

第9课 u-boot 分析

9.1: u-boot 打补丁,编译,烧写,试验 (第9课第1节 u-boot 分析之编译体验)

对比 PC 和 linux 是如何启动的

PC: 上电->启动 Bios ->引导操作系统 WINDOWS->识别 C,D 盘等 ->运行应用程序(QQ..)

嵌入式系统: 上电->启动 bootloader->引导内核 ->挂载跟文件系统 ->运行应用程序

(PC 和 linux 上面的都是——对应的)

Bootloader 的最终目的: 启动内核

PC:是从硬盘上读入内核 (他的内核就是 windows)

Linux: 从 flash 上读出内核, 读出来放入 sdram, 然后从启动内核

为了满足上面的目的, uboot 应该做的事:

1: 能读 flash

2: 能写 sdram, 要写的话必须先初始化 sdram

3: 启动内核

另外其他还要:

还要初始化时钟 (一开始开发板是以 12M 运行的, 为了加快速度)

为了观察输出的信息, 必须初始化串口

加入写 flash 功能 (我们的 uboot 是从 PC 通过 USB 或者烧录器, 或者网卡烧到 flash 上的, 所以要能写)

由于上面的要求可能还要加入 USB, 网卡的支持

1.1: 解压缩命令: tar xjf u-boot-1.1.6.tarbz2

1.2: u-boot 打补丁:

打补丁命令: patch 命令

1.3: 配置: make 100ask24x0_config

1.4: 编译: make

1.5: 烧写: 将编译好的 u-boot.bin 文件烧写到开发板

硬件连接: 板子接 openjtag, 电源, 串口 USB (方便通过串口将板子的信息显示在串口上)

如何烧写 (以下步骤在 windows 下进行, 在 linux 也可以烧写 (也使用 oflash)):

1.5.1: 打开 cmd, 进入编译好的 u-boot.bin 文件的目录

1.5.2: 输入 oflash 命令进行烧录: oflash u-boot.bin

烧录过程中串口并无数据, 烧写重启后才会有信息

注:

A: 烧写完成后, 重启开发板 (后续可以通过看 uboot 的 start.s 文件知道 uboot 在 reset 后才会运行, 运行后才有打印信息出来, 所以烧写的时候是没有信息的, 只有通过 cmd 在烧写过程中输出的 pppp 完成后才知道是否烧写 uboot 完成)

B: reset 后再倒数计时 0 之前就要按空格键才能进入菜单选项

```
-----+-----+
| S3C2440A USB Downloader ver R0.03 2004 Jan |
+-----+-----+
USB: IN_ENDPOINT:1 OUT_ENDPOINT:3
FORMAT: <ADDR(DATA):4>+<SIZE(n+10):4>+<DATA:n>+<CS:2>
NOTE: Power off/on or press the reset button for 1 sec
      in order to get a valid USB device address.

Hit any key to stop autoboot: 0
##### 100ask Bootloader for OpenJTAG #####
[n] Download u-boot to Nand Flash
[k] Download Linux kernel uImage
[j] Download root_jffs2 image
[y] Download root_yaffs image
[d] Download to SDRAM & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the Nand Flash
[s] Set the boot parameters
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection:
```

上面的菜单是自己实现的, 原版的 uboot 是没有这些菜单的

C: 按 q 退出菜单进入 uboot 界面, 可以在里面输入各种命令,

通过输入 ? 或者 help 可以知道 uboot 有哪些命令, (注意中文? 和英文的? 不一样, 中文的识别不了)

通过? 命令 可以知道该命令的具体的用法

通过输入 menu 命令可以进入菜单界面 (这个 uboot 里面 menu 这个命令也是自己加进去的)

D: 这些命令可以下载内核, 可以下载 uboot 等等

E: uboot 里面几个命令 (区别 linux 里面的环境命令):

Print(应该是 printenv, 不确定):

这个是输出 uboot 的环境变量: 其中有 Bootdelay 这个环境变量: 倒数计时, 单位 s

Set (应该是 setenv): 设置环境变量

Save (应该是 saveenv) :保存已修改的环境变量

Menu: 进入菜单

Reset: 重启开发板

例子: 设置环境变量 bootdelay 为 10s

- 1.1: print 输出有哪些环境变量 (可省)
- 1.2 : 设置 bootdelay :set bootdelay 10 (注意不是 set bootdelay=10)
- 1.3: 保存:save
- 1.4: 重启查看效果: reset (可省)

9.2: u-boot 功能, 结构, 结合 makefile 进行分析 (第9课第2节 uboot 分析之 Makefile 结构分析)

结合 makefile 分析

1: 分析配置过程(应用手册里面总结了 makefile 里面的过程可看, 这里没做笔记, 以后总结)

2: 分析编译过程

2.1: 第一个文件: start.S

2.2 : 链接地址

分析一个文件的时候想知道他的结构和怎么链接的最好的方法就是分析 makefile

我们编译 uboot 的时候是

1:配置 :make xxx_config

2:编译 :make

之所以知道要这样做是因为在 uboot 里面有个说明顶层文件目录 readme(里面介绍了 ubootd 一些约定)

结合 makefile 分析 uboot

1:分析配置过程:

配置的时候是执行 make 100ask2410_config 命令

A:在 makefile 中搜索 100ask2410_config

1:make 配置过程分析:

make 100ask24x0_config 这条配置命令干了什么?

a:确定 mkconfig 中的变量 BOARD_NAME = \$1 = 100ask24x0

100ask24x0_config : unconfig

c:在 include 目录下创建 config.mk 内容为: (从 makefile 可看出在执行 make 命令时包含了 config.mk,文件,所以要在其之前产生该文件)

```
ARCH = arm
CPU = arm920t
BOARD = 100ask24x0
SOC = s3c24x0
```

d:在 include 目录下创建 config.h 内容为:

```
/* Automatically generated - do not edit */
#include <configs/100ask24x0.h>
```

```
@$(MKCONFIG) $(@:_config=) arm arm920t 100ask24x0 NULL s3c24x0
```

unconfig:

```
@rm -f $(obj)include/config.h $(obj)include/config.mk \
$(obj)board/*/config.tmp $(obj)board/*/config.tmp
```

```
@rm -f $(obj)include/config.h $(obj)include/config.mk $(obj)board/*/config.tmp $(obj)board/*/config.tmp
```

unconfig 下面的命令清理上一次执行 make *_config 时生成的头文件和 makefile 的包含文件。主要是 include/config.h 和 include/config.mk 文件。

```
MKCONFIG := $(SRCTREE)/mkconfig
SRCTREE := $(CURDIR)
pwd :/root/WorkSpace/System/u-boot-1.1.6
```

```
/root/WorkSpace/System/u-boot-1.1.6/mkconfig 100ask24x0 arm arm920t 100ask24x0 NULL s3c24x0
$0 $1 $2 $3 $4 $5 $6
```

```
/root/WorkSpace/System/u-boot-1.1.6/mkconfig 100ask24x0 arm arm920t 100ask24x0 NULL s3c24x0
```

mkconfig 分析

```
[ "${BOARD_NAME}" ] || BOARD_NAME="$1" || cmd1 正确,cmd2 不执行
BOARD_NAME=100ask24x0
```

```
echo "Configuring for ${BOARD_NAME} board..."
echo "Configuring for 100ask24x0 board..."
```

```
if [ "$SRCTREE" != "$OBJTREE" ]
SRCTREE = /root/WorkSpace/System/u-boot-1.1.6
OBJTREE := $(if $(BUILD_DIR),$(BUILD_DIR),$(CURDIR))
OBJTREE= CURDIR = SRCTREE
```

```

else
    cd ./include
    rm -f asm
    ln -s asm-$2 asm //建立一个asm文件并指向asm-$2 即asm-arm

rm -f asm-$2/arch
    if [ -z "$6" -o "$6" = "NULL" ] //字符串比较 -z string 判断字符串是否为0 -o(即or)两者中任何一个成立即为真
$6=s3c24x0 所以不成立

else
    ln -s ${LNPREFIX}arch-$6 asm-$2/arch //这里LNPREFIX未定义为空
    即 ln -s arch-$6 asm-$2/arch
    ln -s arch-s3c24x0 asm-arm/arch //疑问:当前目录为include,下面只有asm-arm,arch-s3c24x0这
个文件并不在include下面?

if [ "$2" = "arm" ] ; then
    rm -f asm-$2/proc
    ln -s ${LNPREFIX}proc-armv asm-$2/proc
fi

执行
    rm -f asm-$2/proc
    ln -s ${LNPREFIX}proc-armv asm-$2/proc

```

到目前创建了3个链接文件:

建立一个asm文件并指向asm-\$2 即asm-arm

建立一个asm-arm/arch文件并指向arch-s3c24x0

建立一个asm-arm/proc文件并指向proc-armv

```

echo "ARCH = $2" > config.mk 若config.mk不存在则创建 //疑问:这里的echo为什么在make 100ask24x0不显示?
echo "CPU = $3" >> config.mk
echo "BOARD = $4" >> config.mk
>, >> 数据流重导向: 输出导向, 分别是『替代』和『累加』

```

编译第二部:make命令

```
ALL = $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U_BOOT_NAND)
```

```
all: $(ALL)
```

前面有

```

ifneq ($(OBJTREE),$(SRCTREE))
obj := $(OBJTREE)/
src := $(SRCTREE)/
else
obj :=
src :=
endif
export obj src

```

```

SRCTREE := $(CURDIR)
pwd :/root/WorkSpace/System/u-boot-1.1.6

```

```

all: $(ALL)即
all: /u-boot-1.1.6/u-boot.srec /u-boot-1.1.6/u-boot.bin /u-boot-1.1.6/System.map

```

```

$(obj)u-boot: depend version $(SUBDIRS) $(OBJ3S) $(LIBS) $(LDSCRIPT)
UNDEF_SYM=$(OBJDUMP) -x $(LIBS) | sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`; \
cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJ3S) \
    其实就是把start.o和各个子目录makefile生成
的库文件按照LDFLAGS连接在一起,生成ELF文件u-boot和连接时内存分配图文件u-boot.map。
--start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
-Map u-boot.map -o u-boot

```

分析 LNDIR := \$(OBJTREE) 为当前目录

```

depend dep:
    for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir _depend ; done
SUBDIRS= tools \
examples \
post \
post/cpu

相当于:
make -C tools _depend
make -C examples _depend
make -C post _depend
make -C post/cpu _depend

```

而在顶层makefile中找不到_depend的定义,发现有include \$(SRCTREE)/rules.mk

在rules.mk中有定义:

```

_depend:      $(obj).depend

$(obj).depend:$(src)Makefile $(TOPDIR)/config.mk $(SRCS)
    @rm -f $@
    @for f in $(SRCS); do \
        g=`basename $$f` | sed -e 's/\(.*\)\\.w/\1.o/'; \
        $(CC) -M $(HOST_CFLAGS) $(CPPFLAGS) -MQ $(obj)$$g $$f >> $@ ; \
    done

```

而 SRCS 这个变量在各个子目录下 makefile 中有定义

.depend 文件的生成规则为一个 for 循环,也就是找出各个源文件的依赖关系并放入.depend 文件

```

version:

    @echo -n "#define U_BOOT_VERSION \"U-Boot \" > $(VERSION_FILE); \
    echo -n "$(U_BOOT_VERSION)" >> $(VERSION_FILE); \
    echo -n "$(shell $(CONFIG_SHELL) $(TOPDIR)/tools/setlocalversion \
        $(TOPDIR)) >> $(VERSION_FILE); \
    echo "\" >> $(VERSION_FILE)

    最终在 include/version_autogenerated.h 写入 #define U_BOOT_VERSION "U-Boot 1.1.6"

```

```

$(SUBDIRS):

    $(MAKE) -C $@ all

```

```

分析:SUBDIRS = tools \
              examples \
              post \
              post/cpu

              等价于
              make -C tools all
              make -C examples all
              make -C post all
              make -C post/cpu all
              执行 tools ,examples ,post,post\cpu 子目录下面的 make 文件

```

```

$(OBJJS):

    echo $(OBJJS)
    $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))

```

分析:

```

OBJJS = cpu/$(CPU)/start.o
ifeq ($(CPU),i386)
OBJJS += cpu/$(CPU)/start16.o
OBJJS += cpu/$(CPU)/reset.o
endif
ifeq ($(CPU),ppc4xx)
OBJJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc83xx)
OBJJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc85xx)
OBJJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc86xx)
OBJJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),bf533)
OBJJS += cpu/$(CPU)/start1.o cpu/$(CPU)/interrupt.o cpu/$(CPU)/cache.o
OBJJS += cpu/$(CPU)/cplbhdr.o cpu/$(CPU)/cplbmgr.o cpu/$(CPU)/flush.o
endif
#调用 addprefix 函数,在每个 OBJJS 前面加上 obj
OBJJS := $(addprefix $(obj),$(OBJJS))

```

分析:

```
OBJJS = cpu/arm920t/start.o
```

```
OBJJS := $(addprefix $(obj),$(OBJJS))
```

分析:

```

OBJJS := $(addprefix $(obj),cpu/arm920t/start.o)
obj := $(OBJTREE)/
obj := $(CURDIR)/
OBJJS := $(CURDIR)/cpu/arm920t/start.o

```

```

$(OBJJS):

    echo $(OBJJS)
    $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))
    $(CPU)在 config.mk 中定义 CPU =arm920t
    分析: make -C cpu/arm920t $(OBJJS)
    make -C cpu/arm920t $(OBJJS)
    make -C cpu/arm920t $(CURDIR)/cpu/arm920t/start.o
    即:
    make -C arm920t start.o

```

```

LIBS = lib_generic/libgeneric.a
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC

```



```
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a \
        fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
LIBS += net/libnet.a
LIBS += disk/libdisk.a
LIBS += rtc/librtc.a
LIBS += dtb/libdtb.a
LIBS += drivers/libdrivers.a
LIBS += drivers/nand/libnand.a
LIBS += drivers/nand_legacy/libnand_legacy.a
LIBS += drivers/usb/libusb.a
LIBS += drivers/sk98lin/libsk98lin.a
LIBS += common/libcommon.a
LIBS += $(BOARDLIBS)

LIBS := $(addprefix $(obj),$(LIBS))

$(LIBS):
        $(MAKE) -C $(dir $(subst $(obj),,$@))
即:make -C $(dir $(subst $(obj),,$(LIBS)))
```

顶层 makefile 中并没有定义 \$(obj)u-boot: depend version \$(SUBDIRS) \$(OBJJS) \$(LIBS) \$(LDSCRIPT) 中的 LDSCRIPT 规则

而是包含了

```
include $(OBJTREE)/include/config.mk
```

在其中我们可以看到

```
ifndef LDSCRIPT
#LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds.debug
ifeq ($(CONFIG_NAND_U_BOOT),y)
LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot-nand.lds
else
LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds
endif
endif

LDLFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE) $(PLATFORM_LDFLAGS)
```

没找到哪有定义 CONFIG_NAND_U_BOOT =y

所以

```
LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds
```

```
UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`; \
cd $(LNDIR) && $(LD) $(LDLFLAGS) $$UNDEF_SYM $(__OBJJS) \
--start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
-Map u-boot.map -o u-boot
```

分析:

1:字符“`”,叫做反引号:这个反引号表示在 shell 脚本中,用于包含要执行的命令,否则不加反引号,后面的内容就视为普通字符串了。

\$(OBJDUMP) -x \$(LIBS) |sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq 将前面的执行结果送给 UNDEF_SYM

2:关于 空格加上反斜杠

即" \" ,意思是,一行内容太多了,分成多行书写,此时,就要用到上面一行的后面,加上空格再加一个反斜杠\",然后接下来写后面的内容,这样,对于系统处理,就可以把你的输入看成是一行的输入,否则,不加空格和反斜杠,就视为两行

\$(OBJDUMP)在 config.mk 中定义 OBJDUMP = \$(CROSS_COMPILE)objdump

即 OBJDUMP = arm-linux-objdump -x 用于导出库文件中所有的头信息

objdump -x 参数的含义:

" -x, --all-headers Display the contents of all headers"

即显示(.a 库文件)所包含全部的头(信息).因此上面的就是把所有的.a 库文件中的头信息都导出来给那个变量。

3:用 sed 处理字符串:后面的字符"|",表示管道,即前面的内容,字节通过管道,送到后面的 sed 命令去处理。

```
sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
```

先看参数 -n,-e:

-n, --quiet, --silent 取消自动打印模式空间

-e 脚本, --expression=脚本 添加“脚本”到程序的运行列表

-e 的含义,简单的来说,就是把-e 后面跟的内容,当做要执行的脚本(命令)放入到命令列表中(以待后面分别执行这些命令去处理字符流/串)

对于符合.*\(__u_boot_cmd_.*\),即以单个任意字符或者多个任意字符开头,后面符合__u_boot_cmd_,后面再跟上单个任意字符或者多个任意字符的字符串,用-u 再加上__u_boot_cmd_.*,即__u_boot_cmd_跟上单个任意字符或者多个任意字符,来代替

最后所有的处理的结果,就是这样的了:

```
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_bdfinfo
-u__u_boot_cmd_bdfinfo
-u__u_boot_cmd_go
-u__u_boot_cmd_reset
-u__u_boot_cmd_go
-u__u_boot_cmd_reset
-u__u_boot_cmd_bootm
-u__u_boot_cmd_bootm
-u__u_boot_cmd_flinfinfo
-u__u_boot_cmd_erase
-u__u_boot_cmd_protect
-u__u_boot_cmd_flinfinfo
-u__u_boot_cmd_erase
```

.....

(4)sort 和 uniq:后面用|sort|uniq,即通过管道送给 sort 去排序,然后将排序后的结果用 uniq 去去掉重复的

最后得到这样的结果:

```
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_base
-u__u_boot_cmd_bdfinfo
-u__u_boot_cmd_bootm
-u__u_boot_cmd_bootp
-u__u_boot_cmd_cmp
-u__u_boot_cmd_cp
-u__u_boot_cmd_crc32
-u__u_boot_cmd_end
-u__u_boot_cmd_erase
-u__u_boot_cmd_flinfno
-u__u_boot_cmd_go
-u__u_boot_cmd_help
-u__u_boot_cmd_loadb
-u__u_boot_cmd_loads
.....
```

(5)分号用于单行输入中,分隔多个命令:

执行玩前面的 UNDEF_SYM 赋值后面加了个分号";"用于在一次单行输入中,分割多个命令,这里的意思就是,

先 UNDEF_SYM=XXXX,然后加上;,结束了前面的变量赋值,后面又去执行其他的命令,这里的是

```
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome
```

(6)&&,表示逻辑与,前面为真,后面的命令才执行:

后面的是:

```
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome && arm-linux-ld XXXX
```

意思是先 cd 切换目录,如果切换目录成功,接着执行 arm-linux-ld XXXX,

(7)ld 的执行:

```
arm-linux-ld -Bstatic -T /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome/board/ams/as3536/u-boot.lds -Ttext 0x00000000 $UNDEF_SYM
cpu/arm926ejs/start.o \
--start-group lib_generic/libgeneric.a ..... board/ams/as3536/libas3536.a --end-group -L /home/crifan/develop/buildroot/buildroot-
2009.11/output/staging/usr/bin/./lib/gcc/arm-linux-uclicgnueabi/4.3.4 -lgcc \
-Map u-boot.map -o u-boot
```

分析:

(A)-Bstatic:意思是后面的接下来要处理的库,都是 static,静态方式链接进来.

(B)-T /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome/board/ams/as3536/u-boot.lds

-T FILE 或-script FILE 读链接描述文件名,以确定符号等的定位地址

这里,意思就是,使用 u-boot.lds,以确定符号等的定位地址,即根据这个 lds 中的指示,把对应的不同段的数据,放在指定的地方。

(C)u-boot.lds 的简单分析:

首先使用下面的语句连接得到 ELF 格式的 u-boot.最后转化为二进制格式的 u-boot.bin,S-Record 格式的 u-boot.srec. LD_FLAGS 确定了连接的方式,其中“-T board/smdk2410/u-

boot.lds -Ttext 0x33f80000”字样指定了程序的布局和地址。u-boot.lds 的文件如下:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
/*指定输出可执行文件是 elf 格式,32 位 ARM 指令,小端*/
OUTPUT_ARCH(arm)
/*指定输出可执行文件的平台为 ARM*/
ENTRY(_start)
/*指定输出可执行文件的起始代码段为_start*/
(.globl _start _start: b      start_code//cpu/arm920t/start.S)
SECTIONS
{
/*指定可执行 image 文件的全局入口点,通常这个地址都放在 ROM(flash)0x0 位置。必须使编译器知道这个地址,通常都是修改此处来完成*/
. = 0x00000000; /*;从 0x0 位置开始*/
. = ALIGN(4);/*代码以 4 字节对齐*/
.text
:
{
cpu/arm920t/start.o (.text) /*代码的第一个代码部分*/
*(.text) /*其它代码部分*/
}

. = ALIGN(4);
.rodata : { *(.rodata) } /*指定只读数据段*/

. = ALIGN(4);
.data : { *(.data) } /*指定读/写数据段*/

. = ALIGN(4);
.got : { *(.got) } /*指定 got 段, got 段是 uboot 自定义的一个段,非标准段*/

. = .;

/*把__u_boot_cmd_start 赋值为当前位置,即起始位置*/
__u_boot_cmd_start = .;

/*指定 u_boot_cmd 段, uboot 把所有的 uboot 命令放在该段.*/*
.u_boot_cmd : { *(.u_boot_cmd) }

/*把__u_boot_cmd_end 赋值为当前位置,即结束位置*/
__u_boot_cmd_end = .;

. = ALIGN(4);
__bss_start = .; /*把__bss_start 赋值为当前位置,即 bss 段的开始位置*/
.bss (NOLOAD) : { *(.bss) } /*指定 bss 段,告诉加载器不要加载这个段*/
_end = .;

/*把_end 赋值为当前位置,即 bss 段的结束位置*/
```

```
}
```

这样代码的都是以 `0x33f80000+0x0` 为基准开始, 如果你从 `nandflash` 启动, 测试前 4K 的代码的地址都是在 `0x0`, 那么 4K 的代码的实现可以通过位置无关指令 `b` 来实现。`b` 指令的程序不依赖代码存储的位置 - 即不管这条代码放在什么位置, `B` 指令都可以跳转到正确的位置。

`bootloader`, 内核等程序刚开始运行时。他们所处的地址通常不等于运行地址, 在程序的开头, 先使用 `b, bl.mov` 等位置无关的指令将代码从 `flash` 等设备中复制到内存的运行地址处, 然后跳转到运行地址去执行。

9.3: u-boot 命令分析

9.3.1: u-boot 命令分析 (第 9 课第 3 节 uboot 分析之源码第一阶段)

本期视频目标:

- 1: u-boot 的核心: 命令
- 2: 实现
- 3: 添加一个命令

这一节可以参考 pdf: <<Uboot 中 start.S 源码的指令级的详尽解析 v1.6>>

之前的裸板程序我们写的流程是:

- 1: 硬件相关的初始化:
关看门狗
初始化时钟
初始化 sdram
- 2: 若程序很大:
把程序从 `nandflash` 拷贝到 `sdram`
- 3: 设置栈: 调用 C 函数

而 `uboot` 的实现流程其实就比较复杂的单片机程序而已, 所以流程其实和单片机程序差不多。

Uboot 的流程

- 1: 硬件相关的初始化
- 2: 设置栈: 所谓设置栈就是让 `sp` 寄存器指向某块内存
- 3: 调用 `c` 函数: `c` 函数里面实现读出内核, 启动内核等等

我们的第一个文件之前分析的是: `cpu/arm920t/start.S` 从这个文件入手分析:

分析 `start.s`

第一条指令:

```
_start:    b        reset                                //跳转到 reset

reset:                                           //进入 SVC32 模式
/*
 * set the cpu to SVC32 mode
 */
/* turn off the watchdog */                      //关闭看门狗
/*
 * mask all IRQs by setting all bits in the INTMR - default //屏蔽所有的中断
 */
//Cpu 初始化
//设置栈: 这里有讲解 uboot 的栈的排布, 可以看下
//初始化时钟
//代码重定位
//清除 bss 段: 就是初始值为 0 的变量, 所以没必要保存在 bss 段里
//调用 c 函数: start_arm_boot
```

可以看出主要的流程和单片机的程序其实差不多, 这里调用 `c` 函数之前的部分称为 `uboot` 的第一阶段

9.3.2: u-boot 命令分析 (第 9 课第 3 节 uboot 分析之源码第二阶段)

本期视频目标:

- 1: u-boot 的核心: 命令
- 2: 实现
- 3: 添加一个命令

讲解第二阶段代码:

Uboot 目标: 启动内核

- 1: 从 `flash` 中读出内核
- 2: 启动内核

以此为目的去分析第二阶段源码(其他的忽略掉):

- 1: 从 `start.s` 中找到 `_start_armboot: .word start_armboot`, 进入分析 `start_armboot` 函数

start_armboot (void)函数分析:

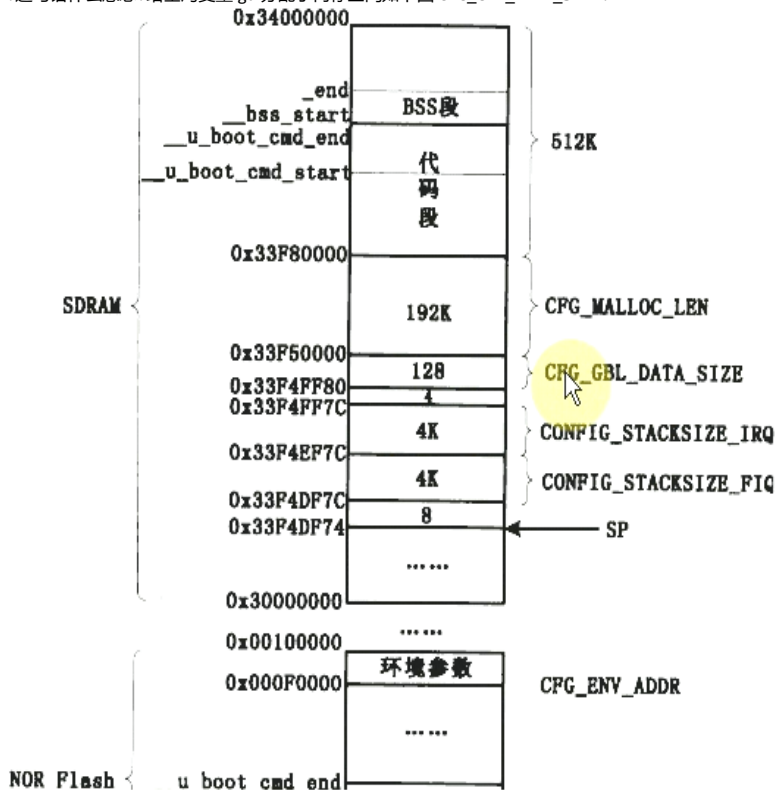
```
gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
```

/*分析: 1:gd 这个结构体指针在那定义:

arch/arm/include/asm/global_data.h 中定义了 gd 这个全局变量:

```
#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r8")
```

2:这句话什么意思?给全局变量 gd 分配了内存空间如下图 CFG_GBL_DATA_SIZE:



*/

```
for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {  
    if ((*init_fnc_ptr)() != 0) {  
        hang ();  
    }  
}
```

/*分析:这里调用 init_sequence,他里面有一系列的初始化,比如

Board_init:1:它里面设置了将机器 ID 保存在全局变量 gd 中

2:将 uboot 要传递给内核的启动参数也设置了保存的地址为 0x30000100*/

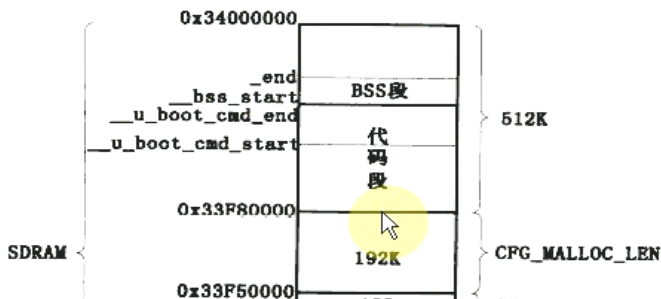
```
size = flash_init (); //对 norflash 的初始化  
/* flash_init 调用的是 cfi_flash.c 文件中的,  
if (!flash_detect_legacy (BANK_BASE(i), i)) //这个函数就用来识别是哪种 flash*/  
flash_get_size (BANK_BASE(i), i);  
*/
```

由于我们要从 flash 中读取出内核,那么就必须有以下能力,

支持 flash 的读:uboot 为了方便还增加了写功能,所以必须支持 flash 的读写,但是由于对于

Nandflash 和 norflash,都必须要知道是哪种型号的 norflash 或者 nandflash,所以要对检测 norflash 的型号的识别,nandflash 在后面会做*/

```
mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);
```



/*这里的 mem_malloc_init 对于的就是 CFG_MALLOC_LEN,由于 uboot 是单片机程序,其中用到的什么 malloc 和 free 都需要我们自己实现,malloc 称为堆,而这 192K 就是用来实现堆的分配和释放,而 mem_malloc_init 就是对堆的一些初始化,(这里自己要好好理解下,单片机的堆和栈)*/

```
nand_init(); /* go init the NAND */  
nand_init_chip  
nand_scan
```

/*这个 nand_init 函数用来识别出 flash 的型号,norflash 在上面已经做了*/

```
/* initialize environment */
```

```
env_relocate (); //初始化环境变量
```

```
/*在 uboot 中输入 print 打印出来的就是环境变量:
```

```

OpenJTAG>
OpenJTAG>
OpenJTAG> print
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
ipaddr=192.168.7.17
serverip=192.168.7.11
netmask=255.255.255.0
mtdids=nand0=nandflash0
mtdparts=mtdparts=nandflash0:256k@0<bootloader>,128k<params>,2m<kernel>,-<root>
bootdelay=10
stdin=serial
stdout=serial
stderr=serial
partition=nand0.0
mtddevnum=0
mtddevname=bootloader

Environment size: 444/131068 bytes
OpenJTAG>

```

每个环境变量等号后面就是他们的值，

那么环境变量的值哪里来的，来自两种：

1:在代码中写死的，这种环境变量称为默认的，

2:flash 上保存的

当 uboot 启动后要读取环境变量的值，他首先去 flash 上读取，如果 flash 上没有，就使用默认的环境变量。而 env_relocate (); 这个函数就是实现这个的

// start_armboot 在这里之前我们只是知道我们的 uboot 有了 norflash 和 nandflash 的读写能力，还没有做我们的读取内核，启动内核的功能*/

```

for (;;) {
    main_loop ();

    s = getenv ("bootdelay"); //获得环境变量 bootdelay 的值
    // 我们的环境变量的值是个字符串，simple_strtol 将字符串转成数字
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
    s = getenv ("bootcmd");

    if (bootdelay >= 0 && s && !abortboot (bootdelay)) {
        # ifdef CONFIG_AUTOBOOT_KEYED
            int prev = disable_ctrlc(1); /* disable Control C checking */
        # endif
        # ifdef CFG_HUSH_PARSER
            {
                printf("Booting Linux ...\n");
                run_command (s, 0);
            }
        # else
            printf("Booting Linux ...\n");
            run_command(s,0);
        # endif
    }
}

```

/*上面这段代码的意思：如果我们在倒计时的时间内没有按下空格的话就会打印

printf("Booting Linux ...\n");并 run_command(s,0)，而这个命令 s 的值的来源是：

s = getenv ("bootcmd");即 bootcmd 的这个环境变量的值，这个环境变量的值我们查看

uboot 的环境变量看出是

```

OpenJTAG> print
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
baudrate=115200

```

可以看出 bootcmd 就只有两个参数，其实是两个命令：

nand read.jffs2 0x30007FC0 kernel

这个参数实现的就是从 flash 读取内核：nand read.jffs2 是读取内核的命令，0x30007FC0 是将内核读到哪里的地址，从 kernel 这里读，kernel 是个分区

Bootm 0x30007FC0 这条命令就是从 0x30007fc0 这里启动内核

之前分析的都是倒计时的时间内未按下空格的情况，下面分析的是在倒计时的时候按下空格的情况

ifdef CONFIG_MENUKEY//这里开始的就是未按下空格的分支部分：

```

...
run_command("menu", 0); //由于前面的宏都没有定义(CONFIG_MENUKEY,
//CONFIG_AMIGAONEG3SE 等等)，所以会直接运行到这里
//其实这句是韦东山，原版的 uboot 没有，他实现的就是如下：
never reach in order to get a valid USB device address.
Hit any key to stop autoboot: 0
##### 100ask Bootloader for OpenJTAG #####
[n] Download u-boot to Nand Flash
[k] Download Linux kernel uImage
[j] Download root_jffs2 image
[y] Download root_yaffs image
[d] Download to SDRAM & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the Nand Flash
[s] Set the boot parameters
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection: q
OpenJTAG>
OpenJTAG>
Ready

```

当我们按下 q 的时候就会退出来，然后等待我们输入命令，其实就是进入了个死循环，然后他会根据我们输入的命令来解析命令

```

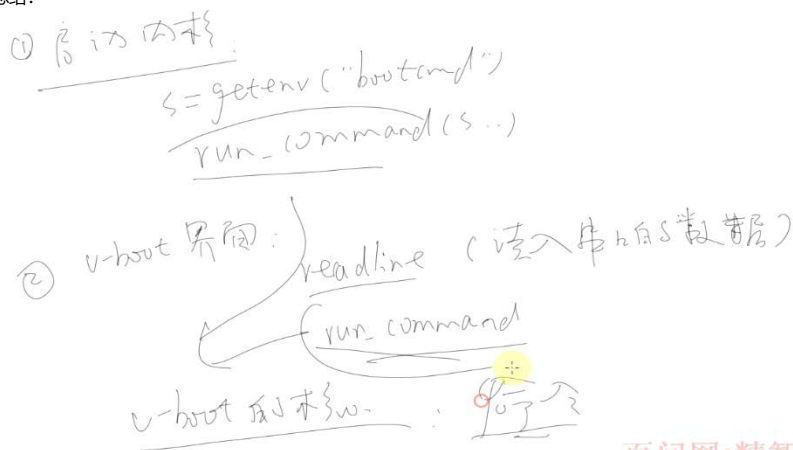
#else
for (;;) {
    //这里就是上面说的进入死循环，
    //等待我们串口输入东西

    #ifdef CONFIG_BOOT_RETRY_TIME
    len = readline (CFG_PROMPT); //读取串口输入的数据
    ...经过一系列解析
    rc = run_command (lastcommand, flag); //运行命令

```

}

总结:



百问网·精智24

这节我们想知道 u-boot 是如何读取内核并且启动的,发现是先读取环境变量的值,然后发现环境变量的值其实就是两个命令,最终使用通过 run_command 来实现的,而我们在 u-boot 界面 (这里的界面不是老师自己做的 menu,而是 u-boot 自带的等待我们输入命令的界面)中也是通过命令来实现各个动作的,所以 u-boot 的核心就是命令,我们接下来就要来知道 u-boot 的命令实现

Q: mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);这个老师说说的是设置堆,这个怎么理解?

堆和栈的区别:<http://blog.csdn.net/yinqiaohua/article/details/44947375>,而这里的这个函数就实现了堆的实现

9.4: u-boot 命令分析 (第9课第4节 u-boot 分析之 u-boot 命令实现)

本期视频目标:

- 1: u-boot 的核心: 命令
- 2: 实现
- 3: 添加一个命令

命令如何实现呢?

我们在串口上输入一些字符串,然后开发板会执行一些动作(动作函数)

我们 u-boot 里面会有很多命令的结构体,

其中肯定会包含命令的名字,和命令的执行动作

然后 run_command 就会在这些结构体里面依次去找命令的名字,如果名字匹配,就会调用相应的命令函数,那么我们分析下 run_command 函数看他是如何执行命令的

```
run_command (const char *cmd, int flag)
{
    for (inquotes = 0, sep = str; *sep; sep++) {
        if ((*sep == '\\' &&
            (*(sep-1) != '\\'))
            inquotes=!inquotes;

        if (!inquotes &&
            (*sep == ';' && /* separator */
            (sep != str) && /* past string start */
            (*(sep-1) != '\\')) /* and NOT escaped */
            break;
    }
}
```

/*分析

解析命令,将命令一个个提取出来,如果串口以下输入多个命令要以分号隔开,他就会分别执行这多个命令,比如同时输入 print 和 md.w 0 两条命令,他是可以分别解析执行的:

```
OpenJTAG> print; md.w 0
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
ipaddr=192.168.7.17
serverip=192.168.7.11
netmask=255.255.255.0
mtdids=nand0=nandflash0
mtdparts=mtdparts=nandflash0:256k@0 (bootloader),128k (params),2m (kernel),-(root)
bootdelay=10
stdin=serial
stdout=serial
stderr=serial
partition=nand0,0
mtddevnum=0
mtddevname=bootloader
*/
```

提取命令的参数

```
process_macros (token, finaltoken); //解析宏
if ((argc = parse_line (finaltoken, argv)) == 0) {
    rc = -1; /* no command at all */
    continue;
}
```

/*用 parse_line 来解析提取命令参数:比如说我们串口输入的命令是 md.w 0

这个对于 u-boot 来说是个字符串,他就会被解析为:argv[0]= "md.w"argv[1]="0"*/
if ((cmdtp = find_cmd(argv[0])) == NULL)

/*argv[0]里面一般都是存放命令的名字,后面的都是命令参数,cmdtp 就是我们之前说的每个命令都被保存在了一个结构体里面,这里显然就是通过命令的名字来查找对应的命令,

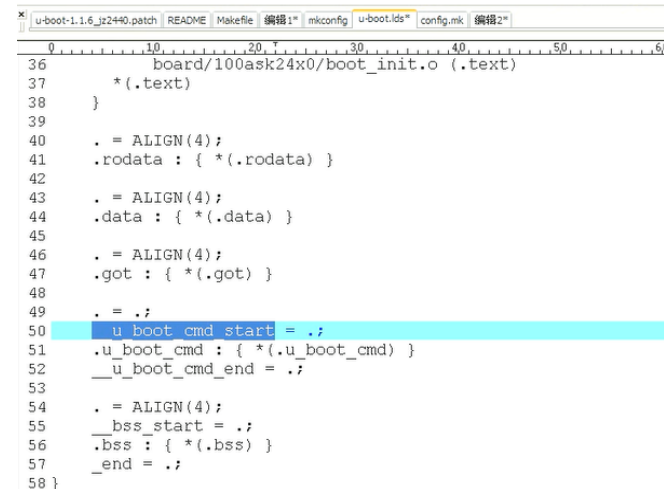
命令的结构体如下:

```
struct cmd_tbl_s {
    char      *name;           /*命令的名字          */
    int        maxargs;        /*最大的参数的个数    */
    int        repeatable; /*知否可重复执行,如果可以重复执行的命令,直接一直按回车就可以一直重复执行 */
    int        (*cmd)(struct cmd_tbl_s *, int, int, char *[]);/* 这条命令对应的执行函数*/
    char      *usage;          /* 短的使用信息,比如直接输入 help 出来的就是短的信息 */
#ifdef CFG_LONGHELP
    char      *help;           /* 长的帮助信息,比如我们输入 help+某个命令,就是长的信息 */
#endif
};
```

接下来我们来看看他是如何通过结构体的名字来查找命令的,即分析:

```
find_cmd (const char *cmd)
{
    for (cmdtp = &__u_boot_cmd_start;
         cmdtp != &__u_boot_cmd_end;
         cmdtp++) {
        if (strcmp (cmd, cmdtp->name, len) == 0) {
            if (len == strlen (cmdtp->name))
                return cmdtp;
        }
    }
}
```

/*__u_boot_cmd_start 和 __u_boot_cmd_end 是通过链接脚本传递的参数



```
*
board/100ask24x0/boot_init.o (.text)
36      *(.text)
37
38      }
39
40      . = ALIGN(4);
41      .rodata : { *(.rodata) }
42
43      . = ALIGN(4);
44      .data : { *(.data) }
45
46      . = ALIGN(4);
47      .got : { *(.got) }
48
49      . = .;
50      __u_boot_cmd_start = .;
51      .u_boot_cmd : { *(.u_boot_cmd) }
52      __u_boot_cmd_end = .;
53
54      . = ALIGN(4);
55      __bss_start = .;
56      .bss : { *(.bss) }
57      _end = .;
58 }
```

上面的 for 循环就是从保存在__u_boot_cmd_start 和 __u_boot_cmd_end 之间的所有命令结构体和形参传递过来的命令的名字依次比较,如果匹配,就把保存在__u_boot_cmd_start 和 __u_boot_cmd_end 直接的该命令的结构体读取出来

接下来,我们来看看 uboot 是如何将这些命令的结构体保存在

__u_boot_cmd_start 和 __u_boot_cmd_end 这个段之间的,从链接脚本中可以知道这两个之间的段称为.u_boot_cmd 段

搜索.u_boot_cmd,在 command.h 中找到了:

```
#define Struct_Section __attribute__((unused,section (".u_boot_cmd")))
```

我们以 bootm 命令为例,看他是如何调用到 bootm 对应的执行函数的

搜索 bootm,我们在 cmd_bootm.c 中发现,下面实际上是宏的调用,只不过这个宏是定义结构体变量,所以可以在函数体外调用,如果宏是个函数的话,肯定不能放在函数体外调用的

```
U_BOOT_CMD(
    bootm,          CFG_MAXARGS,    1,    do_bootm,
    "bootm - boot application image from memory\n",
    "[addr [arg ...]]\n" - boot application image stored in memory\n"
    "\tpassing arguments 'arg ...'; when booting a Linux kernel,\n"
    "\t'arg' can be the address of an initrd image\n"
#ifdef CONFIG_OF_FLAT_TREE
    "\tWhen booting a Linux kernel which requires a flat device-tree\n"
    "\ta third argument is required which is the address of the of the\n"
    "\tdevice-tree blob. To boot that kernel without an initrd image,\n"
    "\tuse a '-' for the second argument. If you do not pass a third\n"
    "\ta bd_info struct will be passed instead\n"
#endif
);
```

/*分析:

其中 U_BOOT_CMD 他是一个宏,它在 command.h 中定义:

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage, help}
```

那么上面的就被替换为:

```
cmd_tbl_t __u_boot_cmd_bootm Struct_Section = { "bootm", CFG_MAXARGS, 1, do_bootm, usage, help} //usage 和 help 我没有替换,太长了  
展开时注意的是
```

1: ##代表连字符, #表示将后面的转为字符串

2: Struct_Section 我们之前在搜索.u_boot_cmd 这个段的时候发现了

```
#define Struct_Section __attribute__((unused,section (".u_boot_cmd")))
```

所以最终替换为:

```
cmd_tbl_t __u_boot_cmd_bootm __attribute__((unused,section (".u_boot_cmd")))
= { "bootm", CFG_MAXARGS, 1, do_bootm, usage, help} //usage 和 help 我没有替换,太长了
```

那么上面的语句代表什么意思呢? 定义了一个结构体 __u_boot_cmd_bootm,这个结构体的类型是 cmd_tbl_t,这个结构体有个属性: __attribute__((unused,section

```
("u_boot_cmd"))
```

将它的段属性强制存放在了 `u_boot_cmd`,

这样在代码中所有用 `U_BOOT_CMD` 这个宏来定义的结构体都会保存在这个段中,这样就实现了在 `__u_boot_cmd_start` 和 `__u_boot_cmd_end` 中存放所有命令的结构体.

好,这里我知道了 `find_cmd` 如何通过命令的名字来查找相应的命令结构体,回过头来继续分析 `run_command` 函数,刚才分析到了 `run_command` 的:

```
if ((cmdtp = find_cmd(argv[0])) == NULL)
```

后面的就是分析匹配的命令的结构体的参数,并且调用命令的实现函数了

我们这里:

新增一个命令:hello 命令,输入该命令时,他要打印 `helloworld`

这个具体的可以看看网上的,没做记录

实现要点

1:通过 `U_BOOT_CMD` 这个宏来定义 hello 命令对应的 `cmd_tbl_t` 结构体

2:实现 `cmd_tbl_t` 结构体里面的动作函数

3:将上述的两个东西都放入 `common` 文件夹内,并修改 `common` 里面的 `makefile` 让他们编译

总结:这一节我们了解到了 `uboot` 是如何去读取串口上传入的参数,并如何进行解析,并根据解析的结果来实现相应的处理动作的

我们之前分析过,`uboot` 最终要读出内核并且启动他,发现是先读取环境变量的值,然后发现环境变量的值其实就是两个命令,最终使用通过 `run_command` 来实现的,那么下节就分析,`uboot` 是如何通过这两个命令来实现他的终极目标的

9.5: 启动内核的过程分析 (第9课第5节 u-boot 分析_uboot 启动内核)

本期视频目标:

1: 读出内核

2: `uImage` 头部

3: 设置启动参数:TAG

通过 9.3.2 节的分析我们知道了 `uboot` 读出内核和启动内核是依赖与以下的语句:

```
s = getenv ("bootcmd");
```

```
.....
    if (bootdelay >= 0 && s && !abortboot (bootdelay)) {
        .....
        printf("Booting Linux ...\n");
        run_command (s, 0);
    }
.....
```

而 `s` 就是环境变量的 `bootcmd` 的值即如下两条命令:

```
nand read.jffs2 0x30007FC0 kernel 和 bootm 0x30007FC0
```

然后通过 `run_command` 来解析和执行这两条命令

下面来分别来分析这两条命令:

```
nand read.jffs2 0x30007FC0 kernel
```

这条命令就是从 `nandflash` 把内核读到 `0x30007FC0` 这个地址去

这条命令需解决两个问题

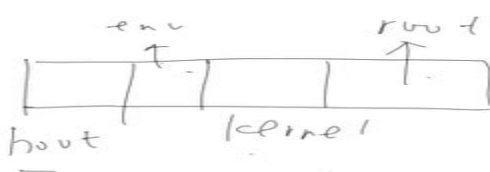
1:从 `nandflash` 哪里读内核 :从 `kernel` 分区读

2:读到哪里去: 读到 `0x30007FC0` 去,这个可以修改的

分区的概念:

PC 机上,每个硬盘前面会有一个分区表,我们可以把它分成好几个区,然后放到不同的 PC 机上,分区都不会变

对于嵌入式 linux 来说,我们的 `flash` 上面没有分区表,我们在脑子里面把这个 `flash` 分为几个区:bootloader,环境变量 `env` 区,内核区,文件系统分区(一般称为 `root` 区)



既然没有分区表,也就是没有东西记录我们的分区的方式,那么这些分区怎么体现呢,只能在源码中写死,那么我们的 `uboot` 是在哪里写死呢?

