第 6 章　嵌入式 Linux 的内核开发

内容简介：

我们介绍如何开发 Linux 的内核？通过这一章，大家应该了解 Linux 的配置、编译过程，并且能够形成最后的 Linux 映象文件，掌握自己如何加入驱动程序的方法，真正对嵌入式 Linux 的内核有深刻的了解。

1．英贝德给你提供的内核开发资源

当然，作为产品开发商，英贝德提供你可以对内核操作的所有资源，包括

- EBD9200 的内核原码
- 交叉编译工具（arm-linux-）
- 映象文件的形成工具（mkimage）
- 必不可少的关键性说明

这些内容的提供，使得你可以轻松面对内核的操作。

2．开发 Linux 内核的步骤

2．1 准备工作

交叉开发环境的建立

在第 5 章中，我们已经介绍了交叉开发环境，我们应该记住位置/usr/local/arm/2.95.3/bin 或者其它位置。

内核原码的安装

在光盘中找到内核的源码程序 linux-2.4.27.tar.gz，拷贝到/usr/src/arm 下面，并且解压缩，得到内核原码，你也可以安装到别的地方。

$ tar　-zxvf　linux-2.4.27.tar.gz

2．2 配置内核前的必要设置

主要在内核原码中设置 Makefile 文件，

用下列指令打开 Makefile 文件:

$vi　Makefile

在 Makefile 中主要设置两个地方：ARCH 和 CROSS_COMPILE。

ARCH :=arm　　；表示目标板为 arm。

CROSS_COMPILE=交叉编译工具的地址　　；设置交叉编译工具的地址。

例如 CROSS_COMPILE= /usr/local/arm/2.95.3/bin/arm-linux。

还要在脚本文件 mkimage 中把路径改为 9200/bootldr/u-bootFor16M/u-boot-1.1.1/tools 。（具体的路径和你的 u-boot 放的位置有关）

