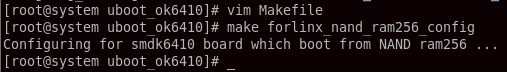
|  |
| --- |
|  |
| **Linux驱动程序设计基础** |
| V2.0 |
| 专题二 Uboot新手入门  专题三 Linux内核制作  专题四 Linux根文件系统  专题五 Linux内核模块  专题六 Linux内核子系统  专题七 Linux内核链表  专题八 系统调用  专题九 驱动前奏  专题十 字符设备驱动模型  专题十一 字符设备控制技术 |
|  |
| 2017-07-21 |

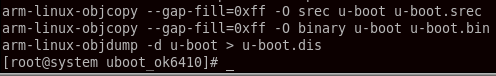
**专题二 Uboot新手入门**

**2.1编译u-boot：**

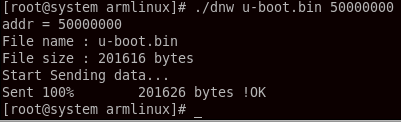
将OK6410的uboot压缩文件通过smb拷贝到linux系统当中并完成解压缩，并且配置需要编译的uboot文件，Vim makefile ：/6410 查找到相关的配置文件名。配置编译：



开始编译：输入**make ARCH=arm CROSS\_COMPILE=arm-linux-** 开始执行编译，大概30秒左右，最后生成了文件如下所示：

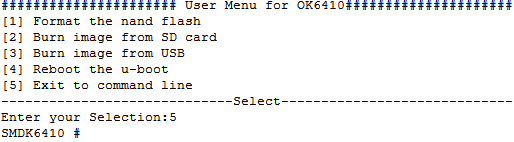


将制作好的镜像下载到开发板中：



**2.2 U-boot命令：**

启动开发板，在控制台上进入uboot模式，而不是bootloader。



**（1）#help**  帮助命令

**（2）#printenv**  查看系统环境变量

范例：#printenv ipaddr

**（3）#setenv** 添加、修改、删除内存中的环境变量

#setenv myboard 210

**（4）#saveenv** 保存环境变量

**2.3 文件下载：**

范例：#tftp 0x50000000 uImage

把tftp服务器上的uImage下载到0x50000000处。

**tftp** 通过网络下载文件

#setenv ethaddr 12:34:56:78:9A:BC

#setenv ipaddr 192.168.0.3

#setenv serverip 192.168.0.2 (tftp服务器的地址)

#ping 192.168.0.2 (tftp服务器的地址)

**2.4 执行程序**

范例：#bootm 0xc0008000

bootm {addr} {arg}

执行固定格式的2进制程序

**2.5 查看内存内容**

md 显示内存区的内容。

md采用十六进制和ASCII码两种形式来显示存储单元的内容。

这条命令还可以采用长度标识符.l, .w和.b ：

md [.b, .w, .l] address

范例：md.w 100000

00100000: 2705 1956 5050 4342 6f6f 7420 312e 312e

00100010: 3520 284d 6172 2032 3120 3230 3032 202d

**2.6 擦除\写\读 nand flash**

**nand erase** 起始地址start 长度len，擦除start处开始的，长度为len的区域

**nand write** 内存起始地址flash 长度len，将内存起始地址处，长度为len的数据，写入flash起始地址处。

**nand read** 内存起始地址flash起始地址长度len，将flash起始地址处，长度为len的数据，读到内存起始地址处。

范例：#nand erase 0x400000 0x500000

范例：#nand write c0008000 400000 500000

范例：#nand read c0008000 400000 500000

**2.7 设置自启动**

1: 设置从nand flash自动启动

#setenv bootcmd nand read c0008000 400000 500000 ; bootm c0008000

2：设置自动下载内核到内存后启动

#setenv bootcmd tftp c0008000 uImage ; bootm c0008000

**小结：**

官方提供的uboot压缩包使用的网卡是CS8900，而不是开发板上面的DM9000，这部分的驱动在网上找到了一个替代的bin文件。

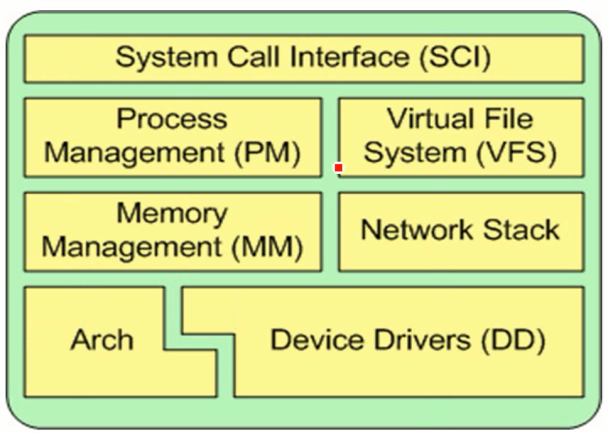
**专题三 Linux内核制作**

**3.1.1 Linux体系结构：**

Linux由用户空间和内核空间组成。User space包含了User Applications和 GNU C Library，而Kernel space则包含了System call interface 、Kernel 和 Architecture-Dependent Kernel Code三个部分。

划分为两个部分的原因是为了保护操作系统本身能够正常的运行。

**3.1.2 Linux内核架构：**



SCI：用于实现调用内核的函数。

PM：用于管理进程。

MM：内存管理。

Arch：和处理器相关的部分，用于系统移植。

VFS：

Net：网络协议栈。

DD：硬件设备驱动。

**3.2 Linux内核：**

配置内核就是按照系统软件和硬件设计的需要，将多余的文件去掉。比方说将ARCH中除ARM之外的所有处理器都去掉，就是在对内核进行配置。配置内核的方法：

make config：基于文本模式的交互式配置。

make menuconfig：基于文本模式的菜单型配置。

在内核文件目录下进入menuconfig配置文件界面。

1. 选择Processor type and features配置处理器平台，在处理器家族中选择对应的宿主机的处理器系列。
2. 配置Networking support。
3. 配置Generic Driver Options。
4. 配置File system。

< \* >选择 最后生成zImage内核映像

< M >选择为内核模块，需要时就加载，使用完成之后就去掉

< >不选择

但是更多的时候是使用参考的结果文件进行对应的修改即可。

**3.3编译步骤：**

(1)编译内核：zImage

zImage编译内核小于512K的内核文件；

bzImage用于编译内核较大的内核文件；

如需获取详细编译信息，可使用：

make zImage V=1 make bzImage V=1

编译好的内核位于arch/<cpu>/boot/目录下。

(2) 编译模块：生成.ko文件

编译过程：make modules 时间会比较长

复制所有文件到 /lib/modules下面。执行移动make modules\_install

(3)生成内核模块文件的打包文件：

mkinitrd rd-2.6.32 2.6.32

(4)清理内核源代码：

整个过程中生成了.config文件 .o文件 .ko文件这三类文件，清理时使用make clean清除.o文件。Make disclean 可以清除所有的中间文件。

* 1. **Arm-Linux内核制作：**

(1)执行make distclean清除掉之前所有的中间文件和相关的配置文件。

(2)配置内核，使用命令为**make menuconfig ARCH=arm**。

(3)在这个图形界面中加载配置文件config\_file。

(4)检查system type中的各项配置是否正确，如有需要可进行相应的配置。

(5)将内核文件中的mkimage工具拷贝到/bin/这一目录之下。源文件路径为/home/armlinux/uboot/linux/uboot\_ok6410/tools。

(6)编译内核：make uImage ARCH=arm CROSS\_COMPILE=arm-linux-，确定使用的编译的平台和交叉工具链。

**注意：下载使用的地址务必是0x50000000。使用0x50008000时，出现了卡在Starting Kernel...的现象。**

**专题四 Linux根文件系统**

**4.1 建立根文件系统目录与文件：**

**[1] 创建目录如下：**

#mkdir rootfs

#cd rootfs

#mkdir bin dev etc lib proc sbin sys usr mnt tmp var

#mkdir usr/bin usr/lib usr/sbin lib/modules

**[2] 创建设备文件：**

#cd rootfs/dev

#mknod -m 666 console c 5 1

#mknod -m 666 null c 1 3

**[3]加入配置文件：**

#tar etc.tar.gz

#mv etc/\* …/root/etc/ -rf

**[4]添加内核模块：**

#cd …/linux

#make modules ARCH=arm CROSS\_COMPILE=arm-linux-

#make modules\_install ARCH=arm INSTALL\_MOD\_PATH=…/rootfs

**[5] 编译/安装busybox**

BusyBox 是一个集成了一百多个最常用linux命令和工具的软件。BusyBox 包含了一些简单的工具，例如ls、cat和echo等等，还包含了一些更大、更复杂的工具，例grep、find、mount以及telnet。有些人将 BusyBox 称为 Linux 工具里的[瑞士军刀](http://baike.baidu.com/view/4053.htm" \t "http://baike.baidu.com/_blank)。简单的说BusyBox就好像是个大工具箱，它集成压缩了 Linux 的许多工具和命令，也包含了 Android 系统的自带的shell。