是在每个型号的配置文件中写死的,我们是在 `100ask24x0.h` 中:

```
#define MTDPARTS_DEFAULT "mtdparts=nandflash0:256k@0(bootloader)," \
    "128k(params)," \
    "2m(kernel)," \
    "-(root)"
```

/*定义了一个 `mtdparts` 分区,这个分区位于 `nandflash` 上面,前面的 256K 是 `bootloader`,128k 是环境变量,2M 是内核,剩下的是 `root`,这些分区的名字不重要,重要的是分区的地址

我们可以在 `uboot` 的命令行中输入 `mtd` 来看下他的分区信息


```
Environment size: 444/131068 bytes
OpenJTAG> ntd

device nand0 <nandflash0>, # parts = 4
#: name      size      offset      mask_flags
0: bootloader 0x00040000 0x00000000 0
1: params     0x00020000 0x00040000 0
2: kernel     0x00200000 0x00060000 0
3: root       0x0fda0000 0x00260000 0

active partition: nand0,0 - <bootloader> 0x00040000 @ 0x00000000

defaults:
ntdids : nand0=nandflash0
ntdparts=nandflash0:256k@0 (bootloader),128k (params),2m (kernel),-(root)
OpenJTAG>
```

可以知道 kernel 分区的起始地址是 0x00060000,大小为 0x00200000(2M)

那么这句:

nand read.jffs2 0x30007FC0 kernel 就可以替换为:

nand read.jffs2 0x30007FC0 0x00060000 0x00200000

这句话的意思是从 nandflash 0x00060000 去读内核,读 0x00200000 这么大,读到 0x30007FC0 去

这句和 nand read.jffs2 0x30007FC0 kernel 是一样的效果

再次声明,这个分区名字不重要,重要的是他的起始地址和长度

然后我们来看看这个命令的相应的执行函数是什么,怎么找到他的执行函数呢?我自己总结了三种方法:

1:先找他的命令的结构体,肯定也是 U_BOOT_CMD 这个宏来定义的,搜索命令的名字 nand

就可以找到了,

2:其实我们之前分析了 bootm 这个命令,发现他的实现这个命令的文件是放在 common 文件里面,而且源码文件的名字是 cmd_bootm,我们也可以知道直接在 common 中找 cmd_nand.c 就行了,

3:其实还可以我们之前分析了 bootm 命令的执行函数是 do_bootm,那么 nand 的执行函数的名字一般就是 do_nand

最终结果是

```
U_BOOT_CMD(nand, 5, 1, do_nand,
1: "nand - NAND sub-system\n",
1: "info - show available NAND devices\n",
1: "nand device [dev] - show or set current device\n",
1: "nand read[jffs2] - addr off[ partition size]\n",
1: "nand write[jffs2] - addr off[ partiton size - read/ write 'size' bytes starting\n",
1: " at offset 'off' to/ from memory address 'addr'\n",
1: "nand read.yaffs addr off size - read the 'size' byte yaffs image starting\n",
1: " at offset 'off' to memory address 'addr'\n",
1: "nand write.yaffs addr off size - write the 'size' byte yaffs image starting\n",
1: " at offset 'off' from memory address 'addr'\n",
1: "nand read.raw addr off size - read the 'size' bytes starting\n",
1: " at offset 'off' to memory address 'addr', without oob and ecc\n",
1: "nand write.raw addr off size - write the 'size' bytes starting\n",
1: " at offset 'off' from memory address 'addr', without oob and ecc\n",
1: "nand erase [clean] [off size] - erase 'size' bytes from\n",
1: " offset 'off' (entire device if not specified)\n",
1: "nand bad - show bad blocks\n",
1: "nand dump[.oob] off - dump page\n",
1: "nand scrub - really clean NAND erasing bad blocks (UNSAFE)\n",
1: "nand markbad off - mark bad block at offset (UNSAFE)\n",
1: "nand biterr off - make a bit error at offset (UNSAFE)\n",
1: "nand lock [tight] [status] - bring nand to lock state or display locked pages\n",
1: "nand unlock [offset] [size] - unlock section\n");
```

具体的如何去读内核的 do_read 函数我就不分析了,稍微说下这个命令:

nand read.jffs2 这个为什么 read.jffs2 呢?jffs2 是一种文件格式,但这里和文件格式没什么关系,这用用 jffs2 这个后缀的话是读内核的时候不需要页对齐的,所以比较方便

接下来我们看看 uboot 是如何启动内核的

Bootm 0x30007FC0,直接找到 bootm 的执行函数 do_bootm

先讲下内核的格式:

Flash 上存的内核的格式称为 uImage

uImage 即 头部(64BYTE)+真正的内核

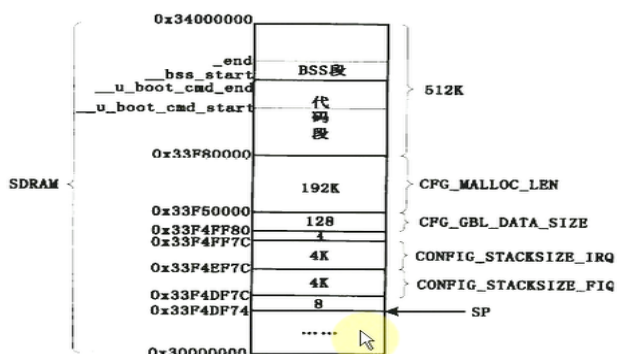
头部的作用:

我们先看下头部的结构体(这个结构体在 do_bootm 函数里面有使用到,所以先分析下,uImage,然后再分析 do_bootm 函数)

```
typedef struct image_header {
.....
uint32_t ih_load; /* Data Load Address:表示加载地址,就代表内核运行的时候你要把它先放在哪里 */
uint32_t ih_ep; /* Entry Point Address:入口地址,就是你要运行这个内核的时候只要直接跳到这个入口地址即可 */
.....
} image_header_t;
```

我们的读取内核的命令是 nand read.jffs2 0x30007FC0 kernel

这个 0x30007c0 是读取内核到什么地方的地址,我们之前说过这个 0x30007c0 可以随意写,只要不破坏 SDRAM 的前面的内容即可就是 0x34000000 到 0x33f4df74 之间不可以即:



我们之所以可以随便设置读取内核到某处的地址,就是因为 uImage 有这个头部,他知道内核的加载的地址和入口地址,如果他发现真正的内核不在他的加载地址的话,他就会将内核移动到他的加载地址,最后再跳到入口地址

如果说我们的真正的内核刚好就处于 ih_load 这个加载地址的时候,就不用移动真正的内核了,就节约了移动的时间,这就是我们这个 0x30007FC0 的由来.

对于我们这个开发板的内核的加载地址是:0x30008000

(Q:难道每个开发板的内核存放地址不一样?在哪里看的)

0x30008000- x30007FC0=64BYTE(刚好就是内核头部的大小)

```
int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
    memmove (&header, (char *)addr, sizeof(image_header_t)); //读出内核头部
    if(ntohl(hdr->ih_load) == data) {如果头部的加载地址等于内核真正的存放地址
        printf (" XIP %s ... ", name);
    } else
        memmove ((void *) ntohl(hdr->ih_load), (uchar *)data, len); //将真正的内核移动到//ih_load 这个加载地址去,这个 data 就是我们真正的内核
    do_bootm_linux (cmdtp, flag, argc, argv, addr, len_ptr, verify); //启动内核
    /*理论上启动内核只要跳转到入口地址即可,但是之前还需要做一些事情,
    1:uboot 告诉内核一些启动参数:
    2:跳转到入口地址,启动内核
    这些都在 do_bootm_linux 函数中实现(在 armlinux.c 中定义)
    */
}
```

do_bootm_linux 中的设置启动参数的实现

由于 uboot 读取内核到某个地址之后,就会跳转去执行内核了,那么 uboot 就不存在了, uboot 和内核其实是两个独立的

映像, 内核启动后 uboot 映像便不起作用了, 而且其空间也会被内核初始化

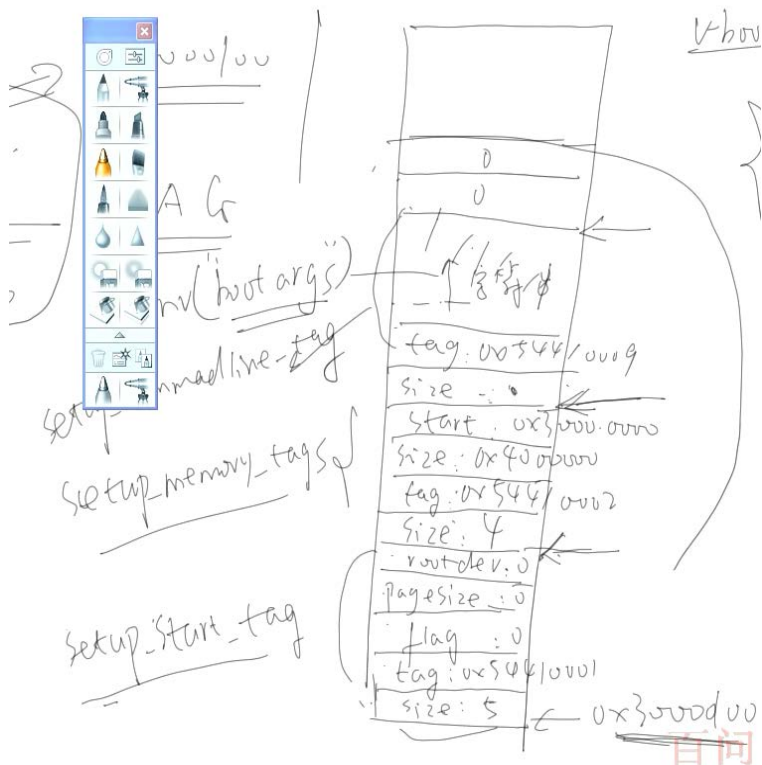
怎么实现让内核去读取 uboot 传过来的参数呢?

在某个地址按某种格式保存数据即可,然后内核去规定的地址读取数据

这个格式称为 TAG,这个地址对我们开发板来说是 0x30001000

我们在 do_bootm_linux 发现有很多 tag,其中 start_tag 和 end_tag 是必须的,其他的是可选的,我们下面分析下面几个 tag

```
setup_start_tag (bd);
    params = (struct tag *) bd->bi_boot_params;
    //搜索 bi_boot_params,发现
    ---- bi boot params Matches (7 in 4 files) ----
    100ask24x0.c (board\100ask24x0): gd->bd->bi_boot_params = 0x30000100;
    所以这个参数是放在 0x30000100 的
setup_memory_tags (bd);
    这里面告诉内核我们内存的大小
    这里的数据来源与我们的 board.c 中的 start_armboot->init_sequence 中的->dram_init
    int dram_init (void)
    {
        gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
        gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;
        return 0;
    }
    就是把这里的参数传递给内核
setup_commandline_tag (bd, cmdline);
    这里面告诉内核我们的命令行中的 bootargs
    我们这里 uboot 中的 bootargs 的内容是
    OpenJTAG> print
    bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
    bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
setup_end_tag (bd);
```



do_bootm_linux 中的启动内核的实现

下面三步就完成了启动内核的动作

```
void (*theKernel)(int zero, int arch, uint params);//定义 theKernel 为函数指针
theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);//将 theKernel 指向头部的入口地址
theKernel (0, bd->bi_arch_number, bd->bi_boot_params);//直接调用 theKernel,就可以跳到入口地址去执行了,这就启动内核
```

theKernel 的三个参数,第二个参数是机器 ID,第三个参数就是之前的 u-boot 要传递给内核的启动参数的地址

Q1:u-boot 给内核是如何传递参数的?这些参数的值是怎么来的,保存在哪的,从哪来开始读取的

Q2:我们说我们这个板子的内核的装载地址是 0x30008000,这是从哪确定的

第 10 课 内核启动流程分析

1. 内核打补丁,编译,烧写,试验
2. 内核功能,结构,结合 Makefile , Kconfig 进行分析
3. 内核启动的过程分析
 - 3.1: 比较机器 ID
 - 3.2: 解析 u-boot 传入的启动参数
 - 3.3: 挂接根文件系统,执行第一个应用程序

10.1: 内核打补丁,编译,烧写,试验 (第 10 课第 1 节 内核启动流程分析之编译体验)

1: 解压缩

```
tar -xjf linux-2.6.22.tar.bz2
```

2: 打补丁

```
patch -p1 < ../linux-2.6.22_patch
```

3: 配置: 有三种方法

3.1: 执行 make menuconfig,从头到尾每一项都自己配置(里面有成千上万种配置项,很复杂)

3.2: 使用默认的配置,在默认的配置上基础上修改

方法: 目前我们不知道内核有哪些默认配置,我们如何查找当前内核有哪些默认配置呢?

1: 在内核文件中搜索 find -name "*defconfig"

发现每个架构的都有配置文件,其中 arm 架构的均在 ./arch/arm/configs 目录下

2: 进入 ./arch/arm/configs 目录下查看有哪些文件: cd ./arch/arm/configs,其中的 s3c2410_defconfig 的配置文件与我们要找的单板相似

3: 再次进入顶层 makefile 目录即 linux_2.6.22.6 的目录,执行 make s3c2410_defconfig

4: 再 执行 make menuconfig (这样就会在 make defconfig 的默认配置的基础上去执行)

执行完 make s3c2410_defconfig 后他最后提示 configuration written to .config

说明这条命令最后是将配置写入了 .config 文件里面,而后的 make menuconfig 就是读取这个 .config 文件

3.3:使用厂家提供的配置文件

方法:我们可以猜测 厂家的配置文件应该是和我们要条用的.config 文件是同等级的,而且 make menuconfig 是调用.config 的,所以我们只要将厂家的 config 文件复制成为.config 文件就 ok 了,然后再执行 make menuconfig 就能间接的执行厂家的 config 文件了

对于本内核的厂家配置文件就是韦老师提供的 config_ok 文件

步骤:

```
1:cp config_ok .config
2:make menuconfig
```

4:编译:

方法一:

直接执行 make 或者

方法二:如果我们想生成 uimage:

Uboot 支持的内核是有个头部+真正的内核

而 uimage 就是生成一个 uboot 支持的内核(即满足以上的格式)

执行 make uImage(注意 I 是大写)

我们想通过 uboot 里面的菜单来下载内核,而里面的菜单命令都是通过 usb 来下载的

将 uImage 文件烧录到开发板上,通过 USB 来烧录,所以要安装 USB 的驱动:

驱动的路径: F:\linux\资料盘\windows\drivers\win7-64-DNW-USB

安装成功后在设备管理器会有:100ask eBlocks SEC SOC Test Board

Windows10 的话提示获取设备描述符失败,按照网上的教程

通过 uboot 的命令来下载内核到单板上

1:必须有 uboot 在单板上

2:重启开发板,在倒计时为 0 前进入 uboot 的命令菜单

3:选择下载 uImage 内核命令 k:串口上会提示: USB host is connected. Waiting a download.

4:打开 DNW 工具来发送要烧写的 uImage(选择 DNW 里面的 USB port->transmit->选择发送的文件)

5:通过 DNW 发送的过程中可以在串口上看到它一边接收一边发送,在接收到之后就会擦出 NAND flash 然后烧写

6:输入 b 命令来启动内核

10.2.1: 内核功能,结构,结合 Makefile,Kconfig 进行分析 (第 10 课第 2 节 内核启动流程分析之配置)

I:配置过程:

备注:vim 里面的用法:

1:搜索 vim 里面的某个字符 /abc 通过 n 和 N 来向上或向下选择

2:跳转 vim 某一行 :23

配置的结果是产生了一个.config 文件,那么.config 文件里面有什么东西呢?

打开.config 可以看到很多配置项:

例如:

```
CONFIG_NO_IOPORT=y
CONFIG_INIT_ENV_ARG_LIMIT=32
CONFIG_BLK_DEV_NBD=m (这就是作为一个模块编译)
```

以 DM9000 为例分析:

```
CONFIG_DM9000=y
```

这个配置项=y:表示对 DM9000 的这个网卡的支持将会被编译到内核,若不设置,则内核不支持网卡

若=m:表示最终会被编译成一个模块,这个模块以后可以动态加载

A:DM9000 这个配置项谁来使用,在内核目录搜索 grep "CONFIG_DM9000"-nr

发现有以下几类文件有这个配置项:

1:C 源码文件:使用宏来定义,那么这个宏只能在 c 文件或者.h 文件里面定义,那么只能来自于

include/linux/autoconf.h

2:子目录的 Makefile:例如: drivers/net/Makefile

3: include/config/auto.conf

4: include/linux/autoconf.h:通过这个文件的名字可以猜测这个文件是自动生成的,而这个文件的内容来源于.config 文件

B:我们来分析下 autoconf.h 这个文件的内容是什么,打开之后搜索下 DM9000,发现是定义成了一个宏#define CONFIG_DM9000 1

我们发现里面的文件所有的配置项都是定义为 1,就是说不管在.config 文件里面该配置项是配制成 y 或者 m,在 autoconf.h 都是被定义为 1,这样我们 c 语言源码就能直接使用这些宏了

C:那么=y 和=m 的差别在哪里体现呢,既然在 c 源码和.h 文件体现不出来,那应该是在 makefile 里面体现,我们分析下子目录的 makefile

先了解下对于内核 makefile 它的子目录 makefile 格式(注意是子目录的 makefile):

obj-y+=xxx.o 这句表示 xxx.c 文件最后会被编译进内核

obj-m+=yyy.o 这句表示 yyy.c 文件最后会被编译成一个可加载的模块 yyy.ko

在 drivers/net/Makefile 搜索 dm9000 发现为: # obj-\$(CONFIG_DM9000) += dm9000.o

这就说明是根据 CONFIG_DM9000 这个变量的值(y 或 m)来判断是编译进内核还是编译成模块(这里就体现出了 y 和 m 的差别,如果这个变量值没有被设置的时候,

即 obj-+= dm9000.o 那么这条语句将不会被处理),那么子目录的变量 CONFIG_DM9000 又是由谁来定义的呢,是由 include/config/auto.conf 来定义的,auto.conf 是怎么来的呢,也是由.config 生成的,显然这个文件要被别人包含,是有顶层的 makefile 包含的

总结:

- 1:配置内核的时候生成了.config 文件
- 2:然后我们去编译,make uImage 的时候做了哪些事?
- 2.1:.config 用来自动生成了一个 autoconf.h(这个头文件是被源代码使用的)
- 2.2:.config 用来自动生成一个 auto.conf(这个文件是被顶层 makefile 包含的,由子目录的 makefile 来使用:obj-\$(CONFIG_DM9000)+=dm9000.o 这里的\$就是对变量的引用)

10.2.2: 内核功能,结构,结合 Makefile,Kconfig 进行分析 (第 10 课第 3 节 内核启动流程分析之 makefile)

分析过程:

内核的 makefile 的分类(应用手册里面介绍):

- 1.1:顶层 makefile
- 1.2:.config
- 1.3arch/\$(ARCH)/makefile:架构相关的 makfile
- 1.4:scripts/makefile.*
- 1.5:kbuild makefile:子目录的 makefile

内核 makfile 的详细介绍在

D:\Linux_Share_Place\linux-2.6.22.6_PATCHED\Documentation\kbuild\makefile.txt

1:先分析最简单的 makefile:子目录的 makefile (即 kbuild makefile):每个子目录都有一个 makefile

子目录在 D:\Linux_Share_Place\linux-2.6.22.6_PATCHED\drivers 这里

子目录的 makefile 的语法规则在上一节笔记中有即:

obj-y+=xxx.o 这句表示 xxx.c 文件最后会被编译进内核
obj-m+=yyy.o 这句表示 yyy.c 文件最后会被编译成一个可加载的模块 yyy.ko

Q1:假如有两个文件 a.c b.c 两个文件:如果要被编译进内核

A:Obj-y+=a.o b.o

Q2:假如有两个文件 a.c b.c 两个文件:如果要组成一个模块

A:obj-m+=ab.o(这个 ab 的名字可以随便定义,也可以是 xxx.o)

ab-objs :=a.o b.o

这样做的结果是什么呢:

a.c 会被编译成 a.o,b.c 会被编译成 b.o

a.o 和 b.o 会被组成 ab.ko 模块

上述的方法在 Documentation\kbuild\makefile.txt 有说明(L:189)

2:分析顶层目录的 makefile:

- 1:分析 makefile:目的:
- 1.1:确定第一文件:通过第一个文件可以分析它的启动过程
- 1.2:链接脚本:表示的是内核是放在哪里,里面的内容是如何排布的

我们编译的时候是执行 make 或者 make uImage:这条语句如何实现呢?

我们在顶层 makefile 搜索 uImage 发现搜索不到,而是在架构相关的 makefile 里面: arch/\$(ARCH)/makefile

说明架构相关的 makefile 会被包含进顶层 makefile,里面有:

include \$(srctree)/arch/\$(ARCH)/Makefile

搜索 ARCH: ARCH ?= arm 这是在打上补丁后实现的

在架构相关的 makefile 搜索目标 uImage:

zImage Image xipImage bootpImage uImage: vmlinux
\$(Q)\$(MAKE) \$(build)=\$(boot) MACHINE=\$(MACHINE) \$(boot)/\$@

而 uImage 依赖于 vmlinux

我们之前说过 uImage 就是头部+真正的内核,而真正的内核就是 vmlinux

而 vmlinux 的依赖在顶层的 makefile 里面:

vmlinux: \$(vmlinux-lds) \$(vmlinux-init) \$(vmlinux-main) \$(kallsyms.o) FORCE
链接脚本 初始化代码 主要代码

继续分析 vmlinux 的依赖:

```
a:vmlinux-init := $(head-y) $(init-y)
    head-y 发现在顶层目录 makefile 未定义,应该是在架构相关的 makefile 定义:
    head-y      := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
    MMUEXT 这个变量未定义
    init-y 是在顶层 makefile 定义的
    init-y      := init/
    init-y      := $(patsubst %, %/built-in.o, $(init-y)) 这里用到了 makefile 的函数
    也就是 init-y= init/ built-in.o
b:vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
    core-y      := usr/
    core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
    core-y      := $(patsubst %, %/built-in.o, $(core-y))
    即最终为 core-y := usr/ built-in.o kernel/ built-in.o mm/ built-in.o fs/ built-in.o ipc/ built-in.o security/ built-in.o crypto/ built-
    in.o block/ built-in.o
    剩下三个的依赖结果为:
    libs-y      :=lib/lib.a lib/built-in.o
    drivers-y   := drivers/ built-in.o sound/ built-in.o
    net-y       := net/ built-in.o
c:vmlinux-all := $(vmlinux-init) $(vmlinux-main)
d:vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

我们想知道上述的 vmlinux 是如何生成的,一个办法是分析 makefile(这个里面的脚本太多,没有继续分析)第二种方法是直接编译

我们用方法二来分析 vmlinux 是如何通过上面的依赖来生成的

步骤:

- 1:若之前编译过内核,则把 vmlinux 删除 `rm vmlinux`
- 2:执行编译内核: `make uImage V=1` (注:V=1 的意思是会将命令更加详细的列出来)
- 3:我们只要关心最后一条命令即可:

```
arm-linux-ld -EL -p --no-undefined -X -o vmlinux (链接,输出 vmlinux)
-T arch/arm/kernel/vmlinux.lds(这个是链接脚本:决定了这些文件在链接的时候如何排布) arch/arm/kernel/head.o arch/arm/kernel/init_task.o init/built-
in.o (这些就是编译时需要的原材料和我们上面的刚好对应)
--start-group usr/built-in.o arch/arm/kernel/built-in.o arch/arm/mm/built-in.o arch/arm/common/built-in.o arch/arm/mach-s3c2410/built-in.o
arch/arm/mach-s3c2400/built-in.o arch/arm/mach-s3c2412/built-in.o arch/arm/mach-s3c2440/built-in.o arch/arm/mach-s3c2442/built-in.o
arch/arm/mach-s3c2443/built-in.o arch/arm/nwpe/built-in.o arch/arm/plat-s3c24xx/built-in.o kernel/built-in.o mm/built-in.o fs/built-
in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o arch/arm/lib/lib.a lib/lib.a arch/arm/lib/built-in.o
lib/built-in.o drivers/built-in.o sound/built-in.o net/built-in.o --end-group tmp_kallsyms2.o
```

结论:

a:内核的编译时的第一文件为 arch/arm/kernel/head.o(这个 head.o 应该是汇编生成的,应该里面有个 head.s)

b:链接脚本: arch/arm/kernel/vmlinux.lds

接下来分析链接脚本:在 kernel 目录下没有找到 vmlinux.lds 文件,但有 vmlinux.lds.S 文件

可以知道这个链接脚本是我们根据这个.s 文件在编译过程中会生成

3:分析.config:

我们之前分析,.config 文件会自动生成一个头文件 autoconf.h(这是给源码用的)

还有会生成一个 auto.conf(这个文件之前说被顶层目录 makefile 包含的)

可以在顶层的 makefile 搜索下: `-include include/config/auto.conf`

10.3: 内核启动的过程分析 (第 10 课第 4 节 内核启动流程分析之内核启动)

I: 猜测在分析内核启动过程之前我们先猜测内核启动要做什么事情:

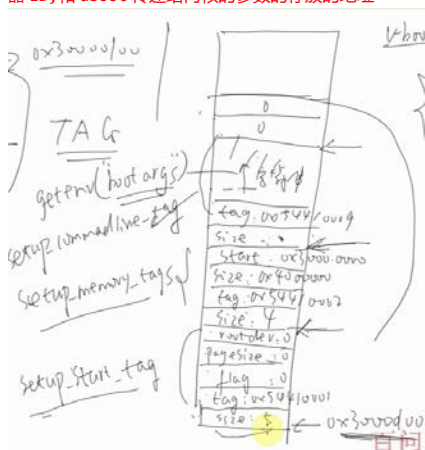
由于我们之前在 uboot 的讲解的时候,我们的 uboot 给内核传递了很多参数,然后才启动内核的:

Uboot 传入的参数包括参数包括了机器 ID,和 TAG 标记的参数(这里面包括了 tag 标记的起始,tag 标记的内存信息地址,tag 标记的 bootargs,tag 标记的 end 信息)

这些参数 Uboot 都是通过

```
theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
```

来传递给内核的,其中 thekernel 函数的地址是内核的入口地址,所以运行了 thekernel 函数就相当于跳转到了内核的入口地址,也就是启动了内核,这个函数的传入的参数分别是 0,机
器 ID,和 uboot 传递给内核的参数的存放的地址



那么内核肯定要处理这些参数,其次,内核的最终目的是为了运行应用程序,而应用程序存放与根文件系统中,所以肯定要挂接根文件系统,

I 总结:

所以猜测内核需要做如下的事情:

- 1:处理 uboot 传入的参数:内核传进了机器 ID,启动参数,就是环境变量 bootargs 里面的参数:
- 2:内核的最终目的是为了运行应用程序,那么应用程序在跟文件系统中,所以要挂接根文件系统

II: 分析内核的启动过程看是否做了我们猜测的事情

分析步骤:

1:

从编译的第一个文件开始分析:即 arch/arm/kernel/head.o (也就是 head.s)

我们利用 sourceinsight 查找 head.s 的时候发现内核文件有两个 head.s

一个是 head.S (d:\linux_share_place\linux-2.6.22.6_patched\arch\arm\boot\compressed)

另一个是: head.S (d:\linux_share_place\linux-2.6.22.6_patched\arch\arm\kernel)

显然我们要的是第二个,那么第一个是什么呢?

是由于一般我们的内核比较大,所以需要压缩以下,一般压缩后的文件由两部分构成

自解压代码+压缩的内核代码,我们执行压缩的文件时先执行自解压代码,自解压代码会将后面的压缩代码解压,再执行解压后的代码

第一个 head.s 就是自解压的代码

接下来就分析 head.s 来分析内核的处理流程

从头往下看(这里只截取了重要的语句)

step1:判断是否支持 uboot 传进来的处理器

```
ENTRY(stext)
msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ ensure svc mode
                                                    @ and irq disabled
mrc p15, 0, r9, c0, c0 @ get processor id
bl __lookup_processor_type @ r5=procinfo r9=cupid
movs r10, r5 @ invalid processor (r5=0)?
beq __error_p @ yes, error 'p'
bl __lookup_machine_type @ r5=machineinfo
movs r0, r5 @ invalid machine (r5=0)?
```

分析: mrc p15, 0, r9, c0, c0 @ get processor id //这些寄存器里面保存了我们的开发板的 CPU 的 ID
// (这里是 CPU 的 ID 或者称为处理器的 ID,并不是机器 ID)
bl __lookup_processor_type @ r5=procinfo r9=cupid //我们一个内核能够处理哪些处理器,这个在编译内核
//的时候就决定了,这里就是去将读到的寄存器和
//我的内核支持的 CPU 进行查找,看看能不能支持,
//若不能则跳到下面的 beq __error_p
movs r10, r5 @ invalid processor (r5=0)?
beq __error_p @ yes, error 'p'

step2: 判断是否支持 uboot 传进来的这个单板(这个单板的机器 ID 是 uboot 启动内核时通过 thekernel 传入的机器 ID)

这里要区分个概念,判断是否支持某种单板是通过比较机器 ID 来实现的

```
bl __lookup_machine_type @ r5=machineinfo //这里就是来查找机器 ID,这个机器 ID 就是 uboot 里面 theKernel 函数
//的第二个参数来和内核支持的机器 ID 比较
movs r8, r5 @ invalid machine (r5=0)?
beq __error_a @ yes, error 'a'
bl __create_page_tables
```

在 uboot 中搜索 thekernel 的第二个参数 bi_arch_number ,在 100ask24x0.c 中找到

```
/* arch number of SMDK2410- Board */
gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
#define MACH_TYPE_TOTO 361
#define MACH_TYPE_S3C2440 362
#define MACH_TYPE_KS8695P 363
```

也就是说我们这个单板被定义为了是 362 这个数字,我们就要看看内核里面是否有这个机器 ID

内核自己能支持哪些单板,他自己肯定知道的,我们来看下内核是如何实现 判断是否支持 uboot 传进来的单板的 ID

也就是分析 __lookup_machine_type

```
__lookup_machine_type:
    type __lookup_machine_type, %function
    adr r3, 3b @ r3 = address of 3b, real address, phy address
    ldmia r3, {r4, r5, r6} @ r4 = " virtual address of 3b, r5 = __arch_info_begin, r6 = __arch_info_end
    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, #MACHINE_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
```

上面大概就是实现了将 uboot 传入的单板的机器 ID 和内核中支持的机器 ID 进行逐一比较,看是否匹配,

而内核中支持的所有单板信息(这些单板信息中就包含了机器 ID),被保存在了 __arch_info_begin 和 __arch_info_end 这两个地址之间

Q: 内核是如何将支持的单板信息存放在 __arch_info_begin 和 __arch_info_end 这两个地址之间?

A: __arch_info_begin 和 __arch_info_end 是在链接脚本 vmlinux.lds 中定义的

```
.init : { /* Init code and data */
    *(.init.text)
    __inittext = .;
    __proc_info_begin = .;
    *(.proc.info.init)
    __proc_info_end = .;
    __arch_info_begin = .;
    *(.arch.info.init)
    __arch_info_end = .;
    __tagtable_begin = .;
    *(.taglist.init)
    __tagtable_end = .;
}
```

可以看出来 __arch_info_begin 和 __arch_info_end 之间段称为 .arch.info.init

联想我们之前的 uboot 的命令也是将所有的 uboot 的命令定义成一个结构体,这个结构体里面要有命令的名字,和命令的执行函数,然后让这些结构体都增加一个段属性,然后保存在同一个段中,当 uboot 的命令行界面输入命令时,就去这个段去依次查找命令名相同的结构体,然后执行这个结构体中的执行函数,这样就完成了命令的实现过程

那么对于这里也是一样的,肯定是内核支持的所有机器 ID 的开发板都分别是一个结构体,然后放在了这个 .arch.info.init 段中,而且肯定使用宏来实现这个结构体的定义的,我们直接在内核中搜 .arch.info.init 看看哪些使用了这个段属性,

```
book@book-desktop:~/work/system/linux-2.6.22.6$ grep ".arch.info.init" * -nr
Binary file arch/arm/mach-s3c2443/mach-smdk2443.o matches
Binary file arch/arm/mach-s3c2443/built-in.o matches
Binary file arch/arm/mach-s3c2410/mach-smdk2410.o matches
Binary file arch/arm/mach-s3c2410/mach-gt2410.o matches
Binary file arch/arm/mach-s3c2410/built-in.o matches
Binary file arch/arm/mach-s3c2440/built-in.o matches
Binary file arch/arm/mach-s3c2440/mach-smdk2440.o matches
arch/arm/kernel/vmlinux.lds:306: *(.arch.info.init)
arch/arm/kernel/vmlinux.lds:39: *(.arch.info.init)
Binary file arch/arm/mach-s3c2412/mach-smdk2413.o matches
Binary file arch/arm/mach-s3c2412/built-in.o matches
Binary file arch/arm/mach-s3c2412/mach-vstms.o matches
include/asm/mach/arch.h:53: __attribute__((__section__(".arch.info.init"))) = { \
include/asm-arm/mach/arch.h:53: __attribute__((__section__(".arch.info.init"))) = { \
book@book-desktop:~/work/system/linux-2.6.22.6$
```

从上面截图可以看出:

1:可以看出我们的内核支持上面黑色标记的这些单板:smdk2443,smdk2410,qt2410 等等,

2:在 asm-arm/mach/arch.h 中找到定义:

```
#define MACHINE_START( type, name)
static const struct machine_desc __mach_desc_##type \
__used
__attribute__((section(".arch.info.init"))) = { \
    .nr = MACH_TYPE_##type, \
    .name = name, \
}

#define MACHINE_END
};
```

注意,这里的 MACHINE_END 被定义为 }; (别看错了)

那么我们 uboot 传入的机器 ID 肯定也是用这个 MACHINE_START 来定义里面的结构体的,所以搜索 MACHINE_START

```
Mach-nexcoder.c (arch/arm/mach-s3c2440):MACHINE_START(NEXCODER_2440, "NexVision - Nexcoder 24
Mach-osiris.c (arch/arm/mach-s3c2440):MACHINE_START(OSIRIS, "Simtec-OSIRIS")
Mach-otom.c (arch/arm/mach-s3c2410):MACHINE_START(OTOM, "Nex Vision - Otom 1.1")
Mach-qt2410.c (arch/arm/mach-s3c2410):MACHINE_START(QT2410, "QT2410")
Mach-rx3715.c (arch/arm/mach-s3c2440):MACHINE_START(RX3715, "IPAQ-RX3715")
Mach-smdk2410.c (arch/arm/mach-s3c2410):MACHINE_START(SMDK2410, "SMDK2410") /* @TODO: request
Mach-smdk2440.c (arch/arm/mach-s3c2440):MACHINE_START(S3C2440, "SMDK2440")
Mach-vr1000.c (arch/arm/mach-s3c2410):MACHINE_START(VR1000, "Thorcom-VR1000")
```

我们的 S3C2440 使用该宏来定义的结构体的语句如下:

```
>MACHINE_START(S3C2440, "SMDK2440")
/* Maintainer: Ben Dooks <ben@fluff.org> */
.phys_io = S3C2410_PA_UART,
.io_pg_offst = (((u32)S3C24XX_VA_UART) >> 10) & 0xffff,
.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
```

将上面的宏展开结果为:

```
static const struct machine_desc __mach_desc_S3C2440 \
__used
__attribute__((section(".arch.info.init"))) = { \
    .nr = MACH_TYPE_S3C2440, \
    .name = "SMDK2440", \
    /* Maintainer: Ben Dooks <ben@fluff.org> */
    .phys_io = S3C2410_PA_UART,
    .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 10) & 0xffff,
    .boot_params = S3C2410_SDRAM_PA + 0x100,

    .init_irq = s3c24xx_init_irq,
    .map_io = smdk2440_map_io,
    .init_machine = smdk2440_machine_init,
    .timer = s3c24xx_timer,
};
```

也就是定义了一个 machine_desc 类型的结构体 __mach_desc_S3C2440,他有一个段属性,

这个结构体中的.nr 就是机器 ID, 显然,内核支持多少种单板,就有多少个这种结构体,这个结构体里面不仅仅包含机器 ID,还有其他的参数, .bootparams 是指 uboot 传进的 TAG 参数存放的位置(这个结构体的详细描述具体的看 machine_desc 的定义)

也就是说我们的 s3c2440 的单板信息就在这里被保存在了 .arch.info.init 段中

我们 2440 的这个结构体刚才说是在 arch/arm/mach-s3c2440/mach-smdk2440 中定义的,也就是说,如果这个文件在编译内核的时候被包含了话,那么这个内核就支持这种单板,

现在我们继续分析 head.s

```
.type __lookup_machine_type, %function
__lookup_machine_type:
    adr r3, 3b @ r3 = address of 3b, real address, phy address
    ldmia r3, {r4, r5, r6} @ r4 = ". virtual address of 3b, r5 = arch_info_begin, r6 = arch_info_end
    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
```

下面就是逐一比较机器 ID 了:

```
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
```

若比较成功之后(也就是内核支持我的 uboot 传进来的单板)就会继续回到

```
ENTRY(stext)
    msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ 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 __create_page_tables
```

Step3:创建页表:

```
movs r8, r5 @ invalid machine (r5=0)?
beq __error_a @ yes, error 'a'
bl __create_page_tables
```


从我们的链接脚本可以看到,我们的内核是从 0xc0000000+0x00008000 开始的,这个地址并不对应真实的存在的内核的,我们的内存是从 0x30000000 开始的,所以要建立一个页表,启动 MMU,来进行地址的映射

Step4:启动 MMU:

从上面的注释可以看出:当 mmu 使能之后会跳转到__switch_data 里去,至于如何跳转的,可以看看__enable_mmu 的实现,这里就不分析了,我们直接进入__switch_data,这个是在 head-common.S 中定义的

然后进入__mmap_switched,要注意的是这个__mmap 还是在 head-common.S 中,跳转到 start_kernel:如下:

Step5:跳转执行 start_kernel,这个函数就是我们内核的第一个C函数

所以我们还要继续分析 start kernel,看他做了哪些事情:

```
start kernel(void)
```

```
printk(linux_banner); //输出内核版本信息,linux_banner 定义如下
```

对应的串口显示为：

接下来就是处理 uboot 传入的启动参数:

```
setup_arch(&command_line); //用来处理 uboot 传进来的启动参数的,只是保存在数组中
```

```
struct machine_desc *mdesc; //这个结构体就是我们之前分析单板信息的那个,那个结构体里面包含了机器 ID 和
```

//uboot 传入给内核参数保存的地址,

if (mdesc->boot_params)//这个就是 uboot 保存传入参数的地址

```
..... //后续就是将 uboot 的那些 tag 分别取出来
```

```
init_mm.start_code = (unsigned long) &text;
init_mm.end_code   = (unsigned long) &etext;
init_mm.end_data    = (unsigned long) &edata;
init_mm.brk        = (unsigned long) &end,
```

```
setup_command_line(command_line); //用来处理 uboot 传进来的启动参数的,只是将启动参数保存到数组中:
rest_init();
```

```
kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND); //创建一个内核线程, 暂且认为这里就是调用
//kernel_init 函数
```

```
kernel_init
prepare_namespace();
mount_root(); //这里就挂接了根文件系统
init_post(); //这里就是开始执行应用程序
```

总结: 通过分析 start_kernel(void) 函数, 他里面做了

- 1: 解析 uboot 传入的启动参数 (通过 setup_arch(&command_line) 和 setup_command_line(command_line) 这两个函数实现)
- 2: 挂接根文件系统 (mount_root(); 函数来实现)
- 3: 启动第一个应用程序 (init_post();)

我们之前只是说 setup_arch(&command_line) 和 setup_command_line(command_line) 这两个函数只是将启动参数给保存起来了, 并没有什么其他动作:

启动参数就是

```
OpenJTAG> print
bootargs=bootmtd root=/dev/mtdblock3 init=/linuxrc console=ttysAO0
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
```

我们看到了一个 root=/dev/mtdblock3 这个启动参数, 我们就以他为切入点来看看内核是怎么处理启动参数的

我们知道这个启动参数的意思是我们的根文件系统放在第四个分区上面, 也就是 mtdblock3

而我么内核在是通过 mount_root() 这个函数来挂接根文件系统的, 在挂接根文件系统之前, 他肯定要确定要挂接哪个根文件系统

所以分析 mount_root 前的一个函数 prepare_namespace()

分析

```
prepare_namespace();
if (saved_root_name[0]) {
    root_device_name = saved_root_name;
    if (!strcmp(root_device_name, "mtd", 3))
//这里么发现我们的根文件系统分区的名字貌似是被保存在了 saved_root_name[0] 这个全局数组中, 搜索这个 saved_root_name 在哪里被赋值, 在 Do_mounts.c 中发现如下的赋值:
static int __init root_dev_setup(char *line)
{
    strcpy(saved_root_name, line, sizeof(saved_root_name));
    return 1;
}
```

```
__setup("root=", root_dev_setup);
```

我们在解析我们的命令行时候发现 root=xxx 的时候, 就会根据这个 root=xxx 找到 root_dev_setup 这个函数, 然后调用这个函数

root_dev_setup(char *line) 这个函数的作用就是把我们找到的 xxx 保存到 saved_root_name

很明显这个 __setup 这个宏肯定是定义某个结构体或者变量, 最终我在 init.h 找到了:

```
/*
#define __setup_param(str, unique_id, fn, early) \
static char __setup_str_##unique_id[] __initdata = str; \
static struct obs_kernel_param __setup_##unique_id \
__attribute__((__section__(".init.setup"))) \
__attribute__((aligned((sizeof(long)))) \
= ( __setup_str_##unique_id, fn, early )

#define __setup_null_param(str, unique_id) \
__setup_param(str, unique_id, NULL, 0)

#define __setup(str, fn) \
__setup_param(str, fn, fn, 0)
*/
```

这个结构体的段属性为 .init.setup 段, 显然这个段在链接脚本里面定义的

```
. = ALIGN(16);
__setup_start = .;
*(.init.setup)
__setup_end = .;
__early_begin = .;
*(.early_param.init)
__early_end = .;
```

也就是说我们的所有的启动参数都会被放入 .init.setup 这个段中, 肯定是最后将 uboot 传入给内核的命令依次去在 __setup_start 和 __setup_end 这个范围去比较, 如果比较成功, 就调用相应的函数, 那么我们只要搜索下, 看看 __setup_start 看看有谁在使用, 就知道了那些启动参数是怎样被调用了 (注意, 这里搜索的是地址的段的范围, 而不是段的属性, 好好理解下)

搜索 __setup_start

```
--- __setup_start matches (0 in 3 files) ---
Do_mounts.c (init): root_mount_data = str; __setup_start
Main.c (init): extern struct obs_kernel_param __setup_start[], __setup_end[];
Main.c (init): p = __setup_start;
Main.c (init): for (p = __setup_start; p < __setup_end; p++) {
vmlinux.lds.S (arch/arm/kernel): __setup_start = .;
```

最后发现只有两个函数在用

```
static int __init do_early_param(char *param, char *val)
{
    struct obs_kernel_param *p;
    for (p = __setup_start; p < __setup_end; p++) {
        if (p->early && strcmp(param, p->str) == 0) {
            if (p->setup_func(val) != 0)
                printk(KERN_WARNING
                    "Malformed early option '%s'\n", param);
        }
    }
}

static int __init obsolete_checksetup(char *line)
{
    struct obs_kernel_param *p;
    int had_early_param = 0;
    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (!strcmp(line, p->str, n)) {
            if (p->early) {
                /* Already done in parse_early_param?
                */
            }
        }
    } while (p++);
}
```

(这两个函数应该是根据 early 参数的不同而选择不同执行), early 参数如下:

```
#define __setup_param(str, unique_id, fn, early) \
static char __setup_str_##unique_id[] __initdata = str; \
```

A:

首先分析下 do_early_param ,我们看看是谁调用的他

```
void __init parse_early_param(void)
    __init do_early_param
```

而__init parse_early_param(void)则是在 void __init start_kernel(void)中被调用

B:

另外的

static int __init obsolete_checksetup(char *line)这个函数是在 static int __init unknown_bootoption(char *param, char *val)调用

而 unknown_bootoption 在 void __init start_kernel(void)中被调用

总体来说如下:

```
start_kernel
1  setup_arch // 解析u-boot传入的启动参数
2  setup_command_line // 解析u-boot传入的启动参数
3  parse_early_param
4  do_early_param
5  从 __setup_start到 __setup_end, 调用early函数
6  unknown_bootoption
7  obsolete_checksetup
8  从 __setup_start到 __setup_end, 调用非early函数
9  rest_init
10 kernel_init
11 prepare_namespace
12 mount_root // 挂载根文件系统
13 init_post
14 // 执行应用程序
```

现在我们知道,我们内核要挂载哪个分区是由命令行参数来决定的,命令行参数一开始是被保存在某个字符串中的,然后调用

do_early_param 和 obsolete_checksetup 来——分析他们.

III:如何给修改内核分区:

另外说一个问题,

我们现在是 root=/dev/mtdblock3 这个分区,我们之前说过,flash 里面没有分区表,那么这些是如何体现的呢,我们是在代码里面写死的,在书上有讲的,

另外的方法:

我们启动内核,内核会把分区信息打印出来:

```
new 0x00000000-0x00000000:
Creating 4 MTD partitions on "NAND 256MiB 3.3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
s3c2410-nhci s3c2410-nhci: S3C2410 OHCI
s3c2410-nhci s3c2410-nhci: new USB bus registered, assigned bus number 1
```

我们随便搜索个"bootloader"

```
static struct mtd_partition smdk_default_nand_part[] = {
[0] = {
    .name = "bootloader",
    .size = 0x00040000,
    .offset = 0,
},
[1] = {
    .name = "params",
    .offset = MTDPART_OFST_APPEND,
    .size = 0x00020000,
},
[2] = {
    .name = "kernel",
    .offset = MTDPART_OFST_APPEND,
    .size = 0x00200000,
},
[3] = {
    .name = "root",
    .offset = MTDPART_OFST_APPEND,
    .size = MTDPART_SIZ_FULL,
},
};
```

人家搜索"bootloader"的时候双引号加了\,(猜测:应该是要转义字符吧)

找到如下:

```
static struct mtd_partition smdk_default_nand_part[] = {
[0] = {
    .name = "bootloader",
    .size = 0x00040000,
    .offset = 0,
},
[1] = {
    .name = "params",
    .offset = MTDPART_OFST_APPEND,
    .size = 0x00020000,
},
[2] = {
    .name = "kernel",
    .offset = MTDPART_OFST_APPEND,
    .size = 0x00200000,
},
[3] = {
    .name = "root",
    .offset = MTDPART_OFST_APPEND,
    .size = MTDPART_SIZ_FULL,
},
};
```

上面的 MTDPART_OFST_APPEND 是紧接上一个分区的意思,我们的分区就是在这里写死的

Q1:核是如何将支持的单板信息存放在__arch_info_begin 和 __arch_info_end 这两个地址之间?

A1:搜索本节 Q1 回复

第 11 课 构建根文件系统

1. 内核怎么启动第一个应用程序
2. Init 进程分析
 - 2.1 读取配置文件
 - 2.2 执行
3. 配置,编译 busybox
4. 构建根文件系统
 - 4.1 内容:可执行程序,库,配置文件,设备节点
 - 4.2 制作映像文件
 - 4.3 烧写

11.1: 内核怎么启动第一应用程序 (第 11 课第 1 节 构建根文件系统之启动第 1 个程序)

- 1: Uboot 的目的: 启动内核
- 2: 内核的目的: 启动应用程序
应用程序位于的位置: 跟文件系统
- 3: 本节目的: 构建根文件系统

应用程序有很多, 比如 A1, A2, A3

内核启动之后运行哪个应用程序呢?

分析步骤:

上章分析到: mount_root : 挂接好根文件系统

init_post: 启动应用程序

- 1: 在内核文件中搜索 init_post 发现在 Main.c 中

分析 init_post 函数:

```
static int noinline init_post(void)
{
    free_initmem();
    unlock_kernel();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();
    //step 1: 打开一个设备/dev/console, 这个设备对应的文件叫做终端, 我们这里对应的设备是串口 0, 对于其他的也许是液晶, 或者键盘等等
    if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
        printk(KERN_WARNING "Warning: unable to open an initial console.\n");
    //step 2: 将之前的打开的文件复制两个, 构成 3 个文件, 这三个文件夹分别构成标准输入, 输出, 错误,
    /*我们写程序的时候经常会用到
    printf: 这个成为标准输出
    scanf: 这个成为标准输入
    err: 这个成为标准错误
    所谓的标准输入, 标准输出, 标准错误是指三个文件
    */
    (void) sys_dup(0); //这个是复制的意思
    (void) sys_dup(0);

    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n",
               ramdisk_execute_command);
    }
    //step3: 通过 run_init_process 来启动程序, 那么启动哪个应用程序呢?
    /*1: 在命令行中指定 (是通过 uboot 命令行传给内核的): 在命令行里面的传入的 init= xxxxx
    2: 若上述未定义则依次执行/sbin/init, 如果前面的不成功就执行后续的 (应为应用程序一般是个死循环)
    所以一般来说: 运行的第一个程序一般是命令行参数指定的, 要么是/sbin/init
    */
    * We try each of these until one succeeds.
    *
    * The Bourne shell can be used instead of init if we are
    * trying to recover a really broken machine.
    */
    /*
    if (execute_command)这句分析:
    搜索 execute_command 在 static int __init init_setup(char *str)函数中找到 execute_command = str;这句
    execute_command = str;
    __setup("init=", init_setup);
    分析这两句: __setup 是我们在内核命令中传入的命令行参数, 也就是说如果我们设置了 init=什么东西,
    那么 execute_command 就会等于那个东西
    我们通过串口打印板子信息在命令行里面输入 print 可以发现这有句
    bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
    这里的 execute_command 就等于 linuxrc
    */
    if (execute_command) { //如果定义了某个 execute_command
        run_init_process(execute_command); //就启动这个应用程序
        printk(KERN_WARNING "Failed to execute %s. Attempting "
               "defaults...\n", execute_command);
    }
    //需要注意的是: 如果我们上面的 execute_command 被定义了, 就会执行那个应用程序, 如果那个应用程序是死循环的, 就不会进入这里了
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel.");
}
```

验证一 (根文件系统没有内容会怎样):

板子内部已经烧写好了内核

- 1: 首先删除掉之前的文件系统: 输入命令 nand erase root
- 2: 然后启动命令 boot

```

Hit any key to stop autoboot: 0

##### 100ask Bootloader for openJTAG #####
[n] Download u-boot to Nand Flash
[k] Download Linux kernel uImage
[j] Download root_jffs2 image
[y] Download root_yaffs image
[d] Download to SDRAM & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the Nand Flash
[s] Set the boot parameters
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection: q
openJTAG> print
bootargs=noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
ipaddr=192.168.7.17
serverip=192.168.7.11
netmask=255.255.255.0
mtdids=nand0=nandflash0
mtdparts=mtdparts=nandflash0:256k@0(bootloader),128k(params),2m(kernel),-(root)
bootdelay=10
stdin=serial
stdout=serial
stderr=serial
partition=nand0,0
mtddevnum=0
mtddevname=bootloader

Environment size: 444/131068 bytes
OpenJTAG> nand erase root

NAND erase: device 0 offset 0x260000, size 0xfda0000
Skipping bad block at 0x08f60000
Skipping bad block at 0x0cd20000
Skipping bad block at 0x0cf00000
Skipping bad block at 0x0ea80000
Erasing at 0xffe0000 -- 100% complete.
OK
OpenJTAG> boot

```

截取最后几行结果如图:

```

yaffs: passed flags
yaffs: Attempting MTD mount on 31.3, "mtdblock3"
yaffs: auto selecting yaffs2
block 1129 is bad
block 1623 is bad
block 1638 is bad
block 1858 is bad
VFS: Mounted root (yaffs filesystem).
Freeing init memory: 140K
Warning: unable to open an initial console.
Failed to execute /linuxrc. Attempting defaults...
Kernel panic - not syncing: No init found. Try passing init= option to kernel.

```

启动 boot 后发现:

VFS: Mounted root (yaffs filesystem). 这句说明已经挂接上跟文件系统了

对于我们的 flash 来说把它擦除之后相当于把它格式化, 可以把它当作任何一种文件系统比如说 yaffs, jffs2, 这里默认是 yaffs 文件系统。虽然这里挂载成功了, 但里面是没有内容的, 就不能启动应用程序。

根据上面分析的启动第一个应该程序的函数 init_post

Warning: unable to open an initial console.

后续都是该函数的打印信息

验证二: 现在我们烧录下根文件系统对比下结果有何不同

- 1: 先给板子关机, 我们用 usb 来烧录, 因为我们的板子没有 usb 的时能开关引脚, 要么拔插下 usb 线, 要么重启开发板
- 2: 重新启动单板, 进入命令行界面
- 3: 输入 y 命令进入下载 yaff 镜像命令
- 4: 找到对应的文件 F:\linux\资料盘\bin\ fs_mini.yaffs2
- 5: 打开 dnw 工具->USB port->transmit 来烧录
- 6: 烧写成功后在命令行中输入 b 命令启动

截图如下:

```

yaffs: dev is 32005839 name is mtdblock3
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.3, "mtdblock3"
yaffs: auto selecting yaffs2
block 1129 is bad
block 1623 is bad
block 1638 is bad
block 1858 is bad
VFS: Mounted root (yaffs filesystem).
Freeing init memory: 140K
init started: BusyBox v1.7.0 (2008-01-22 10:04:09 EST)
starting pid 764, tty '': '/etc/init.d/rcS'

```

```

Please press Enter to activate this console.
starting pid 767, tty '/dev/ttySAC0': '/bin/sh'

```

```

#
# ps
# PID  Uid          VSZ  Stat  Command
# 1 0          3092  S    init
# 2 0              SW<  [kthreadd]
# 3 0              SWN  [ksoftirqd/0]
# 4 0              SW<  [watchdog/0]
# 5 0              SW<  [events/0]
# 6 0              SW<  [khelper]
# 55 0             SW<  [kblockd/0]
# 56 0             SW<  [ksuspend_usbd]
# 59 0             SW<  [khubd]
# 61 0             SW<  [kseriod]
# 73 0             SW   [pdflush]
# 74 0             SW   [pdflush]
# 75 0             SW<  [kswapd0]
# 76 0             SW<  [aio/0]
# 710 0            SW<  [mtdblockd]
# 745 0            SW<  [kmmcd]
# 767 0            S    -sh
# 768 0            3096  R    ps
#
# ls
# bin      lib      mnt      sbin     usr
dev        linuxrc  proc     sys
etc        lost+found  root    tmp
# █

```

7:我们执行 ps 命令可以看到当前的执行的应用程序,可以看到第一个为 init

Q1:uboot 如何将启动参数传递给内核?

Q2:内核如何接收 uboot 传递的参数并如何解析的

11.2: init 进程分析 (第 11 课第 2 节 构建根文件系统之 init 进程分析)

根据上节可知,第一个启动的应用程序为 init

上面的文件系统有很多命令,比如 ls cp 等等,这些命令其实对应一个应用程序

假如说我们有很多命令,如果每个命令都要去找它们对于的源码然后编译显然很费事,

在嵌入式系统里面有个 busybox,他就是这些命令的组合,我们去编译 busybox 时候生成一个一个应用程序 busybox,然后我们执行这些命令的时候就到这些 busybox 对于的链接,我们执行这些命令的时候就相当于执行 busybox 应用程序

输入 ls -l /bin/ls

ls -l /bin/cp

```

# ls -l /bin/ls
lrwxrwxrwx 1 1000 1000 7 Jan 6 2010 /bin/ls -> busybox
# ls -l /bin/cp
lrwxrwxrwx 1 1000 1000 7 Jan 6 2010 /bin/cp -> busybox
#

```

通过结果可以看到 ls 和 cp 命令都是到 busybox 的链接

可以知道我们的 init 进程其实也是到 **init** 的链接(这里是不是写错了?):

```

# ls -l /sbin/init
lrwxrwxrwx 1 1000 1000 14 Jan 6 2010 /sbin/init -> ../bin/busybox

```

所以我们想知道 init 进程做了什么就必须去找到 busybox 的源码

分析 busybox:

我们在 busybox 里面的文件里可以发现每个命令都有一个.c 文件

init 这个应用程序会做什么事情呢?

由于我们内核的最终目的是启动应用程序,而我们启动的第一个程序是 init 程序,但是我们最终肯定是要执行我们想执行的某个或某些程序所以:

1:init 程序里面应该有个配置文件,里面指定了后续要执行的程序,所以 init 程序肯定要读取这个配置文件

2:读取之后要解析这个配置文件

3:根据配置文件来执行用户的程序

按照上面的分析来分析 init 应用程序:

我们搜索下 init 的文件:发现里面有个 int init_main(int argc, char **argv)函数

1:如何打开配置文件表找到 init 文件:调用过程如下(注意下面的前后带表执行的顺序)

Busybox->init_main//在 init_main 中调用 parse_inittab

parse_inittab //里面代开配置文件和解析

fopen(INITTAB, "r");//打开配置文件表"/etc/inittab"

1:如果文件为空(即没有配置文件),则会执行一套默认的配置,至里面调用了 new_init_action 这个函数

2:如果有配置文件,最后还是执行了 new_init_action 这个函数

分析 new_init_action 函数:


```

new_init_action//在 Parse_initab 解析完成配置文件后调用该函数
run_actions(SYSINIT);//后面的函数都是执行完 Parse_initab 继续反回//init_main 后执行的
    waitfor(a, 0);//等待该应用程序执行完毕
    run(a);//创建 process 子进程
    waitpid(runpid, &status, 0)//等待它结束
    delete_init_action(a);//将这个结构从链表中删除

run_actions(WAIT);

    waitfor(a, 0);//等待该应用程序执行完毕
    run(a);//创建 process 子进程
    waitpid(runpid, &status, 0)//等待它结束
    delete_init_action(a);//将这个结构从链表中删除

run_actions(ONCE);//和前面两类时机不同,这个不会等他她结束
    run(a);//创建 process 子进程
    delete_init_action(a);//将这个结构从链表中删除
while (1) {
    run_actions(RESPAWN);//重复运行已经退出来的子进程
    if (a->pid == 0) {
        a->pid = run(a);
    }

    run_actions(ASKFIRST);//重复运行已经退出来的子进程
    和 RESPAWN 的区别,在执行这类应用程序的时候会
    1:印"\nPlease press Enter to activate this console. ";
    2:等待回车
    创建子进程
    if (a->pid == 0) {
        a->pid = run(a);
    }

    wpid = wait(NULL); //等待任意子进程退出
    while (wpid > 0) {
        a->pid = 0;//该子进程退出后,这个子进程的 pid=0
    }
}

```

以默认配置中调用的

```

new_init_action(ASKFIRST, bb_default_login_shell, VC_2);为例如分析这个函数
分析: bb_default_login_shell 其实就是"/bin/sh" VC_2 就是 "/dev/tty2"
static void new_init_action(int action, const char *command, const char *cons)
结合配置文件表(inittab)可知
可以知道 action 其实就是执行的时机,commond 就是执行的应用程序,cons 就是 id
进入 new_init_action 函数里面看,其中有个链表:
struct init_action {
    struct init_action *next;
    int action;//这里就是执行时机
    pid_t pid;//进程号
    char command[INIT_BUFFS_SIZE];//要执行的应用程序
    char terminal[CONSOLE_NAME_SIZE];//对应的终端即 id(因为 id 在配置文件表中的作用介绍是说用作终端)
};

```

通过这个链表我们可以知道 new_init_action 这个函数要做什么事情

1:创建一个 init_action 结构

2:将创建的这个结构放入我们链表中(如果该结构已经存在,则覆盖)

总结:也就是说 init_action 是把所有我们要执行的应用程序假如一个链表中,但没有去执行,至于何时执行是要在 init_main 函数中对所有的应用程序进行了分类,然后分类执行然而我们有哪些类呢?

我们可以在配置文件表中找到关于 action 的解释:

```

# <action>: Valid actions include: sysinit, respawn, askfirst, wait, once,
# restart, ctrlaltdel, and shutdown.

```

也就是说 action 的默认执行程序就以上几类

很多类的 action 在最开始的时候分析了,还有一些没有分析,是在 init_main 函数最开始的时候有很多信号量函数,比如这句: signal(SIGINT, ctrlaltdel_signal);

当按键按下 ctrl+alt+del 时,会发出信号,去执行相应的应用程序

2:如何解析配置文件表

我们打开配置文件 inittab 文档可以发现他的格式里面介绍是

```
<id>:<runlevels>:<action>:<process>
```

我们可以猜测下:

2.1:这个配置文件里面肯定指定了很多的应用的程序,

2.2:这个配置文件的应用程序什么时候会去执行肯定会有指示

分析 inittab 文件:

inittab 格式为:

```
<id>:<runlevels>:<action>:<process>
```

该文件中介绍:

```
<id>:
The id field is used by BusyBox init to specify the controlling tty for the specified process to run on. The contents of this field are appended to "/dev/" and used as-is. The contents of this field are appended to "/dev/" and used as-is.

```

说明 id 项前面会加上一个 dev 前缀,即:

id =>/dev/id //用做终端:终端就是我们的标准输入,输出,错误等等

runlevels:可以忽略

action:执行时机,用来只是何时执行的

process:应用程序或脚本

Q:从 parse_inittab 函数中的默认配置来推到对应的配置文件表

```
if (file == NULL) {
    /* No inittab file -- set up some default behavior */
#endif
    /* Reboot on Ctrl-Alt-Del */
    new_init_action(CTRLALTD, "reboot", "");
    /* Umount all filesystems on halt/reboot */
    new_init_action(SHUTDOWN, "umount -a -r", "");
    /* Swapoff on halt/reboot */
    if (ENABLE_SWAPONOFF) new_init_action(SHUTDOWN, "swapoff -a", "");
    /* Prepare to restart init when a HUP is received */
    new_init_action(RESTART, "init", "");
    /* Askfirst shell on tty1-4 */
    new_init_action(ASKFIRST, bb_default_login_shell, "");
    new_init_action(ASKFIRST, bb_default_login_shell, VC_2);
    new_init_action(ASKFIRST, bb_default_login_shell, VC_3);
    new_init_action(ASKFIRST, bb_default_login_shell, VC_4);
    /* sysinit */
    new_init_action(SYSINIT, INIT_SCRIPT, "");

    return;
}
#endif
配置文件格式      <id>:<runlevels>:<action>:<process>
Action 时机的分类: sysinit, respawn, askfirst, wait, once, restart, ctrlaltdel, and shutdown.
上述的 new_init_action 函数分别对应为:
::ctrlaltdel:reboot
::shutdown:umount -a -r
::restart: init
:: askfirst:-/bin/sh
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh
::sysinit:/etc/init.d/rcS
```

总结最小根文件系统包含:

- 1:终端:dev/console /dev/null(这个是当我的应用程序错误的时候用的)
- 2:init 程序(来自 busybox)
- 3:/etc/inittab:配置文件表
- 4:配置文件里指定的应用程序
- 5:库:应用程序会用到很多函数,由 C 库提供

Q1:我们在分析内核启动第一个应用程序的时候发现:

如果我们的 uboot 的启动的时候若 init=没有设置启动参数的时候,内核就会启动

执行/sbin/init 做为第一个启动的应用程序

而我们启动开发板的时候发现 uboot 的 init=/linuxrc

而我通过编译安装 busybox 的时候,发现里面的目录刚好有一个 linuxrc 和/sbin/init

```
lrwxrwxrwx 1 root root 11 Jan 31 20:43 linuxrc -> bin/busybox
lrwxrwxrwx 1 root root 14 Jan 31 20:43 init -> ../bin/busybox
```

是不是就是说,这两个都是可以作为 uboot 启动的第一个程序呢?那么这个两个程序有什么差别呢?

并且 uboot 里面的 init=/linuxrc 的这个第一个应用程序是不是就是指这个 busybox 里面的 linuxrc 呢?

Q2:为什么要创建 console 和 NULL

11.3: 配置,编译 busybox (第 11 课第 3 节 构建根文件系统之 busybox)

1:在虚拟机解压 busybox-1.7.0.tar.bz2

```
tar -xjf busybox-1.7.0.tar.bz2
```

2:在 busybox-1.7.0.tar.bz2 里面找到 readme 看看里面有没有配置编译说明,发现没有,在 INSTALL 文件中有:

```
make menuconfig # This creates a file called ".config"
make             # This creates the "busybox" executable
make install     # or make CONFIG_PREFIX=/path/from/root install
```

需要注意的是第三步中的 make install 是默认安装在 PC 机上,而我们要安装在嵌入式开发板上所以需要执行后面的命令 make CONFIG_PREFIX=/path/from/root install

3:配置 busybox:make menuconfig

提示 Makefile:405: *** mixed implicit and normal rules. Stop.

提示 Makefile:1246: *** mixed implicit and normal rules. Stop.

方法:修改 makefile

进入配置界面:做以下微调:

3.1:busybox settings->build busybox as a static binary

如果选择把 busybox 编译为静态的话,则构建最小跟文件系统的条件 4--库就不需要了,除非客户自己添加的应用程序需要库,否则就不需要了,我们这里选择动态的,这个不用选

3.2:设置交叉工具链的,

方法一:发现在配置界面找不到,去 makefile 里面修改(在 1.20.0 的 busybox 的配置界面可以找到):

搜索 CROSS,一般交叉工具链都是用 cross 来开头的

将 CROSS_COMPILE ?=

修改为 CROSS_COMPILE ?= arm-linux-

方法二:在 make 的时候加上 make CROSS_COMPILE= arm-linux-

3.3:增加 tab 补全功能

配置界面-> busybox settings ->busybox library tuning->tab completion

其他详细的可以看开发手册

修改完配置界面后退出保存

4:执行 make 命令

我有用过 arm-linux-gcc 4.3.2 版本来编译,提示:

```
In file included from /root/Dev/Work_Space/Tool/usr/local/arm/4.3.2/bin/../arm-none-linux-gnueabi/libc/usr/
include/linux/netfilter_ipv4.h:8,
      from ipsvd/tcpudp.c:33:
/root/Dev/Work_Space/Tool/usr/local/arm/4.3.2/bin/../arm-none-linux-gnueabi/libc/usr/include/linux/netfilter
.h:44: error: field 'in' has incomplete type
/root/Dev/Work_Space/Tool/usr/local/arm/4.3.2/bin/../arm-none-linux-gnueabi/libc/usr/include/linux/netfilter
.h:45: error: field 'in6' has incomplete type
ipsvd/tcpudp.c: In function 'tcpudpsvd_main':
ipsvd/tcpudp.c:314: warning: ignoring return value of 'write', declared with attribute warn_unused_result
scripts/Makefile.build:192: recipe for target 'ipsvd/tcpudp.o' failed
make[1]: *** [ipsvd/tcpudp.o] Error 1
```

方法:

解决办法:

arm-linux 交叉编译工具链

在 usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/usr/include/linux/netfilter.h 头文件的开头

添加缺少的头文件:

```
#include <netinet/in.h>
```

5:执行安装命令,千万不要直接 make install 否则会破坏 PC 机

5.1 现在虚拟机上创建一个文件夹 mkdir first_fs(注:这个文件可以自己随便命名)

5.2 make CONFIG_PREFIX=/path/from/root install 将/path/from/root 更改为自己创建的文件夹路径即 make CONFIG_PREFIX=/xxxxxx/ first_fs install

6:进入刚刚安装 busybox 的文件夹,输入 ls -l 可以发现里面已经有了如下应用程序

```
drwxr-xr-x 2 root root 4096 Jan 31 20:43 bin
lrwxrwxrwx 1 root root 11 Jan 31 20:43 linuxrc -> bin/busybox
drwxr-xr-x 2 root root 4096 Jan 31 20:43 sbin
drwxr-xr-x 4 root root 4096 Jan 31 20:43 usr
```

上面也就完成了 busybox 的配置和编译安装,也就完成了最小跟文件系统的 init,也就是第 2 点

11.4: 构建根文件系统 (第 11 课第 4 节 构建根文件系统之构建根文件系统)

1.1 内容:可执行程序,库,配置文件,设备节点

1.2 制作映像文件

1.3 烧写

之前我们知道了要创建最小根文件系统需要:

1:终端:dev/console /dev/null(这个是当我的应用程序错误的时候用的)

2:init 程序(来自 busybox)

3:/etc/inittab:配置文件表

4:配置文件里指定的应用程序

5:库:应用程序会用到很多函数,由 C 库提供

Q1:构建根文件系统的流程是怎样的?

(具体的参考保存的网页: 详解制作根文件系统.html)

A1:我们打开虚拟机,然后打开虚拟机的根目录,来观察他的根文件系统的布局:

目录名	存放的内容
/bin	必备的用户命令，例如ls、cp等
/sbin	必备的系统管理员命令，例如ifconfig、reboot等
/dev	设备文件，例如mtdblock0、tty1等
/etc	系统配置文件，包括启动文件，例如inittab等
/lib	必要的链接库，例如C链接库、内核模块
/home	普通用户主目录
/root	root用户主目录
/usr/bin	非必备的用户程序，例如find、du等
/usr/sbin	非必备的管理员程序，例如chroot、inetd等
/usr/lib	库文件
/var	守护程序和工具程序所存放的可变，例如日志文件
/proc	用来提供内核与进程信息的虚拟文件系统，由内核自动生成目录下的内容
/sys	用来提供内核与设备信息的虚拟文件系统，由内核自动生成目录下的内容
/mnt	文件系统挂接点，用于临时安装文件系统
/tmp	临时性的文件，重启后将自动清除

所有的 linux 发行版在对根文件系统布局上都遵循 FHS 标准的建议规定
制作根文件系统就是要建立以上的目录，并在其中按照上面的存放内容建立完整目录内容。其过程大体包括：

- 编译 / 安装 busybox，生成/bin、/sbin、/usr/bin、/usr/sbin 目录
(对应韦老师的 5-2: init 程序(来自 busybox))
- 利用交叉编译工具链，构建/lib 目录
(对应韦老师的 5-5: 库:应用程序会用到很多函数,由 C 库提供)
- 手工构建/etc 目录

(对应韦老师的 5-3: /etc/inittab:配置文件表)

- 手工构建最简化的/dev 目录
(对应韦老师的 5-1: 终端:dev/console /dev/null)
- 创建其它空目录
- 配置系统自动生成/proc 目录
- 利用 udev 构建完整的/dev 目录
- 制作根文件系统的 jffs2 映像文件

下面依次分析：

1:编译 / 安装 busybox，生成/bin、/sbin、/usr/bin、/usr/sbin 目录
A:为什么要编译 busybox 呢?
busybox 简介

busybox 是一个集成了一百多个最常用 linux 命令和工具的软件,他甚至还集成了一个 http 服务器和一个 telnet 服务器,而所有这一切功能却只有区区 1M 左右的大小.我们平时用的那些 linux 命令就好比是分立式的电子元件,而 busybox 就好比是一个集成电路,把常用的工具和命令集成压缩在一个可执行文件里,功能基本不变,而大小却小很多倍,在嵌入式 linux 应用中,busybox 有非常广的应用,另外,大多数 linux 发行版的安装程序中都有 busybox 的身影,安装 linux 的时候案 ctrl+alt+F2 就能得到一个控制台,而这个控制台中的所有命令都是指向 busybox 的链接。

而 linux 命令有很多,我们不可能每个命令都写个 .c 文件
如果要自己编写这几百个常用命令的源程序, my god, 这简直是一个噩梦! 好在我们有嵌入式 Linux 系统的瑞士军刀—busybox, 事情就简单很多。
另外这些目录下存储的主要是常用命令的二进制文件。有没有发现编译完 busybox 基本上自动生成的目录都是包含命令的目录,所以我们可以猜测到 busybox 的作用就是：
常用 linux 命令和工具的软件合集

2:利用交叉编译工具链，构建/lib 目录
光有应用程序（命令）是不够的，因为应用程序本身需要使用 C 库的库函数，因此还必需制作 for ARM 的 C 库，并将其放置于/lib 目录。my god, 要自己写 C 库的源代码吗？不用！还记得交叉编译工具链的 3 个组成部分吗？交叉编译器、for ARM 的 C 库和二进制工具。哈哈，for ARM 的 C 库是现成的，我们只需要拷贝过来就可以了。遗憾的是：整个 C 库目录下的文件总大小有 26M。而我们根文件系统所在分区不过区区 16M 而已，根本 放不下。怎么办呢？
dennis@dennis-desktop:/work/nfs_root/fs_mini3\$ du -s --si /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
26M /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
需要 C 库目录下所有的文件吗？no, absolutely no! 让我们来分析一下 glibc 库目录下内容的组成。该目录下的子目录和文件共分 8 类：

- 目标文件，如 crtn.o, 用于 gcc 链接可执行文件
- libtool 库文件（.la），在链接库文件时这些文件会被用到，比如他们列出了当前库文件所依赖的其它库文件，程序运行时无需这些文件
- gconv 目录，里面是各种链接脚本，在编译应用程序时，他们用于指定程序的运行地址，各段的位置等
- 静态库文件（.a），例如 libm.a, libc.a

- 动态库文件 (.so、.so.[0-9]*)
- 动态链接库加载器 ld-2.3.6.so、ld-linux.so.2
- 其它目录及文件

很显然，第 1、2、3、4、7 类文件和目录是不需要拷贝的。

由于动态链接的应用程序本身并不含有它所调用的 C 库函数的代码，因此执行时需要动态链接库加载器来为它加载相应的 C 库文件，所以第 6 类文件是需要拷贝的。

除此之外，第 5 类文件当然也要拷贝。但第 5 类文件的大小也相当大。

```
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ du -c --si *.so*
7.2M    total
```

需要全部拷贝吗？非也，非也！其实，需要哪些库完全取决于要运行的应用程序使用了哪些库函数。如果我们只制作最简单的系统，那么我们只需要运行 busybox 这一个应用程序即可。通过执行

```
dennis@dennis-desktop:/work/nfs_root/fs_mini3$ arm-linux-readelf -a bin/busybox | grep 'Shared'
0x00000001 (NEEDED)             Shared library: [libcrypt.so.1]
0x00000001 (NEEDED)             Shared library: [libm.so.6]
0x00000001 (NEEDED)             Shared library: [libc.so.6]
```

可知：busybox 只用到了 3 个库：通用 C 库（libc）、数学库（libm）、加密库（libcrypt），因此我们只需要拷贝这 3 个库的库文件即可。但是每个库都有 4 个文件，4 个文件都要拷贝吗？当然不是。

```
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ ls -l libcrypt[.]*
```

```
-rwxr-xr-x 1 dennis dennis 30700 2008-01-22 05:32 libcrypt-2.3.6.so
-rw-r--r-- 1 dennis dennis 23118 2008-01-22 05:32 libcrypt.a
lrwxrwxrwx 1 dennis dennis 13 2008-12-22 15:38 libcrypt.so -> libcrypt.so.1
lrwxrwxrwx 1 dennis dennis 17 2008-12-22 15:38 libcrypt.so.1 -> libcrypt-2.3.6.so
```

```
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ ls -l libm[.]*
```

```
-rwxr-xr-x 1 dennis dennis 779096 2008-01-22 05:31 libm-2.3.6.so
-rw-r--r-- 1 dennis dennis 1134282 2008-01-22 05:32 libm.a
lrwxrwxrwx 1 dennis dennis 9 2008-12-22 15:38 libm.so -> libm.so.6
lrwxrwxrwx 1 dennis dennis 13 2008-12-22 15:38 libm.so.6 -> libm-2.3.6.so
```

```
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ ls -l libc[.]*
```

```
-rwxr-xr-x 1 dennis dennis 1435660 2008-01-22 05:48 libc-2.3.6.so
-rw-r--r-- 1 dennis dennis 2768280 2008-01-22 05:31 libc.a
-rw-r--r-- 1 dennis dennis 195 2008-01-22 05:34 libc.so
lrwxrwxrwx 1 dennis dennis 13 2008-12-22 15:38 libc.so.6 -> libc-2.3.6.so
```

4 个文件中的 .a 文件是静态库文件，是不需要拷贝的。另外 3 个文件是：

- 实际的共享链接库：libLIBRARY_NAME-GLIBC_VERSION.so。当然需要拷贝。
- 主修订版本的符号链接，指向实际的共享链接库：libLIBRARY_NAME.so.MAJOR_REVISION_VERSION，程序一旦链接了特定的链接库，将会参阅该符号链接。程序启动时，加载器在加载程序前，会检索该文件。所以需要拷贝。
- 与版本无关的符号链接，指向主修订版本的符号连接（libc.so 是唯一的例外，他是一个链接命令行：libLIBRARY_NAME.so，是为编译程序时提供一个通用条目）。这些文件在程序被编译时会被用到，但在程序运行时不会被用到，所以不必拷贝它。

关于共享库的 2 个符号链接的作用的特别说明：

当我们使用 gcc hello.c -o hello -lm 编译程序时，gcc 会根据 -lm 的指示，加头（lib）添尾（.so）得到 libm.so，从而沿着与版本无关的符号链接（libm.so -> libm.so.6）找到 libm.so.6 并记录在案（hello 的 ELF 头中），表示 hello 需要使用 libm.so.6 这个库文件所代表的数学库中的库函数。而当 hello 被执行的时候，动态链接库加载器会从 hello 的 ELF 头中找到 libm.so.6 这个记录，然后沿着主修订版本的符号链接（libm.so.6 -> libm-2.3.6.so）找到实际的共享链接库 libm-2.3.6.so，从而将其与 hello 作动态链接。可见，与版本无关的符号链接是供编译器使用的，主修订版本的符号链接是供动态链接库加载器使用的，而实际的共享链接库则是供应用程序使用的。

通过以上分析，我们只需要拷贝 3 个库（每个库各 1 个主修订版本的符号链接和 1 个实际的共享链接库）以及动态链接库加载器（1 个符号链接和 1 个实体文件）。步骤如下：

```
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ mkdir /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libcrypt-* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -l libcrypt.so.* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libm-* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -l libm.so.* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libc-* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -l libc.so.* /work/nfs_root/fs_mini3/lib
dennis@dennis-desktop:/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -l ld-* /work/nfs_root/fs_mini3/lib
```

3:手工构建/etc 目录

/etc 目录存放的是系统程序的主配置文件，因此需要哪些配置文件取决于要运行哪些系统程序。即使最小的系统也一定会运行 1 号用户进程 init，所以我们至少要手工编写 init 的主配置文件 inittab。busybox 的 inittab 文件的语法、语义与传统的 SYSV 的 inittab 有所不同。

inittab 文件中每个条目用来定义一个需要 init 启动的子进程，并确定它的启动方式，格式为 <id>:<runlevel>:<action>:<process>。例如：
ttySAC0::askfirst:-/bin/sh

4:手工构建简化的/dev 目录

A:为什么要创建或者我怎么知道需要创建 console 和 NULL 两个设备文件呢？

在 linux 机器上，执行 ls /dev 可看到几百个设备文件，我需要手工创建它们吗？maybe，我只需要手工创建几个设备文件！我怎么知道我应该创建哪几个设备文件呢？管它呢，先看看开发板上可爱的 linux 的反应再说。

启动 Linux 操作系统，显示：

```
VFS: Mounted root (nfs filesystem).
Freeing init memory: 112K
Warning: unable to open an initial console.
```

这说明，内核已经成功挂载根文件系统，但却未能成功启动第 1 个用户进程 init。通过错误消息“unable to open an initial console”搜索内核源代码，找到 init/main.c 文件。

```
748 static int noinline init_post(void)
749 {
750     free_initmem();
751     unlock_kernel();
752     mark_rodata_ro();
753     system_state = SYSTEM_RUNNING;
754     numa_default_policy();
755
756     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
757         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
758
759     (void) sys_dup(0);
760     (void) sys_dup(0);
```

```

761
762     if (ramdisk_execute_command) {
763         run_init_process(ramdisk_execute_command);
764         printk(KERN_WARNING "Failed to execute %s\n",
765                 ramdisk_execute_command);
766     }
767
768     /*
769     * We try each of these until one succeeds.
770     *
771     * The Bourne shell can be used instead of init if we are
772     * trying to recover a really broken machine.
773     */
774     if (execute_command) {
775         run_init_process(execute_command);
776         printk(KERN_WARNING "Failed to execute %s. Attempting "
777                 "defaults...\n", execute_command);
778     }
779     run_init_process("/sbin/init");
780     run_init_process("/etc/init");
781     run_init_process("/bin/init");
782     run_init_process("/bin/sh");
783
784     panic("No init found. Try passing init= option to kernel.");
785 }

```

显然，内核错误是由 175 行不能打开/dev/console 所致。通过查看已经安装好的 linux 机器的/dev/console 设备文件，可知其是字符设备文件，主设备号为 5，次设备号为 1：

```

dennis@dennis-desktop:/work/nfs_root/fs_mini3/etc$ ls -l /dev/console
crw----- 1 root root 5, 1 2010-04-08 08:40 /dev/console

```

因此，我们使用下面的命令创建它：

```

dennis@dennis-desktop:/work/nfs_root/fs_mini3/dev$ sudo mknod console c 5 1

```

还需要创建其它设备文件吗？只有天知道！再看看 linux 的反应。

```

VFS: Mounted root (nfs filesystem).

```

```

Freeing init memory: 112K

```

```

init: can't open '/dev/null': No such file or directory

```

这次我们有经验了，如法炮制，创建/dev/null 设备文件：

```

dennis@dennis-desktop:/work/nfs_root/fs_mini3/dev$ sudo mknod null c 1 3

```

再次重启开发板上的 linux，显示

```

VFS: Mounted root (nfs filesystem).

```

```

Freeing init memory: 112K

```

```

init started: BusyBox v1.7.0 (2010-04-03 23:53:55 CST)

```

```

starting pid 229, tty '': '/etc/init.d/rcS'

```

```

Please press Enter to activate this console.

```

```

starting pid 231, tty '': '/bin/sh'

```

```

#

```

5:创建其它空目录

6:配置系统自动生成/proc 目录

7:利用 udev 构建完整的/dev 目录

8:制作根文件系统的 jffs2 映像文件

根文件系统已经制作完毕，最后一个步骤是将其打包为 jffs2 映像文件，以供 bootloader 将其烧录到 nand flash 上。

一：内容:可执行程序,库,配置文件,设备节点

创建最小根文件系统步骤

1:创建终端:即创建这两个设备文件: dev/console /dev/null

1.1:首先我们看下在 linux PC 机上这两个设备:如下:

```

[root@fedora ~]# ls /dev/console /dev/null -l
crw----- 1 root root 5, 1          Feb  1  2017 /dev/console
crw-rw-rw- 1 root root 1, 3          Feb  1  2017 /dev/null

```

其中 crw 中 c 表示为字符设备 5 为主设备号, 1 为次设备号

1.2:在我们上节下载 busybox 的文件中创建这两个设备:

1.2.1:先创建 dev 目录:mkdir dev

1.2.2:进入 dev 目录并创建两个设备:

```

cd dev

```

```

mknod console c 5 1    (这里 c 表示为字符设备,5 为主设备号,1 为次设备号)

```

如果如果提示权限不够,则在前面加上 sudo

```

mknod null c 1 3

```

2: init 程序(来自 busybox)这项在上小结完成了

3:创建/etc/inittab:配置文件表

3.1:在 dev 同等级目录中创建 etc 文件 mkdir etc

```

vim inittab    (创建 inittab)

```

在其中添加我们 init 应用进程中的默认配置中的一个:

```

:: askfirst:~/bin/sh

```

并且在这个配置文件中假如终端,即 id,最后在 init 文件中内容为:

```

console:: askfirst:~/bin/sh

```

加入这个终端后,这个应用程序~/bin/sh 的标准输入,输出,错误就会定位到 console 中

4: 配置文件里指定的应用程序:这个由于我们根本没有指定用户程序,可以不用了

5:c 库

步骤:进入我们之前的安装的交叉编译工具的目录:

```

cd Tool/gcc-3.4.5-glibc-2.3.6/arm-linux/lib

```

```

ls lib -l

```

可以看到里面就有所有的库

其中以.a结尾的是静态库:例如: libutil.a

librt.so -> librt.so.1 这种就表示 librt.so 文件是链接到 librt.so.1 这里的

我们使用 cp 命令来拷贝库到我们的最小根文件系统的时候,如果我们不加 -d 选项的话 librt.so 就会被拷贝成 librt.so.1 这个文件的内容,那样的话会库会大很多

1:首先在 nfs_root/first_fs 中创建 lib 目录:mkdir lib

2:进入 Tool/gcc-3.4.5-glibc-2.3.6/arm-linux/lib 这个目录

3:将所有的动态库文件拷贝至我们的创建的 lib 文件夹中

cp *.so* /root/Work/nfs_root/first_fs/lib/ -d

注意的是 a:通配符的使用,我们要的是动态库 b:cp 选项字 -d 不要忘了

至此我们就完成了最小的根文件系统

然后我们如何将这最小的根文件系统烧录到开发板上呢?就需要我们制作映像文件

二:制作 yaffs2 映像文件

制作 yaffs2 方法(应用手册是 yaffs1 的制作方法:是针对小页的 nandflash 来使用的,每页 512 字节而 yaffs2 是每页 2048byte):

1:将 yaffs_source_util_larger_small_page_nand.tar.bz2 复制到虚拟机中并解压(这个文件既可以支持大页也可以支持小页)

tar -xjf yaffs_source_util_larger_small_page_nand.tar.bz2

解压出的文件为: Development_util_ok

2:进入解压出来的文件进入以下目录:

/root/Work/System/Development_util_ok/yaffs2/utlis

可以参照手册上的修改步骤来修改,这个是已修改好的

3:然后直接 make

然后生成了很多工具:

```
[root@fedora utlis]# ls
Makefile          mkyaffsimage.c  yaffs_ecc.o      yaffs_tagsvalidity.c
mkyaffs2image     mkyaffsimage.o  yaffs_packedtags1.c yaffs_tagsvalidity.o
mkyaffs2image.c   nand_ecc.c      yaffs_packedtags1.o
mkyaffs2image.o   nand_ecc.o      yaffs_packedtags2.c
mkyaffsimage      yaffs_ecc.c     yaffs_packedtags2.o
而我们要使用的是: mkyaffs2image
```

4:将上述工具拷贝至 PC 机系统目录中

cp mkyaffs2image /usr/local/bin/

并给该文件加上可执行属性:

chmod +x /usr/local/bin/mkyaffs2image

现在就可以使用这个工具了

5:进入我们的最小根文件文件夹

输入 mkyaffs2image 命令就可以看到这个工具如何使用:

```
[root@fedora first_fs]# mkyaffs2image
mkyaffs2image: image building tool for YAFFS2 built Feb  1 2017
usage: mkyaffs2image dir image_file [convert]
        dir          the directory tree to be converted
        image_file   the output file to hold the image
        'convert'    produce a big-endian image from a little-endian machine
```

注意:我们的最小根文件系统在 nfs_root/first_fs 中,我们制作映像文件要在 nfs_root 下,而不是在我们的最小根文件系统 first_fs 中执行

```
[root@fedora nfs_root]# mkyaffs2image first_fs first_fs.yaffs2
```

6:可以发现在 nfs_root 下生成了第一个映像文件: first_fs.yaffs2

7:烧录:用 usb 来下载

7.1:将映像文件 first_fs.yaffs2 拷到 windows 下

7.2:重启开发板进入 uboot 命令行界面:按下 y 命令进入下载 yaffs 镜像

7.3:打开 dnw 通过 USB 来下载到开发板

7.4:输入 b 命令来启动内核

最后可以看到最后一句:

```
Please press Enter to activate this console.
starting pid 763, tty '': '/bin/sh'
```

这里可以看出来,我们这里的配置文件只有

console:: askfirst:~/bin/sh 这一句,

它的启动时机是 askfirst 我们之前分析过如果是这类启动时机的应用程序他相对于

RESPAWN 的区别是:

1:印"\nPlease press Enter to activate this console. ";

2:等待回车

这里就完成了并烧录了第一个根文件系统

三:完善根文件系统

完善我们的最小根文件系统

完善 1:通过修改配置文件来完善

1:我们输入 ps 命令,发现提示没有这个文件

```
# ps
```

```
PID  Uid          VSZ  Stat Command
```

```
ps: can't open '/proc': No such file or directory
```

- 2: 创建一个 proc 文件夹
mkdir proc
- 3: 还要挂载一个虚拟的根文件系统
mount -t proc none /proc
- 4: 再次 ps 发现可以成功:

```
# ps
  PID  uid      VSZ  Stat  Command
    1   0        3092  S     init
    2   0          SW<   [kthreadd]
    3   0          SWN   [ksoftirqd/0]
    4   0          SW<   [watchdog/0]
    5   0          SW<   [events/0]
    6   0          SW<   [khelper]
   55   0          SW<   [kblockd/0]
   56   0          SW<   [ksuspend_usbd]
   59   0          SW<   [khubd]
   61   0          SW<   [kseriod]
   73   0          SW   [pdflush]
   74   0          SW   [pdflush]
   75   0          SW<   [kswapd0]
   76   0          SW<   [aio/0]
  710   0          SW<   [mtdblockd]
  745   0          SW<   [kmmcd]
  763   0        3096  S     -sh
  786   0        3096  R     ps

# cd /proc
# ls
1      745      diskstats  locks      sys
2      73       driver     meminfo    sysrq-trigger
3      76       execdmain misc        sysvipc
4      763      fb         modules    timer_list
5      787      filesystems mounts      tty
55     asound    fs         mtd         uptime
56     buddyinfo interrupts net         version
59     bus      iomem     partitions vmstat
6      cmdline  ioports   scsi        yaffs
61     cpu      irq       self        zoneinfo
710    cpuinfo  kallsyms  slabinfo
73     crypto  kmsg      stat
74     devices loadavg   swaps

# cd 1
# ls
auxv      environ  maps      oom_adj    stat
clear_refs exe      mem        oom_score  statm
cmdline   fd       mounts     root        status
cwd       fdinfo   mountstats root         task