一般需要详细阅读 readme 文件。

2．3 内核配置

make menuconfig ；菜单界面

make xconfig ；图形界面

2．4 内核编译

make clean

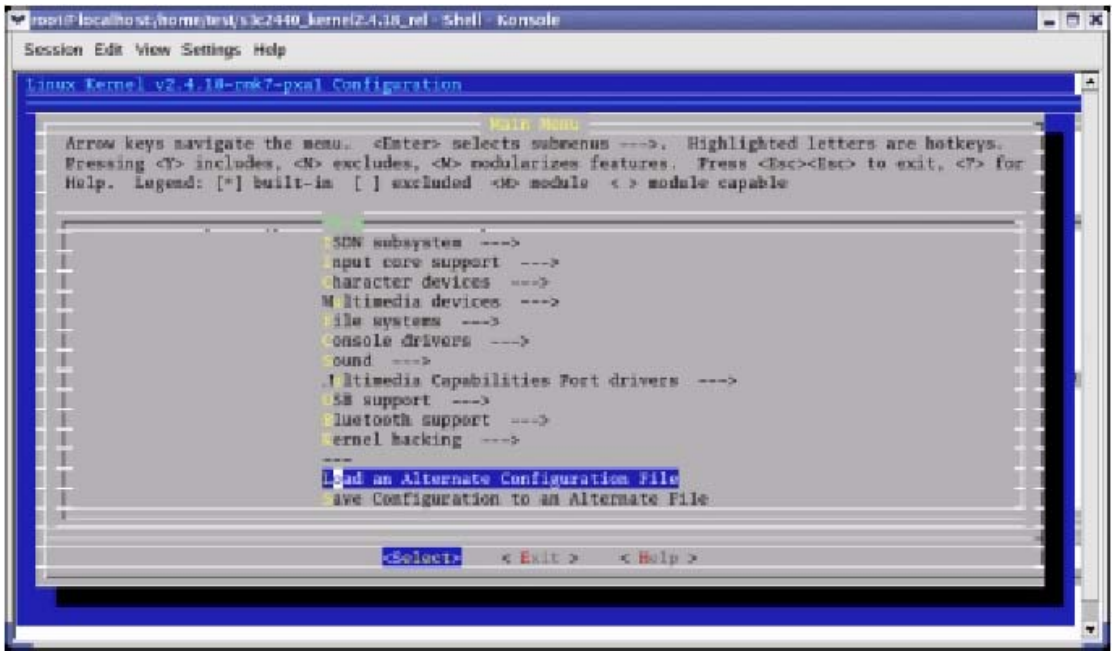
make dep

./mkimage //运行 mkimage 脚本文件

最终编译得到的映象文件 uImage

3．如何配置和裁剪 Linux 的内核

这一节我们来配置 linux 的内核，在 Linux 下，用 make menuconfig 或 make xconfig 进入配置的界面。在内核配置中，一般有四种选择：Y（选种）、N（不选）、M（模块）和数字，用户可以根据裁剪需要进行设置，最后配置完毕，选择是否对配置结果进行保存？在 Linux 中的图形配置界面如下：



Block Devices:

➔ Network block device support : n

➔ -> Ramdisk support: y

➔ -> Default Ramdisk size: 20000

➔ -> Initial Ramdisk (initrd) support: y

File System -> Network File systems:

➔ NFS file system support : n

➔ -> NFS server support: n

Console drivers -> Frame Buffer Support:

➔ support for frame buffer devices (exp): y

➔ -> epson LCD/CR/TV controller support: y
➔ -> epson S1 D1 3806 support for AT91RM9200DK: y
➔ -> virtual frame buffer support (only for testing!): n
➔ -> advanced low level driver option: y
➔ -> 16 bpp packed pixeld support: y (others n)

USB Support:

➔ support for USB: y
➔ -> AT91RM9200 OHCI- compatible host interface: y
➔ -> USB mass storage support : y
➔ -> USB Human Interface device (full HID) support: y
➔ -> HID input layer support : y

Kernel configuration when using NFS ramdisk:

Below the details of the parameters when typing: make xconfig:

General setup:

-> Default kernel string : Erase its contents

Block Devices:

➔ Network block device support : y
➔ -> Ramdisk support: n

File System:

➔ Quota support: n
➔ -> Kernel automounter support: y
➔ -> DOS FAT fs support: y
➔ -> VFAT (Windows 95) fs support: y
➔ -> Journalling flash file system v2 (JFFS2) support: 0
➔ -> /proc file system support: y
➔ -> /dev file system support (EXP): y
➔ -> Automatically mount a boot: y
➔ -> Second extended fs support: y

File System -> Network File systems:

➔ NFS file system support : y
➔ -> provide NFSv3 client support: y
➔ -> root file system on NFS: y
➔ -> NFS server support: n

当然，更多的东西，请你恭身入局，实际在 linux 下看看，你才会真正了解。

4．如何添加新的内核配置和驱动

英贝德公司为你提供了丰富的驱动和配置，对于很多客户，可能只需要进行裁剪，去掉自己不需要的内容，然后重新编译内核，形成自己的内核映象文件，而对于一些构造复杂系统的客户，可能还需要在原来的基础上添加新的设备和驱动程序，比如：你可能需要增加一

个标准并口的支持，以挂接打印机，可能增加 IDE 接口，对系统挂接标准硬盘，当然，还有很多客户开发属于自己接口的各种驱动程序等等，这时，我们在裁剪内核的基础上，可能就需要添加内核和驱动程序。

实际上，不论是操作系统还是用户的上层程序，实际上都是程序代码，可以这么说：你可以编写从底层操作系统到上层实现的一系列流程。当然如果你的设备要求不高，你可以把你的驱动初始化就直接写到你的上层程序中，当然，你也可以把两者分开，但是这样你肯定会花更多的精力，我们在此介绍如何添加新的内核配置和驱动？