1.配置busybox

#make menuconfig

Busybox Settings build Options->“Build busybox as a static binary”->

Cross Compiler prefix (arm-linux-)->Installation Options->选中“Don‘t use /usr”,

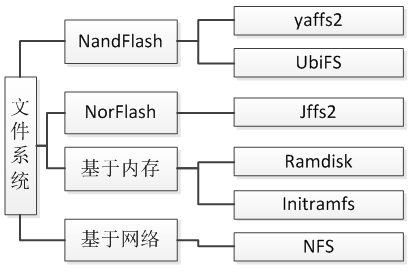
选中该项可避免busybox 被安装到宿主系统的/usr目录下，破坏宿主系统。

Busybox Installation Prefix (/xxx/rootfs)该选项表明编译后的busybox的安装位置。

**[6]编译、安装busybox**

make

make install

**4.2 挂载根文件系统到内核：**

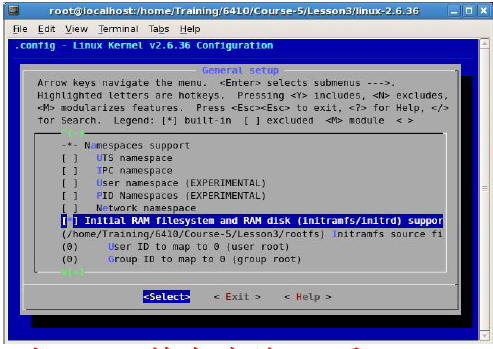
文件系统在集中载体下的格式：

**[1]使用Initramfs**

1. #cd …/rootfs/

#ln –s ./bin/busybox init 创建软连接

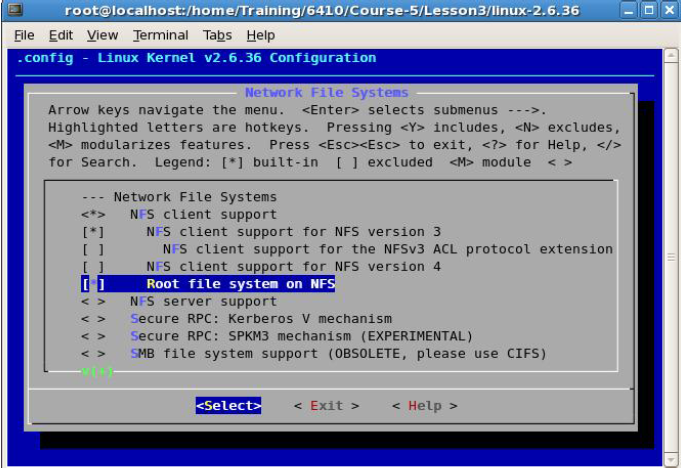
2.配置Linux内核，支持initramfs



3.编译内核

4.setenv bootargs noinitrd console=ttySAC0,115200

**[2]使用NFS**



setenv bootargs noinitrd

console=ttySAC0,115200 init=/init

root=/dev/nfs rw

nfsroot=192.168.1.17:/home/armlinux/rootfs, ip=192.168.1.80:192.168.1.17:192.168.1.0:255.255.255.0::eth0:off

**专题五 Linux内核模块**

整个Linux内核是非常庞大的，不可能全部都编译到内核文件之中。因此，通常只是将必要的文件进行编译，其他的文件作为内核模块在需要使用的时候进行动态调用。

内核模块的特点：

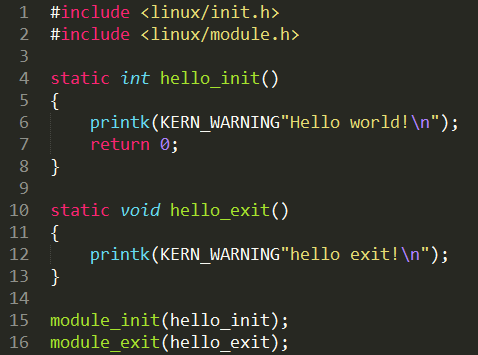
模块本身不会被编译进入到内核文件中；

可以更具需要，在内核运行的时候，动态的安装、卸载。

安装：insmod /home/dnw\_usb.ko

**查看：lsmod**

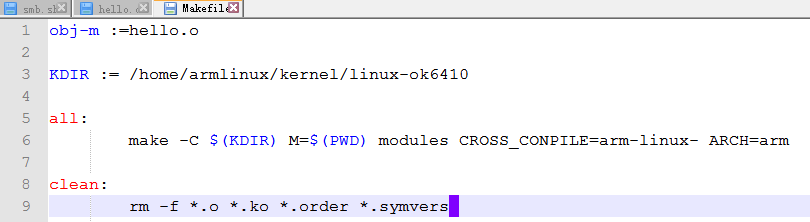
卸载：rmmod dnw\_usb

**5.1内核模块设计：**

内核模块中并没有作为程序入口的main函数。而是使用了module\_xxx的方式来作为函数的入口。由此可知内核模块的三个要素为**头文件**、**加载函数**、**卸载函数**。

module\_init作为入口函数，而module\_exit函数则是作为了卸载函数。**KERN\_WARNING**用于标识打印的优先级。

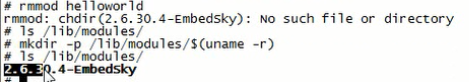
**5.2编译内核模块：**



需要配置相应的makefile文件：

Obj-m 用于指示生成的.o文件，KDIR变量用于指定内核代码所在的文件夹，生成了.ko文件之后，将这个文件拷贝到NFS目录中，即可在NFS根文件系统中使用这个内核模块。这里使用的linux内核是3.0.1版本的。

当文件系统中没有和内核版本相同的目录的时候，无法正常卸载内核模块。就需要创建一个对应的目录。在没有的情况下，可以通过下面的命令自行创建一个。



**5.3.1内核模块可选信息-模块申明**

例: int a = 3;char \*st;

module\_param(a,int,S\_IRUGO)； module\_param(st,charp, S\_IRUGO)；

**1.MODULE\_LICENSE（"**遵守的协议**"）**

申明该模块遵守的许可证协议，如：**"**GPL**"**、**"**GPL v2**"**等

**2.MODULE\_AUTHOR("**作者**")**

申明模块的作者

**3.MODULE\_DESCRIPTION("**模块的功能描述**")**

申明模块的功能

**4.MODULE\_VERSION("**V1.0**")**

申明模块的版本

**5.3.2内核模块可选信息-模块参数**

通过宏module\_param指定保存模块参数的变量。模块参数用于在加载模块时传递参数给模块。

module\_param(name,type,perm)

使用时： insmod \*.ko a=10

name:变量的名称；

type:变量类型，bool:布尔型int:整型charp:字符串型；

perm是访问权限。S\_IRUGO:读权限S\_IWUSR:写权限；

**5.3.3内核模块可选信息-符号输出**

**不同的模块**之间函数相互调用的时候必须申明符号供其他内核模块使用，否则在insmod模块的时候会报错，找不到对应的符号。

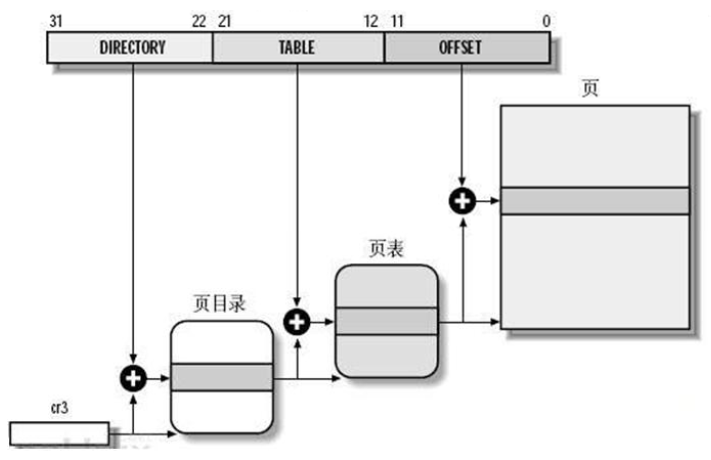
内核符号的导出使用宏：

EXPORT\_SYMBOL(符号名)