# ls -l fd
lrwx----- 1 0 0 64 Jan 1 00:26 0 -> /dev/console
lrwx----- 1 0 0 64 Jan 1 00:26 1 -> /dev/console
lrwx----- 1 0 0 64 Jan 1 00:26 2 -> /dev/console
```

如果我们想不用每次手动挂载的话:

方法一:

- 1: first_fs 中添加 proc 文件 mkdir proc
- 2: 修改配置文件:
cd etc
vim inittab
在里面添加 ::sysinit:/etc/init.d/rcS
即最终的 inittab 中的内容为:
::askfirst:/bin/sh
::sysinit:/etc/init.d/rcS (这个进程是运行一个脚本文件, 这样我们通过创建修改这个脚本, 在脚本里加入挂载虚拟文件 proc 就好了)
- 3: 新增一个脚本
[root@fedora first_fs]# mkdir etc/init.d
[root@fedora first_fs]# vi etc/init.d/rcS (mkdir 创建的是文件夹或目录不是文件, 直接用 vim 来创建)
在 rcS 中添加一句: mount -t proc none /proc
- 4: 修改这个脚本的可以执行属性
chmod +x etc/init.d/rcS
这样就可以重新烧录到开发板, 执行 ps 命令就可以成功

方法二:
和方法一一样, 将 rcS 中的内容更改为 mount -a, 这样可以不用创建 proc 文件夹
但是这个命令需要依赖于另一个文件: /etc/fstab (这样就根据 fstab 这个配置文件的内容来挂载文件系统)
这样我们就要另外在/etc 中创建一个 fstab:

Mkdir fstab
里面的内容为 (具体的 fstab 的格式可以看应用手册)
具体的格式为
Device mount-point type options dump fsck order
device: 挂载哪个设备, 对于我们而言, 由于 proc 是虚拟的文件系统, 所以这里的 device 可以随便写
mount-point: 挂载到哪里去
type: 什么文件系统
后面的几个选项都是参数
例如我们这里在 fstab 中写的是:
proc /proc proc default 0 0
这样也可实现同样的效果

具体的步骤:
1: 修改 etc/init.d/rcS 这个脚本文件: 将方法一之前的内容屏蔽掉
最终结果为 mount -a
2: 创建在/etc 中创建一个 fstab
vim fstab (不要用 mkdir, 它会创建成文件夹)
3: 修改 fstab 里面的内容 (第一行是注释):
#device mount-point type options dump fsck order
proc /proc proc defaults 0 0
这里就完成了, 我们烧录下看行不行

1: 创建映像文件: 注意要退回 nfs_root 目录下
[root@fedora nfs_root]# mkyaffs2image first_fs first_fs.yaffs2
2: 重启开发板进入 uboot 命令行进行烧录 按如 y 命令烧写 yaffs2 映像文件
3: 将文件拷贝之 windows, 用 dnw 烧录进开发板

4:烧录完成按 b 启动内核

5:

5.1:输入 ps 发现该命令可以实现

5.2:输入 cat /proc/mounts 该语句用来查看当前挂载了哪些文件系统,结果为:

```
# cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / yaffs rw 0 0
proc /proc proc rw 0 0
```

可以看到 proc 也挂载成功了

完善 2:完善 最小根文件系统 dev 目录

目前我们的 dev 中只有两个设备 console 和 null

假如我们有很多设备需要添加,不可能每个都自己添加

可以通过一个 udev 工具来实现,而我们用的是 mdev,其实就是 udev 的简化版

Udev 和 mdev 都可以自动创建 dev 目录下面的设备节点

如何使用 mdev:

在 busybox-1.70\docs\mdev.txt 里面有说明如何使用:截取步骤如下:

Here's a typical code snippet from the init script:

```
[1] mount -t sysfs sysfs /sys
```

```
[2] echo /bin/mdev > /proc/sys/kernel/hotplug
```

```
[3] mdev -s
```

Of course, a more "full" setup would entail executing this before the previous code snippet: [4] mount -t tmpfs mdev /dev

```
[5] mkdir /dev/pts
```

```
[6] mount -t devpts devpts /dev/pts
```

具体实施步骤:

1:由于上面说要挂载 mount -t sysfs sysfs /sys ,所以首先在最小根文件系统创建 sys 文件:

```
Mkdir sys
```

2:由于用到了 mount 这个命令根据完善 1 的步骤可以知道,如果要用到 mount 的话必须修改/etc/fstab 中,最终修改为(第一行的是在完善 1 中本来就存在的):

```
proc /proc      proc      default 0 0
sysfs /sys       sysfs     defaults 0 0
tmpfs /dev       tmpfs     defaults 0 0
```

至此完成了 mdev 的第 1 和 4 步

3:将 mdev 2,3,5,6 步骤的内容拷贝到 rcS 脚本中,最终内容为:

```
#mount -t proc none /proc          (这一步是完善 1 中的方法一的)
```

```
mount -a                          (这一步是完善 1 中的方法二的)
```

```
mkdir /dev/pts                    (后面的对于步骤 2,3,5,6)
```

```
mount -t devpts devpts /dev/pts
```

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

```
mdev -s
```

4:重新将新的文件系统烧录进开发板

5:烧录成功

5.1:输入 cat /proc/mounts 该语句用来查看当前挂载了哪些文件系统:

```
# cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / yaffs rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /dev tmpfs rw 0 0
devpts /dev/pts devpts rw 0 0
```

5.2:我们未假如 mdev 之前 dev 中只有两个设备 console 和 null,现在输入 ls /dev/可以发现有很多,这些都是 mdev 帮我们自动创建的

至此我们的最小的根文件系统就比较完善了

四: 制作 jffs2 映像文件

一般来说 jffs2 是用在 norflash 上的,也可以用在 nandflash 上,方法如下:

1:首先也得制作工具

视频从 30min 开始到 35min,这里就不做笔记了

五: nfs

我们每次在 linux 上修改了我们的最小根文件系统就要通过 dnw 烧写到开发板,很麻烦,可以通过网络文件系统 nfs,也就是说这个文件系统是放在服务器上面的,内核启动时,自动识别这个文件目录,把它当作我们的文件系统,这样就不需要烧写了

1:将网线连接上

2:用 ifconfig 命令看看有没有设置 ip,若没有则设置,(要在同一网段)

```
ifconfig eth0 xxxxxxxxx(要注意自己是不是 eth0)
```

3:看看能不能 ping 通

4:从 flash 上启动根文件系统,在用命令挂载 nfs

要挂载 nfs 需要满足的条件:

1:服务器允许那目录可被别人挂载

现在我们是想让 linux 的 first_fs 这个存放了我们的文件系统的目录能被别人挂载:

- 1.1:进入该文件系统
 - 1.2:输入 vim /etc/exports
 - 1.3:在里面加入一句
- 2:单板去挂接

第 12 课 字符设备驱动程序

1. 概念介绍
 - 1.1. 应用程序与内核,驱动的关系
 - 1.2. 框架
2. 第一个驱动程序:点灯
 - 2.1. 源码分析:驱动框架,硬件操作(ioremap),用户空间与内核空间的数据传输
 - 2.2. 怎么编译驱动:Makefile
 - 2.3. 测试
3. 第二个驱动程序:查询方式获取按键值
 - 3.1. 源码分析:
 - 3.2. 测试
4. 第三个驱动程序:中断方式获取按键值
 - 4.1. Linux 中断系统架构
 - 4.2. 源码分析:
 - 4.3. 测试

12.1: 概念介绍 (第 12 课第 1 节 字符设备驱动程序之概念介绍)

- 1 概念介绍
- 1.1. 应用程序与内核,驱动的关系
 - 1.2. 框架

Uboot 任务是启动内核
内核任务是启动应用程序
应用程序里面肯定会涉及

- 1:很多文件操作:读写文件
- 2:硬件操作:点灯,获取按键值

对于应用程序而言:只会操作标准接口(比如 read ,open,write)来调用应用程序

以点灯为例:

应用程序的 open 最终会调用对应驱动程序的 led_open
应用程序的 read 最终会调用对应驱动程序的 led_read
应用程序的 write 最终会调用对应驱动程序的 led_write

如图:

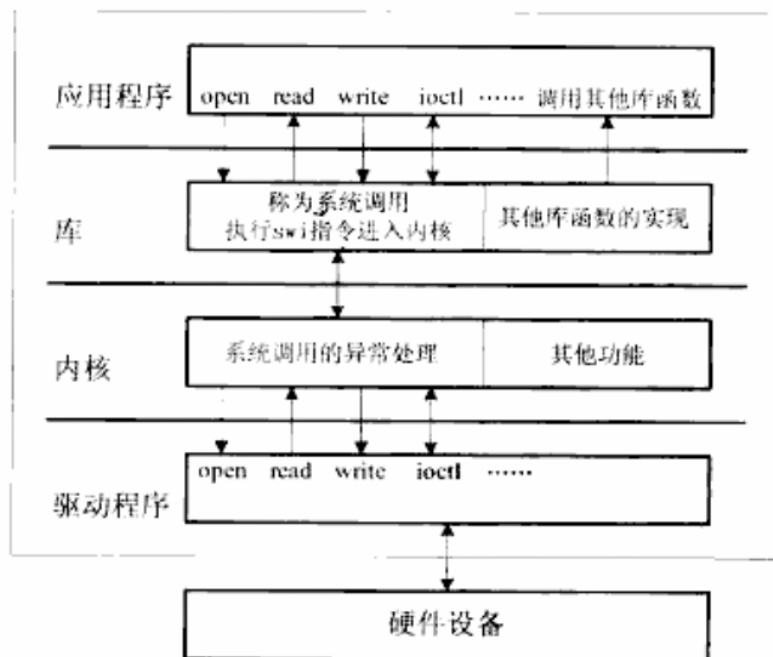
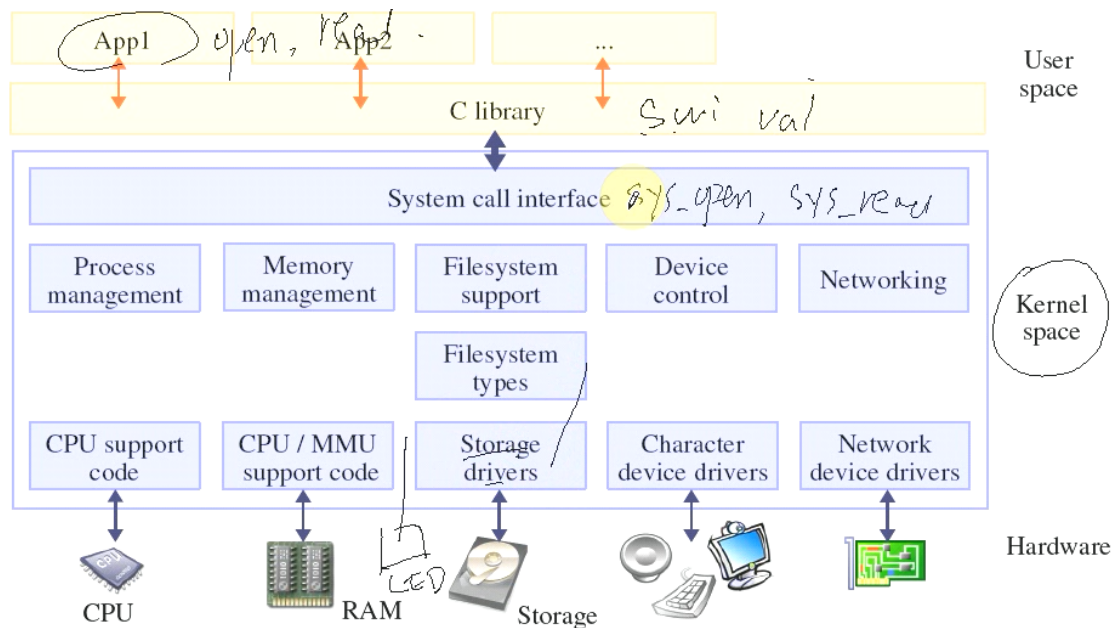


图 19.1 Linux 软件系统的层次关系 (swi 是 ARM 指令)



Q1: 应用程序中的 `open`, `read` 等函数并不是我们实现的, 那么是谁实现的?

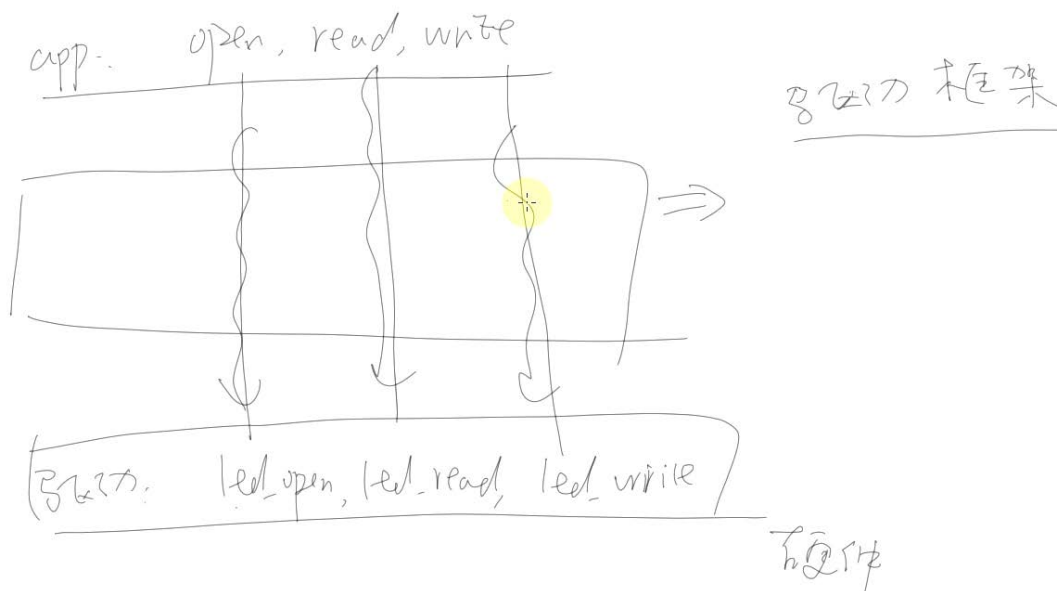
AN: 是由 C 库实现的, C 库和 `open` 这些函数都属于应用层, 当我们应用程序调用 `open` 这些系统调用接口的时候就会进入到内核, 内核包含上图的内核和驱动程序部分, (驱动也属于内核)

Q2: 那么 C 库是如何进入内核的?

AN: 当应用程序执行 `open` 等函数的时候实际上是执行一条 `swi` 指令

(具体的指令格式为: `swi + 某个值 val`), 这条指令就会引发一个异常, 相当于触发一个中断, 当发生中断时, 会进入我们内核的异常处理函数中, 就是上图中的系统调用的异常处理, 这里就会根据触发中断的值来做相应的处理: 即调用不同的处理的函数, 比如应用程序调用 `open` 函数, 传进来的是 `val1`

12.2: 第一个驱动程序: 点灯 (第 12 课第 2.1 节 字符设备驱动程序之 LED 驱动程序编写编译)



我们知道我们的应用程序最终会调用驱动程序中的对应函数

,这直接是如何实现这种对应关系的呢?就是通过我们的驱动框架来实现的,下面详细介绍:

步骤 1:先写出驱动程序中的 led_open 等等函数

步骤 2:有了这些驱动程序,那么怎么告诉内核已经有了这些函数呢?

2.1 通过定义一个结构体:file_operations,通过这个结构体的定义我们可以看到,应用程序中有哪些接口是和这个结构体——对应的

2.2 有了这个结构体,必须还要将这个结构体注册到驱动程序,通过 register_chrdev 来实习

2.3 有了这个函数,那么谁来调用呢?即驱动的入口是谁?

由 first_dirve_init,这个函数里面调用 register_chrdev

如果有很多驱动的话,那么这个驱动入口函数就会有多个,而且名字都是各不相同

2.4:为了让这个驱动入口函数能够被识别,必须做一定的修饰

通过 module_init 来实现:其实里面的就是定义一个结构体,结构体中有个函数指针来指向入口函数

12.3: 第二个驱动程序:查询获取按键值 (第 12 课第 1 节 构建根文件系统之启动第 1 个程序)

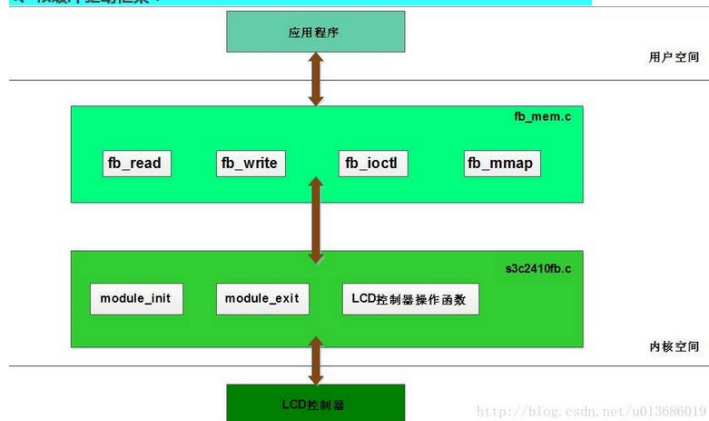
12.4: 第三个驱动程序:中断方式获取按键值 (第 12 课第 1 节 构建根文件系统之启动第 1 个程序)

第 15 课 LCD 驱动程序

明白几点

1: LCD 驱动程序的框架:

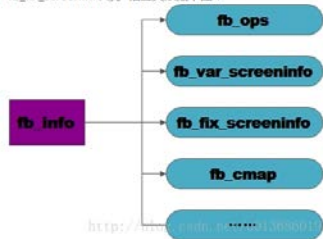
4、帧缓冲驱动框架:



LCD 这种设备我们称为帧缓冲设备,也属于字符设备中的一种,它的步骤其实和之前的字符设备一样,设备号, fileoperation 结构体,注册,出口函数,入口函数.

内核只不过将帧缓冲设备又进行了一次层次的分层,如上图所示,其中帧缓冲相关的代码已经在 fb_mem.c 中完成,他里面就已经完成了我们之前的字符设备中的所有操作,其中还有帮我们完成了创建设备节点,只不过对于我们而言,我们只要给提供一个 fb_info 的结构体即可

帧缓冲设备的数据结构在内核include/linux/fb.h文件中，关键的有fb_info、fb_ops、fb_var_screeninfo、fb_fix_screeninfo等。相互关系如下图：



我们可以看到 fb_info 结构体中又有了 fb_ops, linux 内核将帧缓冲层已经写好, 但是对于具体的帧缓冲设备肯定每个操作函数会有不一样, 所以我们要完成上面 fb_info 结构体的初始化, 注册到内核就好了

2: 假设我们已经实现了一个 fb_info 结构体, 并注册到内核了, 那么当用户空间对我们的帧缓冲设备进行读写或其他操作的时候, 又是怎么通过帧缓冲层来调用我们自己写的 fb_info 结构体中的 fb_ops 里面的函数的呢?

假设 用户空间进行 open 操作:

app: open("/dev/fb0", ...) 主设备号: 29, 次设备号: 0

kernel:

```
fb_open
int fbidx = iminor(inode);
struct fb_info *info = registered_fb[0];
```

假设用户空间进行 read 操作:

app: read()

kernel:

```
fb_read
int fbidx = iminor(inode);
struct fb_info *info = registered_fb[fbidx];
if (info->fbops->fb_read)
    return info->fbops->fb_read(info, buf, count, ppos);

src = (u32 __iomem *) (info->screen_base + p);
dst = buffer;
*dst++ = fb_readl(src++);
copy_to_user(buf, buffer, c)
```

我们可以看到上面举的两个例子, 用户空间对想对 LCD 操作的时候, 先找到帧缓冲层中对应的 open, read 函数

```
94:
95: static const struct file_operations fb_fops = {
96:     .owner = THIS_MODULE,
97:     .read = fb_read,
98:     .write = fb_write,
99:     .ioctl = fb_ioctl,
100:     .compat_ioctl = fb_compat_ioctl,
101:     .mmap = fb_mmap,
102:     .open = fb_open,
103:     .release = fb_release,
104:     .get_unmappable_area = get_fb_unmappable_area,
105:     .fsync = fb_deferred_io_fsync,
106: };
107:
108: #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
109: #define get_fb_unmappable_area get_unmappable_area
110: #endif
111: #ifdef CONFIG_FB_DEFERRED_IO
112: #define fb_deferred_io_fsync fb_deferred_io_fsync
113: #endif
114: }
```

帧缓冲层中的这些函数已经自己实现了, 他们的作用就是找到我们自己 LCD 注册到内核中 fb_info 结构体中对应的 open 等函数的, 上面的 open, 和 read 流程中有分析他们都是通过了一个 registered_fb[] 数组来查找要调用的 fb_info 结构体的, 我们自己的驱动中就是在注册的时候, 把我们的 fb_info 结构体注册到了这个数组中的

```
ret = register_framebuffer(fbinfo);
registered_fb[i] = fb_info;
```

第 17 课 USB 驱动程序

17.1 USB 驱动程序之概念介绍

现象: 把 USB 设备接到 PC

1. 右下角弹出"发现 android phone"
2. 跳出一个对话框, 提示你安装驱动程序

引出问题:

问 1. 既然还没有"驱动程序", 为何能知道是"android phone"

答 1. windows 里已经有了 USB 的总线驱动程序, 接入 USB 设备后, 是"总线驱动程序"知道你是"android phone"
提示你安装的是"设备驱动程序"

USB 总线驱动程序负责: 识别 USB 设备, 给 USB 设备找到对应的驱动程序

问 2. USB 设备种类非常多, 为什么一接入电脑, 就能识别出来?

答 2. PC 和 USB 设备都得遵守一些规范。

比如: USB 设备接入电脑后, PC 机会发出"你是什么"?

USB 设备就必须回答"我是 xxx", 并且回答的语言必须是中文 (格式一样)

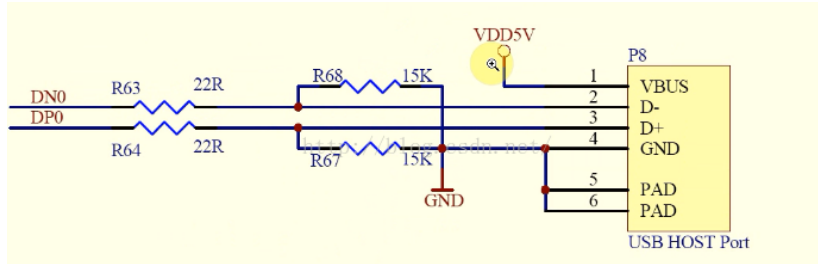
USB 总线驱动程序会发出某些命令想获取设备信息(描述符),

USB 设备必须返回"描述符"给 PC

问 3. PC 机上接有非常多的 USB 设备, 怎么分辨它们?

USB 接口只有 4 条线：5V, GND, D-, D+

- 答 3. 每一个 USB 设备接入 PC 时, USB 总线驱动程序都会给它分配一个编号
接在 USB 总线上的每一个 USB 设备都有自己的编号(地址)
PC 机想访问某个 USB 设备时, 发出的命令都含有对应的编号(地址)



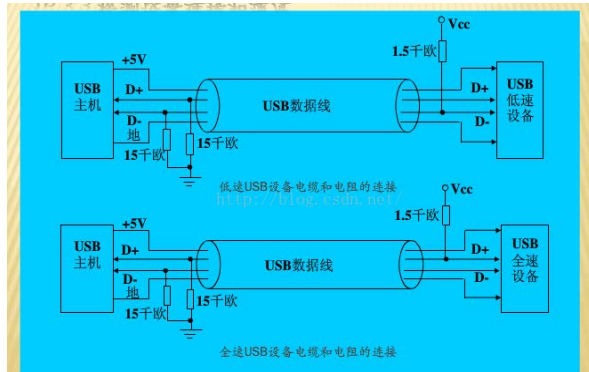
问 4. USB 设备刚接入 PC 时, 还没有编号; 那么 PC 怎么把"分配的编号"告诉它?

答 4. 新接入的 USB 设备的默认编号是 0, 在未分配新编号前, PC 使用 0 编号和它通信。

问 5. 为什么一接入 USB 设备, PC 机就能发现它?

答 5. PC 的 USB 口内部, D-和 D+接有 15K 的下拉电阻, 未接 USB 设备时为低电平

USB 设备的 USB 口内部, D-或 D+接有 1.5K 的上拉电阻; 它一接入 PC, 就会把 PC USB 口的 D-或 D+拉高, 从硬件的角度(电平由低变高)通知 PC 有新设备接入。下面的全速设备和低速设备区别在于上拉电阻接在 D+还是 D-端。在 usb device 端, 这时就有高速低速设备的区别了, 高速设备: d+ 接一个 1.5kohm 的上拉电阻, d-不接; 低速设备则相反。



其他概念:

1. USB 是主从结构的(连续不断地查询)

所有的 USB 传输, 都是从 USB 主机这方发起; USB 设备没有"主动"通知 USB 主机的能力。

例子: USB 鼠标滑动一下立刻产生数据, 但是它没有能力通知 PC 机来读数据, 只能被动地等得 PC 机来读。

2. USB 的传输类型:

- a. 控制传输: 可靠, 时间有保证, 比如: USB 设备的识别过程
- b. 批量传输: 可靠, 时间没有保证, 比如: U 盘
- c. 中断传输: 可靠, 实时, 比如: USB 鼠标 (只是借助中断这个概念, 没有中断能力, 使用查询方式来实现实时性)
- d. 实时传输: 不可靠, 实时, 比如: USB 摄像头

3. USB 传输的对象: 端点(endpoint)

我们说"读 U 盘"、"写 U 盘", 可以细化为: 把数据写到 U 盘的端点 1, 从 U 盘的端点 2 里读出数据

除了端点 0 外, 每一个端点只支持一个方向的数据传输

端点 0 用于控制传输, 既能输出也能输入

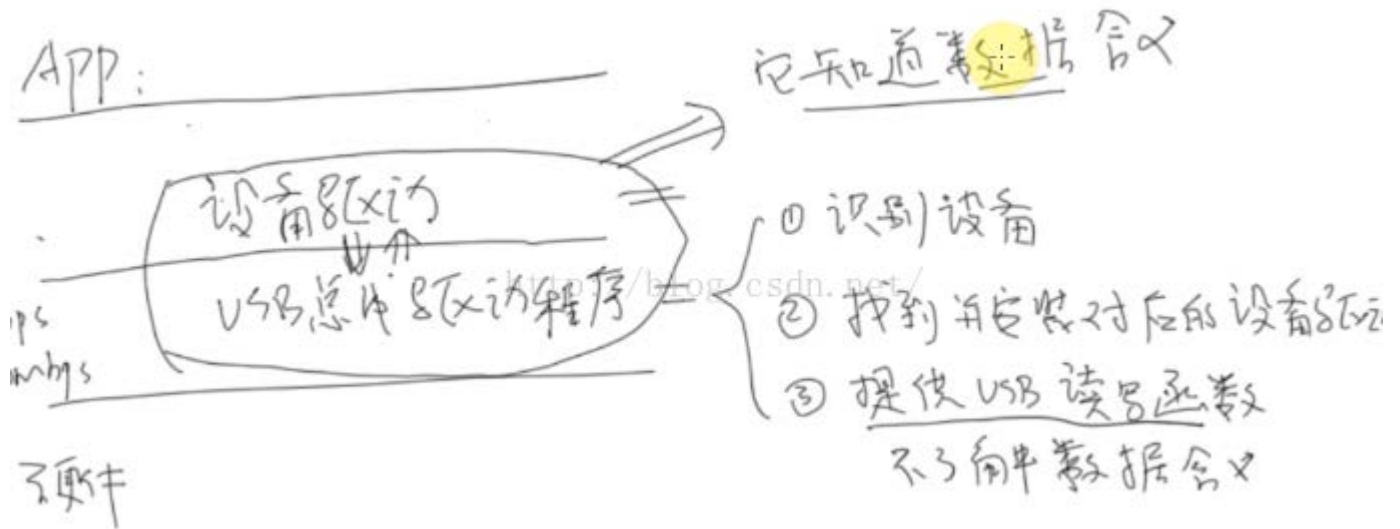
4. 每一个端点都有传输类型, 传输方向

5. 术语里、程序里说的输入(IN)、输出(OUT) "都是" 基于 USB 主机的立场说的。

比如鼠标的数据是从鼠标传到 PC 机, 对应的端点称为"输入端点"

6. USB 总线驱动程序的作用

- a. 识别 USB 设备
- b. 查找并安装对应的设备驱动程序
- c. 提供 USB 读写函数 (只是收发送信, 不知道信的内容)



USB 总线驱动程序已经有了读写函数，能不能让应用程序直接使用这些函数来访问（跨过设备驱动程序）。USB 总线驱动程序里面有 `libusb`（里面封装了接口函数）

USB 驱动程序框架：

```

app:
-----
    USB 设备驱动程序    // 知道数据含义
内核 -----
    USB 总线驱动程序    // 1. 识别, 2. 找到匹配的设备驱动, 3. 提供 USB 读写函数（它不知道数据含义）
-----
    USB 主机控制器
    UHCI OHCI EHCI
硬件 -----
    USB 设备
  
```

UHCI: intel 阵营规范（主硬件）， 低速(1.5Mbps)/全速(12Mbps)

OHCI: microsoft 阵营规范（主软件） 低速/全速

EHCI: 高速(480Mbps)

USB 主机控制器有 3 种规格：

OHCI (Open Host Controller Interface)、

UHCI (Universal HostController Interface)

EHCI (Enhanced Host Controller Interface)。

OHCI 驱动程序用来为非 PC 系统上以及带有 SiS 和 Ali 芯片组的 PC 主板上的 USB 芯片提供支持。

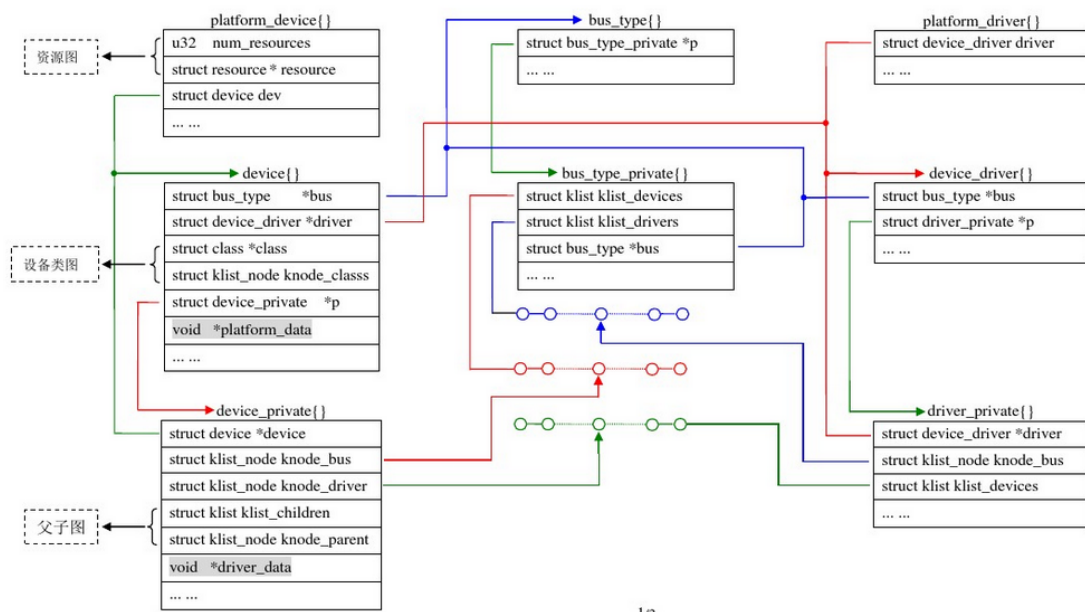
UHCI 驱动程序多用来为大多数其他 PC 主板（包括 Intel 和 Via）的 USB 芯片提供支持。

EHCI 由 USB 2.0 规范所提出，

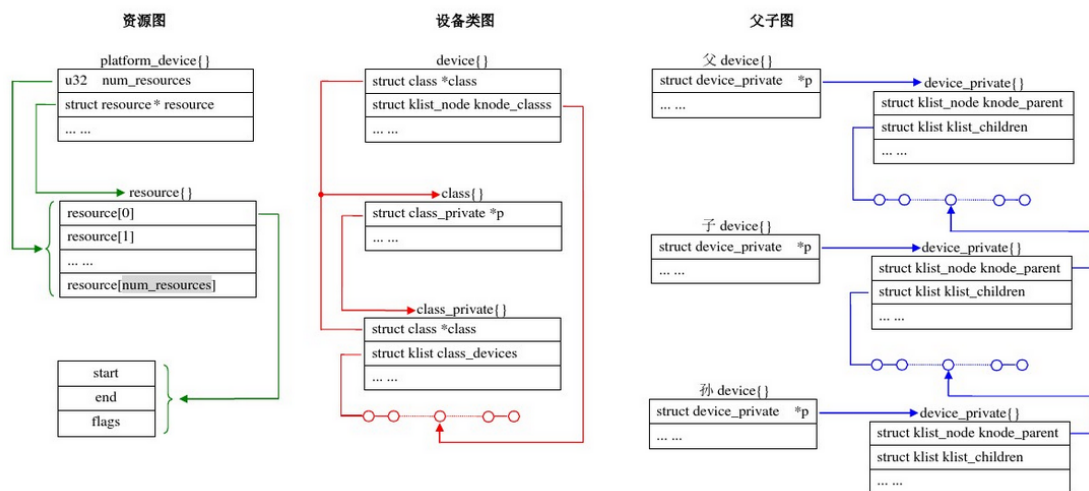
它兼容于 OHCI 和 UHCI。UHCI 的硬件线路比 OHCI 简单，所以成本较低，但需要较复杂的驱动程序，CPU 负荷稍重。本节将重点介绍嵌入式系统中常用的 OHCI 主机控制器驱动。

驱动模型数据结构关系图

2012 年 12 月 13 日

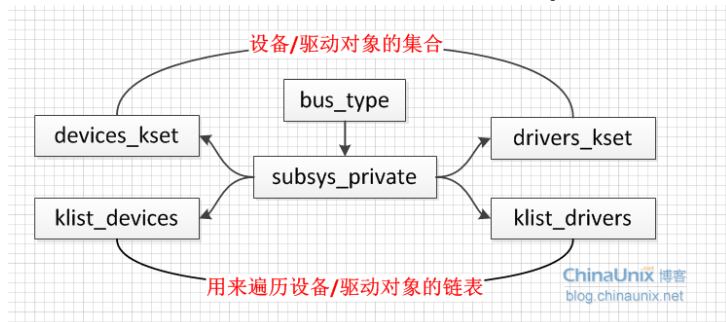


1/3



Linux 设备驱动模型中，按照层次的组织结构，抽象成总线 (struct bus_type)，设备 (struct device)，驱动 (struct device_driver) 的层次组织形式，这是最原始的抽象结构，在此基础上，根据不同类型的总线/设备/驱动，有形成了更高层次的组织结构，如 virtio 总线 (struct bus_type virtio_bus)，virtio 设备 (struct virtio_device)，virtio 驱动 (struct virtio_driver) 等。

不同的抽象层次构成一颗网状的树，linux 内核通过驱动模型：kobject，kset 来组织树的结构。

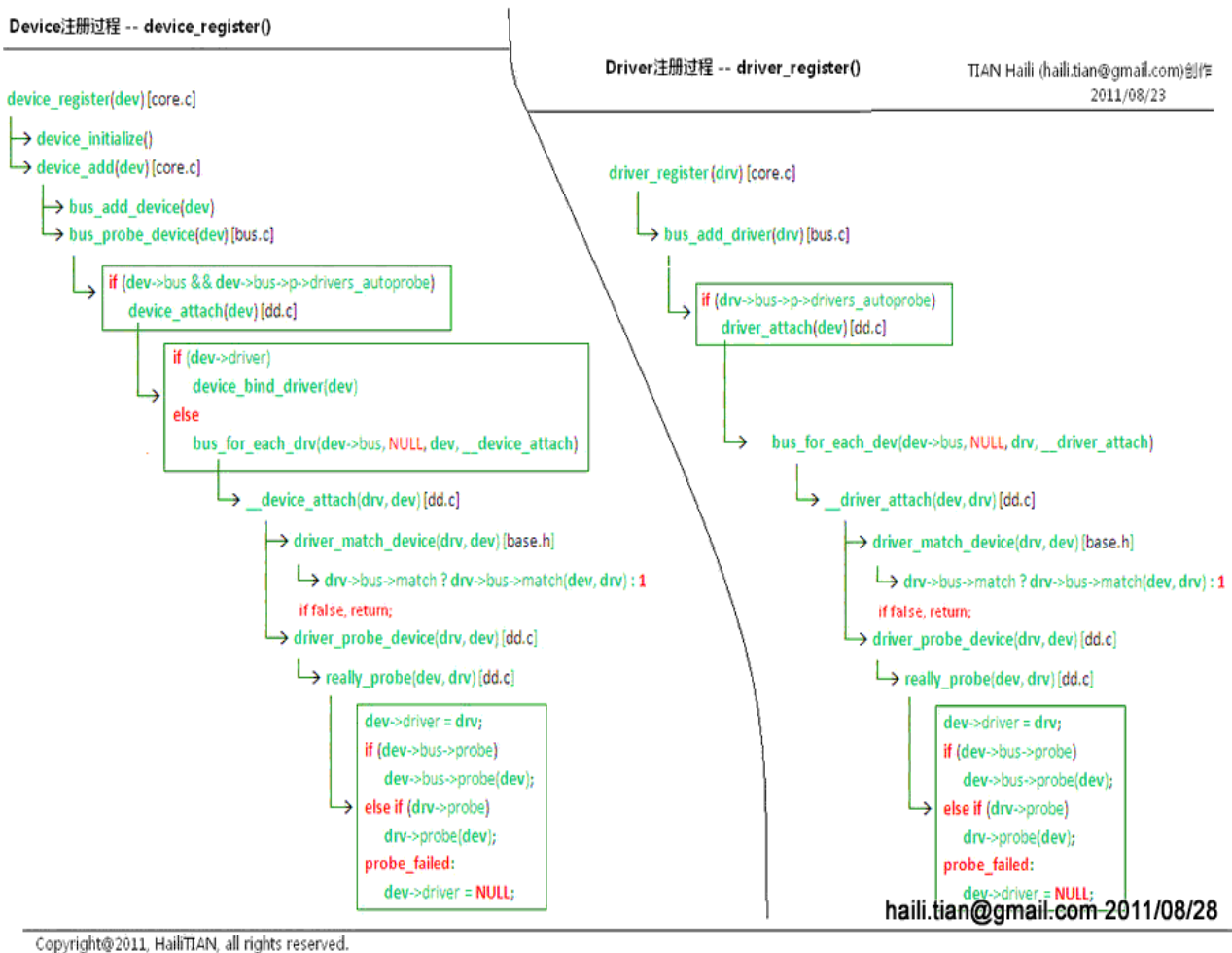


上图说明了总线通过两个数据结构：devices_kset 和 driver_kset 来管理注册在此总线上的所有的设备和驱动，为了方便遍历，linux 增加了 klist_devices 和 klist_drivers 用来实现设备和驱动的遍历。

理解 linux 驱动模型，最重要的是理解设备与驱动的匹配过程，即在何时，驱动与设备如何实现匹配？

总结起来，设备的注册时机为调用 `device_register()`，将 `devices` 注册到总线的 `devices_kset` 上，同理，驱动的注册为调用 `driver_register` 将驱动注册到总线的 `drivers_kset` 上。

下图对照说明了 Device 和 Driver 的注册过程。



也就是说总线平台驱动，不管怎么注册，最终分别注册到内核中的都是 `device` 和 `device_driver` 结构体，我们所谓的 `platform_device` 和 `platform_driver` 都是在之前两个结构体上有封装了一次而已

设备与驱动的匹配主要有以下几个点：

- (1) 如果总线的 `match` 函数非空，调用总线的 `match` 函数
- (2) 如果总线的 `probe` 函数非空，调用总线的 `probe` 函数，然后会在总线的 `probe` 函数中调用驱动的 `probe` 函数
- (3) 如果总线的 `probe` 函数为空，则直接调用驱动的 `probe` 函数

另外，`bus_for_each_drv()` 是对 BUS 上所有的 Driver 都进行 `__device_attach()` 操作；同样的，`bus_for_each_dev()` 是对 BUS 上所有的 Device 都进行 `__driver_attach()` 操作。

那么，这些函数的执行时机在哪呢？

只要有 `device` 或 `device_driver` 被注册到总线上，都会执行上面的过程。

这一节是介绍 USB 的总线驱动程序的，根据 USB 驱动程序的分层，还有设备驱动程序层，在下一节讲到

A:USB 总线驱动程序的作用

1. 识别 USB 设备

1.1 分配地址

1.2 并告诉 USB 设备(set address)

1.3 发出命令获取描述符

描述符的信息可以在 `include/linux/usb/ch9.h` 看到 (Ch9 是指 USB 规范的第九章)

2. 查找并安装对应的设备驱动程序

3. 提供 USB 读写函数

B:分析代码

我们知道我们的 USB 设备驱动的 probe 函数中的一个参数是 interface 结构, 因此一般来说, 一个 USB 设备中的任何一个接口都应该有对应的一个驱动程序, 当然也有例外(如 cdc-acm).