驱动程序的使用可以按照两种方式编译，一种是静态编译进内核，另一种是编译成模块以供动态加载。由于嵌入式 Linux 不能够象桌面 Linux 那样灵活的使用 insmod/rmmod 加载卸载设备驱动程序，因而这里只介绍将设备驱动程序静态编译进 linux 内核的方法。下面以嵌入式 Linux 为例，介绍在一个以字符设备驱动的例子，将其编译进内核的一系列步骤。

（1）： DEMO 驱动程序(demo_drv.c)

```
/*********************************************************************************
*          演示如何设计一个字符型驱动程序
*********************************************************************************/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/sched.h>
#define DEMO_MAJOR     125
#define COMMAND1  1
#define COMMAND2  2
/*********************************************************************************
*          函数声明
/*********************************************************************************/
static int demo_init(void);
static int demo_open(struct inode *inode,struct file *file);
static int demo_close(struct inode *inode,struct file *file);
static ssize_t    demo_read(struct file *file,char *buf,size_t count,loff_t *offset);
static int demo_ioctl(struct inode *inode,struct file *file,unsigned int cmd,unsigned long arg);
static void demo_cleanup(void);
/*********************************************************************************
*          全局变量定义
/*********************************************************************************/
int demo_param =9;
static int demo_initialized =0;
static volatile int demo_flag =0;
static struct file_operations demo_fops={
#if LINUX_KERNEL_VERSION>=KERNEL_VERSION(2,4,0)
    ower:      THIS_MODULE,
#endif
    llseek:    NULL,
```

```
    read:demo_read,
    write:    NULL,
    ioctl:    demo_iocrl,
    open:     demo_open,
    release:demo_close,
};

static int demo_init(void){
    int i;
    /*确定模块以前未初始化*/
    if(demo_initialized==1)
        return 0;
    /*分配并初始化所有数据结构为缺省状态*/
    i=register_chrdev(DEMO_MAJOR,"demo_drv",&demo_fops);
    if(i<0){
        printk(KERN_CRIT "DEMO: i = %d\n",i);
        rerurn -EIO;
    }
    printk(KERN_CRIT "DEMO: demo_drv registerred successfully:)=\n");
    /*请求中断*/
    demo_initialized=1;
    return 0;
}
static int demo_open(struct inode *inode,struct file *file){
    if(demo_flag==1){ /*检查驱动是否忙*/
        return -1;
    }
    /*可以初始化一些内部数据结构*/
    printk(KERN_CRIT "DEMO: demo device open\n");
    MOD_INC_USE_COUNT;
    demo_flag=1;
    return 0;
}
static int demo_close(struct inode *inode,struct file *file){
    if(demo_flag==0){
        return 0;
    }
    /*可以删除一些内部数据结构*/
    printk(KERN_CRIT "DEMO: demo device close\n");
    MOD_DEC_USE_COUNT;
    demo_flag=0;
    return 0;
}
static ssize_t demo_read(struct file *file,char *buf,size_t count,loff_t *offset){
```

```
    /*检查是否有线程在读数据，返回 error*/
    //DEMO_RD_LOCK;
    printk(KERN_CRIT "DEMO: demo is reading,demo_param = %d\n",demo_param);
    //DEMO_RD_UNLOCK;
    /*通常返回成功独到的数据*/
    return 0;
}
static int demo_ioctl(struct inode *inode,struct file *file,unsigned int cmd,unsigned long arg){
    if(cmd==COMMAND1){
        printk(KERN_CRIT "DEMO: set command COMMAND1\n";
        return 0;
    }
    if(cmd==COMMAND2){
        printk(KERN_CRIT "DEMO: set command COMMAND2\n";
        return 0;
    }
    printk(KERN_CRIT "DEMO: set command WRONG\n";
    return 0;
}
static void demo_cleanup(void){
    /*确保要清掉的模块是已初始化的*/
    if(demo_initialized==1){
        /*禁止中断*/
        /*释放该模块的中断服务程序*/
        unregister_chrdev(DEMO_MAJOR,"demo_drv");
        demo_initialized=0;
        printk(KERN_CRIT "DEMO: demo device is cleanup\n");
    }
    return;
}
/*****************************************************************************
*           初始化/清除模块
*****************************************************************************/

#ifdef MODULE
MODULE_AUTHOR("DEPART 901");
MODULE_DESCRIPTION("DEMO driver");
MODULE_PARM(demo_param,"i");
MODULE_PARM_DESC(demo_param,"parameter sent to driver");
int init_module(void){
    return demo_init();
}
void cleanup_module(void){
    demo_cleanup();
```

```
}
#endif
```