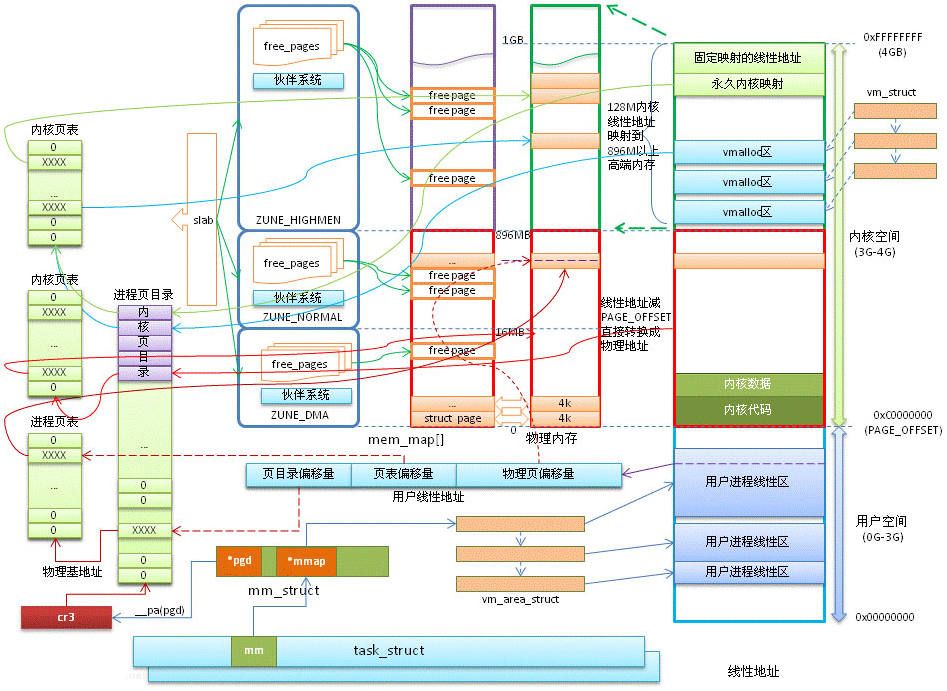
EXPORT\_SYMBOL\_GPL(符号名)

**说明：其中EXPORT\_SYMBOL\_GPL只能用于包含GPL许可证的模块。**

**专题六 Linux内核子系统**

**6.1 内存管理子系统**

CR3作为一个基地址，将DIRECTORY作为偏移找到页表的基地址，以这个地址做基地址，用TABLE作为偏移，找到物理页的基地址，最后用这个物理页的基地址加上OFFSET偏移量，最后就找到了对应的物理地址。

 作为直接映射区，物理地址的0~896MB对应虚拟地址3GB+（0~896MB）。使用时，虚拟地址直接减去3GB就可以获得对应的物理地址，这就可以很方便的完成对硬件寄存器的配置。

vmalloc区，映射关系比较复杂，可以访问的区域比较大。

永久映射区，固定用来访问高端内存的。

固定映射的线性地址，和特殊寄存器建立相应的映射关系。

**物理内存分配**

当进程开始访问虚拟内存的时候，才会分配对应的空间给相应的进程。

kmalloc函数可以直接获取一段内存的使用权限。

**6.2 进程管理子系统**

程序是在磁盘上一系列代码和数据的可执行映像，是一个静止的实体，而进程是一个执行中的程序，是一个动态的实体。

**进程和线程的区别：**进程有独立的用户空间，而线程没有，用共享用户空间的线程为用户线程，否则为内核线程。

在Linux系统中，进程及线程由PCB(process control block)负责管理。在task\_struct结构中。

**调度：**从就绪的进程中选出最合适的一个进程来执行。要素为：调度策略、调度时机、调度步骤。

**[1]调度策略：**

**SCHED\_NORMAL**(SCHED\_OTHER):普通的分时进程

**SCHED\_FIFO** :先入先出的实时进程

**SCHED\_RR**：时间片轮转的实时进程

**SCHED\_BATCH**:批处理进程

**SCHED\_IDLE**: 只在系统空闲时才能够被调度执行的进程

**[2]调度步骤：**

1). 清理当前运行中的进程；

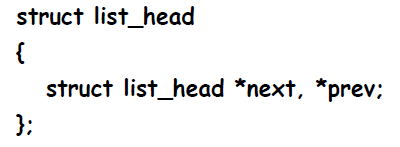
2). 选择下一个要运行的进程；

3). 设置新进程的运行环境；

4). 进程上下文切换。

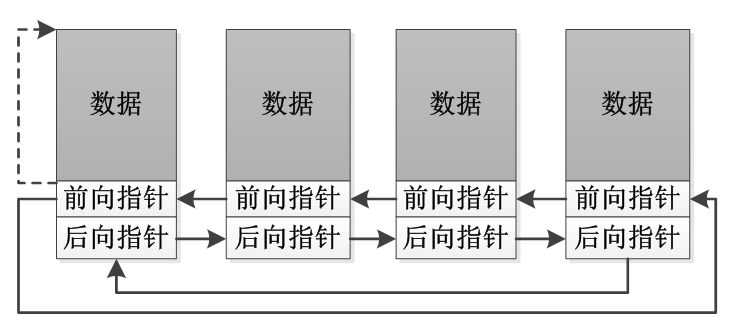
**专题七 Linux内核链表**

Linux内核链表为双向循环链表。



list\_head结构包含两个指向list\_head结构的指针prev和next，由此可见，内核的链表具备双链表功能。

Linux的内核链表不同于传统的链表，其最大的特点在于其连接时，并不将指针指向节点，而是相邻接点的前向或者后向指针，使用数据时，根据前向指针来寻找到节点的首地址，最终实现对数据的操作。



常用的Linux内核链表函数如下：

1. INIT\_LIST\_HEAD:创建链表

2. list\_add：在链表头插入节点

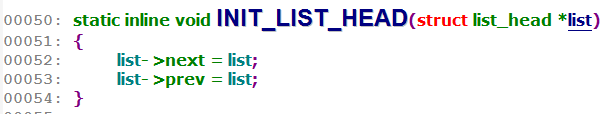
3. list\_add\_tail：在链表尾插入节点

4. list\_del：删除节点

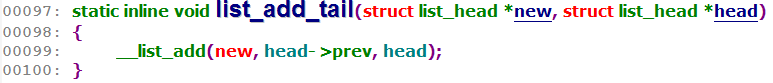
5. list\_entry：取出节点

6. list\_for\_each：遍历链表

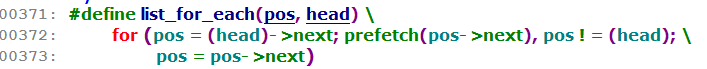
在Linux源码找找到这个INIT\_LIST\_HEAD，实现代码如下：



节点的尾部插入函数实现如下

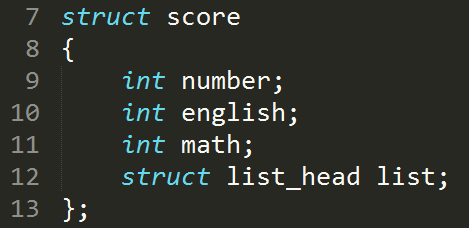
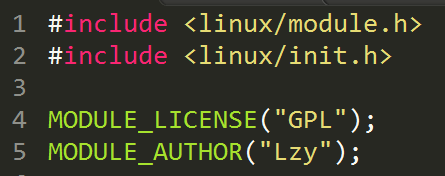


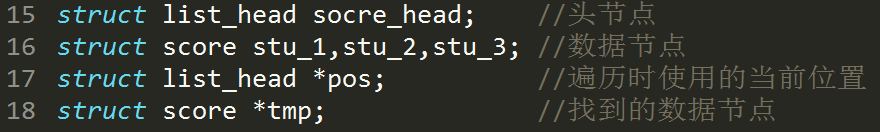
遍历节点的函数实现如下：



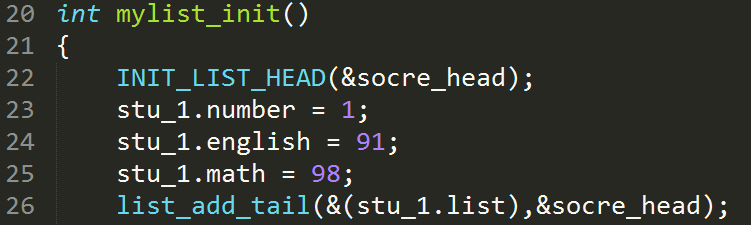
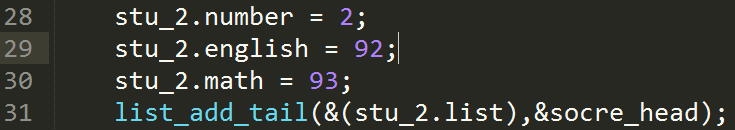
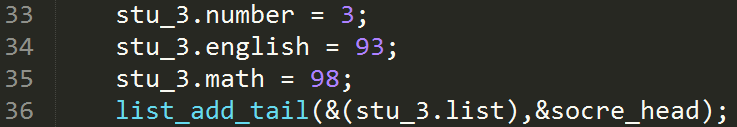
以一个具体的程序设计为例：