```
.4:
.5: static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
.6: {
.7:     struct usb_device *dev = interface_to_usbdev(intf);
```

我们知道 USB 设备都是通过插入上层 HUB 的一个 Port 来连入系统并进而被系统发现的, 当 USB 设备插入一个 HUB 时, 该 HUB 的那个 port 的状态就会改变, 从而系统就会知道这个改变, 此时会调用 `hub_port_connect_change()`

把 USB 设备接到开发板上, 看输出信息:

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: configuration #1 chosen from 1 choice
scsi0 : SCSI emulation for USB Mass Storage devices
scsi 0:0:0:0: Direct-Access    HTC      Android Phone    0100 PQ: 0 ANSI: 2
sd 0:0:0:0: [sda] Attached SCSI removable disk
```

拔掉

```
usb 1-1: USB disconnect, address 2
```

再接上:

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
scsi1 : SCSI emulation for USB Mass Storage devices
scsi 1:0:0:0: Direct-Access    HTC      Android Phone    0100 PQ: 0 ANSI: 2
sd 1:0:0:0: [sda] Attached SCSI removable disk
```

在内核目录下搜:

```
grep "USB device using" * -nr
drivers/usb/core/hub.c:2186:         "%s %s speed %sUSB device using %s and address %d\n",
```



根据打印信息查看信息所在的函数 `hub_port_init()`, 看这个函数的被调用情况

`hub_irq` //这个中断是主机控制器驱动程序注册的中断, 当接上 USB 设备后, USB 设备硬件上的手脚使主机控制器的 D+或 D-引脚发生变化。主机控制器硬件上感知有 USB 控制器接入, 就会产生某个中断, 调用中断函数 `hub_irq`。

```
hub_irq
    kick_khubd
        hub_thread
            hub_events
                hub_port_connect_change

                udev = usb_alloc_dev(hdev, hdev->bus, port1);
                dev->dev.bus = &usb_bus_type;

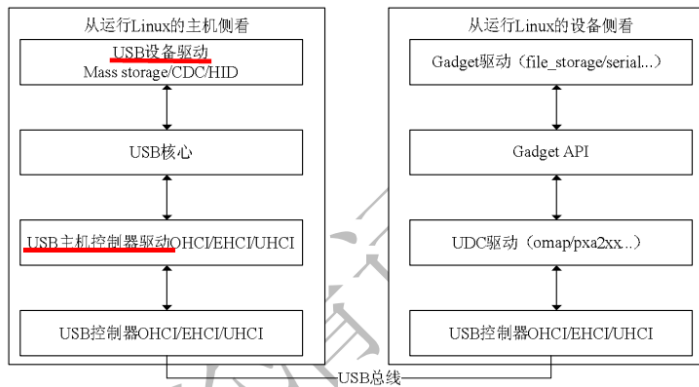
                choose_address(udev); // 给新设备分配编号(地址)

                hub_port_init // usb 1-1: new full speed USB device using s3c2410-ohci and address 3
                hub_set_address // 把编号(地址)告诉USB设备
                usb_get_device_descriptor(udev, 8); // 获取设备描述符
                retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);

                usb_new_device(udev)
                err = usb_get_configuration(udev); // 把所有的描述符都读出来, 并解析
                usb_parse_configuration

                device_add // 把device放入usb_bus_type的dev链表,
                // 从usb_bus_type的driver链表里取出usb_driver,
                // 把usb_interface和usb_driver的id_table比较
                // 如果能匹配, 调用usb_driver的probe
```

再次强调一次, USB 的驱动分为 USB 总线驱动程序和 USB 设备程序层, 我们这一节讲的是 USB 总线驱动, 至于他要完成的事就如开始描述的几点, 所以我们这里就是要知道 USB 总线驱动到底在内核中哪里实现了他的功能, 最后发现, 在上面的函数流程中就实现了



```

hub_irq
kick_khubd
hub_thread
hub_events
hub_port_connect_change

```

```

udev = usb_alloc_dev(hdev, hdev->bus, port1);
dev->dev.bus = &usb_bus_type; //总线设备驱动模型

```

```

choose_address(udev); // 给新设备分配编号(地址)

```

```

hub_port_init // usb 1-1: new full speed USB device using s3c2410-ohci and address 3

```

```

hub_set_address // 把编号(地址)告诉 USB 设备

```

```

usb_get_device_descriptor(udev, 8); // 获取设备描述符, 这里获得 8 个字节是因为还不知道端点 0 一次性能传输多少数据, 从第 8 个字节可以知道端点 0 一次性传输的包大小是多少。

```

```

struct usb_device_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;

    __le16 bcdUSB;
    __u8 bDeviceClass;
    __u8 bDeviceSubClass;
    __u8 bDeviceProtocol;
    __u8 bMaxPacketSize0;
    __le16 idVendor;
    __le16 idProduct;
    __le16 bcdDevice;
    __u8 iManufacturer;
    __u8 iProduct;
    __u8 iSerialNumber;
    __u8 bNumConfigurations;
} __attribute__((packed));

```

```

retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);

```

```

usb_new_device(udev)

```

```

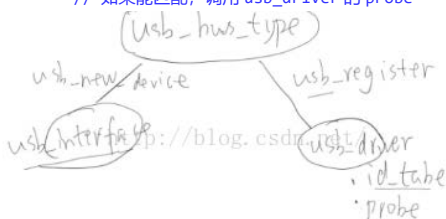
err = usb_get_configuration(udev); // 把所有的描述符都读出来, 并解析
usb_parse_configuration

```

```

device_add // 把 device 放入 usb_bus_type 的 dev 链表,
            // 从 usb_bus_type 的 driver 链表里取出 usb_driver,
            // 把 usb_interface 和 usb_driver 的 id_table 比较
            // 如果能匹配, 调用 usb_driver 的 probe

```



上面分析完了我们 USB 总线驱动程序是如何在代码中实现:

1. 识别 USB 设备

1.1 分配地址

1.2 并告诉 USB 设备(set address)

1.3 发出命令获取描述符

描述符的信息可以在 include/linux/usb/Ch9.h 看到

2. 查找并安装对应的设备驱动程序

3. 提供 USB 读写函数

详解查找并安装对应的设备驱动程序过程:

对于第二步中查找对应的设备驱动程序部分我们只知道是 device_add,那么具体的流程又是怎样的呢?

整个流程大概就是这样:

```

status = usb_new_device(udev);

```

```

err = usb_get_configuration(udev); // 把所有的描述符都读出来, 并解析
result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, bigbuffer, length);
err = device_add(&udev->dev); //udev 类型是 usb_device 结构体, 它里面有个 device 结构体成员 dev
bus_add_device //这个设备添加到相应 bus 的设备列表中去
bus_attach_device(dev); //这个函数就是用来为设备找到相应的设备驱动程序的(通过调用 device_attach()实现). 对于 USB 设备来说第一次调用
bus_attach_device()时的参数 dev 代表的是整个 usb 设备(以后 usb 设备中的 interface 也会作为设备调用这个函数).
device_attach(dev);
ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach); //总线上已注册的所有驱动中找出匹配的驱动程序

```

```

/*
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void *data, int (*fn)(struct device_driver *, void *))
{
while((drv = next_driver(&i)) && !error)
error = fn(drv, data); //返回 0 将继续搜索, 返回错误值将停止搜索.
...
}
该函数遍历 bus 上的所有驱动程序, 并为每个驱动调用 fn() 来查看是否匹配. 这里的 fn 就是 __device_attach.
*/

```

也就是说 bus_for_each_drv 函数会一直从总线上遍历已经注册的 usb 的驱动, 来查找匹配的 usb 驱动
那么怎么才知道这个驱动是不是匹配的呢? 我们从总线上遍历的时候, 找到了一个驱动, 就去调用这个驱动的 match 函数, 看是否匹配
具体的流程如下:

所以接下来我们要分析的是 __device_attach 函数

```

static int __device_attach(struct device_driver *drv, void *data)
{
driver_probe_device(drv, dev);
if(drv->bus->match && !drv->bus_match(dev, drv))
ret = really_probe(dev, drv);
}

```

也就是说对于每个遍历到的 usb 驱动我们都要调用对应 drv 侧的 match 函数来进行匹配

drv->bus_match(dev, drv) 对于这里的 dev 和 drv, dev 我们好理解, 是从一开始 usb_new_device 这个函数就已经初始化了, 但是这个 drv 参数

我们是通过遍历 drv 链表或得到的, 他的对应的 match 函数到底是什么呢?

那么我们的 drv 侧的 match 函数是在哪里被定义的呢?

对于 usb 驱动来说, 我们通过 usb_register() 来注册我们的驱动程序, 这个函数会为我们的驱动程序对象(usb_driver)中的 bus 指定为 usb_bus_type:

```

usb_register
usb_register_driver //在这里我们指定了总线类型为 usb_bus_type
new_driver->drvwrap.driver.bus = &usb_bus_type;

```

这个 usb_bus_type 在内核中已经定义好了的:

```

struct bus_type usb_bus_type = {
.name = "usb",
.match = usb_device_match,
.uevent = usb_uevent,
.suspend = usb_suspend,
.resume = usb_resume,
};

```

也就是说我们注册对应的 drv 侧的驱动的时候就已经规定好了 match 函数为 usb_device_match

所以回到之前的 __device_attach 函数分析部分:

```

static int __device_attach(struct device_driver *drv, void *data)
{
driver_probe_device(drv, dev);
if(drv->bus->match && !drv->bus_match(dev, drv))
ret = really_probe(dev, drv);
}

```

我们先忽略掉 really_probe 这个部分, 待会分析

drv->bus_match(dev, drv) 实际上就是调用 usb_device_match

继续分析:

```

usb_device_match
if(is_usb_device(dev)) /*dev 代表整个 usb 设备*/
else /*dev 代表一个 usb 设备 interface*/
{
usb_match_id();
usb_match_dynamic_id();
}

```

usb_device_match 这个函数只是做一些粗略的匹配, 如果匹配成功则返回 1, 然后由 really_probe 来做进一步的匹配, 如果匹配失败则返回 0, 并且 really_probe 也不会再执行.
这个函数的调用保证了 dev, drv 要么都是设备级别的(即 dev 代表 usb 设备, drv 代表 usb 设备驱动), 要么都是接口级别的(即 dev 代表 usb 设备的一个 interface, drv 代表 usb 接口驱动).*/

如果 usb_device_match 这个函数匹配成功, 我们继续回到前面, 会接着调用 really_probe 来进一步进行匹配

```

static int really_probe(struct device *dev, struct device_driver *drv)
{
dev->driver = drv; //先赋值, 以后的 probe 过程中会用到
else if(drv->probe)
ret = drv->probe(dev);
probe_failed:
dev->driver = NULL; //probe 失败, 重设它
}

```

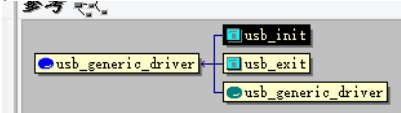
上面我们可以看到通过 really_probe 匹配成功后就会调用对应的驱动的 probe 函数了

对于 usb 来说这个函数的调用有 2 种分支, 1: dev, drv 代表的是设备级别的, 2 dev, drv 代表的是接口级别的. 其他情况组合在 usb_device_match 中被过滤掉了,

分支 1: dev, drv 代表的是设备级别:

此时的 drv 肯定是 `usb_generic_driver`。因为在当前的 usb 系统中只有这个 driver 是代表整个设备的驱动,它是在 `usb_init` 中被注册的,而我们通常写的 usb 驱动都是代表一个 interface 的。

```
struct usb_device_driver usb_generic_driver = {
    .probe = generic_probe,
}
```



因此,此时的 `drv->probe` 将调用 `generic_probe()`。

```
static int generic_probe(struct usb_device *udev)
{
    c = choose_configuration(dev);
    if(c >= 0) {
        err = usb_set_configuration(udev, c); //设置配置,并注册 interface.
    }
}
```

该函数为这个 usb 设备选择一个合适的配置,并注册这个配置下面的 interface。

```
int usb_set_configuration(struct usb_device *dev, int configuration)
{
    for(I = 0; I < nintf; i++) {
        struct usb_interface *intf = cp->interface[i];
        device_add(&intf->dev);
    }
}
```

从这里我们就可以知道,总的来说我们 usb 设备最终会注册一个 `usb_interface` 的结构体到 usb 总线链表中

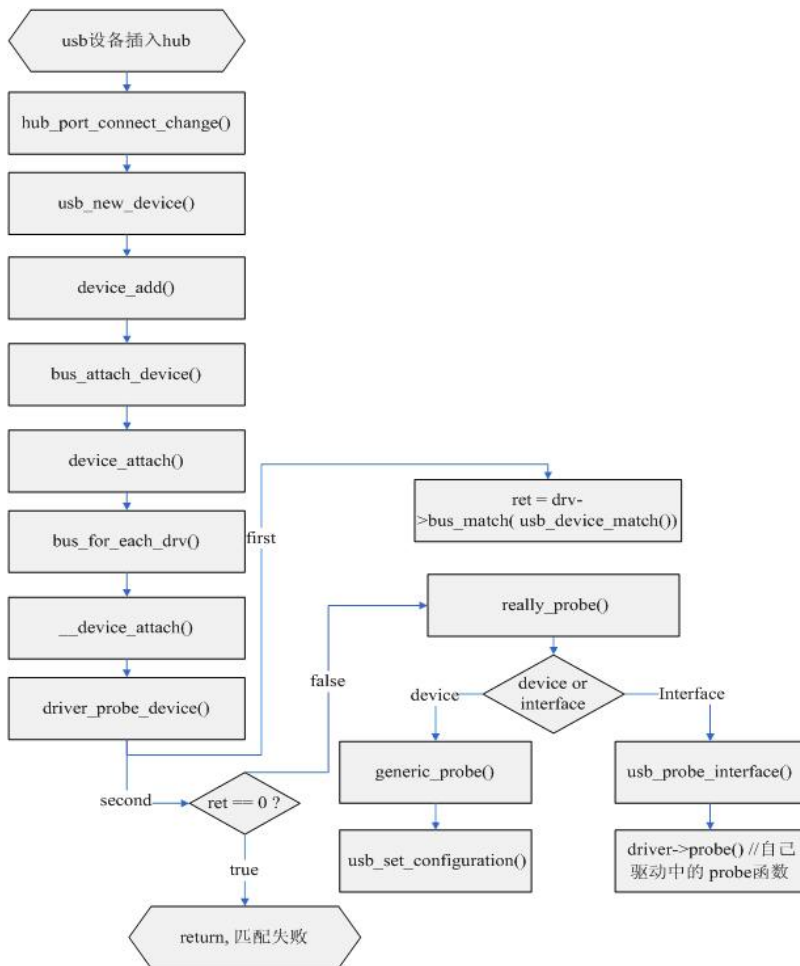
该函数比较重要,但我们只关心 probe 过程因此省掉了很多东西。它为当前配置下的每个 interface 调用 `device_add()` 函数,根据前面的分析可知,这个过程将会走到接下来我们要分析的分支 2。

分支 2: `dev,drv` 代表的是 interface 级别:

此时的 `dev` 代表着一个 interface,而 `drv` 就代表了我们自己的 usb 驱动。但是我们应当看到 `really_probe` 中的 `drv` 是 `device_driver` 类型,而我们写的 usb 驱动的自己定义的结构体类型一般是 `usb_driver`,因此这里的 probe 和我们自己写的 probe 显然不是同一个。实际上这里的 `drv` 是我们的驱动对象里内嵌的一个子对象(因为 linux 下所有的驱动都必须用 `device_driver` 来代表,)。那这个子对象的 probe 函数是在哪里赋值的呢?这就要看 `usb_register` 函数了,

跟踪这个函数我们可以看到这里的 probe 函数实际上是 `usb_probe_interface`(所有的 usb interface 驱动都是一样的)。

```
static int usb_probe_interface(struct device *dev)
{
    struct driver = to_usb_driver(dev->driver); //dev->driver 在 really_probe 中设置.
    error = driver->probe(intf, id); //这个就是我们自己写的 probe 函数了.
}
driver->probe(intf, id); 这就调用到我们自己写的代码里面了
```



USB 协议描述符:

我们之前分析 USB 协议的时候知道,USB 的描述符可以分为 5 种:

标准描述符有设备描述符 (Device Descriptor)、配置描述符 (Configuration Descriptor)、接口描述符 (Interface Descriptor)、端点描述符 (Endpoint Descriptor) 及字符串描述符 (String Descriptor)

那么在 linux 内核源码中我们是如何体现 USB 设备的层次和这几种描述符的关系呢?

- 设备描述符主要记录的信息有: 设备所使用的USB协议版本号、设备类型、端点0的最大包大小、厂商 ID (VID) 和产品ID (PID)、设备版本号、厂商字符串索引、产品字符串索引、设备序列号索引、可能的配置数等。
- 配置描述符主要记录的信息有: 配置所包含的接口数、配置的编号、供电方式、是否支持远程唤醒、电流需求等。
- 接口描述符主要记录的信息有: 接口的编号、接口的端点数、接口所使用的类、子类、协议等。
- 端点描述符主要记录的信息有: 端点号及方向、端点的传输类型、最大包长度、查询时间间隔等。
- 字符串描述符主要是提供一些方便人们阅读的信息, 它不是必需的。

1. 设备描述符

```

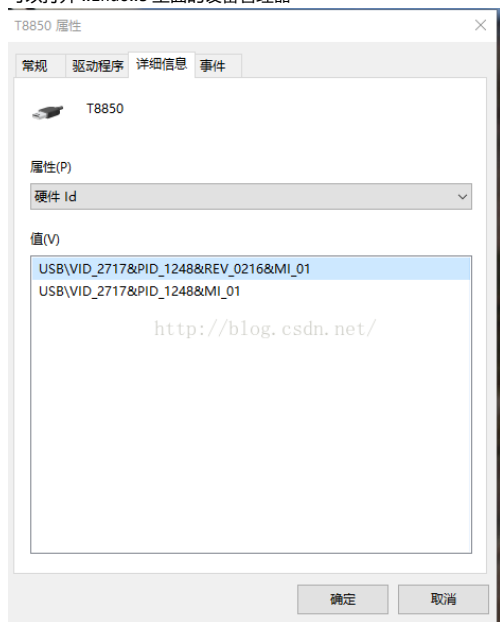
/* USB_DT_DEVICE: Device descriptor */
struct usb_device_descriptor {
    __u8  bLength;
    __u8  bDescriptorType; ①
    __le16 bcdUSB;
    __u8  bDeviceClass;
    __u8  bDeviceSubClass;
    __u8  bDeviceProtocol;
    __u8  bMaxPacketSize0;
    __le16 idVendor;
    __le16 idProduct;
    __le16 bcdDevice;
    __u8  iManufacturer;
    __u8  iProduct;
    __u8  iSerialNumber;
    __u8  bNumConfigurations;
} __attribute__((packed));
  
```

(1)这里 bMaxPacketSize0

最大包大小 0 是指端点 0, 每一个设备都有端点 0, 因为是通过端点 0 识别 USB 设备的, 把命令和地址发给端点 0, 从端点 0 读取到描述符信息

(2)设备描述符有厂家 ID、产品 ID 等

可以打开 windows 里面的设备管理器



(3) 设备描述符里有 bNumConfigurations

是指配置的个数，这个设备有多少种配置。因而有配置描述符。

2、配置描述符

配置描述符里有 bNumInterfaces; /*配置所支持的接口数*

3、接口描述符

接口描述符是指逻辑上的设备，如一个 USB 声卡，硬件上只有一个，但是逻辑上可能有两个功能，一个是录音，一个是播放。
写驱动程序的时候是给逻辑上的设备写的，所以一个 USB 硬件可能安装多个驱动程序。

4、端点描述符

(1) 结构体

```
struct usb_endpoint_descriptor {
3  _u8 bLength; /* 描述符长度 */
4  _u8 bDescriptorType; /* 描述符类型 */
5  _u8 bEndpointAddress; /* 端点地址: 0~3 位是端点号, 第 7 位是方向(0-OUT,1-IN) */
6  _u8 bmAttributes; /* 端点属性: bit[0:1] 的值为 00 表示控制, 为 01 表示同步, 为 02 表示批量, 为 03 表示中断 */
7
8  _le16 wMaxPacketSize; /* 本端点接收或发送的最大信息包的大小 */
9  _u8 bInterval; /* 轮询数据传送端点的时间间隔 */
10 /* 对于批量传送的端点以及控制传送的端点, 此域忽略 */
11 /* 对于同步传送的端点, 此域必须为 1 */
12 /* 对于中断传送的端点, 此域值的范围为 1~255 */
13 _u8 bRefresh;
14 _u8 bSynchAddress;
15 } _attribute_ ((packed));
```

(2)

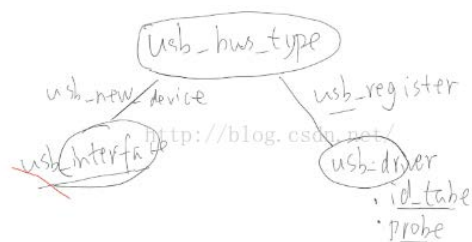
访问设备时：访问某个接口——接口表示逻辑设备
传输数据时：读写某个端点——端点是数据通道

5、字符串描述符

5、把描述符统统读出来后，根据这些信息来找到驱动程序

1、驱动编写分析

(1) usb 总线驱动程序在我们接入 USB 设备的时候会帮我们构造一个新的 usb_device, 注册到总线里面来。左边这一块已经帮我们做好了, 我们要做的是右边这一块。我们要构造一个 usb_driver 结构体, 然后注册进去。usb_driver 结构体里面的 id_table 表示能够支持哪一些设备, 里面的 probe 函数表示能够支持接入设备时会被调用。



(2) 目的

USB 鼠标用作键盘。左键按下时相当于字母 L, 右键按下时相当于字母 S, 鼠标中间的滑轮相当于回车。

(3) 输入子系统

在 probe 函数里面做以下几件事情

- 分配 input_dev 结构体
- 设置它能产生按键类事件
- 注册
- 硬件相关的操作 (对于 usb 鼠标来说, 使用 usb 总线驱动程序提供的读写函数来收发数据)

(4) usb 驱动程序编写

- 分配/设置 usb_driver 结构体
- - .id_table
 - .probe
 - .disconnect
- 注册

2、参考 usbmouse.c 驱动

(1) usb_driver 结构体

```
static struct usb_driver usb_mouse_driver = {
    .name         = "usbmouse",
    .probe        = usb_mouse_probe,
    .disconnect   = usb_mouse_disconnect,
    .id_table     = usb_mouse_id_table,
};
```

(2) id_table

```
static struct usb_device_id usb_mouse_id_table [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
        USB_INTERFACE_PROTOCOL_MOUSE) },
    { } /* Terminating entry */
};
```

```
#define USB_INTERFACE_INFO(cl,sc,pr) \
    .match_flags = USB_DEVICE_ID_MATCH_INT_INFO, .bInterfaceClass = (cl), \
    .bInterfaceSubClass = (sc), .bInterfaceProtocol = (pr)
```

展开上面的宏: USB_INTERFACE_INFO 的展开如下:

结合上面的可以展开为:

```
static struct usb_device_id usb_mouse_id_table [] = {
    { .match_flags = USB_DEVICE_ID_MATCH_INT_INFO,
      .bInterfaceClass = (c1),
      .bInterfaceSubClass = (sc),
      .bInterfaceProtocol = (pr) },
    { } /* Terminating entry */
};
```

分别带入 c1,sc,pr 的值,最终为:

```
static struct usb_device_id usb_mouse_id_table [] = {
    { .match_flags = USB_DEVICE_ID_MATCH_INT_INFO,
      .bInterfaceClass = (USB_INTERFACE_CLASS_HID),
      .bInterfaceSubClass = (USB_INTERFACE_SUBCLASS_BOOT),
      .bInterfaceProtocol = (USB_INTERFACE_PROTOCOL_MOUSE) },
    { } /* Terminating entry */
};
```

(match_flag 是表示匹配 (设备描述符) 的哪一项) int 是 interface 的简写, 匹配接口的信息 (接口描述符), 只要接口描述符里面的接口的类是 c1, 子类是 sc, 协议是 pr, 就能够支持。只要 USB 设备的接口描述符里面的类是 HID 类, 子类是 BOOT, 协议是 MOUSE, 就能够支持。

(3) 接口描述符 (代表逻辑设备)

```
/* USB_DT_INTERFACE: Interface descriptor */
struct usb_interface_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;

    __u8 bInterfaceNumber;
    __u8 bAlternateSetting;
    __u8 bNumEndpoints;
    __u8 bInterfaceClass;
    __u8 bInterfaceSubClass;
    __u8 bInterfaceProtocol;
    __u8 iInterface;
} __attribute__((packed));
```

(4) (id_table) 想支持某一款设备, 厂家 ID 和设备 ID

```
#define USB_DEVICE(vend,prod) \
    .match_flags = USB_DEVICE_ID_MATCH_DEVICE, .idVendor = (vend), \
    .idProduct = (prod)
```

下面可以自己设备厂家 ID 和设备 ID

```
static struct usb_device_id usbmouse_as_key_id_table [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
      USB_INTERFACE_PROTOCOL_MOUSE) },
    { USB_DEVICE(0x1234, 0x5678) },
    { } /* Terminating entry */
};
```

```
#####
#####3
```

(5) probe 函数

```
static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
```

参数 usb_interface 结构体, 一个 usb 硬件设备有多个逻辑上的设备, 逻辑上的设备使用 usb_interface 结构体表示的,

参考 <http://blog.csdn.net/qingkongveyue/article/details/53363298>

```
Image arch/arm/boot/uImage is ready
book@book-desktop:/work/system/linux-2.6.22.6$ cp arch/arm/boot/uImage /work/nfs_root/uImage_nohid
```

编译驱动程序

拷贝别的文件的 Makefile, 然后修改

```
book@book-desktop:/work/system/linux-2.6.22.6$ cd /work/drivers_and_test/12th_usb/1th/
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ cp ../../11th_ts/8th/Makefile .
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ vi M"C
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ ls
Makefile  Module.symvers  usbmouse_as_key.c  usbmouse_as_key.ko  usbmouse_as_key.mod.c  usbmouse_as_key.mod.o  usbmouse_as_key.o
```

```
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ vi Makefile
```

```
KERN_DIR = /work/system/linux-2.6.22.6

all:
    make -C $(KERN_DIR) M=`pwd` modules

clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order

obj-m += usbmouse_as_key.o
```

用 make 命令进行编译

```
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/12th_usb/1th/usbmouse_as_key.o
  Building modules stage 2.
  MODPOST 1 modules
  LD [M] /work/drivers_and_test/12th_usb/1th/usbmouse_as_key.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
```

把生成的驱动程序拷贝到网络文件系统

```
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ cp usbmouse_as_key.ko /work/nfs_root/first_fs
```

使用新内核启动开发板 (reboot 命令重启)

```
Enter your selection: q
OpenJTAG> nfs 30000000 192.168.1.5:/work/nfs_root/uImage_nohid:bootm 30000000
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:00:3e:26:0a:5b
```

挂载网络文件系统, 安装驱动

```
Please press Enter to activate this console.
starting pid 758, tty '/dev/s3c2410_serial0': '/bin/sh'
#
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod usbmouse_as_key.ko
usbcore: registered new interface driver usbmouse_as_key_
```

接上 usb 鼠标 (found usbmouse 是在 probe 函数设置的)

```
#
# usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!
```

想插入 USB 时把厂家 ID 和设备 ID 打印出来, 因为厂家 ID 和设备 ID 都是在设备描述符里面的, 当我们接入 usb 设备后, usb 总线驱动程序把相关的描述符都读出来啦! 我们只要用就可以啦!

在 usb_device 结构体里面有设备描述符, 设备描述符里面就有厂家 ID 和设备 ID、版本号。

```
static int usbmouse_as_key_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *dev = interface_to_usbdev(intf);

    printk("found usbmouse!\n");

    printk("bcdUSB = %x\n", dev->descriptor.bcdUSB);
    printk("VID = 0x%x\n", dev->descriptor.idVendor);
    printk("PID = 0x%x\n", dev->descriptor.idProduct);

    return 0;
}
```

卸载原来的驱动装载新驱动

```
# rmmod usbmouse_as_key
usbcore: deregistering interface driver usbmouse_as_key_
# insmod usbmouse_as_key.ko
usbcore: registered new interface driver usbmouse_as_key_
# usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!
bcdUSB = 200
VID   = 0x46d
PID   = 0xc52f
```

编写驱动 usbmouse_as_key.c (参考 usbmouse.c)

17.4 USB 驱动程序之 USB 设备驱动程序 2 鼠标用作键盘

1. usbmouse.c

(1) probe 函数

```
static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *dev = interface_to_usbdev(intf);
    struct usb_host_interface *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    struct input_dev *input_dev;
    int pipe, maxp;
    int error = -ENOMEM;

    interface = intf->cur_altsetting;
    if (interface->desc.bNumEndpoints != 1)
        return -ENODEV;

    endpoint = &interface->endpoint[0].desc;
    if (!usb_endpoint_is_int_in(endpoint))
        return -ENODEV;

    pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
    maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
```

在这个 probe 函数后判断是不是一个鼠标，先得到 usb_host_interface 结构体，除了端点 0 外，端点个数如果不是 1，返回错误，表示不是自己能支持的设备。如果只有一个端点，放在 endpoint 这个数组里面。endpoint[0] 是表示除了端点 0 外的第一个端点。得到它的端点描述符。

如果不是输入（中断）类型端点，输入输出是站在主机角度说的，鼠标是输入设备（数据输入给主机）。返回错误。可以查看端点描述符里面的属性知道

```
__u8 bmAttributes; //
```

，这个属性表明其类型和方向

(2) 接口描述符

里面有端点的个数（除端点 0 外）（端点是 USB 传输对象）

```
/* USB_DT_INTERFACE: Interface descriptor */
struct usb_interface_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;

    __u8 bInterfaceNumber;
    __u8 bAlternateSetting;
    __u8 bNumEndpoints;
    __u8 bInterfaceClass;
    __u8 bInterfaceSubClass;
    __u8 bInterfaceProtocol;
    __u8 iInterface;
} __attribute__((packed));

#define USB_DT_INTERFACE_SIZE 9
```

2. 分配、设置、注册 usb_driver

(1) 分配设置 usb_driver 结构体

```
/* 1. 分配 / 设置 usb_driver */
static struct usb_driver usbmouse_as_key_driver = {
    .name = "usbmouse_as_key_",
    .probe = usbmouse_as_key_probe,
    .disconnect = usbmouse_as_key_disconnect,
    .id_table = usbmouse_as_key_id_table,
};
```

(2) 注册结构体

```
static int usbmouse_as_key_init(void)
{
    /* 2. 注册 */
    usb_register(&usbmouse_as_key_driver);
    return 0;
}
```

3. probe 函数

(1) 分配, 设置, 注册 input_dev 结构体

```
/* a. 分配 一个 input_dev */
uk_dev = input_allocate_device();

/* b. 设置 */
/* b.1 能产生哪类事件 */
/* 按键类事件和重复类事件: 按下一个字母不动重复打印 */
set_bit(EV_KEY, uk_dev->evbit);
set_bit(EV_REP, uk_dev->evbit);

/* b.2 能产生哪些事件, l、s、回车 */
set_bit(KEY_L, uk_dev->keybit);
set_bit(KEY_S, uk_dev->keybit);
set_bit(KEY_ENTER, uk_dev->keybit);

/* c. 注册 */
input_register_device(uk_dev);
```

(2) 硬件相关操作 (源、目的、长度)

源: 每个 USB 设备都有个地址,

```
usbcore: registered new interface driver usbmouse_as_key_
# usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!
```

宏 `usb_rcvintpipe` 包含有 usb 设备的地址和端点的地址。下面 `PIPE_INTERRUPT` 是中断类型端点, 源 pipe 是一个整数, 这个整数含有端点的类型和端点的方向。

```
#define usb_rcvintpipe(dev, endpoint) \
    ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)

#define usb_rcvintpipe(dev, endpoint) \
    ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
```

这里 `__create_pipe` 里面既含有设备地址也含有端点地址, 下面的 `devnum` 是 usb 设备的地址 (编号), `endpoint` 是端点的地址 (编号)

```
static inline unsigned int __create_pipe(struct usb_device *dev,
    unsigned int endpoint)
{
    return (dev->devnum << 8) | (endpoint << 15);
}

static inline unsigned int __create_pipe(struct usb_device *dev,
    unsigned int endpoint)
{
    return (dev->devnum << 8) | (endpoint << 15);
}
```


目的：缓冲区

分配缓冲区，返回一个虚拟地址，最后一个参数是物理地址，下面的长度 len 是端点描述符的最大包大小

```
/*目的：*/
usb_buf = usb_buffer_alloc(dev, len, GFP_ATOMIC, &usb_buf_phys);
```

```
void *usb_buffer_alloc(
    struct usb_device *dev,
    size_t size,
    gfp_t mem_flags,
    dma_addr_t *dma
)
{
    http://blog.csdn.net/
    if (!dev || !dev->bus)
        return NULL;
    return hcd_buffer_alloc(dev->bus, size, mem_flags, dma);
}
```

长度：在端点描述符里面有长度（最大包大小）

```
/*长度：*/
len = endpoint->wMaxPacketSize;
```

(3) 把三要素（源、目的、长度）用起来

分配 URB(usb request block)

```
/*分配 usb request block*/
uk_urb = usb_alloc_urb(0, GFP_KERNEL);
```

设置（中断类型）URB（填充源、目的、长度），还有中断函数

（鼠标是中断传输，usb 设备没有主动通知（打断）主机的能力，为了保证数据的实时性，主机控制器（外接鼠标）就不断地查询（查询的间隔由 bInterval 决定），查询得到数据后发出中断（中断 CPU），主机控制器有中断 cpu 的能力，usb 设备没有中断主机控制器的能力。）

主机控制器得到数据后（往某个内存里面写，需要告诉它物理地址），usb 总线驱动程序会调用 usbmouse_as_key_irq 函数，

```
/*使用 3 要素设置 urb*/
usb_fill_int_urb(uk_urb, dev, pipe, usb_buf, len, usbmouse_as_key_irq, NULL, endpoint->bInterval);
uk_urb->transfer_dma = usb_buf_phys;
uk_urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

使用 urb(提交 urb)

```
/*使用 URB*/
usb_submit_urb(uk_urb, GFP_KERNEL);
```

4. usbmouse_as_key_irq 函数

(1) 打印键值

```
static void usbmouse_as_key_irq(struct urb *urb)
{
    int i;
    static int cnt = 0;
    printk("data cnt %d: ", ++cnt);
    for (i = 0; i < len; i++)
    {
        printk("%02x ", usb_buf[i]);
    }
    printk("\n");

    /*重新提交 urb*/
    usb_submit_urb(uk_urb, GFP_KERNEL);
}
```

(2) 上报事件

```
static void usbmouse_as_key_irq(struct urb *urb)
{
    static unsigned char pre_val;
    #if 0
        int i;
        static int cnt = 0;
        printk("data cnt %d: ", ++cnt);
        for (i = 0; i < len; i++)
        {
            printk("%02x ", usb_buf[i]);
        }
        printk("\n");
    #endif
    /* USB鼠标数据含义
    * data[0]: bit0-左键, 1-按下, 0-松开
    *          bit1-右键, 1-按下, 0-松开
    *          bit2-中键, 1-按下, 0-松开
    */
    if ((pre_val & (1<<0)) != (usb_buf[0] & (1<<0)))
    {
        /*左键发生了变化*/
        input_event(uk_dev, EV_KEY, KEY_L, (usb_buf[0] & (1<<0)) ? 1 : 0);
        input_sync(uk_dev);
    }
}
```

上一次数据的 bit0 和这一次数据的 bit0 不相等，左键发生了变化

按键类事件 (S)。0 表示松开，1 表示按下

```
if ((pre_val & (1<<1)) != (usb_buf[0] & (1<<1)))
{
    /*右键发生了变化*/
    input_event(uk_dev, EV_KEY, KEY_S, (usb_buf[0] & (1<<1)) ? 1 : 0);
    input_sync(uk_dev);
}

if ((pre_val & (1<<2)) != (usb_buf[0] & (1<<2)))
{
    /*中键发生了变化*/
    input_event(uk_dev, EV_KEY, KEY_ENTER, (usb_buf[0] & (1<<2)) ? 1 : 0);
    input_sync(uk_dev);
}

pre_val = usb_buf[0];

/*重新提交 urb*/
usb_submit_urb(uk_urb, GFP_KERNEL);
} ? end usbmouse_as_key_irq ?
```

上面的 pre_val 用于保存当前值

5. disconnect 函数

这里 len 是长度，usb_buf 是虚拟地址，usb_buf_phys 是物理地址。

```
static void usbmouse_as_key_disconnect(struct usb_interface *intf)
{
    struct usb_device *dev = interface_to_usbdev(intf);

    // printk("disconnect usbmouse! \n");
    usb_kill_urb(uk_urb);
    usb_free_urb(uk_urb);
    usb_buffer_free(dev, len, usb_buf, usb_buf_phys);
    input_unregister_device(uk_dev);
    input_free_device(uk_dev);
}
```

6. 实验:

(1)

测试3th:

1. insmod usbmouse_as_key.ko
2. ls /dev/event*
3. 接上USB鼠标
4. ls /dev/event*
5. 操作鼠标观察数据

查看输出

第一个字表示按键

第一个字节的数据的 bit0 为 1 表示左键，为 0 表示松开。bit1 表示右键，bit2 表示中间的滑轮

第二个字表示 x 方向位移

第三个字表示 y 方向位移

第 4 个字表示滚轮

```
data cnt 305: 00 00 ff ff ff ff 00 00
data cnt 306: 00 00 00 00 ff ff 00 00
data cnt 307: 00 00 ff ff 00 00 00 00
data cnt 308: 00 00 00 00 ff ff 00 00
data cnt 309: 00 00 00 00 ff ff 00 00
data cnt 310: 00 00 00 00 ff ff 00 00
data cnt 311: 00 00 ff ff 00 00 00 00
data cnt 312: 00 00 00 00 ff ff 00 00
data cnt 313: 00 00 ff ff ff ff 00 00
data cnt 314: 00 00 00 00 ff ff 00 00
data cnt 315: 00 00 03 00 00 00 00 00
data cnt 316: 00 00 03 00 ff ff 00 00
data cnt 317: 00 00 02 00 ff ff 00 00
```

(2)

测试 4th:

1. insmod usbmouse_as_key.ko
2. ls /dev/event*
3. 接上 USB 鼠标
4. ls /dev/event*
5. cat /dev/tty1 然后按鼠标键

这里 tty1:对于不同的控制台终端 console,有一些设备特殊文件与之相关联: tty0,tty1

```
# ls /dev/event0
/dev/event0
# cat /dev/tty1
ls
ls
llllllllllllllll
```

6. hexdump /dev/event0

hexdump 参考 <http://blog.csdn.net/qingkongyeyue/article/details/53446839>

这里第一个字是秒,第二个字是微妙,第3个字(第一个字节是按键类,第2个字节表示哪一个按键),第4个字表示(按下或松开)

```
# hexdump /dev/event0
00000000 099a 0000 8eea 000e 0001 0026 0001 0000
00000010 099a 0000 8ef3 000e 0000 0000 0000 0000
00000020 099b 0000 db03 0000 0001 0026 0000 0000
00000030 099b 0000 db0b 0000 0000 0000 0000 0000
00000040 099b 0000 b2e4 0008 0001 001f 0001 0000
00000050 099b 0000 b2ed 0008 0000 0000 0000 0000
00000060 099b 0000 dd8a 000a 0001 001f 0000 0000
00000070 099b 0000 dd91 000a 0000 0000 0000 0000
```

第 20 课 NOR FLASH 驱动程序

20.1 NOR FLASH 原理及硬件操作

使用 UBOOT 体验 NOR FLASH 的操作(开发板设为 NOR 启动,进入 UBOOT)

先使用 OpenJTAG 烧写 UBOOT 到 NOR FLASH

1. 读数据

md.b 0

2. 读 ID

NOR 手册上:

往地址 555H 写 AAH
往地址 2AAH 写 55H
往地址 555H 写 90H
读 0 地址得到厂家 ID: C2H
读 1 地址得到设备 ID: 22DAH 或 225BH
退出读 ID 状态: 给任意地址写 F0H

由于 2440 的 A1 接到 NOR 的 A0, 所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?

往地址 AAAH 写 AAH mw.w aaa aa
往地址 554 写 55H mw.w 554 55
往地址 AAAH 写 90H mw.w aaa 90
读 0 地址得到厂家 ID: C2H md.w 0 1
读 2 地址得到设备 ID: 22DAH 或 225BH md.w 2 1
退出读 ID 状态: mw.w 0 f0

3. NOR 有两种规范, jedec, cfi(common flash interface)
 读取 CFI 信息

NOR 手册:
进入 CFI 模式 往 55H 写入 98H
读数据: 读 10H 得到 0051
 读 11H 得到 0052
 读 12H 得到 0059
 读 27H 得到容量

2440 的 A1 接到 NOR 的 A0, 所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?

进入 CFI 模式 往 AAH 写入 98H mw.w aa 98
读数据: 读 20H 得到 0051 md.w 20 1
 读 22H 得到 0052 md.w 22 1
 读 24H 得到 0059 md.w 24 1
 读 4EH 得到容量 md.w 4e 1
退出 CFI 模式 mw.w 0 f0

4. 写数据: 在地址 0x100000 写入 0x1234

md.w 100000 1 // 得到 ffff
mw.w 100000 1234
md.w 100000 1 // 还是 ffff

NOR 手册:
往地址 555H 写 AAH
往地址 2AAH 写 55H
往地址 555H 写 A0H
往地址 PA 写 PD

2440 的 A1 接到 NOR 的 A0, 所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址
UBOOT 怎么操作?

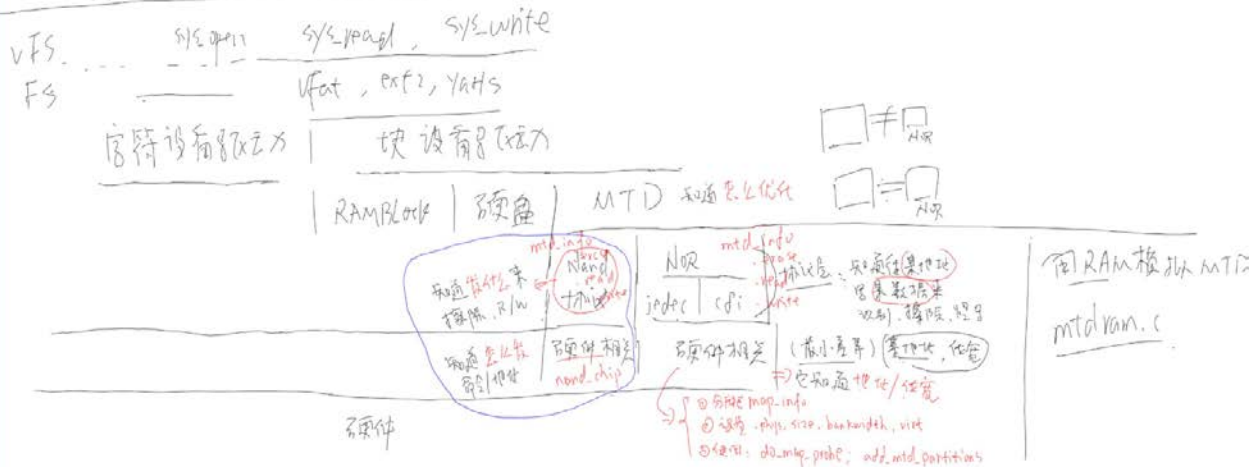
往地址 AAAH 写 AAH mw.w aaa aa
往地址 554H 写 55H mw.w 554 55
往地址 AAAH 写 A0H mw.w aaa a0
往地址 0x100000 写 1234h mw.w 100000 1234

20.2 NOR FLASH 驱动程序之框架

NOR FLASH 驱动程序框架:

Flash 驱动框架

App: open, read, write



上图为 flash 的驱动框架,包括了 nandflash 和 norflash

APP: 用户层调用 open, read 等函数

VFS: 进入虚拟文件系统调用对应的 sys_open, sys_read 等函数

如果是字符设备就直接进入字符设备驱动,如果是普通文件就进入实际的文件系统: ext2, yaffs, jffs2 等文件系统

对于普通文件:

通过实际的文件系统将对文件的操作转化为对块设备的操作: 块设备驱动程序

块设备又分为: 内存块设备 ramblock,

MTD (flash), 硬盘,

MTD 层: norflash | nandflash | 使用 RAM 模拟 MTD MTD 层是知道怎么优化

协议层: nandflash 协议层 | norflash 协议层 (分为 jedec 协议和 cfi 协议)

知道发什么来擦除, R/W | 知道往某地址写某些数据来识别, 擦除, 烧写

不管是 nor 还是 nand 这层都是通过一个 mtd_info 结构体

硬件相关代码层: 硬件相关 | 硬件相关 (给协议层提供上面所说的某些地址和某些数据信息)

nandflash 是 nand_chip 结构体 | nor 是 map_info 结构体 (提供基地址和位宽)

分配, 设置, 使用

20.3 NOR FLASH 驱动程序之编写代码

测试 1: 通过配置内核支持 NOR FLASH

我们目前用的 2.2.6 的内核里面已经有了 norflash 驱动, 只是还未开启支持功能, 我们现在就利用现有的驱动先体验下如何让当前内核支持 norflash

Q1: 如何知道是通过 menuconfig 来设置相应的参数的, 而不是直接改代码里面的长度和位宽

1. make menuconfig

-> Device Drivers

-> Memory Technology Device (MTD) support

-> Mapping drivers for chip access

<M> CFI Flash device in physical memory map

(0x0) Physical start address of flash mapping // 物理基地址

(0x1000000) Physical length of flash mapping // 长度

(2) Bank width in octets (NEW) // 位宽

2. make modules

cp drivers/mtd/maps/physmap.ko /work/nfs_root/first_fs

3. 启动开发板

ls /dev/mtd*

insmod physmap.ko

ls /dev/mtd*

cat /proc/mtd

如何自己写 norflash 驱动:

步骤 1:

搭框架:

```
static int s3c_nor_init(void)
{
}
static void s3c_nor_exit(void)
{
}
```

```

module_init(s3c_nor_init);
module_exit(s3c_nor_exit);

MODULE_LICENSE("GPL");

```

步骤2:完善 static int s3c_nor_init(void)

```

static int s3c_nor_init(void)
{
    /* 1. 分配map_info 结构体 */
    /* 2. 设置: 物理基地址(phys), 大小(size), 位宽(bankwidth), 虚拟基地址(virt) */
    /* 3. 使用: 调用 NOR FLASH 协议层提供的函数来识别 */
    /* 4. add_mtd_partitions */
}

static struct map_info *s3c_nor_map;
static struct mtd_info *s3c_nor_mtd;

static int s3c_nor_init(void)
{
    /* 1. 分配map_info结构体 */
    s3c_nor_map = kzalloc(sizeof(struct map_info), GFP_KERNEL);

    /* 2. 设置: 物理基地址(phys), 大小(size), 位宽(bankwidth), 虚拟基地址(virt) */
    s3c_nor_map->name = "s3c_nor";
    s3c_nor_map->phys = 0;
    s3c_nor_map->size = 0x1000000; /* >= NOR的真正大小 */
    s3c_nor_map->bankwidth = 2;
    s3c_nor_map->virt = ioremap(s3c_nor_map->phys, s3c_nor_map->size);

    simple_map_init(s3c_nor_map);

    /* 3. 使用: 调用NOR FLASH协议层提供的函数来识别 */
    printk("use cfi_probe\n");
    s3c_nor_mtd = do_map_probe("cfi_probe", s3c_nor_map);
    if (!s3c_nor_mtd)
    {
        printk("use jedec_probe\n");
        s3c_nor_mtd = do_map_probe("jedec_probe", s3c_nor_map);
    }

    if (!s3c_nor_mtd)
    {
        iounmap(s3c_nor_map->virt);
        kfree(s3c_nor_map);
        return -EIO;
    }

    /* 4. add_mtd_partitions */

    return 0;
}

```

步骤3:完善 static void s3c_nor_exit(void)

```

static void s3c_nor_exit(void)
{
    iounmap(s3c_nor_map->virt);
    kfree(s3c_nor_map);
}

```

第 21 课 网卡驱动程序

21.1 网卡驱动程序之框架

我们这里说的是网卡驱动程序, 不是网络驱动程序, 网络有七层, 我们写的只是最底层的東西, 网络这么多层, 但是最终你还是要操作硬件啊
所以里面肯定有个硬件相关层, 我们要写的就是硬件相关的驱动程序这一小块。也就是最底层
之前我么字符设备驱动的应用层的时候, 经常好 open 一个设备, 而网卡你不需要打开什么设备, 你只需要 socket 编程就行了
怎么写网卡驱动呢? 和块设备差不多的步骤

1、分配某个结构体

2、设置

3、注册

这有几个问题:

1: 分配什么结构体呢?: 通过参考别人的网卡例程知道是分配一个 net_device 结构体

2: 设置结构体哪些参数呢?

由于网卡主要实现的功能是收发数据, 所以主要要实现这两个函数

2.1 发包函数: hard_start_xmit

2.2 收到数据时(在中断处理函数里)用 netif_rx 上报数据

3: 用什么函数注册: register_netdev(注意不要用 register_netdevice, 老师就是用这个然后报错)

总结如下图:

```

app:  socket
-----
-----  若干层网络协议--纯软件
-----
hard_start_xmit||  /\
                \/  ||  netif_rx   sk_buff
-----
硬件相关的驱动程序(要提供hard_start_xmit, 有数据时要用netif_rx上报)
-----
硬件

```

怎么写网卡驱动程序?

1. 分配一个net_device结构体
2. 设置:
 - 2.1 发包函数: hard_start_xmit
 - 2.2 收到数据时(在中断处理函数里)用netif_rx上报数据
 - 2.3 其他设置
3. 注册: register_netdevice

<http://blog.csdn.net/wangshuchangfrank>

修正一下: 上面第三步应该是: register_netdev

上图分析:

- 1:应用层只要调用 socket 接口就行,而不用打开某个设备
- 2:网络协议层有 7 层,而我们要写的是和硬件相关的网卡驱动
- 3:上图看到的箭头是网卡驱动上层通过 hard_start_xmit 来向网卡驱动层发数据,实际上的意思,上次通过调用网卡驱动层里面实现的 hard_start_xmit 来发数据,通过调用网卡驱动层里面实现的 netif_rx 来收数据.不要被箭头迷惑.