（2）： 应用程序(test.c)

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/rtc.h>
#include <linux/ioctl.h>
#define COMMAND1    1
#define COMMAND2    2
main(){
        int fd;
        int i;
        unsigned long data;
        int retval;
        fd = open("/dev/demo_drv",O_RDONLY);
        if(fd == -1){
                perror("open");
                exit(-1);
        }
        retval = ioctl(fd,COMMAND1,0);
        if(retval == -1){
                perror("ioctl");
                exit(-1);
        }
        for(i=0;i<3;i++){
                retval = read(fd,&data,sizeof(unsigned long));
                if(retval == -1){
                        perror("read");
                        exit(-1);
                }
        }
        close(fd);
}
```

STEP 2：

将 demo_drv.c 复制到…/drivers/char 目录下，并且在…./drivers/char 目录下 mem.c 中， int chr_dev_init( )函数中增加如下代码：

```
#ifdef CONFIG_TESTDRIVE
int demo_init ();
#endif
```

STEP 3：

在…../drivers/char 目录下 Makefile 中增加如下代码：

```
ifeq($(CONFIG_TESTDRIVE),y)
L_OBJS+= demo_drv.o
Endif
```

STEP 4：

在……../arch/m68knommu 目录下 config.in 中字符设备段里增加如下代码：

```
bool 'support for testdrive' CONFIG_TESTDRIVE y
```

STEP 5：

运行 make menuconfig（在 menuconfig 的字符设备选项里你可以看见我们刚刚添加的'support for testdrive'选项，并且已经被选中）；

STEP 6：文件系统的相应改变

在….. /dev/目录下创建设备：

```
mknod demo_drv c 125  0
```

对文件系统的目录进行添加即可。

至此，你已经在 Linux 中增加了一个新的设备驱动程序。

（具体的编写要查看相关资料，LINUX 设备驱动程序第二版是一本不错的书）

5．嵌入式 Linux 内核的编译

对于 Linux 内核的编译，上面已经进行了说明，我们在此解释一下 make dep：

dependence 从字面上是依赖的意思，make dep 的意思就是说：如果你使用程序 A（比如支持特殊设备），而 A 需用到 B(比如 B 是 A 的一 个模块/子程序)。而你在做 make config 的时候将一个设备的驱动 由内核支持改为 module,或取消支持，这将可能影响到 B 的一个参数的设置，需重新编译 B，重新编译或连接 A....如果程序数量非常多， 你是很难手工完全做好此工作的。 所以，你要 make dep。如果你 make menuconfig 或 make config 或 make xconfig 后，直接 reboot，会更快。只是你的内核根本没有任何改变。一般 linux 内核全部编译需要下述命令。

```
make menuconfig（xconfig）；
make dep;
make clean;
./mkimage
```

6．对于嵌入式 Linux 的内核开发，你还需要什么？

我想你应该对于嵌入式 Linux 的内核开发有个了解，并且可以对 EBD9200 进行 Linux 内核的编译和裁剪了，同时，我们还介绍了你的新的驱动程序以及设备如何添加？如果你的驱动程序已经具备，你完全可以进行 Linux 的内核开发了。

但是，你现在可能还在疑惑两个问题。一是，我怎么去编写一个新的设备驱动程序，这个问题就比较复杂了，对于不同的设备，你的代码都千差万别，你千万别试图从这本说明书中找到答案，应该去了解关于 Linux 驱动程序开发的相关内容。另外一个问题就是文件系统的问题了，因为内核启动起来后，关键是你什么都看不到，这时，你需要继续阅读下一章的内容。