struct socre是一个数据节点，该节点包括了数据，和数据之下的节点指针域。

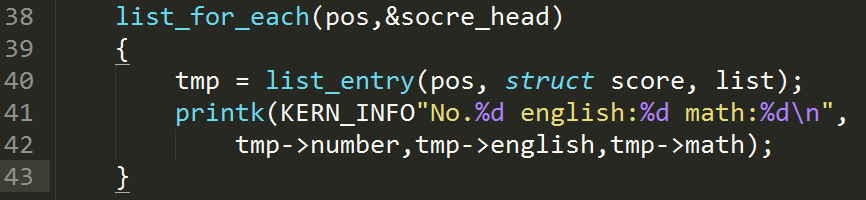




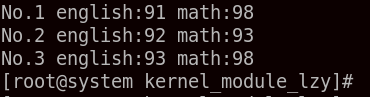
初始化节点数据，并插入到链表中。



遍历内核链表：

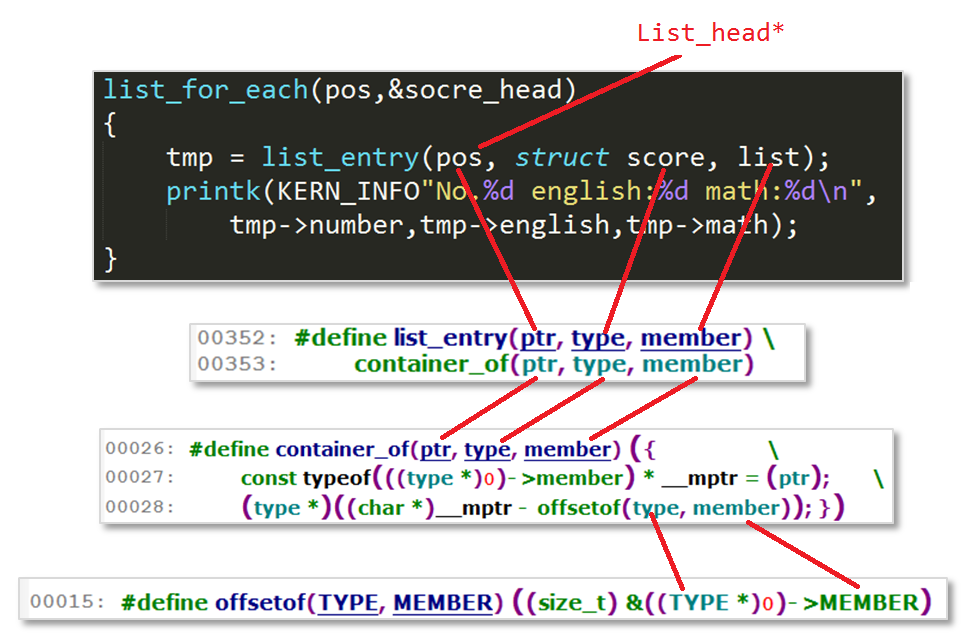


输出结果：



在整个内核链表中，实现极为精妙的函数是list\_entry这个获取节点的方法。

将节点结构的首地址定义为0x00，利用这个指针找到member在这个结构中的偏移地址，最后使用数据节点的指针域的地址减去刚才计算出来的偏移地址，最后就得到了整个数据节点的起始地址。

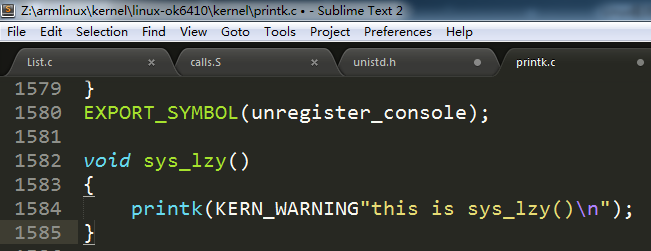


**专题八 Linux系统调用：**

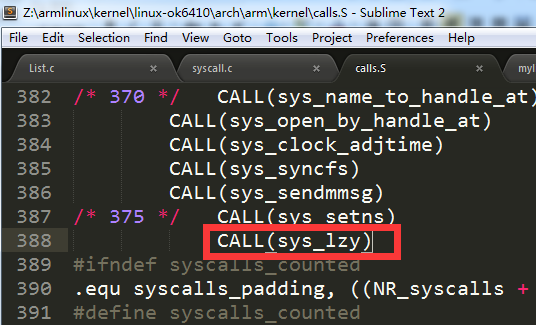
系统调用通过SWI软件中断来进行，系统会从用户空间转换到内核空间中，进而实现内核代码的系统调用。

软中断进行时，使用swi指令，将系统调用的函数编号装载到R7寄存器，swi就回去系统中查询编号为R7中数值的系统函数进行执行。

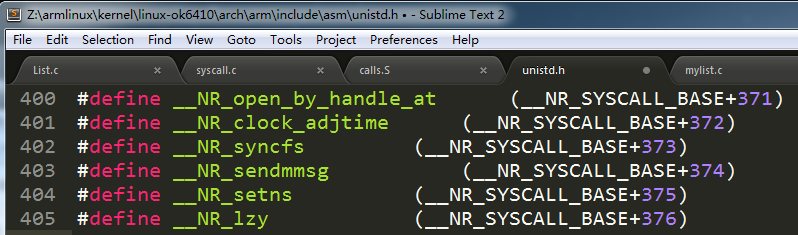
首先在linux-ok6410/arch/arm/kernel/printk.c中编写一个自定义的系统函数：



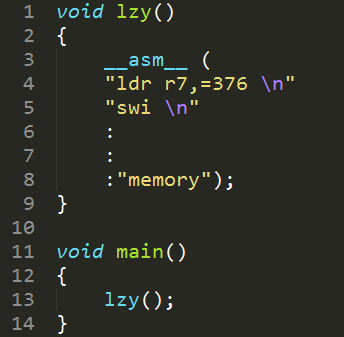
在linux-ok6410/kernel文件夹中的对应.C文件中添加相应的sys\_\*\*\*的函数，并在linux-ok6410/arch/arm/kernel/calls.S的文件夹中添加相应的编号，且务必加在最后面。



此外还要在unistd.h中添加相应的宏定义。该文件存在的目录的地址为linux-ok6410/arch/arm/include/asm。



在实现调用时，使用汇编语句进行编写，由于mov指令无法装载作为系统调用函数编号的363，故而只能使用ldr伪指令将待调用的363号函数装载到R7寄存器中，这个寄存器是有内核代码所指定的，因此只能使用这个寄存器。配置完成了这个寄存器之后，使用swi命令，将程序从用户空间转移到内核空间，并执行对应的系统函数的调用。



完成用户层程序设计后，使用命令arm-linux-gcc -static syscall.c –o syscall\_lzy进行静态编译即可。

**专题九 Linux驱动前奏**

**9.1 驱动分类：**

常规分类：**字符设备**、**块设备**、**网络设备**。**字符设备**是按照字节来访问的设备，块设备是按照一定最小单位进行访问的设备，Unix为512字节，而Linux则没有限制，网络设备为网络接口。总线分类：USB设备、PCI设备、平台总线设备。

**9.2 学习方法：**

学习重点是驱动模型、硬件操作。前半年不要去过多阅读内核代码。

**9.2 硬件访问：**

在Linux系统中，所有程序（包括内核）使用**虚拟地址**，而芯片手册中给出的硬件寄存器地址或者RAM地址则是物理地址，无法直接使用。因此，读写寄存器的第1步就是将将它的物理地址映射为虚拟地址。

**动态映射：**是指在驱动程序中采用ioremap函数将物理地址映射为虚拟地址。

原型： void \* ioremap(physaddr, size)

参数： Physaddr：待映射的物理地址

Size: 映射的区域长度

返回值：映射后的虚拟地址

**静态映射：**根据事先指定的映射关系，在内核启动时，自动地将物理地址映射为虚拟地址。在静态映射中，通过map\_desc结构来指明物理地址与虚拟地址的映射关系，该结构存放在Linux内核源码的cpu.c中。

struct map\_desc {

unsigned long virtual; /\* 映射后的虚拟地址\*/

unsigned long **pfn**; /\* 物理地址所在的页帧号\*/

unsigned long length; /\* 映射长度\*/

unsigned int type; /\* 映射的设备类型\*/

};