1:实现一个虚拟网卡

说到这块, 写一个网卡驱动很简单, 我们先写一个虚拟的网卡

参考/driver/net/cs89x0.c

框架如图

```

static struct net_device *vnet_dev;

static int virtnet_init(void)
{
    /*1.分配一个net_device结构体*/
    /*2.设置*/
    /*3.注册*/
    return 0;
}

static void vitrnet_exit(void)
{
}

module_init(virtnet_init);
module_exit(vitrnet_exit);
MODULE_LICENSE("GPL");

```

我们来看一下范例里面分配的

```
struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
```

但是我们不想用 alloc_etherdev 这个宏, 我们进去看一下这个宏

因为通过追踪这个函数,发现这个函数分配的网卡的名字都是以 eth 开头的

```

00331: struct net_device *alloc_etherdev(int sizeof_priv)
00332: {
00333:     return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
00334: }
00335: EXPORT_SYMBOL(alloc_etherdev);

```

这里的名字用的是 eth0 啊 eth1 这种, 我不想用这个名字, 我想换一个名字, 所以我们直接用 alloc_netdev 这个函数来做

这个函数的第一个参数是私有数据的长度,在 linux 中一个结构体里面可以存放私有数据,我们这里设置为 0

最终最简单的网卡驱动如下:

```

static struct net_device *vnet_dev;

static int virt_net_init(void)
{
    /* 1. 分配一个net_device结构体 */
    vnet_dev = alloc_netdev(0, "vnet%d", ether_setup); /* alloc_etherdev */

    /* 2. 设置 */

    /* 3. 注册 */
    //register_netdevice(vnet_dev);
    register_netdev(vnet_dev);

    return 0;
}

static void virt_net_exit(void)
{
    unregister_netdev(vnet_dev);
    free_netdev(vnet_dev);
}

module_init(virt_net_init);
module_exit(virt_net_exit);

MODULE_AUTHOR("thisway.diy@163.com,17653039@qq.com");
MODULE_LICENSE("GPL");

```

测试 1th:

1. insmod virt_net.ko
 2. ifconfig vnet0 3.3.3.3
- ifconfig // 查看成功显示网卡名称

3. ping 3.3.3.3 // 成功 //说明 ping 本地 ip 并未经过网卡驱动
ping 3.3.3.4 // 死机 //由于未提供 hard_start_xmit 发送函数

完善代码,提供一个发送函数,里面就只打印好了,什么都不做,对比 1th 代码只改变了如下部分:

```
static struct net_device *vnet_dev;

static int virt_net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
    static int cnt = 0;
    printk("virt_net_send_packet cnt = %d\n", ++cnt);
    return 0;
}

static int virt_net_init(void)
{
    /* 1. 分配一个net_device结构体 */
    vnet_dev = alloc_netdev(0, "vnet%d", ether_setup); /* alloc_etherdev */

    /* 2. 设置 */
    vnet_dev->hard_start_xmit = virt_net_send_packet;

    /* 3. 注册 */
    register_netdevice(vnet_dev);
    register_netdev(vnet_dev);

    return 0;
}

static void virt_net_exit(void)
{
    unregister_netdev(vnet_dev);
    free_netdev(vnet_dev);
}

module_init(virt_net_init);
module_exit(virt_net_exit);

MODULE_AUTHOR("thisway.diy@163.com,17653039@qq.com");
MODULE_LICENSE("GPL");
```

继续测试:

测试 2th:

1. insmod virt_net.ko
2. ifconfig vnet0 3.3.3.3
ifconfig // 查看成功显示网卡名称
3. ping 3.3.3.3 // 成功 //说明 ping 本地 ip 并未经过网卡驱动
ping 3.3.3.4 // 死机 //由于未提供 hard_start_xmit 发送函数

21.2 网卡驱动程序之编写虚拟网卡

第 22 课 网卡驱动程序

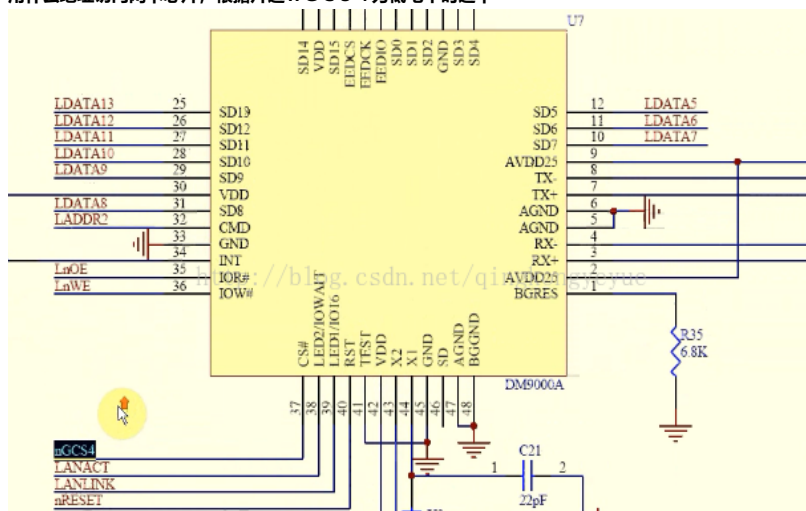
22.1 移植 DM9000C 驱动程序之确定差异性

内核自带的 DM9000 驱动程序不能用在 DM9000C 上面

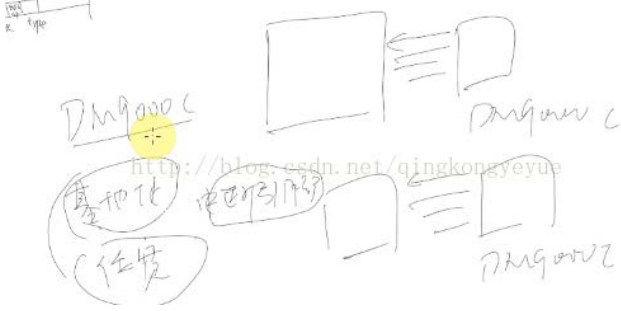
厂家驱动 dm9dev9000c

一 原理图

用什么地址访问网卡芯片, 根据片选 nGCS4 为低电平选中



所以移植的关键是找出相异性并进行修改



二厂家驱动修改

1 dmfe_probe1 函数

(1) 入口函数调用的 dmfe_probe 函数

分配 设置 注册 net_device 结构体

如何设置到函数 dmfe_probe1 里面看

想读寄存器，先把地址写到索引寄存器里面，再读数据寄存器

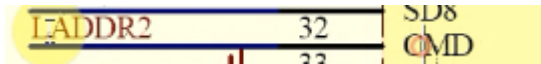
```
outb(DM9KS_VID_L, iobase); /*写 DM9000C 的索引寄存器(cmd 引脚为 0) */
```

DM9KS_VID_L 是寄存器地址

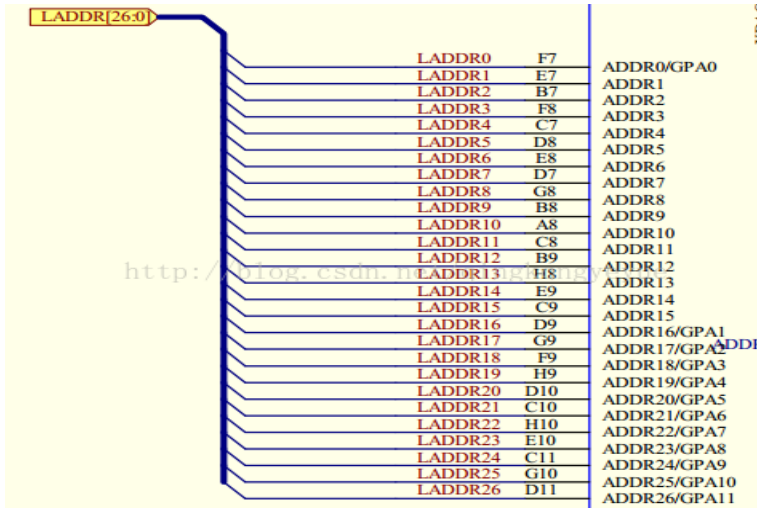
```
id_val = inb(iobase + 4); /* 读 DM9000C 的数据寄存器(cmd 引脚为 1) */
```

这里地址值加 4，表示 bit2，

DM9000 网卡部分



s3c2440 部分



为什么 2440 只有 27 根地址线？

答：2440 地址空间分为 8 个 bank，每个 bank 为 128M（即为 2^{27} ），ngCS[0:7] 作为 bank 的片选信号，总的寻址空间为 $128M \times 8 = 1G$ 。

```
outb(DM9KS_VID_L, iobase); //将要操作的寄存器地址写入 DM9000C 的索引寄存器
```

```
id_val = inb(iobase + 4); //从 DM9000C 的数据寄存器里读出数据
```

那么怎么分辨是操作的索引寄存器还是数据寄存器呢？就是通过 CMD 引脚！我们看到 CMD 引脚接的是 LADDR2，当 LADDR2 为 0 时表示操作的是索引寄存器，当 LADDR2 为 1 时，表示操作的是数据寄存器。

DM9000 的地址线只有 LADDR2。

```
int __init dmfe_probe1(struct net_device *dev)
{
    struct board_info *db; /* Point a board information structure */
    u32 id_val;
    u16 i, dm9000 found = FALSE;
    u8 MAC_addr[6] = {0x00, 0x60, 0x6E, 0x33, 0x44, 0x55};
    u8 HasEEPROM = 0, chip_info;
    DMFE_DEBUG(0, "dmfe_probe1()", 0);

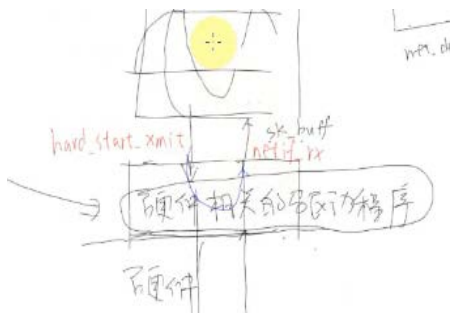
    /* Search All DM9000 serial NIC */
    do {
        outb(DM9KS_VID_L, iobase);
        id_val = inb(iobase + 4);
        outb(DM9KS_VID_H, iobase);
        id_val |= inb(iobase + 4) << 8;
        outb(DM9KS_PID_L, iobase);
        id_val |= inb(iobase + 4) << 16;
        outb(DM9KS_PID_H, iobase);
        id_val |= inb(iobase + 4) << 24;
    } while (0);
}
```

(2) iobase 的设置（入口函数）

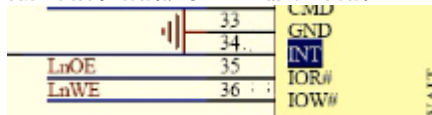
```
iobase = (int)ioremap(0x20000000, 1024); // 基地址是 0x20000000，映射 1M
```

(3) 中断号

看原理图，网卡芯片收到数据后会产生 1 个中断，网卡芯片里面也会有内存的，收到数据后会先放在内存，然后产生中断。在中断服务程序里面把数据从网卡芯片里面拿出来。构造 1 个 sk_buff，然后提交上去，发送的时候从 sk_buff 把数据拿出来，扔给网卡芯片发送出去，发送完后产生 1 个中断，需要中断号。



看原理图看中断引脚接到 2 4 4 0 的哪 1 个引脚



外部中断 7



在入口函数设置中断

```
irq = IRQ_EINT7;
```

2 dmfe_open 函数

注册中断 (可以是上升沿触发或下降沿触发)

```
request_irq(dev->irq, &dmfe_interrupt, IRQF_TRIGGER_RISING, dev->name, dev)
```

irq 是中断号

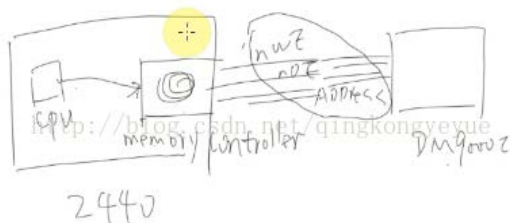
dmfe_interrupt 是中断处理函数

IRQF_TRIGGER_RISING 表示上升沿触发

3 时序修改

读写脉冲信号时序肯定会有要求, 有时间上的要求

网卡接在 2 4 4 0 上的, 内存控制器负责发出读写信号, 地址信号, 信号时间的设置在内存控制器里面设置, 这里没有设置也能用是因为 u b o o t 帮我们设置好了。想写 1 个不依赖 u b o o t 的驱动程序需要在入口函数中设置



```
/* 设置 S3C2440 的 memory controller */
```

```
bwscon = ioremap(0x48000000, 4); // 总线位宽和等待状态寄存器
```

```
bankcon4 = ioremap(0x48000014, 4);
```

片选 4

BWS CON

BUS WIDTH & WAIT CONTROL REGISTER (BWS CON)

Register	Address	R/W	Description	Reset Value
BWS CON	0x48000000	R/W	Bus width & wait status control register	0x000000

ST4	[19]	Determines SRAM for using UB/LB for bank 4. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0]) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS4	[18]	Determines WAIT status for bank 4. 0 = WAIT disable. 1 = WAIT enable	0
DW4	[17:16]	Determine data bus width for bank 4. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0

```
/* DW4[17:16]: 01-16bit 位宽
```

```
* WS4[18] : 0-WAIT disable 不使用等待信号
```

```
* ST4[19] : 0 = Not using UB/LB (The pins are dedicated nWBE[3:0])
```

不了解取缺省值

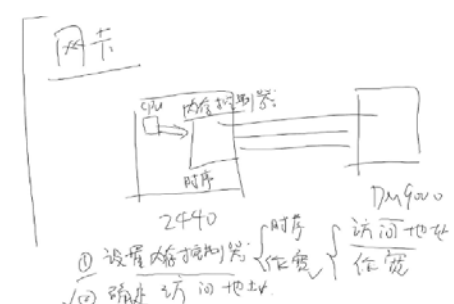
```
*/
```

```
val = *bwscon;
```

```
val &= ~(0xf<<16); //先将 4 个 bit 位置 0
```

```
val |= (1<<16); //再进行置位
```

```
*bwscon = val;
```



对于内存类接口的设备,比如网卡,norflash 等等,不通信的芯片差异的地方无非就是时序和位宽还有基地址
而时序和位宽是在 s3c2440 内部集成的内存控制器里面设置的
首先我们来确定时序部分:

1: 确认内存控制器关于 DM9000 对应时序寄存器参数

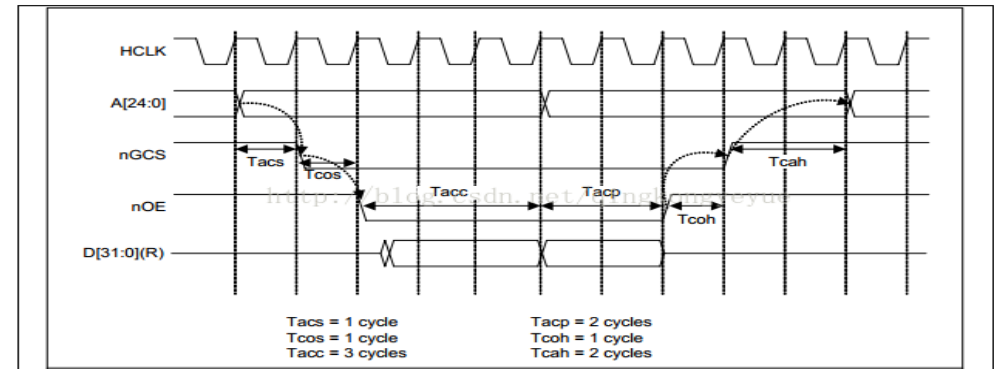
通过 2440 和 DM9000 时序图的分析来确定内存控制器关于 DM9000 时序的设置

1.1: S3C2440 内存控制器时序部分

Tacp 是页模式访问周期

BANKCONn	Bit	Description	Initial State
Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tcos	[12:11]	Chip selection set-up time before nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacc	[10:8]	Access cycle 000 = 1 clock 001 = 2 clocks 010 = 3 clocks 011 = 4 clocks 100 = 6 clocks 101 = 8 clocks 110 = 10 clocks 111 = 14 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	111
Tcoh	[7:6]	Chip selection hold time after nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	000
Tcah	[5:4]	Address hold time after nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00

从上图 s3c2440 的芯片手册截图可以看出,s3c2440 里面内存控制器外接了好几个 bank 也就是好几个内存类的接口,这些 bank 如果要使用就要设置时序,比如我们之前的 uboot 中的 start.s 里面就有设置 sdram 的时序,其实也就是设置了内存控制器中的对应的 bank 寄存器,现在我们要设置网卡的时序,也就是设置上面的寄存器
结合 s3c440 芯片手册中的内存控制器里面的寄存器介绍和 dm9000 的读写时序对比可得出:



上图是 2440 程序控制器的读时序和 dm9000 的读时序对应

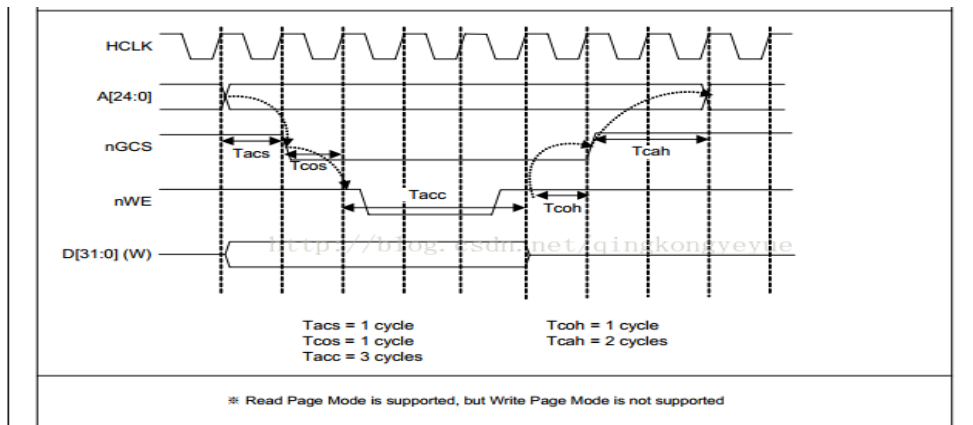


Figure 5-12. S3C2440A nGCS Timing Diagram

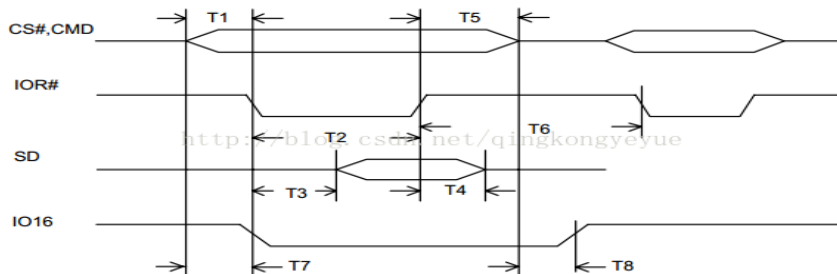
上图是 2440 程序控制器的写时序和 dm9000 的写时序对应

2.2: DM9000 网卡读写时序部分

读时序

读数据，先把地址值写到 DM9000 的某个寄存器

10.3.4 Processor I/O Read Timing

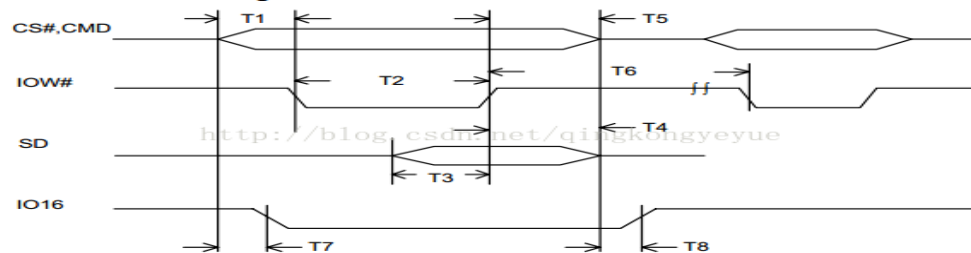


Symbol	Parameter	Min.	Typ.	Max.	Unit
T1	CS#,CMD valid to IOR# valid	0			ns
T2	IOR# width	10			ns
T3	System Data(SD) Delay time			19	ns *2
T4	IOR# invalid to System Data(SD) invalid			6	ns
T5	IOR# invalid to CS#,CMD invalid	0			ns
T6	IOR# invalid to next IOR#/IOW# valid When read DM9000C register	2			clk*
T2+T6	IOR# valid to next IOR#/IOW# valid When read DM9000C memory with F0h register	4			clk*
T2+T6	IOR# valid to next IOR#/IOW# valid When read DM9000C memory with F2h register	1			clk*
T7	CS#,CMD valid to IO16 valid			5	ns
T8	CS#,CMD invalid to IO16 invalid			5	ns

写时序

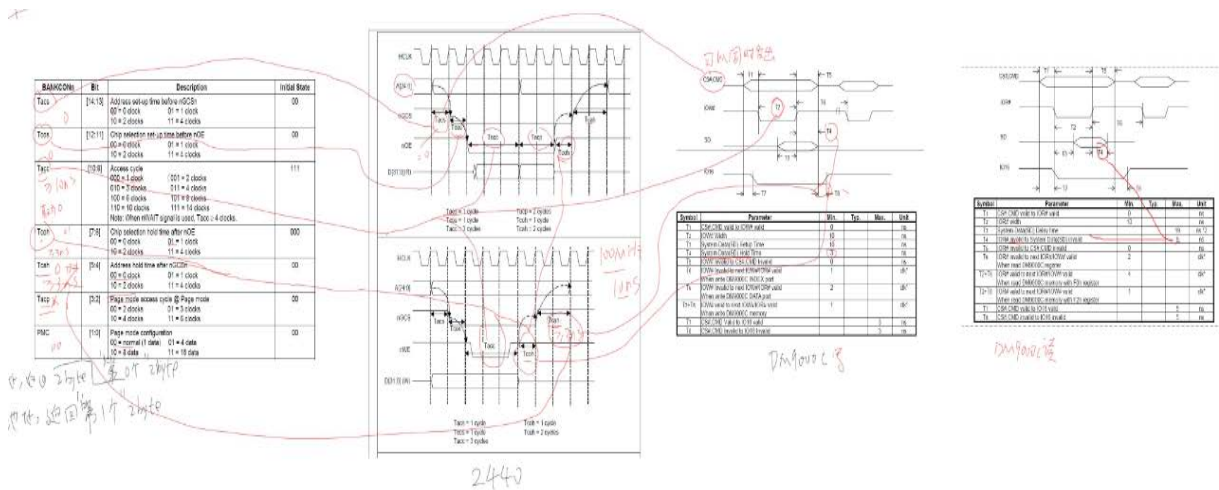
CMD 引脚就是 2440 接出的地址引脚

10.3.5 Processor I/O Write Timing



Symbol	Parameter	Min.	Typ.	Max.	Unit
T1	CS#,CMD valid to IOW# valid	0			ns
T2	IOW# Width	10			ns
T3	System Data(SD) Setup Time	10			ns
T4	System Data(SD) Hold Time	3			ns
T5	IOW# Invalid to CS#,CMD Invalid	0			ns
T6	IOW# Invalid to next IOW#/IOR# valid When write DM9000C INDEX port	1			clk*
T6	IOW# Invalid to next IOW#/IOR# valid When write DM9000C DATA port	2			clk*
T2+T6	IOW# valid to next IOW#/IOR# valid When write DM9000C memory	1			clk*
T7	CS#,CMD Valid to IO16 valid			3	ns
T8	CS#,CMD Invalid to IO16 Invalid			3	ns

最终的结论如下：



2、程序编写

(1) 时间参数设置 (在入口函数中)

BANKCON4 寄存器里面的值设置参考 DM9000 手册 可以先选最大值, 然后根据 DM9000 芯片手册修改这些值

```
/*
 * Tacs[14:13]: 发出片选信号之前, 多长时间要先发出地址信号
 *              DM9000C 的片选信号和 CMD 信号可以同时发出,
 *              所以它设为 0
 * Tcos[12:11]: 发出片选信号之后, 多长时间才能发出读信号 nOE
 *              DM9000C 的 T1>=0ns,
 *              所以它设为 0
 * Tacc[10:8]: 读写信号的脉冲长度,
 *              DM9000C 的 T2>=10ns, 怕太临界
 *              所以它设为 1, 表示 2 个 hclk 周期, hclk=100MHz, 就是 20ns
 * Tcoh[7:6]: 当读信号 nOE 变为高电平后, 片选信号还要维持多长时间
 *              DM9000C 进行写操作时, nWE 变为高电平之后, 数据线上的数据还要维持最少 3ns
 *              DM9000C 进行读操作时, nOE 变为高电平之后, 数据线上的数据在 6ns 之内会消失
 *              我们取一个宽松值: 让片选信号在 nOE 放为高电平后, 再维持 10ns,
 *              所以设为 01
 * Tcah[5:4]: 当片选信号变为高电平后, 地址信号还要维持多长时间
 *              DM9000C 的片选信号和 CMD 信号可以同时出现, 同时消失
 *              所以设为 0
 * PMC[1:0]: 00-正常模式
 */
/*bankcon4 = (1<<8)|(1<<6);*/ 对于 DM9000C 可以设 Tacc 为 1, 对于 DM9000E, Tacc 要设大一点, 比如最大值 7
*bankcon4 = (7<<8)|(1<<6); /* TQ2440 和 MINI2440 使用 DM9000E, Tacc 要设大一点 */
```

22.3 移植 DM9000C 驱动程序之测试及内存控制器简介

1、测试

(1) 修改好驱动, 重新编译内核 make uImage 并拷贝到网络文件系统中, 重启开发板

```
CHK include/linux/compile.h
CC drivers/net/dm9dev9000c.o
LD drivers/net/built-in.o
LD drivers/built-in.o
GEN http://version.csdn.net/qingkongyeyue
CHK include/linux/compile.h
UPD include/linux/compile.h

Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1828644 Bytes = 1785.79 kB = 1.74 MB
Load Address: 0x30008000
Entry Point: 0x30008000
Image arch/arm/boot/uImage is ready
book@book-desktop:/work/system/linux-2.6.22.6$ cp arch/arm/boot/uImage /work/nfs_root/uImage_dm9000c
```

驱动没有设置内存控制器时可以用, 因为内存控制器在 uboot 里面已经设置好了, DM9000 驱动程序能够运行依赖于 uboot 的设置, 写 1 个不依赖 uboot 的驱动程序需要设置内存控制器。而我们之前的 norflash 的驱动时候, 它也是内存类接口, 所以没有设置内存控制器中关于它的时序是因为用默认的参数就可以运行

Q: 虽然我们的 uboot 有设置内存控制器, 但是引导完内核后, uboot 不是已经完全消失了么, 为什么不用重新设置, 之前的 uboot 里面的看门狗 sdram 等等都需要在内核中重新设置吧?

测试DM9000C驱动程序:

1. 把dm9dev9000c.c放到内核的drivers/net目录下

2. 修改drivers/net/Makefile

把

obj-\$(CONFIG_DM9000) += dm9000.o

改为 <http://blog.csdn.net/qingkongyeyue>

obj-\$(CONFIG_DM9000) += dm9dev9000c.o

3. make uImage

使用新内核启动

5. 成功的条件

1:使用 NFS 启动

或

2:

ifconfig eth0 192.168.1.17

ping 192.168.1.1

之所以只要 NFS 能启动就说明网卡的驱动成功了的原因是 NFS 就是要通过 ip 来挂载的

配置 IP 并 ping 其他 网址

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:60:6E:33:44:55
          inet addr:192.168.1.17  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1494 errors:0 dropped:0 overruns:0 frame:0
          TX packets:581 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1990393 (1.8 MiB)  TX bytes:90702 (88.5 KiB)
          Interrupt:51 Base address:0xa000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

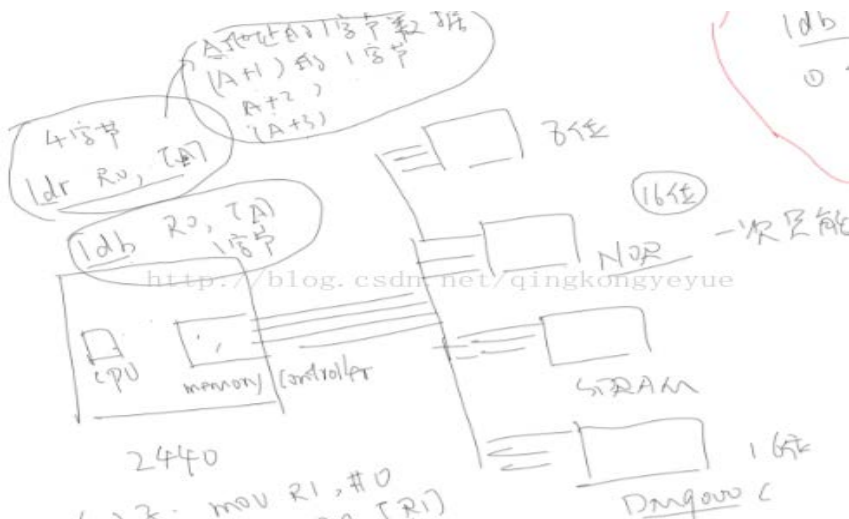
```
# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=4.346 ms
64 bytes from 192.168.1.1: seq=1 ttl=64 time=1.132 ms

--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 1.132/2.739/4.346 ms
#
#
# ifconfig eth0 192.168.1.17
# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=1.342 ms
```

2. 内存控制器

CPU 执行指令

得到 4 个地址的数据, 对于 CPU, 1 个地址对应的数据是 1 字节



上图的分析:

芯片内部有 CPU 和一个内存控制器, 它分为多个 bank, 每个 bank 都接有一个内存类接口的外设, 这些外设可能是 8bit, 可以是 16bit 的, 这个可以通过查看芯片有几根数据线就知道了

对于 cpu 执行指令的时候比如:

ldr r0 这条指令: 他是要求处理 4byte 的数据

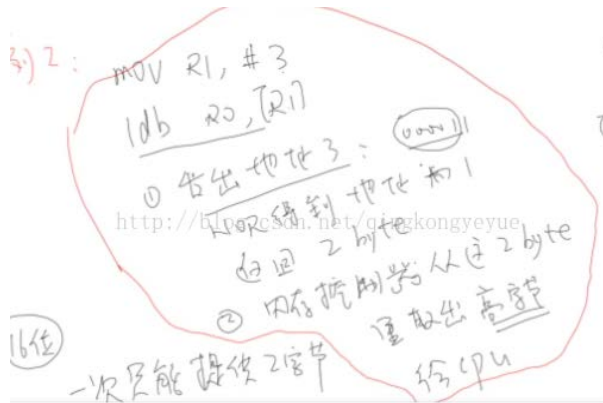
ldb r0 这条指令: 他是处理 1byte 的数据

也就是说对于 cpu 而言, 他并不知道外设是多少位的, 他只是执行指令, 有的指令是 4byte 的, 他就要 4byte 的数据, 不管你你外设的类型, 这就要通过内存控制器把数据拼凑成 cpu 要的数据长度, 所以每个内存控制器外接的设备都要设置位宽

(1) 例 1

CPU 要读 4 字节数据, 但是 nor flash 一次只能提供 2 字节, 内存控制器要发起 2 次传输, 先发送 0 地址, 得到 2 字节, nor flash 得到 0 地址, 返回 2 字节; 发出地址 2, 得到 2 字节, nor flash 得到 1 地址, 返回 2 字节; 把得到的 4 字节返回给 CPU, 需要设置位宽 BWSCON 寄存器, 当 CPU 想得到 4 个字节数据时, 内存控制器要发出 2 次操作, 如果位宽是 32, 那么只需要 1 次操作。

(2) 例 2



上图分析:

Cpu 执行 `mov r1, #3; ldb r0, [r1]`

也就是说 cpu 要把 norflash 的地址 3 的数据赋给 r0, 所以 cpu 发出了地址 3, 而对于 norflash 而言, 他一个地址里面存放的是 2byte 的数据, 理论上如果 cpu 发出 3 这个地址, norflash 接收到了 3 这个地址话, 就会返回下图的 3 地址这个数据, 但实际 cpu 的意思是返回下图 1 地址的高 8bit, 所以 norflash 要实际接收到 1 地址才对, 所以 norflash 和 cpu 的接线的 A0 并没有接在一起, 就是这个原因

0地址	xxxxxxxx	xxxxxxxx	(2字节)
1地址	xxxxxxxx	xxxxxxxx	(2字节)
2地址	xxxxxxxx	xxxxxxxx	(2字节)
3地址	xxxxxxxx	xxxxxxxx	(2字节)

nor flash 的位宽由硬件设置

DW0	[2:1]	Indicate data bus width for bank 0 (read only). 01 = 16-bit, 10 = 32-bit The states are selected by M[1:0] pins	-
Reserved	[0]	Reserve to 0	0

驱动没有设置内存控制器时可以用, 因为内存控制器在 uboot 里面已经设置好了, DM9000 驱动程序能够运行依赖于 uboot 的设置, 写 1 个不依赖 uboot 的驱动程序需要设置内存控制器。而我们之前的 norflash 的驱动时候, 它也是内存类接口, 之所以没有设置内存控制器中关于它的时序是因为用默认的参数就可以运行。

Q: 虽然我们的 uboot 有设置内存控制器, 但是引导完内核后, uboot 不是已经完全消失了么, 为什么不用重新设置, 之前的 uboot 里面的看门狗

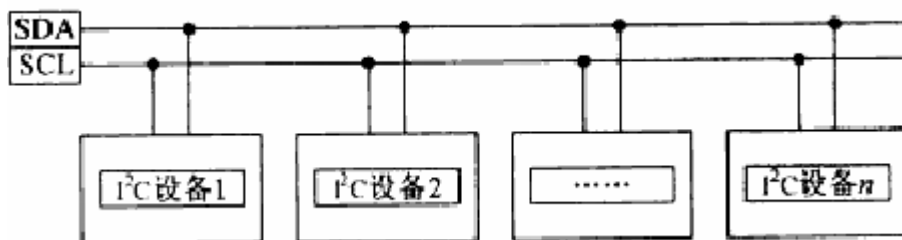
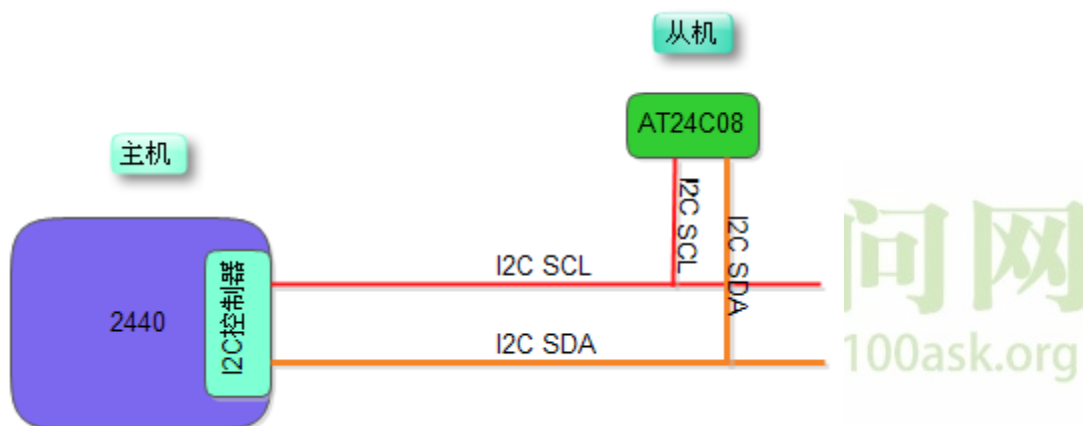
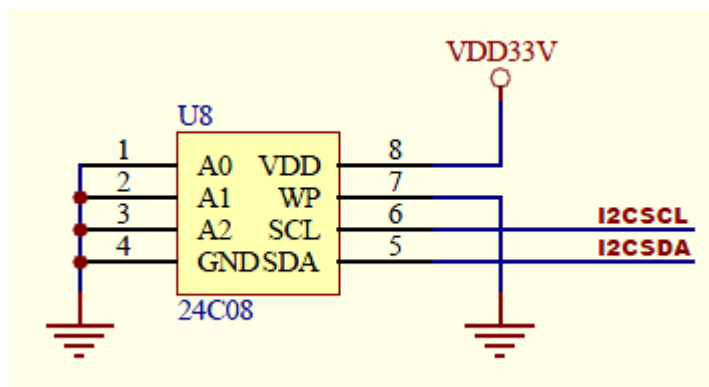
sdrum 等等都需要在内核中重新设置吧?

这里需要重新听下视频, 感觉表达的意思有问题

第 23 课：I2C 设备裸板程序

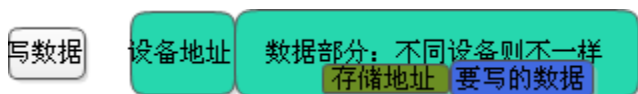
23.1 I2C 设备裸板程序之 I2C 总线

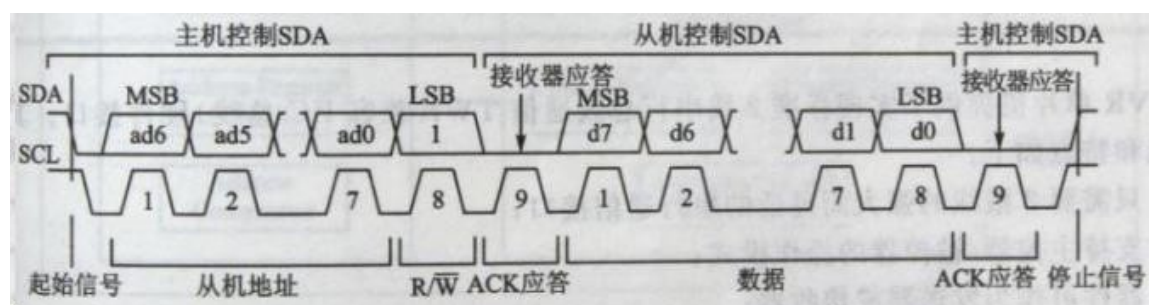
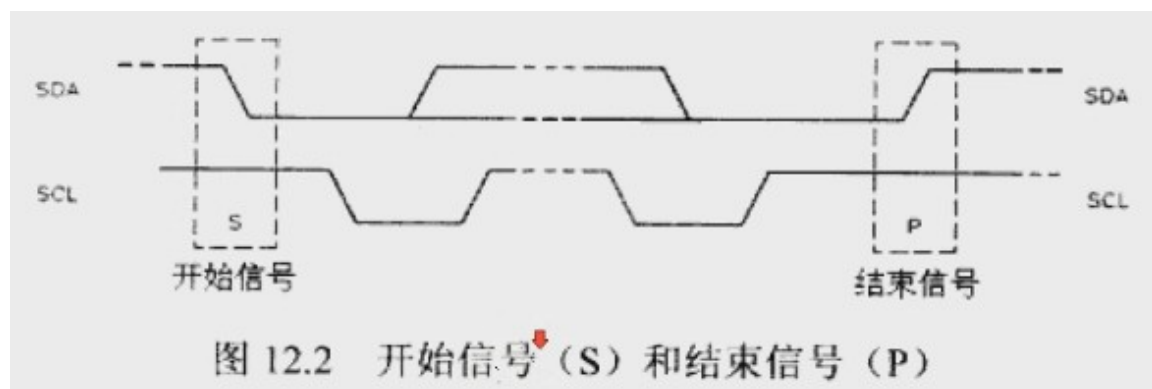
1: 电路图



I2C 是个主从结构，即所有的传输都是从主机发起。从机不可能主动引起数据的传输。对于 I2C 协议来说，它只能规定到发出的第一个数据是“地址”，后面发出的内容是什么，每个 I2C可能不同。

下面是一种情况：





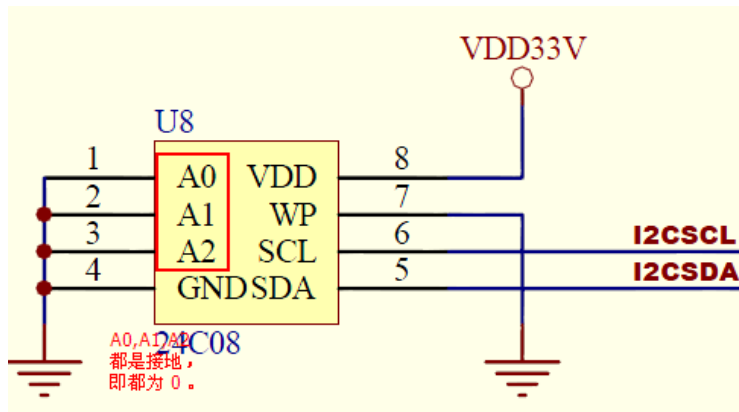
- 1, 平时 SDA和SCL 都是高电平。
- 2, 开始信号：当 SCL 为高电平的同时，SDA有个脉冲。
- 3, 数据在 SCL 低电平时变化，在 SCL脉冲为高电平间稳定。
- 4, 发出地址，每个 I2C芯片里面都有一个地址，是固化在芯片里面。

一，设备地址：寻址过程 与 写数据

1, 设备地址总共是 7bit,“1010”固定，后面“A2,A1,A0”可以从硬件上敲定。

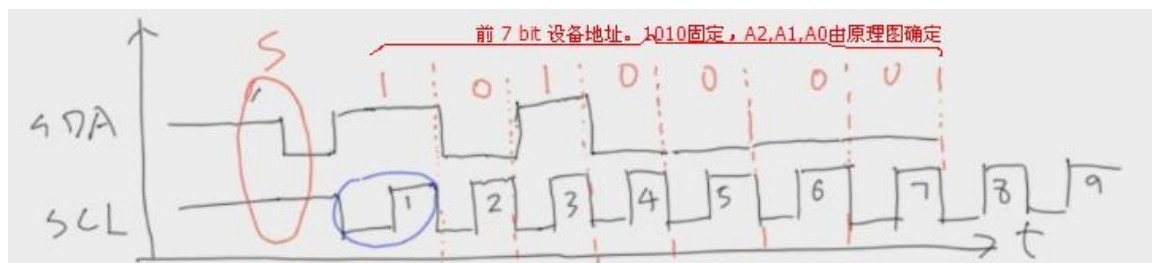
看开发板实际的接线。

2K	1	0	1	0	A ₂	A ₁	A ₀	R/W
	MSD							LSB
4K	1	0	1	0	A ₂	A ₁	P0	R/W
8K	1	0	1	0	A ₂	P1	P0	R/W



开发板上, A0, A1, A2 都是接地, 即此三位都是 0。

2, 则此 AT24C08 的地址 :



发出的前面 7 位就是设备地址。此时后面接的 I2C 设备, 接收到一个 start 信号时, 就知道下面紧接着的就是 7bit 的设备地址。若发现此地址与固化在设备内的地址相同时, 就知道是访问自己。

3, 访问时, 是“读”还是“写”: 由 7bit 后的第 8 位决定。

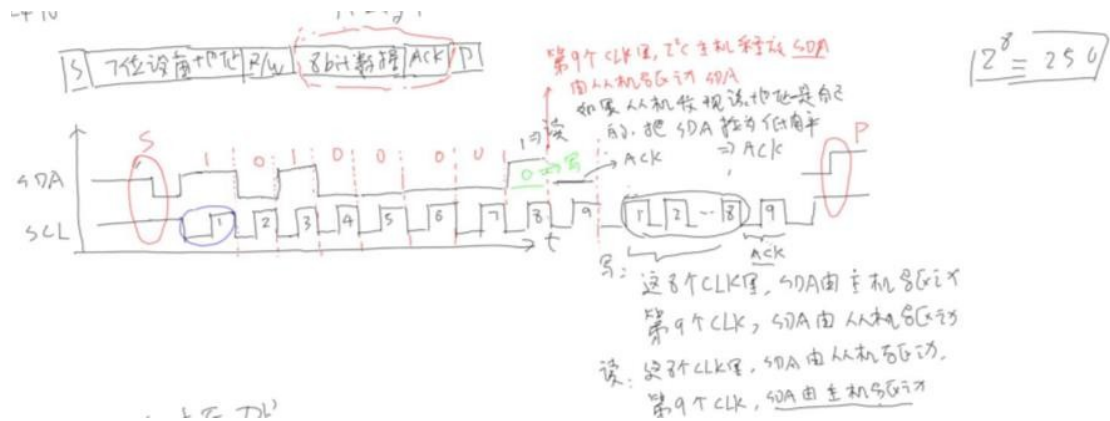
启动一个传输时, 主机先发出 S 信号, 然后发出 8 位 数据。这 8 位数据的前 7bit 为从机的地址, 第 8bit 表示传输的方向 (0 表示写操作, 1 表示读操作)。

4, ACK 回应信号 :

在第 9 个 CLK 里, I2C 主机释放 SDA, 由从机驱动驱动 SDA。若从机发现该“设备地址”是自己的, 是把 SDA 拉为低电平。这时主机就能知道此“设备地址”的设备是存在的, 后面就能再发数据了。

5. 后面接着再 8 个 CLK 时钟, 是具体的数据, 是与设备有关的。

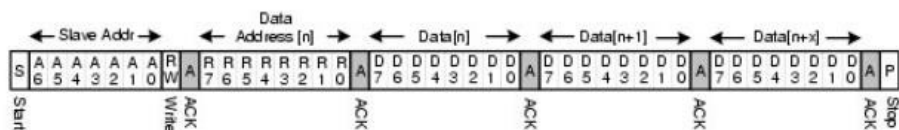
后面还有一个第 9bit, 是从机把 SDA 拉低确认 ACK。



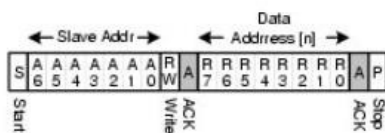
若为主机到从机的读：

寻址后的 8 个时钟是由 从机响应，从机会把设备驱动到 SDA 中，最后的第 9 个时钟也是 ACK，但是由主机响应。就是说主机已经接收到数据了，就到 ACK 拉为低电平。

