe          = s3c2412fb_probe,
        .remove         = s3c2410fb_remove,
        .suspend = s3c2410fb_suspend,
        .resume         = s3c2410fb_resume,
        .driver         = {
                .name   = "s3c2412-lcd",
                .owner  = THIS_MODULE,
        },
};

int __init s3c2410fb_init(void)
{
        int ret = platform_driver_register(&s3c2410fb_driver);

        if (ret == 0)
                ret = platform_driver_register(&s3c2412fb_driver);

        return ret;
}
```

framebuffer驱动实现

```
static int __init s3c2412fb_probe(struct platform_device *pdev)
{
        return s3c24xxfb_probe(pdev, DRV_S3C2412);
}
static int __init s3c24xxfb_probe(struct platform_device *pdev,
                                  enum s3c_drv_type drv_type)
{
        struct s3c2410fb_info *info;
        struct s3c2410fb_display *display;
        struct fb_info *fbinfo;
        struct s3c2410fb_mach_info *mach_info;
        struct resource *res;
        int ret;
        int irq;
        int i;
        int size;
        u32 lcdcon1;
        ..................................................
}
```

LCD控制器数据结构

```
struct s3c2410fb_mach_info {

        struct s3c2410fb_display *displays;    /* attached diplays info */
        unsigned num_displays;                 /* number of defined
displays */
        unsigned default_display;

        /* GPIOs */

        unsigned long       gpcup;
        unsigned long       gpcup_mask;
        unsigned long       gpccon;
        unsigned long       gpccon_mask;
        unsigned long       gpdup;
        unsigned long       gpdup_mask;
        unsigned long       gpdcon;
        unsigned long       gpdcon_mask;

        /* lpc3600 control register */
        unsigned long       lpcsel;
};
```

LCD控制器数据结构

```
struct s3c2410fb_display {
        /* LCD type */
        unsigned type;
        /* Screen size */
        unsigned short width;
        unsigned short height;
        /* Screen info */
        unsigned short xres;
        unsigned short yres;
        unsigned short bpp;

        unsigned pixclock;              /* pixclock in picoseconds */
        unsigned short left_margin;  /* value in pixels (TFT) or HCLKs (STN) */
        unsigned short right_margin; /* value in pixels (TFT) or HCLKs (STN) */
        unsigned short hsync_len;    /* value in pixels (TFT) or HCLKs (STN) */
        unsigned short upper_margin;/* value in lines (TFT) or 0 (STN) */
        unsigned short lower_margin; /* value in lines (TFT) or 0 (STN) */
        unsigned short vsync_len;      /* value in lines (TFT) or 0 (STN) */

        /* lcd configuration registers */
        unsigned long         lcdcon5;
};
```