在完成地址映射后，就可以读写寄存器了，Linux内核提供了以下函数，来读写寄存器。

unsigned ioread8(void \*addr)

unsigned ioread16(void \*addr)

unsigned ioread32(void \*addr)

unsigned readb(address)

unsigned readw(address)

unsigned readl(address)

void iowrite8(u8 value, void \*addr)

void iowrite16(u16 value, void \*addr)

void iowrite32(u32 value, void \*addr)

void writeb(unsigned value, address)

void writew(unsigned value, address)

void writel(unsigned value, address)

**专题十 字符型驱动程序**

在Linux系统中，驱动程序开发的实质就是编译/安装一个内核模块。

**10.1 字符设备文件**

通过**字符设备文件**，应用程序可以使用相应的字符设备驱动程序来控制字符设备。创建字符设备文件的方法一般有两种：



方法1：使用mknod命令 mknod /dev/文件名 c 主设备号 次设备号

方法2：使用函数在驱动程序中创建

主设备号可以通过cat /proc/devices 查询所有主设备号。

次设备号可以设置为任意非负的数字。

例如：mknod /dev/memdev0 c 253 0

当在ARM平台下执行应用程序时，系统提示找不到文件时，在宿主机上使用arm-linux-readelf -d XXX查看是否使用了相应的动态链接库，如果是，则需要在ARM平台的/lib文件夹中补上相应的库文件，或者使用静态编译程序。

**10.2 字符设备驱动模型**

struct cdev {

struct kobject kobj;

struct module \*owner;

const struct file\_operations \*ops; //设备操作集

struct list\_head list;

dev\_t dev; //设备号

unsigned int count; //设备数

};

**设备描述结构：**驱动模型中，设备都会用内核中的一种结构来描述。字符设备使用struct cdev描述。

**主设备号：**字符**设备文件**与字符**驱动程序**对应的依据。

次设备号：同一类型设备使用一个驱动程序的识别，例如串口1和串口2。

Linux内核中使用dev\_t类型来定义设备号，dev\_t这种类型其实质为32位的unsigned int，其中**高12位为主设备号**，**低20位为次设备号。**

合并主设备号和次设备号：dev\_t dev = MKDEV(主设备号，次设备号)

主设备号=MAJOR(dev\_t dev)

次设备号=MINOR(dev\_t dev)

在分配主设备号的时候，有**静态申请**和**动态申请**两种分配方式。静态申请函数为register\_chrdev\_region，动态申请的函数为alloc\_chrdev\_region。驱动程序退出时需要注销设备号，使用函数为unregister\_chrdev\_region。

struct file\_operations dev\_fops =

{

.llseek = NULL,

.read = dev\_read,

.write = dev\_write,

.ioctl = dev\_ioctl,

.open = dev\_open,

.release = dev\_release,

};

**[1]操作函数集合**

Struct file\_operations是一个**函数指针的集合**，定义能在设备上进行的操作。结构中的函数指针指向驱动中的函数,这些函数实现一个针对设备的操作, 对于不支持的操作则设置函数指针为NULL。例如：

**[2]字符设备初始化**



**1.分配cdev**

cdev变量的定义可以采用静态和动态两种办法：

静态分配struct cdev mdev;

动态分配 struct cdev \*pdev = cdev\_alloc()

**2.初始化cdev**

cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops)

**3.注册cdev**

cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)

**[3]实现设备操作（设备方法）**



int (\*open) (struct inode \*, struct file \*) 打开设备，响应open系统调用

int (\*release) (struct inode \*, struct file \*) 关闭设备，响应close系统调用

loff\_t (\*llseek) (struct file \*, loff\_t, int) 重定位读写指针，响应lseek系 统调用

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*)

从设备读取数据，响应read系统调用

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*)

向设备写入数据，响应write系统调用

针对设备操作的函数使用了Struct file和Struct inode结构，相关信息如下：

在Linux系统中，每一个打开的文件，在内核中都会关联一个struct file，它由内核在打开文件时创建, 在文件关闭后释放。

重要成员：

loff\_t f\_pos /\*文件读写位置指针\*/

struct file\_operations \*f\_op /\*该文件所对应的操作\*/

每个文件都会关联一个inode 结构（无论打开与否），该结构用来记录文件物理上的信息。因此, 它和代表打开文件的file结构不同。一个文件没有被打开时不会关联file结构，但是却会关联一个inode 结构。

重要成员：

dev\_t i\_rdev：设备号

**3.1-open操作**

open设备方法是驱动程序用来为以后的操作完成初始化准备工作的。在大部分驱动程序中，open完成如下工作：

1.标明次设备号、2.启动设备

**3.2-release操作**

release方法的作用正好与open相反。这个设备方法有时也称为close。

**3.3-read操作**

read设备方法通常完成2件事情：

1.从设备中读取数据(属于硬件访问类操作)

2.将读取到的数据返回给应用程序

ssize\_t (\*read) (struct file \*filp, char \_\_user \*buff, size\_t count, loff\_t \*offp)

参数分析：

filp：与字符设备文件关联的file结构指针, 由内核创建。

buff : 从设备读取到的数据，需要保存到的位置。由read系统调用提供该参数。

count: 请求传输的数据量，由read系统调用提供该参数。

offp: 文件的读写位置，由内核从file结构中取出后，传递进来

**需要注意的是：buff参数是来源于用户空间的指针，这类指针都不能被内核代码直接引用，必须使用专门的函数:**

int copy\_from\_user(void \*to, const void \_\_user \*from, int n)

int copy\_to\_user(void \_\_user \*to, const void \*from, int n)

**3.4-write操作**

write设备方法通常完成2件事情：

从应用程序提供的地址中取出数据

将数据写入设备(属于硬件访问类操作)

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*)

**[4] 驱动注消**

当我们从内核中卸载驱动程序的时候，需要使用**cdev\_del**函数来完成字符设备的注销。**10.3 自己动手写一个驱动：**

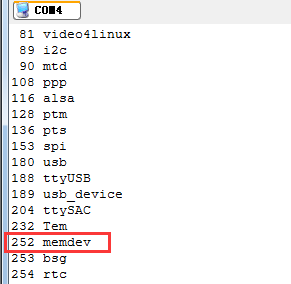
整个字符驱动模型的结构如下：



在初始化cdev的时候从linux源码中找到cdev\_init的模型，并复制到自己的文件中，然后根据函数模型配置需要初始化的mdev结构体和file\_operations函数操作集。

为了获得驱动编号，使用alloc\_chrdev\_region函数动态获取驱动编号，然后使用这个编号通过cdev\_add函数完成驱动程序的注册。由于此次使用的是虚拟出来的设备，因此就不需要再对硬件进行相应的初始化。

程序编译完成之后，放入NFS服务器根文件系统中，进入Linux系统之后，使用cat /proc/devices获取对应驱动的主设备号。



然后创建设备，使用mknod来生成设备文件。

命令 设备文件 type 主设备号 次设备号

mknod /dev/memdev0 c 252 0

编译应用程序的时候要使用arm-linux-gcc进行静态编译，使用gcc编译器编译的程序在开发板上运行程序会出现问题。

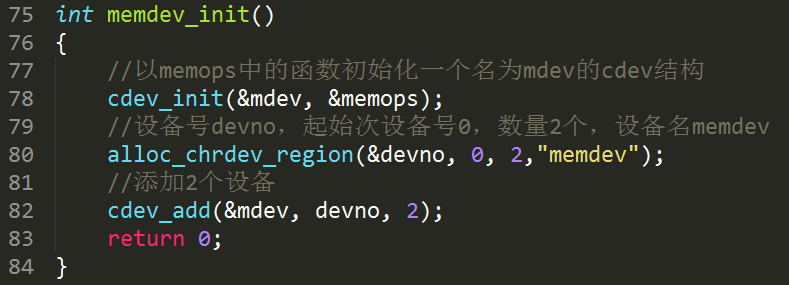
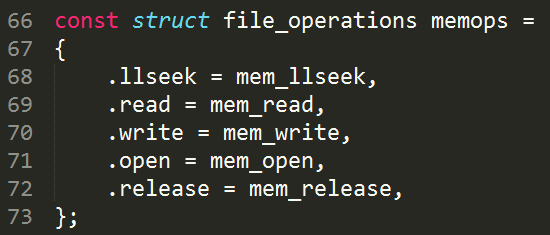
**小结：**

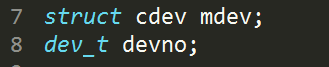
字符驱动的cdev结构体中，重要参数是设备的**操作函数集、设备号（包括主设备号和次设备号）以及设备数量**。