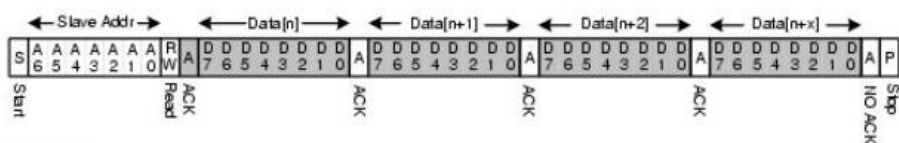
Write x Bytes to I2C Slave



Set Slave Data Pointer



Read x Bytes from I2C Slave



Master
Slave

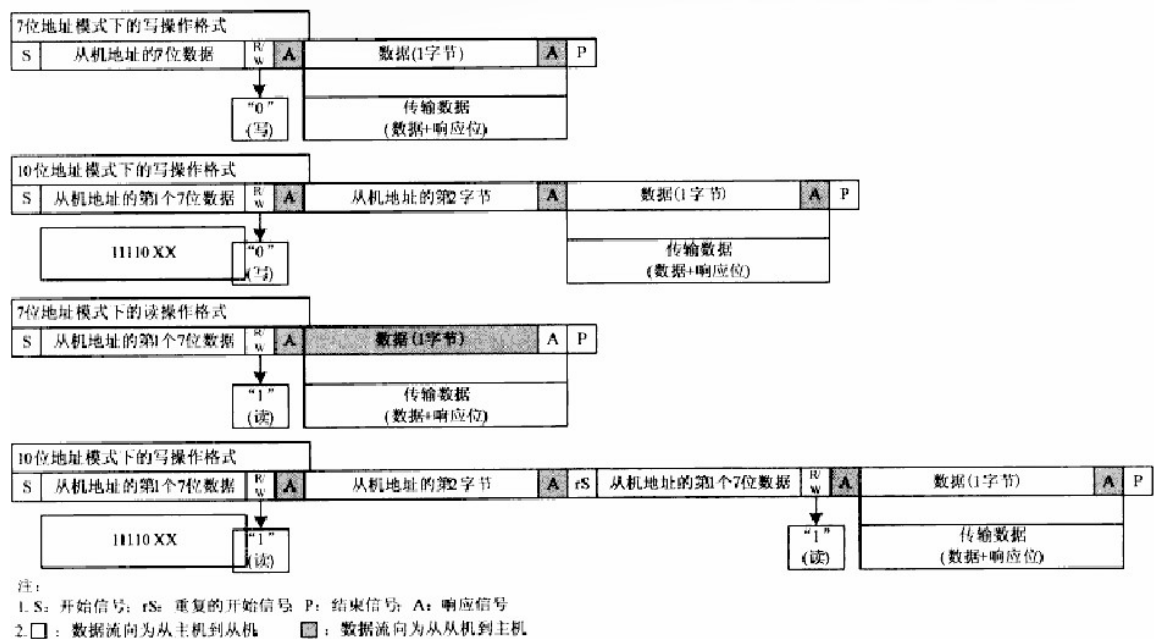


图 12.5 I²C 总线上数据传输的格式

结束传输：

SCL时钟信号在高电平期间，SDA由低电平变成高电平时结束传输。

I2C裸机程序

看一个裸板程序从“入口函数”开始看：head.S

一，开发板上电初始化工作：

假设NAND启动，一上电，程序的代码前 4K就会复制到 2440片内内存。

1，第一步：跳转到 reset 处：

```
9  _start:
10 @*****
11 @ 中断向量，本程序中，除Reset和HandleIRQ外，其它异常都没有使用
12 @*****
13 b Reset @跳转到reset处。
```

2，reset 处代码工作：

①，设置栈、关看门狗、设置时钟、设置内存控制器、初始化 NANDFLASH。

```
Reset:
    ldr sp, =4096 @ 设置栈指针，以下都是C函数，调用前需要设好栈
    bl disable_watch_dog @ 关闭WATCHDOG，否则CPU会不断重启
    bl clock_init @ 设置MPLL，改变FCLK、HCLK、PCLK
    bl memsetup @ 设置存储控制器以使用SDRAM
    bl nand_init @ 初始化NAND Flash
```

这些函数现在是在 0地址（片内内存）处运行。

②，把代码复制到 0x30000000 地址处。

```
ldr r0, =0x30000000 @ 复制代码到SDRAM中
mov r1, #0 @ 1. 目标地址 = 0x30000000，这是SDRAM的起始地址
ldr r2, =__bss_start @ 2. 源地址 = 0
sub r2, r2, r0 @ 3. 复制长度
bl CopyCode2SDRAM @ 调用C函数CopyCode2SDRAM
```

复制到 0x30000000处，是因为“链接地址”规定。


```

SECTIONS {
    . = 0x00000000;
    .init : AT(0){ head.o init.o nand.o }
    . = 0x30000000;
    .text : AT(4096) { *(.text) }
    .rodata ALIGN(4) : AT((LOADADDR(.text)+SIZEOF(.text)+3)&~(0x03)) {*(.rodata*)}
    .data ALIGN(4) : AT((LOADADDR(.rodata)+SIZEOF(.rodata)+3)&~(0x03)) { *(.data) }
    __bss_start = .;
    .bss ALIGN(4) : { *(.bss) *(COMMON) }
    __bss_end = .;
}

```

之前是把程序分成了两段，实际上可以分成一段。修改如下：

a, 链接地址：0x30000000

```

. = 0x30000000; 链接地址都是从这里开始

```

b, 代码段：

```

.text : { head.o(.text)
         init.o(.text)
         nand.o(.text)
         *(.text)
       }

```

```

Head.o 的代码      : head.o(.text)
段              : init.o(.text)
Init.o 的代码段    : nand.o(.text)
*      其他文件的代码段 : *(.text)

```

c, 只读数据段：所有文件的只读数据段。

```

.rodata ALIGN(4) : {*(.rodata*)}

```

d, 所有文件的数据段：

```

.data ALIGN(4) : { *(.data) }

```

e, bss段：

```

__bss_start = .;
.bss ALIGN(4) : { *(.bss) *(COMMON) }
__bss_end = .;

```

所有文件的的 bss和COMMON段。

3, 链接地址：

```

ldrsp,=4096
bl  disable_watch_dog
bl  clock_init

```

```
bl memsetup
bl nand_init
```

这些函数现在是在 0 地址（片内内存）处运行。它们的链接地址是“0x30000000”。链接地址并不等于它们当前所在的位置。所以它们就需要以“位置无关码”来写。就是说写这些函数时，要用“位置无关码”来写，不能用全局变量等。

①，关看门狗：

```
/*
 * 关闭WATCHDOG，否则CPU会不断重启
 */
void disable_watch_dog(void)
{
    WTCN = 0; // 关闭WATCHDOG很简单，往这个寄存器写0即可
}
```

②，初始化时钟：

```
#define FCLK      200000000
#define HCLK      100000000
#define PCLK      50000000
#define S3C2410_MPLL_200MHZ ((0x5c<<12)|(0x04<<4)|(0x00))
#define S3C2440_MPLL_200MHZ ((0x5c<<12)|(0x01<<4)|(0x02))
/*
 * 对于MPLLCON寄存器，[19:14]为MDIV，[9:4]为PDIV，[1:0]为SDIV
 * 有如下计算公式：
 * S3C2410: MPLL(FCLK) = (m * Fin)/(p * 2^s)
 * S3C2410: MPLL(FCLK) = (2 * m * Fin)/(p * 2^s)
 * 其中：m = MDIV + 8, p = PDIV + 2, s = SDIV
 * 对于本开发板，Fin = 12MHz
 * 设置CLKDIVN，令分频比为：FCLK:HCLK:PCLK=1:2:4，
 * FCLK=200MHz,HCLK=100MHz,PCLK=50MHz
 */
void clock_init(void)
{
    // LOCKTIME = 0x0fffffff; // 使用默认值即可
    CLKDIVN = 0x03; // FCLK:HCLK:PCLK=1:2:4, HDIVN=1,PDIVN=1

    /* 如果HDIVN非0，CPU的总线模式应该从“fast bus mode”变为“asynchronous bus mode” */
    __asm__(
        "mrc    p15, 0, r1, c1, c0, 0\n"      /* 读出控制寄存器 */
        "orr    r1, r1, #0xc0000000\n"        /* 设置为“asynchronous bus mode” */
        "mcr    p15, 0, r1, c1, c0, 0\n"      /* 写入控制寄存器 */
    );

    /* 判断是S3C2410还是S3C2440 */
    if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
    {
        MPLLCON = S3C2410_MPLL_200MHZ; /* 现在，FCLK=200MHz,HCLK=100MHz,PCLK=50MHz */
    }
    else
    {
        MPLLCON = S3C2440_MPLL_200MHZ; /* 现在，FCLK=200MHz,HCLK=100MHz,PCLK=50MHz */
    }
}
```

③, 内存设置:

```
/*
 * 设置存储控制器以使用SDRAM
 */
void memsetup(void)
{
    volatile unsigned long *p = (volatile unsigned long *)MEM_CTL_BASE;

    /* 这个函数之所以这样赋值，而不是像前面的实验(比如mmu实验)那样将配置值
     * 写在数组中，是因为要生成“位置无关的代码”，使得这个函数可以在被复制到
     * SDRAM之前就可以在steppingstone中运行
     */
    /* 存储控制器13个寄存器的值 */
    p[0] = 0x22011110;    //BWSCON
    p[1] = 0x00000700;    //BANKCON0
    p[2] = 0x00000700;    //BANKCON1
    p[3] = 0x00000700;    //BANKCON2
    p[4] = 0x00000700;    //BANKCON3
    p[5] = 0x00000700;    //BANKCON4
    p[6] = 0x00000700;    //BANKCON5
    p[7] = 0x00018005;    //BANKCON6
    p[8] = 0x00018005;    //BANKCON7

    /* REFRESH,
     * HCLK=12MHz: 0x008C07A3,
     * HCLK=100MHz: 0x008C04F4
     */
    p[9] = 0x008C04F4;
    p[10] = 0x000000B1;    //BANKSIZE
    p[11] = 0x00000030;    //MRSRB6
    p[12] = 0x00000030;    //MRSRB7
}
```

④, 初始化 NAND:

```
/* 初始化NAND Flash */
void nand_init(void)
{
    S3C2410_NAND * s3c2410nand = (S3C2410_NAND *)0x4e000000;
    S3C2440_NAND * s3c2440nand = (S3C2440_NAND *)0x4e000000;

#define TACLS    0
#define TWRPH0   3
#define TWRPH1   0

    /* 判断是S3C2410还是S3C2440 */
    if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
    {
        /* 使能NAND Flash控制器, 初始化ECC, 禁止片选, 设置时序 */
        s3c2410nand->NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);

        /* 复位NAND Flash */
        s3c2410_nand_reset();
    }
    else
    {
        /* 设置时序 */
        s3c2440nand->NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
        /* 使能NAND Flash控制器, 初始化ECC, 禁止片选 */
        s3c2440nand->NFCONT = (1<<4)|(1<<1)|(1<<0);

        /* 复位NAND Flash */
        s3c2440_nand_reset();
    }
}
```

位置无关码不能用全局变量, 这里的 "S3C2410_NAND"是局部变量。所以这个“nand.c”中的代码都要修改。看修改过的“nand.c”把全局变量全换成局部变量, 即把全局变量放到局部中(函数里)。

4, head.S 重定位:

把NANDFLASH 0地址从 0地址拷贝。拷贝的长度是"__bss_start = ."减去". = 0x30000000", 这样便有足够长了。

```
. = 0x30000000; 链接地址都是从这里开始
.text : { head.o(.text)
        init.o(.text)
        nand.o(.text)
        *(.text)
        }
.rodata ALIGN(4) : {*(.rodata*)}
.data ALIGN(4) : { *(.data) }
bss_start = .;
```

修改为如下: 就是不包括 bss段的长度。

```
ldr r2, = bss_start
```

bss段是放那些初始值为 0的全局变量的长度。当没有 1W个0时, 没有必要把所有的 0全放在最后的二进制文件里面, 要是带上这 1W个0就会很浪费空间。那么二进制文件里面就不去含有 bss段。bss段部分需要我们自己来清零。

4, head.S 接着调用: CopyCode2SDRAM函数, 将代码拷贝到内存中去。

```
b1 CopyCode2SDRAM @ 3: 复制代码  
@ 调用C函数CopyCode2SDRAM
```

```
int CopyCode2SDRAM(unsigned char *buf, unsigned long start_addr, int size)
{
    extern void nand_read(unsigned char *buf, unsigned long start_addr, int size);
    nand_read(buf, start_addr, size);
    return 0;
}
```

用"nand_read"函数来拷贝。

5, head.S 中将代码拷贝到内存中后, 接着清 bss段:

就是将“ __bss_start”到“ __bss_end;”间全部清为0.

```
b1 clean_bss @ 清除bss段, 未初始化或初值为0的全局/静态变量保存在bss段
```

```
void clean_bss(void)
{
    extern int __bss_start, __bss_end;
    int *p = &__bss_start;
    for (; p < &__bss_end; p++)
        *p = 0;
}
```

```
__bss_start = .;
.bss ALIGN(4) : { *(.bss) *(COMMON) }
__bss_end = .;
```

这里在链接时会分配链接地址。就把这段清为 0.那么以前访问这些全局变量时就都是 0.

6, head.S 中其他就是中断相关的设置, 最后是 main 函数。

二, 主函数: main.c

1, 初始化串口和 I2C设置:

a, 初始化串口:

```
uart0_init(); // 波特率115200, 8N1(8个数据位, 无校验位, 1个停止位)
```

```

/*
 * 初始化UART0
 * 115200, 8N1, 无流控
 */
void uart0_init(void)
{
    GPHCON |= 0xa0; // GPH2, GPH3 用作 TXD0, RXD0
    GPHUP = 0xc; // GPH2, GPH3 内部上拉

    ULCON0 = 0x03; // 8N1 (8个数据位, 无校验, 1个停止位)
    UCON0 = 0x05; // 查询方式, UART时钟源为PCLK
    UFCON0 = 0x00; // 不使用FIFO
    UMCON0 = 0x00; // 不使用流控
    UBRDIV0 = UART_BRD; // 波特率为115200
}

```

b, 初始化 I2C:

```
i2c_init();
```

```

/*
 * I2C初始化
 */
void i2c_init(void)
{
    GPEUP |= 0xc000; // 禁止内部上拉
    GPECON |= 0xa0000000; // 选择引脚功能: GPE15:IICSDA, GPE14:IICSCL

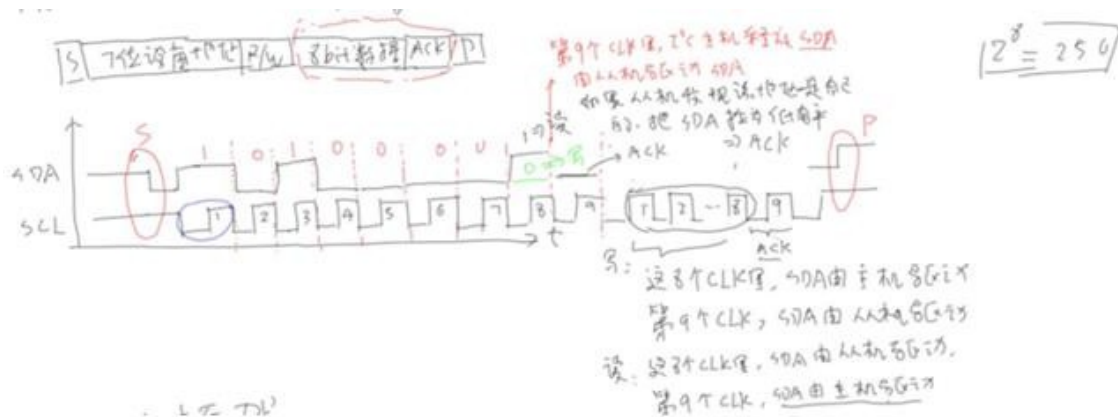
    INTMSK &= ~(BIT_IIC);

    /* bit[7] = 1, 使能ACK
     * bit[6] = 0, IICCLK = PCLK/16
     * bit[5] = 1, 使能中断
     * bit[3:0] = 0xf, Tx clock = IICCLK/16
     * PCLK = 50MHz, IICCLK = 3.125MHz, Tx Clock = 0.195MHz
     */
    IICCON = (1<<7) | (0<<6) | (1<<5) | (0xf); // 0xaf

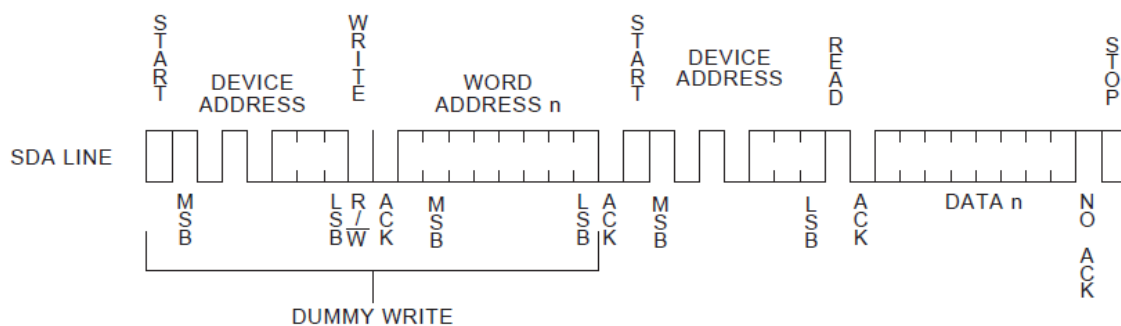
    IICADD = 0x10; // S3C24xx slave address = [7:1]
    IICSTAT = 0x10; // I2C串行输出使能(Rx/Tx)
}

```

3, 两个读写函数:



Random Read 随机读的时序：



前7bit发出设备地址找到从机，第8bit从机回应。紧接着发出存储地址写给从机。接着就开始一个“START”信号，再发出设备地址，这时候表示读。最后读到此数据（WORD ADDRESS n处）。

AT24108, 是8K

(1024*8) 即要10位地址才能表示1024。“WORD ADDRESSn”是8位根本无法寻址1K的空间。2的8次方是最多访问到256个地址。但这里E2PROM的容量有1K、2K时，显然8位的地址是不够的。

Device Addressing 设备寻址：

1K (128 x 8)

2K (256 x 8)

里面的“128”、“256”可以用8位来寻址到。

4K (512 x 8)

8K (1024 x 8)

而512, 1024则有2页或4页数据。

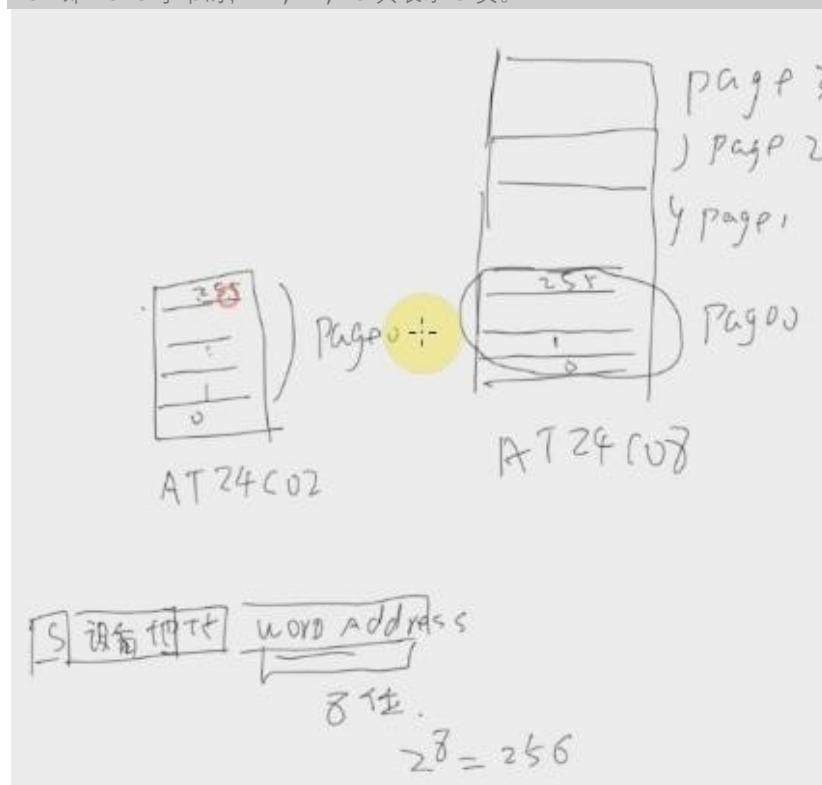
2K	1	0	1	0	A ₂	A ₁	A ₀	R/W
	MSD				LSB			
4K	1	0	1	0	A ₂	A ₁	P ₀	R/W
8K	1	0	1	0	A ₂	P ₁	P ₀	R/W

2K 是 256B 字节时，它所有的存储地址就可以用 8 位来寻址。

4K 即 512 字节时，P₀ 等于 0 时表示访问第一页。P₁ 等于 1 时表示访问第 1 页。

8K 即 1024 字节时，P₁, P₀ 总共就可以表示 4 页。

16K 即 2048 字节时，P₂, P₁, P₀ 共表示 8 页。



对于小容量的 AT24C02 为 256 字节，就可以用“WORD ADDRESS”8 位地址来寻址。但对于上面 AT24C08 这个 1024 字节的大容量 E2PROM 设备，则“WORD ADDRESS”只能寻址它的“PAGE 0”空间。对于其他页的空间，以前面发出“START”信号后“设备地址”：

8K	1	0	1	0	A ₂	P ₁	P ₀	R/W
----	---	---	---	---	----------------	----------------	----------------	-----

A₂ 是硬件引脚，P₁, P₀ 可以变。只要发出“1010+A₂”就能访问到设备（A₂ 硬件上接低电平时，则 A₂ 为 0；硬件上接的是高电平，则 A₂ 为 1），P₁, P₀ 就是用来表示访问哪一页。AT24C08 有 4 页（每页 256 字节）。

最后编译测试：

```
book@book-desktop:/work/drivers_and_test/18th_i2c/at24cxx$ ls -l i2c.bin
-rwxr-xr-x 1 book book 10856 2011-12-26 22:01 i2c.bin
book@book-desktop:/work/drivers_and_test/18th_i2c/at24cxx$
```

```
SECTIONS {
    得到了 bin有10K000000;
    .text : { head.o(.text)
             init.o(.text)
             nand.o(.text)
             *(.text)
            }
    .rodata ALIGN(4) : {*(.rodata*)}
    .data  ALIGN(4) : { *(.data) }
    _bss_start = .;
    .bss ALIGN(4) : { *(.bss) *(COMMON) }
    bss end = .;
}
```

链接脚本要保证“head.o”“init.o”“nand.o”三个文件位于前面4K，因为这个

三个文件在重定位之前，都得位于片内内存里面。可以看反汇编文件：

```
30000ff8: e5d0c000 ldrb ip, [r0]
30000ffc: e1a03004 mov r3, r4
4K=4096=0x1000,即是0x30001000处:
30001000: e158c003 cmp ip, r3
```

```
/* 读函数 */
void nand_read(unsigned char *buf, unsigned long start_addr, int size)
{
```

链接脚本中最后一个 nand.o的最后一个函数是：

在反汇编文件中搜索这个“nand_read”的链接地址：

```
300003e4 <nand_read>:
300003e4: e1a03a81 mov r3, r1, lsl #21
300003e8: e1a03aa3 mov r3, r3, lsr #21
```

链接地址是“3e4”没有超过“4K”。

毕业班第 2 课移植最新 u-boot

毕业班第 2 课第 1 节_移植最新 u-boot 之初试

1:准备移植最新版的 uboot

1.1:下载最新版 u-boot

下载最新版的 uboot,百度 uboot,进入官方网站:<http://www.denx.de/wiki/U-Boot/>

进入源代码 source code,可以通过 git 下载,也可以通过 ftp 服务器

U-Boot Source Code

- The **current source code** is available through the [git](http://git.denx.de) repository at git.denx.de.
- Released Versions** (and some special snapshots) are available from the Amazon Cloud Drive and from the DENX FTP server
- Snapshots** are available using the clever "snapshot" feature of the [git](http://git server) server (see the "snapshot" link after all commit entries)
- NOTE:** even though the download will be a *bzip2* compressed tarball, and the file on your disk will be named **.tar.bz2*, actually **not** compressed at all. This is a "feature" of your web browser, and we cannot do anything to help it.
- www.denx.de also hosts the Custodian git trees
- CVS is no longer supported.
- The old project page at SourceForge is no longer used and **dead**.

File Name	Size	Date
u-boot-2011.09.tar.bz2	8.0 MB	11/9/29 上午12:00:00
u-boot-2011.12-rc1.tar.bz2	8.2 MB	11/12/12 上午12:00:00
u-boot-2011.12-rc2.tar.bz2	8.2 MB	11/12/19 上午12:00:00
u-boot-2011.12-rc3.tar.bz2	8.2 MB	11/12/21 上午12:00:00
u-boot-2011.12.tar.bz2	8.2 MB	11/12/23 上午12:00:00
u-boot-2012.04-rc1.tar.bz2	8.4 MB	12/4/18 上午12:00:00
u-boot-2012.04-rc2.tar.bz2	8.4 MB	12/4/18 上午12:00:00
u-boot-2012.04-rc3.tar.bz2	8.4 MB	12/4/18 上午12:00:00
u-boot-2012.04.01.tar.bz2	8.4 MB	12/4/25 上午12:00:00
u-boot-2012.04.tar.bz2	8.4 MB	12/4/21 上午12:00:00

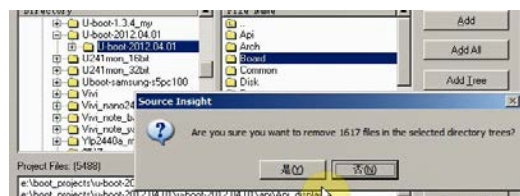
1.2:添加 sourceinsight 工程来分析最新 u-boot

我们的目的是建立能够支持 s3c2440 的 uboot,我们首先在 board 目录下查找是否有 2440 关键字的文件,发现没有,说明,该版本的 uboot 并不支持 2440,

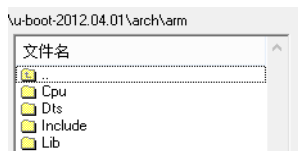
所以找到相近的文件 2410,也就是说我们添加 sourceinsight 工程时,board 目录下只需要添加 2410 的即可

```
[root@fedora board]# find ./ -name "*2410*"
./samsung/smdk2410
./samsung/smdk2410/smdk2410.c
[root@fedora board]#
```

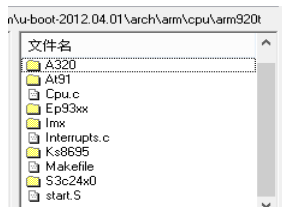
- 1、 先将所有文件都添加进来,点击 "Add All", 选中 "Include top level sub-directories" 和 "Recursively add lower sub-directories" 点击 "OK"
- 2、 选中第一步中已经添加之后的所有文件会在右边显示,在右边选中 board 目录, 选中 "Board" 目录, 点击 "Remove Tree", 去掉总个目录
进入 "Board\Samsung\Smdk2410\", 点击 "Add All"



- 3、 同步步骤 2, 选中 "Arch" 目录, 点击 "Remove Tree", 去掉总个目录
然后进入 arch 目录下我们用的是 arm 架构, 进入 arm 目录:



3.1: 先处理 CPU 目录: CPU 目录中我们用的是 arm920t, 里面也分了很多种:



进入“Arch\Arm\Cpu\Arm920t\”, 首先选中 arm 架构下的通用的文件, 双击选中“Cpu.c”“Interrupts.c”“start.S”, Makefile。

进入“Arch\Arm\Cpu\Arm920t\S3c24x0\”, 点击“Add All”

3.2: 进入“Arch\Arm\”, 选中“Dts”目录(这个目录不确定是干嘛用的), 点击“Add Tree”

3.3: 进入“Arch\Arm\Include\Asm\Arch_s3c24x0”, 点击“Add All”

然后在 Arch\Arm\Include\Asm\Arch_s3c24x0 同一级目录下还有很多.c 等等文件, 看起来是和所有架构都平等的文件, 应该是通用的, 所以加进来

进入“Arch\Arm\Include\Asm\”, 点击“Add All”, 去掉“Include top level sub-directories”和“Recursively add lower sub-directories”前面的勾, 表示只加顶层

目录的文件



进入“Arch\Arm\Include\Asm\”, 单击选中“Proc-armv”(这个文件看起来是通用的), 点击“Add Tree”

进入“Arch\Arm\Lib\”, 点击“Add All”

4: 选中“Include”目录, 点击“Remove Tree”, 去掉总个目录

顶层目录下的文件全部都加, 一般都是通用的

进入“Include\”, 点击“Add All”, 去掉“Include top level sub-directories”和“Recursively add lower sub-directories”前面的勾, 表示只加顶层目录的文件

进入“Include\”, 单击选中“Andestech”, 点击“Add Tree”

进入“Include\”, 单击选中“Asm-generic”, 点击“Add Tree”

进入“Include\”, 同上选中除“Configs”目录外的所有目录, 点击“Add Tree”。“Configs”目录先不加, 下面再议

进入“Include\Configs\”, 双击选中“Smdk2410.h”

4、 同步文件, 完成

1.3: 编译

尝试编译

解压: book@book-desktop:/work/system\$ tar xjf u-boot-2012.04.01.tar.bz2

book@book-desktop:/work/system\$ cd u-boot-2012.04.01/

配置: book@book-desktop:/work/system/u-boot-2012.04.01\$ make smdk2410_config

编译: book@book-desktop:/work/system/u-boot-2012.04.01\$ make

编译不成功, 因为版本太老

查看版本: book@book-desktop:/work/system/u-boot-2012.04.01\$ arm-linux-gcc -v

拷贝解压: book@book-desktop:/work/system/u-boot-2012.04.01\$ cd /work/tools/

book@book-desktop:/work/tools\$ mkdir tmp

book@book-desktop:/work/tools\$ tar xjf arm-linux-gcc-4.3.2.tar.bz2 -C tmp/

book@book-desktop:/work/tools\$ cd tmp/

book@book-desktop:/work/tools/tmp\$ ls usr

book@book-desktop:/work/tools/tmp\$ cd usr/local/arm/4.3.2/

book@book-desktop:/work/tools/tmp/usr/local/arm/4.3.2\$ ls

arm-none-linux-gnueabi bin lib libexec share

book@book-desktop:/work/tools/tmp/usr/local/arm/4.3.2\$ ls bin/

解压到根目录: book@book-desktop:/work/tools\$ sudo tar xjf arm-linux-gcc-4.3.2.tar.bz2 -C /

查看环境变量: book@book-desktop:/work/tools\$ echo \$PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/work/other_board/gcc-3.4.5-glibc-

2.3.6/bin

设置环境变量: book@book-desktop:/work/tools\$ export

PATH=/usr/local/arm/4.3.2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

查看环境变量: book@book-desktop:/work/tools\$ arm-linux-gcc -v

如果不想手动设置, 你可以按如下方式修改:

book@book-desktop:/work/tools\$ sudo vi /etc/environment

重新编译: book@book-desktop:/work/tools\$ cd /work/system/u-boot-2012.04.01/

book@book-desktop:/work/system/u-boot-2012.04.01\$ make distclean

book@book-desktop:/work/system/u-boot-2012.04.01\$ make smdk2410_config

Configuring for smdk2410 board...

book@book-desktop:/work/system/u-boot-2012.04.01\$ make

1.4: 烧些新版 uboot 到开发板:

编译成功之后, 我们要把他烧进我们的开发板, 看看这个 u-boot 能不能支持我们的开发板, 但是我们的开发是应该用 nor 启动还是 nand 启动呢?

我们可以通过 uboot 的重定位代码部分来判断:

```
1: relocate_code:
2:     mov r4, r0 /* save addr_sp */
3:     mov r5, r1 /* save addr of gd */
4:     mov r6, r2 /* save addr of destination */
5:
6:     /* Set up the stack */
7: stack_setup:
8:     mov sp, r4
9:
10:     adr r0, _start
11:     cmp r0, r6
12:     beq clear_bss /* skip relocation */
13:     mov r1, r6 /* r1 <- scratch for copy_loop */
14:     ldr r3, _bss_start_ofs
15:     add r2, r0, r3 /* r2 <- source end address */
16:
17: copy_loop:
18:     ldmia r0!, {r9-r10} /* copy from source address [r0] */
19:     stmia r1!, {r9-r10} /* copy to target address [r1] */
20:     cmp r0, r2 /* until source end address [r2] */
21:     blo copy_loop
22:
```

可以看出我们的这个新版的 uboot 的重定位是简单的复制数据, 所以肯定是 NOR 启动, 因为只有 NOR 启动才可以像内存一样读, NAND 的读的话有一大堆命令

所以我们烧写这个 uboot 的时候开发板要 NOR 启动

为什么我们之前的 uboot 可以用 NAND 启动? 因为老师打了补丁的, 可以对比看下

下载刚编译成功的 u-boot.bin, 发现重新启动, 串口没有任何信息

下节分析原因

毕业班第2课第2.1节_移植最新 u-boot 之分析启动过程之概述

分析新版 uboot 在串口上没有任何信息的原因

分析方法:

1:通过分析 makefile 层层分析

2:通过分析 make 之后的最后一句话找到 arm-linux-ld 这句话:

我们这里采取方法 2:

找到这句:

```
arm-linux-ld -pie -T u-boot.lds -Bstatic -Ttext 0x0 $UNDEF_SYM arch/arm/cpu/arm920t/start.o
```

我们之前的 u-boot.1.1.6 打完补丁之后的对应的语句是:

```
arm-linux-ld -Bstatic -T /root/Dev/Work_Space/System/u-boot-1.1.6/board/100ask24x0/u-boot.lds -Ttext 0x33F80000 $UNDEF_SYM cpu/arm920t/start.o
```

而我们毕业班自己写的 uboot 对应的语句是:

```
arm-linux-ld -Tboot.lds -o boot.elf start.o init.o boot.o
```

通过链接脚本知道: . = 0x00000000;同时-Ttext 0x0, 由此我们知道是从 NOR flash 开始运行, 通过链接脚本还知道第一个运行的是

arch/arm/cpu/arm920t/start.s

分析新版 uboot 的启动过程:

```
/*
 * set the cpu to SVC32 mode
 */
mrs r0, cpsr
bic r0, r0, #0x1f
orr r0, r0, #0xd3
msr cpsr, r0
/*设置为管理模式*/

/*
 * define pWTCN 0x53000000 /* Interrupt-Controller base addresses */
 * define INTMSK 0x4A000000 /* Interrupt-Controller base addresses */
 * define INTSUBMSK 0x4A00001C
 * define CLKDIVN 0x4C000014 /* clock divisor register */
 * endif
 */
ldr r0, =pWTCN
mov r1, #0x0
str r1, [r0]
/*关闭看门狗*/

/*
 * mask all IRQs by setting all bits in the INTMR - default
 */
mov r1, #0xffffffff
ldr r0, =INTMSK
str r1, [r0]
/* if defined(CONFIG_S3C2410)
 * ldr r1, =0x3ff
 * ldr r0, =INTSUBMSK
 * str r1, [r0]
 * endif
 */
/*屏蔽中断*/

/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
ldr r0, =CLKDIVN
mov r1, #3
str r1, [r0]
/*设置时钟,这里只是设置了时钟比例为 1:2:4,但是并没有设置时钟频率*/

/*
 * CONFIG_SKIP_LOWLEVEL_INIT
 * bl cpu_init_crit
 */
/*CPU 初始化: 包含了下面的 I/D caches,禁止 mmu (见/*CPU 初始化内容 1*/)*/

/*和底层初始化 lowlevel_init(这里面其实就是初始化 SDRAM) 见
/*CPU 初始化内容 2*/ */
```

```

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
/*
 * flush v4 I/D caches
 */
mov r0, #0
mcr p15, 0, r0, c7, 0 /* flush v3/v4 cache */
mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */

/*
 * disable MMU stuff and caches
 */
mrc p15, 0, r0, c1, c0, 0
bic r0, r0, #0x00002300 /* clear bits 13, 9:8 (V- - - RS)
bic r0, r0, #0x00000087 /* clear bits 7, 2:0 (B- - - - CAM)
orr r0, r0, #0x00000002 /* set bit 2 (A) Align
orr r0, r0, #0x00001000 /* set bit 12 (I) I-Cache
mcr p15, 0, r0, c1, c0, 0

/*
 * before relocating, we have to setup RAM timing
 * because memory timing is board-dependent, you will
 * find a lowlevel_init.S in your board directory.
 */
mov ip, lr

bl lowlevel_init

mov lr, ip
mov pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */
/*CPU 初始化内容 1*/

```

```

.globl lowlevel_init
lowlevel_init:
/* memory control configuration */
/* make r0 relative the current location so that it */
/* reads SMRDATA out of FLASH rather than memory */
ldr r0, =SMRDATA
ldr r1, =TEXT_BASE
sub r0, r0, r1
ldr r1, =BWSCON /* Bus Width Status Controller */
add r2, r0, #13*4

0:
ldr r3, [r0], #4
str r3, [r1], #4
cmp r2, r0
bne 0

/* everything is fine now */
mov pc, lr

/* the literal pools origin */
SMRDATA:
.word (0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
.word ((B0_Tacc<<13)+(B0_Tosc<<11)+(B0_Tacc<<8)+(B0_Tosc<<4)+(B0_Tacc<<4)+(B0_Tosc<<4)+(B0_Tacc<<4)+(B0_Tosc<<4))
.word ((B1_Tacc<<13)+(B1_Tosc<<11)+(B1_Tacc<<8)+(B1_Tosc<<4)+(B1_Tacc<<4)+(B1_Tosc<<4)+(B1_Tacc<<4)+(B1_Tosc<<4))
.word ((B2_Tacc<<13)+(B2_Tosc<<11)+(B2_Tacc<<8)+(B2_Tosc<<4)+(B2_Tacc<<4)+(B2_Tosc<<4)+(B2_Tacc<<4)+(B2_Tosc<<4))
.word ((B3_Tacc<<13)+(B3_Tosc<<11)+(B3_Tacc<<8)+(B3_Tosc<<4)+(B3_Tacc<<4)+(B3_Tosc<<4)+(B3_Tacc<<4)+(B3_Tosc<<4))
.word ((B4_Tacc<<13)+(B4_Tosc<<11)+(B4_Tacc<<8)+(B4_Tosc<<4)+(B4_Tacc<<4)+(B4_Tosc<<4)+(B4_Tacc<<4)+(B4_Tosc<<4))
.word ((B5_Tacc<<13)+(B5_Tosc<<11)+(B5_Tacc<<8)+(B5_Tosc<<4)+(B5_Tacc<<4)+(B5_Tosc<<4)+(B5_Tacc<<4)+(B5_Tosc<<4))
.word ((B6_MT<<15)+(B6_Trd<<2)+(B6_SCAN))
.word ((B7_MT<<15)+(B7_Trd<<2)+(B7_SCAN))
.word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchc<<15)+REFCNT)
.word 0x0
.word 0x0
.word 0x0

```

然后跳转回

```

4:
5: /* Set stackpointer in internal RAM to call board_init_f */
6: call_board_init_f:
7:     ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
8:     bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
9:     ldr r0, =0x00000000
0:     bl board_init_f
1:

```

/*上面设置栈,分析代码想确定 sp 值,发现比较麻烦,通过将反汇编文件来分析下:

book@book-desktop:/work/system/u-boot-2012.04.01\$ arm-linux-objdump -D u-boot > u-boot.dis

在反汇编中搜索 call_board_init_f

```

00000098 <call_board_init_f>:
98:     e59fd3d8     ldr    sp, [pc, #984] ; 478 <fiq+0x58>
9c:     e3cdd0b7     bic    sp, sp, #7 ; 0x7
a0:     e3a00000     mov    r0, #0 ; 0x0
a4:     eb0007f1     bl     2070 <board_init_f>

```

找到 0x98+984(10 进制)+8 的位置的值

```

468:     4a000008     .word  0x4a000008
46c:     000003ff     .word  0x000003ff
470:     4a00001c     .word  0x4a00001c
474:     4c000014     .word  0x4c000014
478:     30000f80     .word  0x30000f80
47c:     e1a00000     .word  0xe1a00000

```

最终确定 sp=0x30000f80

然后后面通过调用 c 函数 board_init_f 来初始化

```

void board_init_f(ulong bootflag)
{
    bd_t *bd;
    init_func_t **init_func_ptr;
    gd_t *gd;
    ulong addr, addr_sp;
    #ifndef CONFIG_FPM
    ulong reg;
    #endif

    bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_F, "board_init_f");

    /* Pointer is writable since we allocated a register for it */
    gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x7);
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__ ("": : "memory");
}

```

发现 board_init_f 有个 gd,想知道在哪定义的,通过搜索发现在 global_data.h 中定义:


```
#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t "gd asm ("r8")
```

意思是定义一个变量,可以通过这种格式定义为寄存器变量,当你读这个变量的值的时候,就相当于读寄存器 r8 的值

我们可以通过编译的时候找到一些蛛丝马迹:

```
make[1]: Leaving directory '/root/Dev/Work_Space/System/u-boot-2012.04.01/arch/arm-linux-gcc -E -g -Os -fno-common -ffixed-r8 -msoft-float -D_KERNEL -DCCboot_2012.04.01/include -fno-builtin -ffreestanding -nostdinc -isystem /root/Dev
```

在编译的时候,保持 r8 不变,固定他,把他留给 gd

通过上面可以看到 gd 第地址是 `gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);`

而我们之前的 .S 中的栈的地址是

```
ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
```

第二句只是 8 字节对齐

也就是说 gd 指向了栈的最高点

```
: memset((void *)gd, 0, sizeof(gd_t));
:
: gd->mon_len = _bss_end_ofs;
: #ifdef CONFIG_OF_EMBED
: /* Get a pointer to the FDT */
: gd->fdt_blob = _binary_dt_dtb_start;
: #elif defined CONFIG_OF_SEPARATE
: /* FDT is at end of image */
: gd->fdt_blob = (void *)(_end_ofs + _TEXT_BASE);
: #endif
: /* Allow the early environment to override the fdt address */
: gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
: (uintptr_t)gd->fdt_blob); /*gd 参数的初始化*/
:
: for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
:     if ((*init_fnc_ptr)() != 0) {
:         hang();
:     }
: } /*通过调用 init_sequence 函数指针数组来各种初始化*/
```

在 init_sequence 中有个 board_early_init_f 函数,里面才进行了 CPU 时钟的设置,也就是说到目前为止才设置了时钟频率

```
: debug("relocation Offset is: %08lx\n", gd->reloc_off);
: memcpy(id, (void *)gd, sizeof(gd_t));
:
: relocate_code(addr_sp, id, addr);
:
: /* NOTREACHED - relocate_code() does not return */
: } ? end board_init_f ? /*在 board_init_f 最后进行了重定位,这里是 c 语言调用汇编代码来重定位*/
```

这里就要疑惑了:既然我这个 uboot 的连接地址是 0,而且也是从 0 启动,为什么还要重定位呢?

这里就有个 pie 的作用

```
arm-linux-ld -pie -T u-boot.lds -Bstatic -Ttext 0x0 $UNDEF_SYM arch/arm/cpu/arm920t/start.o
```

我们发现在连接的时候,有个 pie 选项,这个下章继续讲

总结:本节主要是分析了新版 uboot 的启动信息,一直分析到了重定位代码:

对比分析自己写 uboot 和新版 uboot 的区别:

自己写 bootloader 的总结的过程:

- 初始化硬件: 关看门狗、设置时钟、设置 SDRAM、初始化 NAND FLASH
- 如果 bootloader 比较大, 要把它重定位到 SDRAM
- 把内核从 NAND FLASH 读到 SDRAM
- 设置"要传给内核的参数"
- 跳转执行内核

新 uboot 的过程:

- 2.1 set the cpu to SVC32 mode
- 2.2 turn off the watchdog
- 2.3 mask all IRQs by setting all bits in the INTMR
- 2.4 设置时钟比例
- 2.5 设置内存控制器
- 2.6 设置栈, 调用 C 函数 board_init_f

2.7 调用函数数组 init_sequence 里的各个函数

2.7.1 board_early_init_f : 设置系统时钟、设置 GPIO

.....

2.8 重定位代码:

这里的 pie 下面讲

毕业班第 2 课第 2.2 节_移植最新 u-boot 之分析启动过程之内存分布

以前我们的程序烧到 nandflash 或者 norflash 上

如果是 nand 启动的话,前 4K 被复制到片内内存 SRAM 中,若程序过大,nandflash 上的程序被拷贝到 sdram 上

至于拷贝到 sdram 上的哪里,是由连接脚本中的链接地址决定的, ,也就是说链接脚本中的地址是程序运行的时候程序应该位于的位置,

如果程序不位于连接地址有什么问题呢?比如某些全局变量的访问,cpu 是通过连接地址去访问的,

至于前面 4K 的代码之所以能够执行,是因为是运用位置无关码写的

而对于我们的新版 uboot,由于是 NOR 启动,他的连接地址和程序的运行地址都是 NORflash 的 0 地址,

连接地址可以看反汇编也可以看出来,反汇编的第一列就是连接地址,理论上通过连接地址去进行变量访问都是可以访问到的,为什么还要重定位呢?

这是由于,如果在程序中存在全局变量,如果只是读的话是没问题,如果要写这个全局变量的话,由于 NORflash 不能像内存一样写,所以肯定写不成功,

所以也要重定位,那么重定位到 sdram 哪里去呢?这是可以随便指定的,这样当我们要访问全局变量的时候,就要将这些变量的地址改为 sdram 中重定位

之后的新地址,这样用新地址去访问这些变量就 ok 了,那么我们怎么知道要修改哪些变量的地址呢?