**操作函数集**用于指示驱动程序对字符设备文件的具体响应函数。**主设备号**用于建立字符设备文件和字符驱动的对应关系，**次设备号**用于识别使用同一个驱动程序的不同设备。

**虚拟字符驱动实例：**

在进行驱动设计的时候，首先要完成设备的初始化，这个过程包含了对设备进行初始化，需要申明一个设备结构体mdev和操作函数集结构体memfops。

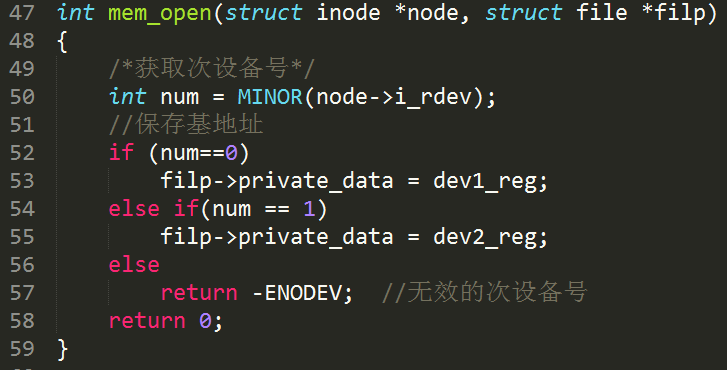




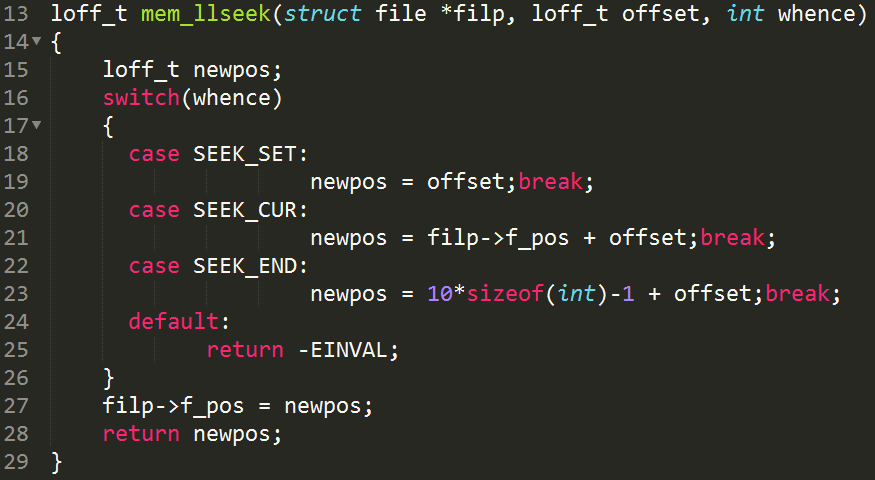
之后就需要动态的从系统内核中获取设备号，并将使用该设备号的驱动设备进行注册。

int mem\_open(struct inode \*node, struct file \*filep)函数的作用是建立使用同一个驱动程序的不同设备的连接，使文件指针能够指向对应的设备文件。

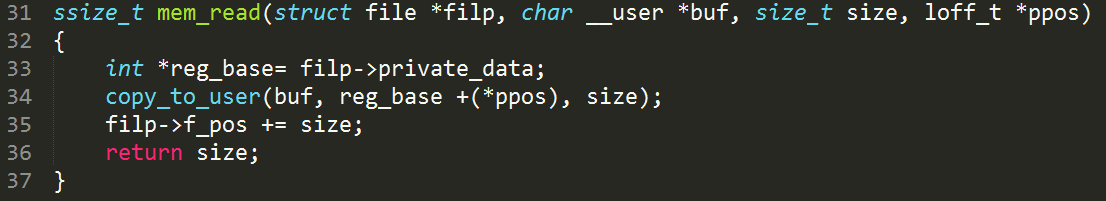
这里从node结构中的i\_rdev中分离出次设备号，并使用file结构中的private\_data成员来保存一个虚拟设备的虚拟地址。private\_data成员是专门设计出来保存一些私有数据的！！！



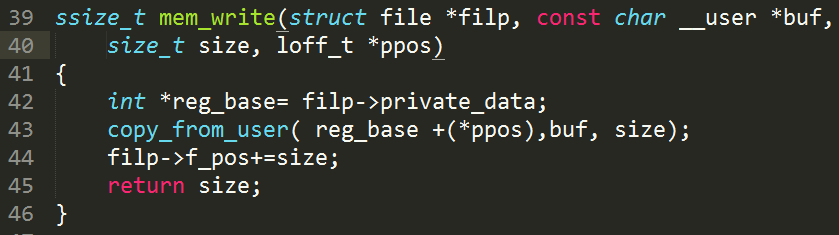
在操作函数集中：loff\_t mem\_lseek用于指明文件操作指针的位置。



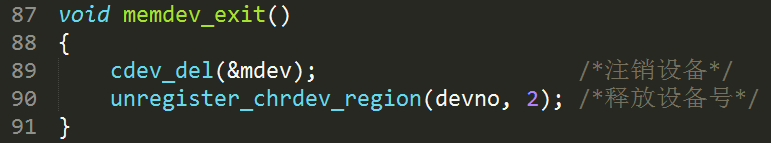
ssize\_t mem\_read函数用于完成对文件的读操作，使用的是copy\_to\_userz这个函数将数据从内核空间转移到了用户空间中。



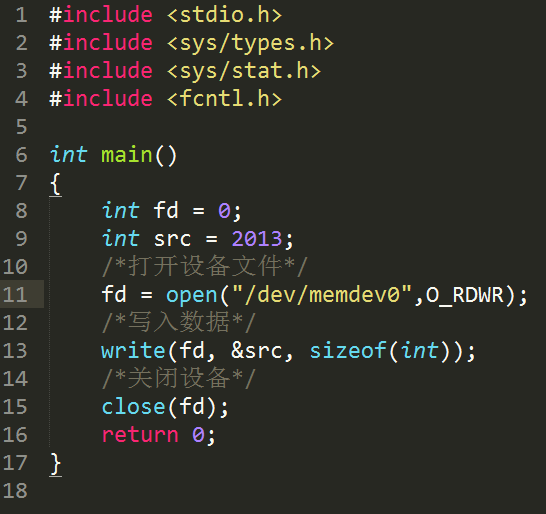
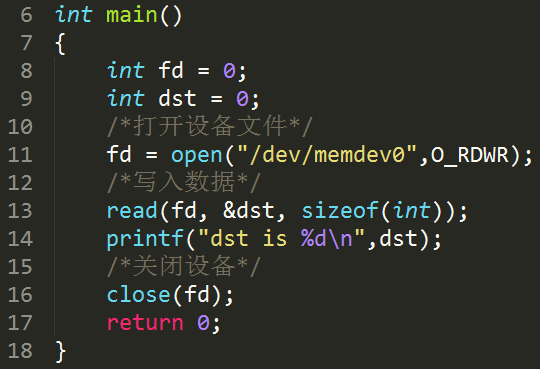
ssize\_t mem\_write函数则是和读函数相反，将数据从用户空间转移到内科空间中。



由于在linux中驱动程序是以内核模块的方式存在的，因此这个驱动程序的出口中就要使用内核模块的方式，依次注销掉设备和设备号。



测试使用的应用程序设计，使用前需用使用mknod /dev/memdev0 c 252 0指令绑定驱动和设备文件。



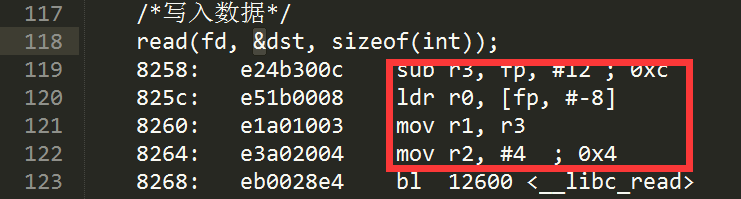
**这一部分的最后总结：**

驱动访问是由应用程序通过内核执行相应的驱动操作函数集的操作方法。用户空间到内核的过程为：应用程序从内核的入口进入内核，内核根据实际情况获取到相应的一个number，根据这个number，查表得出具体的一个操作函数，比如说为用户空间的read函数进入内核之后，固定的number为3，根据这个number，找到对应的内核函数为sys-read -> vfs\_read -> 设备file -> struct file\_operation，在这个操作函数集中就有对应的操作函数。

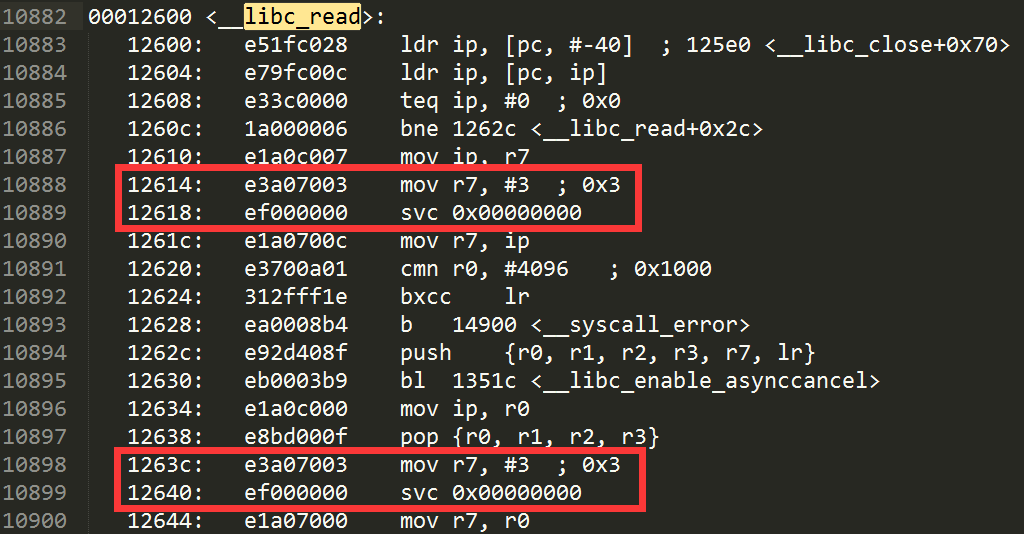
具体的操作流程为：



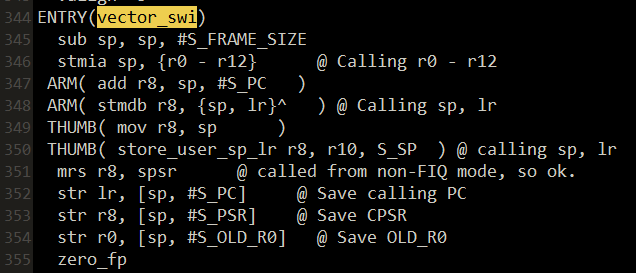
进入汇编代码，找到read的相关部分，发现这里先将参数保存到r0、r1、r2三个寄存器中，然后调用libc\_read函数。



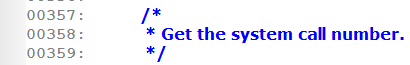
libc\_read函数部分如下，这里关键的是第10888行和10898行代码，分别将立即数3存放到了R7寄存器中（感觉R7寄存器存的数值大多数时候和系统调用的编号有关）。接下来调用了srv指令，这一指令似乎和swi相关。



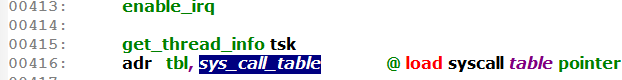
swi指令使用后，pc指针会从用户空间转移到内核控件，且入口固定。进入之后，pc指针回去取R7寄存器中的数字。根据这个数字查找需要调用的系统函数。3对应的调用时sys\_read。



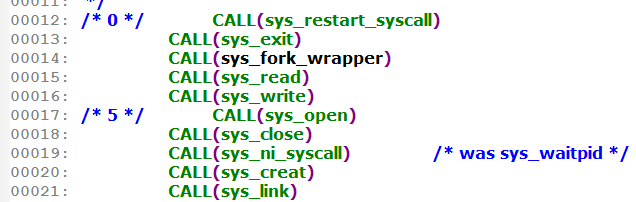
找到enrty-common.S文件的vector\_swi之后，开始向下阅读代码，找关键的注释读。get system call number这个注释告诉我们下面的代码是获取系统调用函数的编号。



再往下阅读就是sys\_call\_table，根据编号找函数。

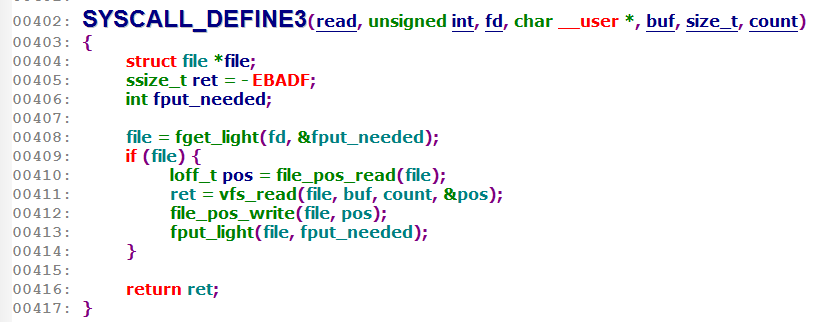


接着向下看，找到sys\_call\_table的入口，调用了一个calls.S。找到打开它。打开这张表，显然就是之前找到的那个系统调用函数表。

接下来为了找到sys\_read，一般途径根本找不到，这里打开read\_write.c这个文件。找到402行的代码，这一行是一个宏代码块。

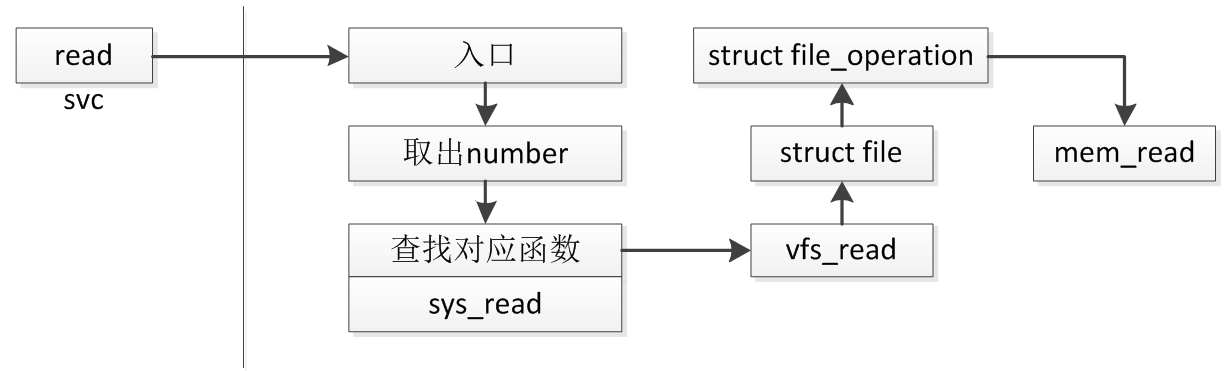
SYSCALL\_DEFIN3这个宏，将第一个参数和sys\_进行拼接，组成sys\_read函数名，后面三个参数作为函数的参数。



函数运行时，需要操作的文件描述符fd通过参数传递进入，通过该fd找到每个打开的文件对应的struct file结构（该结构包括了一个struct file\_operations \*f\_op的指针），并保存到file结构中，并使用vfs\_read函数该文件进行操作。



321行代码是整个操作的关键，使用f\_op指针指向的的read函数去获取底层设备文件的相关数据。整个流程如下：



**专题十一 字符设备控制技术**

大部分驱动程序除了需要提供读写设备的能力外,还需要具备**控制设备**的能力。比如: 改变串口的**波特率**。

**11.1 设备控制-应用函数**

在用户空间，使用ioctl系统调用来控制设备，原型是一个可变参数函数:

**int ioctl(int fd,unsigned long cmd,...)**

fd: 要控制的设备文件描述符

cmd: 发送给设备的控制命令

第3个参数是可选的参数，存在与否是依赖于控制命令(第2 个参数)。例如配置波特率时，第三个变量为波特率。

Linux 2.6.36之后的内核在应用程序使用ioctl系统调用时，由内核中如下函数响应：

**long (\*unlocked\_ioctl) (struct file \*filp, unsigned int cmd, unsigned long arg)**

fd: 要控制的设备文件描述符

cmd: 发送给设备的控制命令

参数cmd: 通过应用函数ioctl传递下来的命令

**11.2 控制实现-定义命令**

**命令的实质是一个整数**，我们通常会把这个整数分为4个段：

Type(类型/幻数) ：表明这是属于哪个设备的命令。

Number(序号) ：用来区分同一设备的不同命令

Direction ：参数传送的方向，可能的值是\_IOC\_NONE(无数据传输), \_IOC\_READ, \_IOC\_WRITE（向设备写入参数）

Size ：参数长度

Linux系统提供了下面的宏来帮助定义命令:

\_IO(type,nr)：不带参数的命令

\_IOR(type,nr,datatype)：从设备中读参数的命令

\_IOW(type,nr,datatype)：向设备写入参数的命令

例：#define MEM\_MAGIC‘m’ //定义幻数

#define MEM\_SET \_IOW(MEM\_MAGIC, 0, int)

**11.3 设备控制-实现操作**

**unlocked\_ioctl**函数的实现通常是根据命令执行的一个switch语句。但是，当命令号不能匹配任何一个设备所支持的命令时，返回-EINVAL。编程模型：

Switch( cmd)

{

Case 命令A： //执行A对应的操作

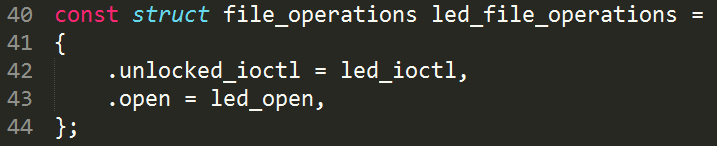
Case 命令B： //执行B对应的操作

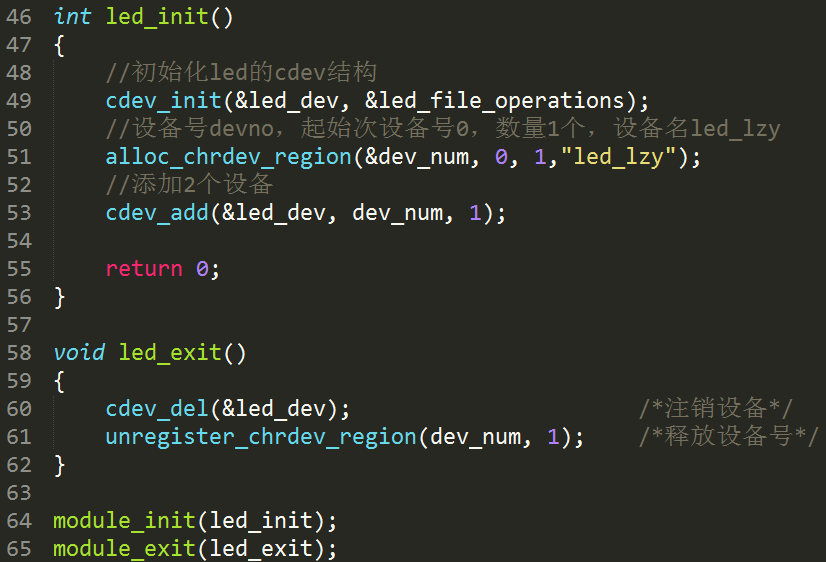
Default: // return -EINVAL

}

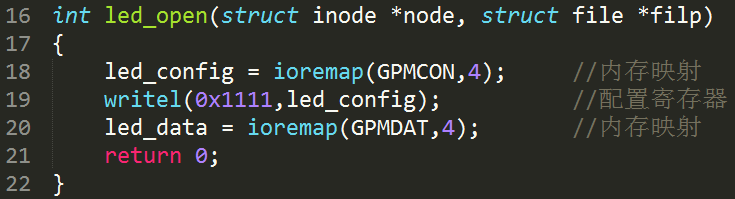
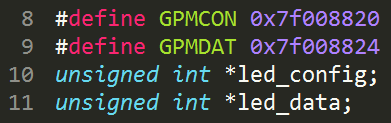
**11.4 LED驱动程序设计**

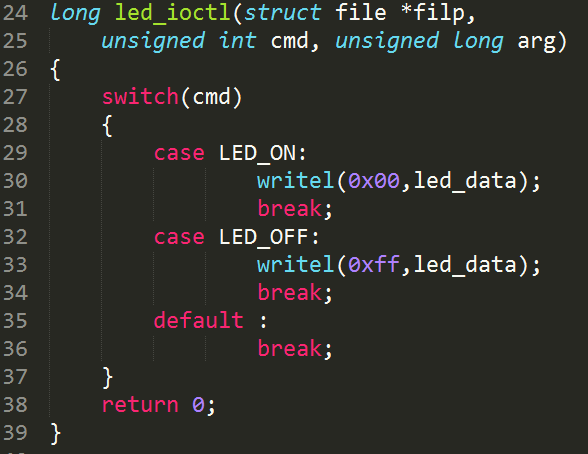
LED作为一个字符型驱动，也是通过一个内核模块来实现的。首先按照正常的流程建立一个内核模块的模型，并完成设备文件的初始和卸载函数。源文件位置：

Z:\kernel\_module\_lzy\3

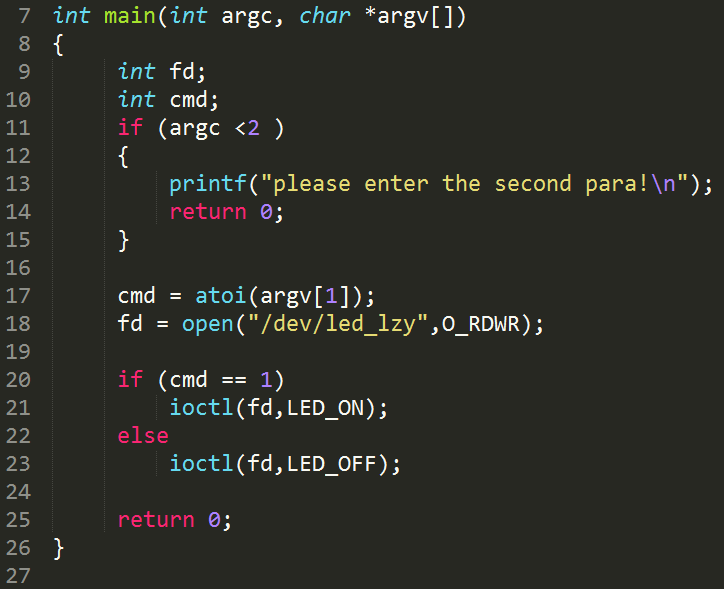


接下来实现操作函数集中的设备打开open函数：



 led控制函数：

这样就完成了驱动程序的设计，应用程序代码设计如下：



将编译好的两个文件都复制到nfs目录中，首先执行：

insmod lzy\_led.ko

 候使用ls /dev/会发现找不到led\_lzy这个设备文件，使用

#cat /proc/devices //这里可以找到led\_lzy的设备号是252

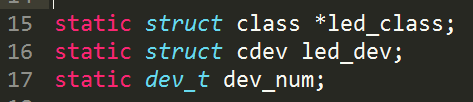
 为了建立设备和驱动程序的连接使用

#mknod /dev/led\_lzy c 252 0

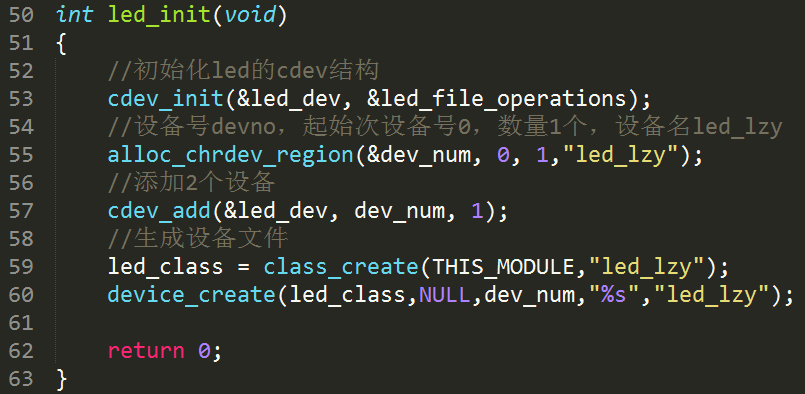
完成这一步之后，/dev/led\_lzy出现，就可以正常使用led的应用程序了。

But这样一种还有手动建立设备文件的方法真的是太操蛋了，于是做了进一步的改进：

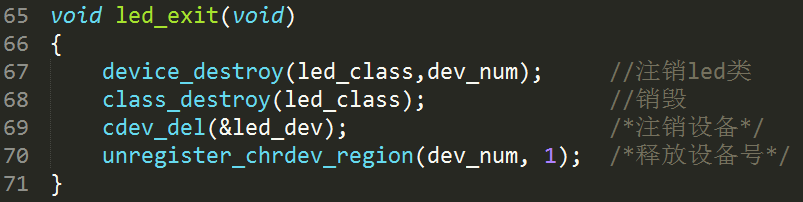
代码15行，新添加了一个设备类，当然也添加了<linux/kernel.h>的头文件。



设备初始化函数也进行了响应的修改，在59行和60行，通过这样的方法来生成一个/dev目录下的led\_lzy设备文件。



当然，卸载函数也就需要修改，原则是先初始化的后释放：



最后进行测试，非常顺利，insmod之后即可直接使用应用程序操作LED。