这就要在编译时添加 pie 选项,这样就会在链接脚本中自动生成要修改代码的段,只要修改这些段就行

```
__image_copy_end = .;
.rel.dyn : {
    __rel_dyn_start = .;
    *(.rel*)
    __rel_dyn_end = .;
}
.dynsym : {
```

也就是说对于 NOR 启动的话,我们不仅要重定位,还要修改代码,让代码的变量和函数使用新的地址,这样才能正确访问到

下面分析新版 uboot 的重定位:

```
debug("relocation Offset is: %08lx\n", gd->reloc_off);
memcpy(d, (void *)gd, sizeof(gd_t));
relocate_code(addr_sp, id, addr);
```

其中 relocate 函数里的参数分别存在 r0,r1,r2 里面去了

这是 C 语言调用汇编,必须声明:

```
void relocate_code(ulong, gd_t *, ulong) __attribute__((noreturn));
```

这样就可以调用到汇编的代码:

```
/*
 * void relocate_code (addr_sp, gd, addr_moni)
 *
 * This "function" does not return, instead it continues in RAM
 * after relocating the monitor code.
 */
.globl relocate_code
relocate_code:
    mov r4, r0 /* save addr_sp */
    mov r5, r1 /* save addr of gd */
    mov r6, r2 /* save addr of destination */
```

现在我们分下,r0,r1,r2 的值是多少

R0= addr_sp 这个参数:

```
*/
addr_sp = addr - TOTAL_MALLOC_LEN;
debug("Reserving %dk for malloc() at: %08lx\n",
```

R0= id 这个参数:

R2= addr 这个参数:

分析 addr 的值:

```
addr = CONFIG_SYS_SDRAM_BASE + gd->ram_size;
```

```
/******
```

```
*****/
```

分析: CONFIG_SYS_SDRAM_BASE 和 gd->ram_size

```
#define CONFIG_SYS_SDRAM_BASE PHYS_SDRAM_1
```

```
#define PHYS_SDRAM_1 0x30000000 /* SDRAM Bank #1 */ //addr = 0x30000000+ gd->ram_size
```

```
: int dram_init(void)
: {
:     /* dram_init must store complete ramsize in gd->ram_size */
:     gd->ram_size = PHYS_SDRAM_1_SIZE;
:     return 0;
: }
```

```
#define PHYS_SDRAM_1_SIZE 0x04000000 /* 64 MB */ //addr = 0x30000000+64M=0X3400 0000
```

```
/******
```

```
*****/
```

```
#if !defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF)
/* reserve TLB table */
addr -= (4096 * 4);
```

```
//addr = addr-4*4096=0X33FF C000 (留 4KB 给 TLB table)
```

```
/* round down to next 64 kB limit */
addr &= ~(0x10000 - 1);
```

```
//addr = 0x33FF0000 (64kb 对齐)
```

```
/* round down to next 4 kB limit */
addr &= ~(4096 - 1);
```

```
// addr = 0x33FF0000(64KB 对齐了 4KB 肯定对齐)
```

```
/* reserve memory for LCD display (always full pages) */
addr = lcd_setmem(addr);
nrl->fh base = addr;
```

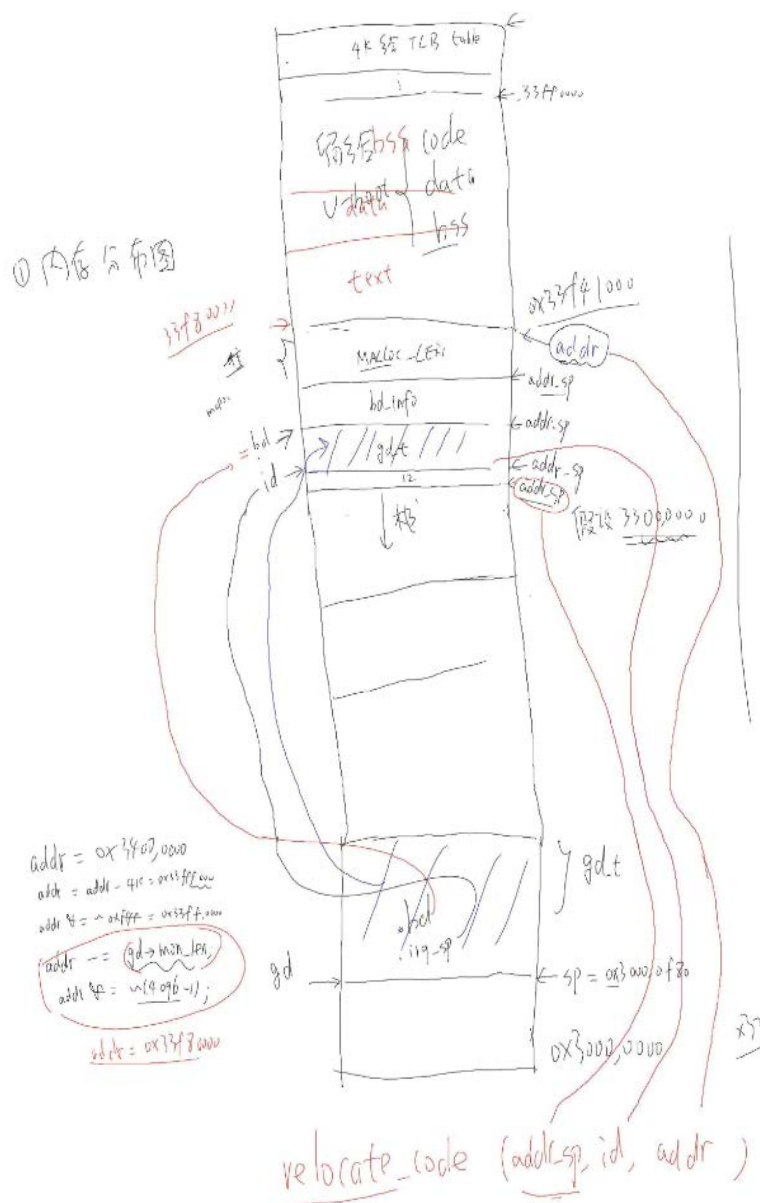
```
//addr = 0x33FF0000 - ???
```

后面还有很多.....

```
/******
```

```
*****/
```

最终得到了内存分布图:



毕业班第2课第2.3节_移植最新 u-boot 之分析启动过程之重定位

```
#define PHYS_SDRAM_1_SIZE 0x04000000 /* 64 MB */
```

毕业班第2课第3.1节_移植最新 u-boot 之修改代码之建新板_时钟_SDRAM_UART

我们用的是 2440 开发板，但在 u-boot 里面只有 2410 的 board/samsung 目录下只有 smdk2410 文件，没有发现 2440。

1、创建新单板

(1)u-boot 的修改

在 u-bootu-boot 目录下,切换到 board/samsung/目录

```
cd board/samsung/
```

新建 smdk2440 文件，并把 smdk2410 文件里的内容完全拷贝到 smdk2440 里面去。

```
cp smdk2410 smdk2440 -rf
```

配置 u-boot 出现错误

```
book@book-desktop:/work/system/u-boot-2012.04.01$ make smdk2440_config
make: *** No rule to make target `smdk2440_config'. Stop.
make: *** [smdk2440_config] Error 1
```

在 u-boot 源代码搜索 smdk2410, 头文件无需理会, boards.cfg 文件还需要修改, 修改 65 行。看文件的前面可知这些参数是什么意思

```
ARCH, CPU, SoC, Vendor, Target
http://prolog.esdm.net/

, 第一个参数是架构,

book@book-desktop:/work/system/u-boot-2012.04.01$ grep "smdk2410" * -nR
arch/arm/include/asm/mach-types.h:1644:# define machine_is_smdk2410() (machine_arch_type == MACH_TYPE_SMDK2410)
arch/arm/include/asm/mach-types.h:1646:# define machine_is_smdk2410() (0)
board/samsung/smdk2440/Makefile:28:COBJS := smdk2410.o
board/samsung/smdk2410/Makefile:28:COBJS := smdk2410.o
boards.cfg:65:smdk2410 arm arm920t -
samsung s3c24x0
MAINTAINERS:750: smdk2410 ARM920T
```

修改 boards.cfg:

仿照

smdk2410	arm	arm920t	-	samsung	s3c24x0
----------	-----	---------	---	---------	---------

添加:

smdk2440	arm	arm920t	-	samsung	s3c24x0
----------	-----	---------	---	---------	---------

再次编译,提示错误:

```
:30: error: configs/smdk2440.h: No such file or directory
:30: error: configs/smdk2440.h: No such file or directory
```

切换到配置文件目录, 每一种单板都有一个配置文件

```
cd ../../include/configs/
```

吧 smdk2410.h 拷贝为 smdk2440.h

```
cp smdk2410.h smdk2440.h
```

重新配置

```
make smdk2440_config
```

编译

```
make
```

(2) 烧写看结果

发现没有任何结果, 其实我们这个在毕业班移植 uboot 的第一课就已经烧写过了, 当时是烧写 2410 的, 而我们这个的结果和 2410 完全没有差别, 所以不用烧写都知道结果了

(3) 调试 (关心内存设置和 CPU 时钟)

a. 阅读代码发现不足:

我们在 start.s 中发现

```
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
ldr r0, =CLKDIVN
mov r1, #3
str r1, [r0]
#endif /* CONFIG_S3C24X0 */
```

我们这里只是设置了 FCLK, HCLK, PCLK 的比例, 并未设置时钟频率, 也就是说时钟还是以 12M 来执行的, 最终是在 board_init_f 这个函数里面的来设置了时钟频率

```
int board_early_init_f(void)
{
    struct s3c24x0_clock_power * const clk_power =
        s3c24x0_get_base_clock_power();
    struct s3c24x0_gpio * const gpio = s3c24x0_get_base_gpio();

    /* to reduce PLL lock time, adjust the LOCKTIME register */
    writel(0xFFFFFFFF, &clk_power->locktime);

    /* configure MPLL */
    writel((M_MDIV << 12) + (M_PDIV << 4) + M_SDIV,
        &clk_power->mpllcon);
}
```

而在这之间我们又做了如下的事情:

1: cpu_init_crit:这里面主要完成了 mmu 的禁止和 sdram 的初始化

2: 设置栈

```
: /* Set stackpointer in internal RAM to call board_init_f */
: call_board_init_f:
:     ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
:     bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
:     ldr r0, =0x00000000
:     bl board_init_f
```

而在初始化 sdram 中代码中写的是以 60M 的时钟频率来初始化的(但实际上是 12M), 这样会不会导致 sdram 的初始化不成功呢?

如果 sdram 初始化不成功, 那么栈的设置也不会成功, 因为当前的栈设置在 sdram 中的

我们可以通过读写 sdram 的地址来调试:

(4) 验证 MPLL 未设置情况下 SDRAM 能否使用 (用 openjtag 调试)

编译出来的 uboot 非常大 (400 多 K), 先烧写主光盘里的 u-boot.bin 到 nor, 然后用这个 uboot 来烧写新的 uboot

我们之所以用老的 UBOOT, 是老的 UBOOT 功能强大, 支持 USB 下载、网络下载; 新的 UBOOT 正在被调试, 它能否下载、烧写, 我们不知道, 所以

不用新的 uboot 来自己升级自己

```
book@book-desktop:/work/system/u-boot-2012.04.01$ ls -l u-boot.bin
-rw-r--r-- 1 book book 472792 2012-05-26 02:57 u-boot.bin
```

查看 u-boot 分区大小, 下面大小是 0x40000, 换算成十进制是 262K, 而需要烧写的 u-boot 大小是 400 多 K,

所以不能用老版 uboot 的菜单页面中的命令烧写

```
never reach
Hit any key to stop autoboot: 0

TRY_TIME
***** 100ask Bootloader for OpenJTAG *****
[n] Download u-boot to Nand Flash
[k] Download Linux kernel uImage
[j] Download root_jffs2 image
[y] Download root_yaffs image
[d] Download to SDRAM & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the Nand Flash
[s] Set the boot parameters
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection: q
OpenJTAG>
OpenJTAG>
Ready
```

只能用 uboot 里面的命令烧写

```
Enter your selection: q
OpenJTAG> mtd

device nand0 <nandflash0>, # parts = 4
#: name          size          offset          mask_flags
0: bootloader    0x00040000    0x00000000      0
1: params        0x00020000    0x00040000      0
2: kernel        0x00200000    0x00060000      0
3: root          0x0fda0000    0x00260000      0

active partition: nand0,0 - (bootloader) 0x00040000 @ 0x00000000

defaults:
mtdids : nand0=nandflash0
mtdparts: mtdparts=nandflash0:256k@0 (bootloader) , 128k (params) , 2m (kernel) , - (root)
```

接上 usb_device 那条线, 打开 dnw,

用 usb 1 30000000 命令下载, usb 是指 usb 命令, 1 是指一直等待直到下载完, 30000000 是指下载到 SDRAM 的地址。然后用 dnw 下载

```
OpenJTAG> usb 1 30000000
USB host is connected. Waiting a download.
```

然后去掉 nor flash 的写保护

```
OpenJTAG> protect off all
Un-Protect_Flash Bank # 1
```

从 0 地址开始擦除, 结束地址是 7FFFF, 共擦除 512K

```
OpenJTAG> erase 0 7FFFF
***** done ***** cadn.net/
Erased 11 sectors
```

从 SDRAM 的 3000, 0000 地址拷贝到 nor flash 的 0 地址, 拷贝大小为 512K

```
OpenJTAG> cp.b 30000000 0 80000
Copy to flash...done csdn.net/
```

烧写完后重启开发板

通过打断点进行对 sdram 的读写发现 sdram 初始化不成功

UBOOT 里先以 60MHZ 的时钟计算参数来设置内存控制器, 但是 MPLL 还未设置, 仍以 12M 频率执行

```
5: #define Tchr 0x2 /* 3clk */
6: #define REFCNT 1118 /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*50) */
7: /*=====*/

1: SMRDATA:
2: .word ((B1_BWSCON<<4)+(B2_BWSCON<<4)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
3: .word ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tapc<<2)+(B0_PMC))
4: .word ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tapc<<2)+(B1_PMC))
5: .word ((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tapc<<2)+(B2_PMC))
6: .word ((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tapc<<2)+(B3_PMC))
7: .word ((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tapc<<2)+(B4_PMC))
8: .word ((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tapc<<2)+(B5_PMC))
9: .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
10: .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
11: .word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
```

处理措施: 把 MPLL 的设置放到 start.S 里 (参考自己写 bootloader 的时钟设置), 取消 board_early_init_f 里对 MPLL 的设置

(1) 时钟的修改

```
#define S3C2440_MPLL_400MHZ ((0x5c<<12) | (0x01<<4) | (0x01))
```

```
5: /* 2. 设置时钟 */
6: ldr r0, =0x4c000014
7: // mov r1, #0x03; // FCLK:HCLK:PCLK=1:2:4, HDIVN=1, PDIVN=1
8: mov r1, #0x05; // FCLK:HCLK:PCLK=1:4:8
9: str r1, [r0]

1: /* 如果 HDIVN 非 0, CPU 的总线模式应该从 "fast bus mode" 变为 "asynchronous bus mode" */
2: mrc p15, 0, r1, c1, c0, 0 /* 读出控制寄存器 */
3: orr r1, r1, #0xc0000000 /* 设置为 "asynchronous bus mode" */
4: mcr p15, 0, r1, c1, c0, 0 /* 写入控制寄存器 */

5: /* MPLLCON = S3C2440_MPLL_200MHZ */
6: ldr r0, =0x4c000004
7: ldr r1, =S3C2440_MPLL_400MHZ
8: str r1, [r0]

1: /* 启动 I-CACHE */
2: mrc p15, 0, r0, c1, c0, 0 @ read control reg
3: orr r0, r0, # (1<<12)
4: mcr p15, 0, r0, c1, c0, 0 @ write it back
```

(2) 内存控制器单元的修改 (个人感觉这里不修改应该也没事, 应该是两种代码的风格, 而且这样写不方便知道每个寄存器的值)

把 \u-boot-2012.04.01\board\samsung\smdk2440\lowlevel_init.S 里的

```
bné ob
/* everything is fine now */
mov pc, lr
.ltorg
/* the literal pools origin */

SMRDATA:
.word ((B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
.word ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tapc<<2)+(B0_PMC))
.word ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tapc<<2)+(B1_PMC))
.word ((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tapc<<2)+(B2_PMC))
.word ((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tapc<<2)+(B3_PMC))
.word ((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tapc<<2)+(B4_PMC))
.word ((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tapc<<2)+(B5_PMC))
.word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
.word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
.word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
.word 0x32
.word 0x30
.word 0x30
```

改为

\韦东山 Linux 视频第 2 期_从零写驱动

\韦东 Linux 视频第 1 第 2 期所有源码文档图片芯片手册

\毕业班_文档_图片_源码_bin

\毕业班第 1 课自己写 bootloader

\32th_my_bootloader

\2th

start.S 里使用的代码


```

SMRDATA:
    .long 0x22011110 //BWSCON
    .long 0x00000700 //BANKCON0
    .long 0x00000700 //BANKCON1
    .long 0x00000700 //BANKCON2
    .long 0x00000700 //BANKCON3
    .long 0x00000700 //BANKCON4
    .long 0x00000700 //BANKCON5
    .long 0x00018005 //BANKCON6
    .long 0x00018005 //BANKCON7
    .long 0x008C04F4 // REFRESH
    .long 0x000000B1 //BANKSIZE
    .long 0x00000030 //MRSRB6
    .long 0x00000030 //MRSRB7

```

(5) 出现 乱码

查看串口波特率的设置, 在 start.s 中找到

```

ldr sp, =(CONFIG_SYS_INIT_SP_AD
bic sp, sp, #7 /* 8-byte alignment f
ldr r0,=0x00000000
bl board_init_f

```

在 board_init_f 中找到对应串口初始化的函数:

```

/*
7: init_func_t *init_sequence[] = {
8: #if defined(CONFIG_ARCH_CPU_INIT)
9:     arch_cpu_init, /* basic arch cpu dependent set
0: #endif
1: #if defined(CONFIG_BOARD_EARLY_INIT_F)
2:     board_early_init_f,
3: #endif
4: #ifdef CONFIG_OF_CONTROL
5:     fdtdec_check_fdt,
6: #endif
7:     timer_init, /* initialize timer */
8: #ifdef CONFIG_FSL_ESDHC
9:     get_clocks,
0: #endif
1:     env_init, /* initialize environment */
2:     init_baudrate, /* initialize baudrate settings */
3:     serial_init, /* serial communications setup */
4:     console_init_f, /* stage 1 init of console */
5:     display_banner, /* say that we are here */
6: #if defined(CONFIG_DISPLAY_CPUINFO)
7:     print_cpuinfo, /* display cpu info (and speed) */
8: #endif
9: #if defined(CONFIG_DISPLAY_BOARDINFO)
0:     checkboard, /* display board info */
1: #endif
2: #if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT
3:     init_func_i2c,
4: #endif
5:     dram_init, /* configure available RAM banks */

```

serial_init->serial_init_dev->_serial_setbrg:

```

/*
9: void _serial_setbrg(const int dev_index)
0: {
1:     struct s3c24x0_uart *uart = s3c24x0_get_base_uart(dev_index);
2:     unsigned int reg = 0;
3:     int i;
4:
5:     /* value is calculated so : (int)(PCLK/ 16./ baudrate) - 1 */
6:     reg = get_PCLK() / (16 * gd->baudrate) - 1;
7:
8:     writel(reg, &uart->ubrddiv);
9:     for (i = 0; i < 100; i++)
0:         /* Delay */ ;
1: }

```

在设置波特率的函数中这里根据 PCLK 除以波特率得到某个系数, 把这个系数 reg 写到串口控制器的寄存器里面就可以设置

,所以肯定是在 get_p1ck 这里面设置的时钟和上 s3c2440 的 p1ck 不匹配导致串口没数据的,

```

/*: ulong get_PCLK(void)
1: {
2:     struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();
3:     return (readl(&clk_power->clkdivn) & 1) ? get_HCLK() / 2 : get_HCLK();
4: }

```

发现 PLCK 依赖与 hclk:

```

: /* return HCLK frequency */
: ulong get_HCLK(void)
: {
:     struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();
:     #ifdef CONFIG_S3C2440
:     switch (readl(&clk_power->clkdivn) & 0x6) {
:     default:
:     case 0:
:         return get_FCLK();
:     case 2:
:         return get_FCLK() / 2;
:     case 4:
:         return (readl(&clk_power->camdivn) & (1 << 9)) ?
:             get_FCLK() / 8 : get_FCLK() / 4;
:     case 6:
:         return (readl(&clk_power->camdivn) & (1 << 8)) ?
:             get_FCLK() / 6 : get_FCLK() / 3;
:     }
:     #else
:     return (readl(&clk_power->clkdivn) & 2) ? get_FCLK() / 2 : get_FCLK();
:     #endif
: } ? end get_HCLK ?

```

发现在 get_HCLK 里没有定义 CONFIG_S3C2440

```

/* value is calculated so : (int)(PCLK/16./baudrate) -1 */
reg = get_FCLK() / (16 * gd->baudrate) - 1;
writel(reg, &uart->ubrdiv);

```

处理措施: include/configs/smdk2440.h: 去掉 CONFIG_S3C2410

```
#define CONFIG_S3C2440
```

然后重新编译,发现报错:

```

protector -Wno-format-nonliteral -Wno-format-security -o s3c2410_nand.o s3c2410_
s3c2410_nand.c: In function 's3c2410_hwcontrol':
s3c2410_nand.c:57: warning: implicit declaration of function 's3c2410_get_base_nand'
s3c2410_nand.c:57: warning: initialization makes pointer from integer without a cast
s3c2410_nand.c:72: error: dereferencing pointer to incomplete type
s3c2410_nand.c:72: error: dereferencing pointer to incomplete type
s3c2410_nand.c:75: error: dereferencing pointer to incomplete type
s3c2410_nand.c:75: error: dereferencing pointer to incomplete type

```

打开这个文件 s3c2410_nand.c 找到 72 行,发现是如下结构体报的错

```

00070:
00071:         if (ctrl & NAND_NCE)
00072:             writel(readl(&nand->nfconf) & ~S3C2410_NFCONF_nFCE,
00073:                 &nand->nfconf);
00074:         else
00075:             writel(readl(&nand->nfconf) | S3C2410_NFCONF_nFCE,
00076:                 &nand->nfconf);

```

找到该结构体的定义如下

```

struct nand_chip *chip = mtd->priv;
struct s3c2410_nand *nand = s3c2410_get_base_nand();

debug("hwcontrol(): 0x%02x 0x%02x\n", cmd, ctrl);

```

而 s3c2410_nand 的结构体的声明如下

```

1: #ifdef CONFIG_S3C2410
2: /* NAND FLASH (see S3C2410 manual chapter 6) */
3: struct s3c2410_nand {
4:     u32 nfconf;
5:     u32 nfcmd;
6:     u32 nfaddr;
7:     u32 nfdata;
8:     u32 nfstat;
9:     u32 nfecc;
10: };
11: #endif
12: #ifdef CONFIG_S3C2440
13: /* NAND FLASH (see S3C2440 manual chapter 6) */
14: struct s3c2440_nand {
15:     u32 nfconf;
16:     u32 nfcont;
17:     u32 nfcmd;
18:     u32 nfaddr;
19:     u32 nfdata;
20:     u32 nfeccd0;
21:     u32 nfeccd1;
22:     u32 nfeccd;
23:     u32 nfstat;
24:     u32 nfstat0;
25:     u32 nfstat1;
26: };
27: #endif

```

发现由于我们之前改串口波特率的时候屏蔽了 CONFIG_S3C2410 的宏,导致了 nand 相关的函数有问题,所以 nand 的驱动需要重新实现

现在我们只想让串口有数据,所以我们将 nand 模块直接屏蔽掉

由于我们刚才看到报错的的是 s3c2410_nand.c 这个文件,我们直接在该文件的同级目录下找到 makefile

发现:

```
COBJS-$(CONFIG_NAND_S3C2410) += s3c2410_nand.o
```

说明这个文件要被编译的话必须依赖于 CONFIG_NAND_S3C2410 这个宏

如暂时不提供 nand flash 支持, 因而不需要吧 s3c2410_nand.c 编进内核, 因而要把 CONFIG_NAND_S3C2410 这个宏定义去掉, 在 u-boot 中搜索可以看到

```
[root@fedora U-boot-2012-04-01]# grep -nr "CONFIG_NAND_S3C2410"
drivers/mtd/nand/Makefile:61:COBJS-$(CONFIG_NAND_S3C2410) += s3c2410_nand.o
include/autoconf.mk:43:CONFIG_NAND_S3C2410=y
include/configs/smdk2410.h:210:#define CONFIG_NAND_S3C2410
include/configs/VCMA9.h:236:#define CONFIG_NAND_S3C2410
include/configs/smdk2440.h:211:#define CONFIG_NAND_S3C2410
```

我们肯定是在 configs/smdk2440.h 里面把这个宏屏蔽掉

```
#ifndef CONFIG_CMD_NAND
#define CONFIG_NAND_S3C2410
#define CONFIG_SYS_S3C2410_NAND_HWECCE
#define CONFIG_SYS_MAX_NAND_DEVICE 1
#define CONFIG_SYS_NAND_BASE 0x4E000000
```

发现 CONFIG_NAND_S3C2410 这个宏 是否被定义是依赖于 CONFIG_CMD_NAND 的宏被定义 ,

因而 // #define CONFIG_CMD_NAND

再次编译

由于我都 uboot 是从官网下下的, 和老师的不一样, 所以还会报一个错:

```
/root/Dev/Work_Space/github/U-boot-2012-04-01/fs/yaffs2/yaffscfg.c:210: undefined reference to `nand_info'
Makefile:472: recipe for target `u-boot' failed
make: *** [u-boot] Error 1
```

同理找对应 makefile:

```
COBJS-$(CONFIG_YAFFS2) := \
    yaffscfg.o yaffs_ecc.o yaffsfs.o yaffs_guts.o yaffs_packedtags1.o \
    yaffs_tagscompat.o yaffs_packedtags2.o yaffs_tagsvalidity.o \
    yaffs_nand.o yaffs_checkptrw.o yaffs_qsort.o yaffs_mtdif.o \
    yaffs_mtdif2.o
```

然后去 smdk2440.h 屏蔽这个宏即可

Q1: 通过老版 uboot 烧些新版 uboot 的时候为什么要先烧到 sdram, 而不是直接烧到 norflash

可不可以烧到 nand 上在复制到 nor 上

A: 这需要你去修改 dnw 的功能, 我们实现 dnw 命令时, 只是把数据通过 USB 口取出保存在内存里, 没有实现直接取到数据烧写到 FLASH 的功能

Q: 通过老版的 uboot 把新版的 uboot 烧到了 sdram 后, 然后擦除 nor, 这样程序不就没了吗, 怎么还可以通过命令行来把 sdram 的新版 uboot 拷贝到 nor?

A: 程序运行在 SDRAM 中, 当然可以把 NOR 上的删掉

Q2 烧些老版 uboot 的时候, 为什么要烧到 norflash, nor 启动和 uboot 是否要在 norflash 上有什么关系?

A2: 如果老版的 uboot 烧些到 nandflash 上, 那么必须用 nand 启动的方式, nand 启动的时候, norflash 是不可用的, (片选引脚被占用), 而新版的

uboot 只支持 nor 启动, 所以必须烧到 nandflash 上

Q: 未修改时钟之前的 sdram 会初始化失败, 如何验证:

A: 通过断点进行调试

毕业班第 2 课第 3.2 节_移植最新 u-boot 之修改代码支持 NAND 启动

原来的代码在链接时加了 "-pie" 选项, 使得 u-boot.bin 里多了 "*(.rel*)", "*(.dynsym)"

使得程序非常大, 不利于从 NAND 启动(重定位之前的启动代码应该少于 4K)

(1) 去掉 "-pie" 选项

可以在 linux 下搜索 -pie 要注意前面的要加 \ grep -nr "\-pie" 我觉得是转义的意思

在文件 arch/arm/config.mk: 75 行 :LDFLAGS_u-boot += -pie 去掉这行

(2) 增加 NAND FLASH 初始化代码

把毕业班里面的 init.c 文件放到\u-boot-2012.04.01\board\samsung\smdk2440 里面，做如下修改

nand_read 改为-》nand_read_ll
nand_init 改为-》nand_init_ll
nand_read 改为-》nand_read_ll

还有第 66 行要也要加上 ll

改为 `nand_read_ll((unsigned int)src, dest, len);`

函数

isBootFromNorFlash

nand_select

nand_deselect

nand_cmd

nand_addr

nand_wait_ready

nand_data

前面都加上 static 表示函数只在本文件使用

(3) 去掉全部串口 0 设置的部分

```
#define PCLK 50000000 // init.c中的clock_init函数
#define UART_CLK PCLK // UART0的时钟源设为PCLK
#define UART_BAUD_RATE 115200 // 波特率
#define UART_BRD ((UART_CLK / (UART_BAUD_RATE * 16)))

/*
 * 初始化UART0
 * 115200, 8N1, 无流控
 */
void uart0_init(void)
{
    GPHCON |= 0xa0; // GPH2, GPH3用作TXD0, RXD0
    GPHUP = 0x0c; // GPH2, GPH3内部上拉

    ULCON0 = 0x03; // SN1 (8个数据位, 无校验, 1个停止位)
    UCON0 = 0x05; // 查询方式, UART时钟源为PCLK
    UFCON0 = 0x00; // 不使用FIFO
    UMCON0 = 0x00; // 不使用流控
    UBRDIVO = UART_BRD; // 波特率为115200
}

/*
 * 发送一个字符
 */
void putc(unsigned char c)
{
    /* 等待, 直到发送缓冲区中的数据已经全部发送出去 */
    while (!(UTRSTAT0 & TXDREADY));

    /* 向UTXH0寄存器中写入数据, UART即自动将它发送出去 */
    UTXH0 = c;
}
```

(4) 添加重定位代码

修改\u-boot-2012.04.01\arch\arm\cpu\arm920t\start.S 文件

旧:

```
/* we do sys-critical inits only at reboot,
 * not when booting from ram!
 */
#ifdef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif

/* Set stackpointer in internal RAM to call board_init_f */
call_board_init_f:
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
    bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
    ldr r0, =0x00000000
    bl board_init_f
```

改为: (添加红色框的代码)

```

/*
 * CONFIG_SKIP_LOWLEVEL_INIT
 */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
#define CONFIG_SKIP_LOWLEVEL_INIT
/*
 * CPU initialization
 */
/*
 * Set stackpointer in internal RAM to call board_init_f */
call_board_init_f:
    ldr r0, =0x00000000
    bl board_init_f
#endif

```

分析

1:之所以 `ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)` 把这两句放到前面只是为了让代码更加有调理,可以不用放

2:为什么 `ldr r1, =_start` 改为 `ldr r1, _TEXT_BASE`?

第一条为伪汇编指令, 当 `_start` 的值很复杂的时候, 存到哪里是由编译器决定的, 自己无法掌握, 有可能就存到 4K 之外去了

第二条 `_TEXT_BASE` 是定死了的

3:最终老师把 `ldr r2, =__bss_start` 这句替换成了 `ldr r2, _bss_start_ofs`, 这两句的作用应该是一样的. 老师最终用的是后者,

在第 5 点有说明

(5) 修改链接地址 (程序运行时应该所在的地址)

修改 `/u-boot-2012.04.01/include/configs/smdk2440.h` 里面 `_TEXT_BASE` 的值:

```
#define CONFIG_SYS_TEXT_BASE 0x0
```

改为

```
#define CONFIG_SYS_TEXT_BASE 0x33f80000
```

(为什么是 `0x33f80000`? 因为 SDRAM 为 64M 就是 `34000000` 减去 `33f80000` 等于 512K 一般 UBOOT 不会超过 512K 就把 UBOOT 放到这个 512K 空间里面去执行, 如果 UBOOT 实在太大会修改 `_TEXT_BASE` 的值 增大程序的存放空间)

回到 `start.S` 文件

再把上图红色框代码的

```

/*
 * CPU initialization
 */
/*
 * Set stackpointer in internal RAM to call board_init_f */
call_board_init_f:
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
    bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
    bl nand_init_ll
    mov r0, #0
    //ldr r1, =_start
    ldr r1, _TEXT_BASE
    ldr r2, =__bss_start
    sub r2, r2, r1
    bl copy_code_to_sdram

```

把蓝色框

```
ldr r2, =__bss_start
```

```
sub r2, r2, r1
```

改为

```

/*
 * CPU initialization
 */
/*
 * Set stackpointer in internal RAM to call board_init_f */
call_board_init_f:
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
    bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
    bl nand_init_ll
    mov r0, #0
    //ldr r1, =_start
    ldr r1, _TEXT_BASE
    ldr r2, _bss_start_ofs
    bl copy_code_to_sdram
    bl clear_bss
    ldr pc, =call_board_init_f

```

其中 `_bss_start_ofs` 为: UBOOT 的大小

结束地址减去开始地址

```
.globl __bss_start_ofs
__bss_start_ofs:
.word __bss_start - _start
```

bl copy_code_to_sdram

然后跳转到 copy_code_to_sdram(unsigned char *src, unsigned char *dest, unsigned int len)

其中 r0=unsigned char *src 源地址 (nand flash 上面的 0 地址)

r1=unsigned char *dest 目标地址 (sdram 上面的 33f80000 地址)

r2 =unsigned int len 长度

这是汇编对 C 语言传参数, 第 1~4 个参数保存在 R0~R3

在传之前也设置了栈

(6) 清除 BSS 段:

先修改 u-boot-2012.04.01\board\samsung\smdk2440\init.c

```
void clear_bss(void)
{
    extern int __bss_start, __bss_end;
    int *p = &__bss_start;
    for (; p < &__bss_end; p++)
        *p = 0;
}
```

在 __bss_end 后面加上两个下划线, 改成

因为链接脚本里面是两个下划线, 保持与 uboot.lds 里面的地址一样

```
3  *(.bss)
4  . = ALIGN(4);
5  __bss_end__ = .;
6  }
7  /DISCARD/ : { *(.dunstr*)
```

继续在 start.S 设置跳转到清除 BSS 函数去

```
bl copy_code_to_sdram
bl clear_bss
```

然后跳到 SDRAM 里面去执行

加入代码:

```
bl copy_code_to_sdram
bl clear_bss

ldr pc,=call_board_init_f

/* Set stackpointer in Internal RAM to call board_init_f */
call_board_init_f:

ldr r0,=0x00000000
bl board_init_f
```

使得程序从片内 4K 内存跳到 SDRAM 去执行了

然后调用 函数 board_init_f 还会去调用里面的重定位代码

但是我们已经做了重定位了

(7) 去除旧代码里面的重定位

所以把 board_init_f 函数里面的重定位去掉

```
display_uart_config(), // and display it //
gd->relocaddr = addr;
gd->start_addr_sp = addr_sp;
gd->reloc_off = addr - TEXT_BASE;
debug("relocation Offset is: %08lx\n", gd->reloc_off);
memcpy(id, (void *)gd, sizeof(gd_t));

//relocate_code(addr_sp, id, addr);

/* NOTREACHED - relocate_code() does not return */
} ? end board_init_f ?
```

(8) 裁剪 start.S

把 start.S 里面的代码

```
*-----*/
/*
 * void relocate_code (addr_sp, gd, addr_moni)
 *
 * This "function" does not return, instead it continues in RAM
 * after relocating the monitor code.
 *
 */

    .globl    relocate_code
relocate_code:

    mov     r4, r0    /* save addr_sp */
    mov     r5, r1    /* save addr of gd */
    mov     r6, r2    /* save addr of destination */

    /* Set up the stack */
stack_setup:

    mov     sp, r4

    adr     r0, _start
    cmp     r0, r6
    beq     clear_bss    /* skip relocation */
    mov     r1, r6        /* r1 <- scratch for copy_loop */
    ldr     r3, _bss_start_ofs
    add     r2, r0, r3    /* r2 <- source end address */

copy_loop:

    ldmia r0!, {r9-r10}    /* copy from source address [r0] */
    stmia r1!, {r9-r10}    /* copy to target address [r1] */
    cmp     r0, r2        /* until source end address [r2] */
    blo     copy_loop

#ifdef CONFIG_SPL_BUILD
/*
 * fix .rel.dyn relocations
 */

    ldr     r0, _TEXT_BASE    /* r0 <- Text base */
    sub     r9, r6, r0        /* r9 <- relocation offset */
    ldr     r10, _dynsym_start_ofs /* r10 <- sym table ofs */
    add     r10, r10, r0        /* r10 <- sym table in FLASH */
    ldr     r2, _rel_dyn_start_ofs /* r2 <- rel dyn start ofs */
    add     r2, r2, r0        /* r2 <- rel dyn start in FLASH */
    ldr     r3, _rel_dyn_end_ofs /* r3 <- rel dyn end ofs */

```



```

        add    r3, r3, r0          /* r3 <- rel dyn end in FLASH */
fixloop:
        ldr    r0, [r2]           /* r0 <- location to fix up, IN FLASH! */
        add    r0, r0, r9          /* r0 <- location to fix up in RAM */
        ldr    r1, [r2, #4]
        and    r7, r1, #0xff
        cmp    r7, #23             /* relative fixup? */
        beq    fixrel
        cmp    r7, #2              /* absolute fixup? */
        beq    fixabs
        /* ignore unknown type of fixup */
        b      fixnext
fixabs:
        /* absolute fix: set location to (offset) symbol value */
        mov    r1, r1, LSR #4      /* r1 <- symbol index in .dynsym */
        add    r1, r10, r1         /* r1 <- address of symbol in table */
        ldr    r1, [r1, #4]        /* r1 <- symbol value */
        add    r1, r1, r9          /* r1 <- relocated sym addr */
        b      fixnext
fixrel:
        /* relative fix: increase location by offset */
        ldr    r1, [r0]
        add    r1, r1, r9
fixnext:
        str    r1, [r0]
        add    r2, r2, #8          /* each rel.dyn entry is 8 bytes */
        cmp    r2, r3
        blo    fixloop
#endif

clear_bss:
#ifdef CONFIG_SPL_BUILD
        ldr    r0, _bss_start_ofs
        ldr    r1, _bss_end_ofs
        mov    r4, r6              /* reloc addr */
        add    r0, r0, r4
        add    r1, r1, r4
        mov    r2, #0x00000000     /* clear          */
        /* clear loop...          */
        clbss_1: str r2, [r0]
        add    r0, r0, #4
        cmp    r0, r1
        bne    clbss_1

```

```

        bl coloured_LED_init

        bl red_led_on

    #endif

```

全部删掉，因为前面我们已经实现了这些功能

(9) 调用第二阶段代码：

```

/* Set stackpointer in internal RAM to call board_init_f */
call_board_init_f:

    ldr r0,=0x00000000
    bl board_init_f
    /*board_init_f的返回值 id 存在 r0里，正好给 board_init_r 使用 */
    ldr r1,TEXT_BASE
    /*调用第二阶段代码 */
    bl board_init_r

```

跳转到第二阶段代码主要是通过 board_init_r 这个函数。这个函数需要两个参数：1.是链接地址也就是我们_TEXT_BASE 2.id

于是我们让 board_init_f 函数的返回值变为 id

```

gd->relocaddr = addr;
gd->start_addr_sp = addr_sp;
gd->reloc_off = addr - TEXT_BASE;
debug("relocation Offset is: %08lx\n", gd->reloc_off);
memcpy(id, (void *)gd, sizeof(gd_t));

//relocate_code(addr_sp, id, addr);

return (unsigned int)id;
/* NOTREACHED - relocate_code() does not return */
} ? end board_init_f ?

```

board_init_f 函数类型改成与返回值相对应 `unsigned int board_init_f(ulong bootflag)`

修改 include\common.h 里面的函数声明

改为：

```

/* arch/${ARCH}/lib/board.c */
unsigned int board_init_f(ulong);
void board_init_r(gd_t *, ulong);
int checkboard(void);

```

把 init.c 放入 board/samsung/smdk2440 目录，

(10) 修改 Makefile

修改/board/samsung/smdk2440/Makefile

增加 init.o

```

COBJS := smdk2410.o init.o
SOBJS := lowlevel init.o

```

(11)修改链接脚本：

把 start.S, init.c, lowlevel.S 等文件放在最前面

因为 init.o, lowlevel.o 会编译成 libsmk2440.o

所以直接加 libsmk2440.o 就可以了

在/arch/arm/cpu/u-boot.lds 加上

```

board/samsung/smdk2440/libsmk2440.o

```

```
SECTIONS
{
    . = 0x00000000;
    . = ALIGN(4);
    .text :
    {
        _image_copy_start = ;
        CPUDIR/start.o (.text)
        board/samsung/smdk2440/libsmdk2440.o (.text)
        *(.text)
    }
}
```

(12)最后我们要修改 arch\arm\lib\board.c

由于我们之前第二阶段的 sp 是由一路计算出来的,而我们之前在 start.s 通过修改宏定义 CONFIG_SYS_TEXT_BASE,强制定义为了 0x300033f8,我们之前的 uboot 的内存图分析的时候的 addr 是计算出来的,指向了 sp,所以我们这里要把它定死

改为:

```
#endif /* CONFIG_FB_ADDR */
#endif /* CONFIG_LCD */

/*
 * reserve memory for U-Boot code, data & bss
 * round down to next 4 kB limit
 */
//addr -= gd->mon_len;
//addr &= ~(4096 - 1);
addr = CONFIG_SYS_TEXT_BASE;

debug("Reserving %ldk for U-Boot at: %08lx\n", gd->mon_len >> 10, addr);
```

烧写的时候 发现这个 uboot 太大了 不止 512K

我们通过 ls uboot.bin 文件的大小

```
oot.dis
book@book-desktop:/work/system/u-boot-2012.04.01$ ls -l u-boot.bin
-rwxr-xr-x 1 book book 433260 2012-05-26 03:28 u-boot.bin
```

只有 400 多 K,但是这是不包含 bss 段的(bin 文件是不包含 bss 段的),要知道程序本身的大小要用_bss_end 减去程序开始才是程序本身的大小

这个值可以通过看反汇编看出来,但是我们在 start.s 中发现:

```
.globl _bss_end_ofs
_bss_end_ofs:
.word _bss_end - _start
```

也就是说我们直接在反汇编搜框中的值即可:

```
14:33f00040 <_bss_end_ofs>:
15:33f80048: 00000000 .word 0x00000000
16:
```

所以我们修改\include\configs\smdk2440.h 里面的

```
#define CONFIG_SYS_TEXT_BASE 0x33f80000
```

改为

```
#define CONFIG_SYS_TEXT_BASE 0x33f00000
```

把程序空间设为 1M

最后我们来看下我的修改成功没有

1:看 dis 文件,找到 nand_init_11 函数有没有超过前面 4K:

```

428 33f80588: <nand_init_ll>:
429 33f80588: e3a0344e      mov r3, #1308622848 ; 0x4e000000
430 33f8058c: e3a02c01      mov r2, #256 ; 0x100
431 33f80590: e5032000      str r2, [r3]
432 33f80594: e3a02013      mov r2, #19 ; 0x13

```

没超过 4K

毕业班第 2 课第 3.3 节_移植最新 u-boot 之修改代码支持 NorFlash

烧写之前的代码,发现串口打印的信息:

```

U-Boot 2012.04.01 (May 26 2012 - 03:31:24)
CPUID: 32440001
FCLK: 400 MHz
HCLK: 100 MHz
PCLK: 50 MHz
DRAM: 64 MiB
WARNING: Caches not enabled
Flash: *** failed ***
## Error ## Please RESET the board ##

```

搜索*** failed *** 发现:是在 uboot 第二阶段代码中打印出的改语句:

Vim 打开文件同时定位到某一行

2013年10月27日 / 综合 / 共 272字 / 字号 小 中 大 / 评论关闭

使用vim 打开文件时,默认情况下光标会停留在文件开头,有时候文件比较大,翻阅和查找都比较麻烦,怎样在打开的时候直接定位到某一行呢?

使用 "+"

vim filename +n 可以在打开文件的同时定位到第n行。

eg: `vim myfile +100` //打开的同时光标停在第100行行首

也可以在打开文件的同时找到第一个匹配的词。

vim filename +/pattern

eg: `vim config.mk +export` //打开的同时光标停在第一个export所在的位置(词首)

[返回](#)

```

static char *failed = "**** failed ***\n";

void board_init_r(gd_t *id, ulong dest_addr)

flash_size = flash_init();

if (flash_size > 0) {

....} else {

    puts(failed);

    hang();

}

```

这是由于我们的 uboot 并不支持我们的 norflash 芯片,所以初始化失败导致报错,更何况我们当前是 nand 启动,norflash 对 CPU 不可见,所以更不可能成功

(1) 去掉卡死代码

先把错误提示信息去掉,改成提示找不到

进入\arch\arm\lib\board.c 529 行

把代码修改成这样

```

}
# else
    puts("\n");
    /* ! CONFIG_SYS_FLASH_CHECKSUM */
    print_size(flash_size, "\n");
# endif /* CONFIG_SYS_FLASH_CHECKSUM */
} ? end if flash_size>0 ? else {
    puts("0KB \n\n");
    //puts(failed);
    //hang();
}
#endif

```

因为 hang()会把程序在这里挂死,所以修改之后,代码就可以继续往下跑,这样就避免 nand 启动时候,会在这里卡死。

(2) 添加调试代码, 获得厂家 ID

我们跟踪初始化的代码, 发现有很多 debug 信息, 发现 debug 是这样的定义的:

```
# ifdef DEBUG
# define debug(fmt,args...) printf(fmt,##args)
# else
# define debug(fmt,args...)
# endif
```

在\drivers\mtd\cfi_flash.c 49 行里面加上 debug 定义 使得 uboot 界面打印调试信息 得到芯片的厂家 ID

```
#include <asm/io.h>
#include <asm/byteorder.h>
#include <environment.h>
#include <mtd/cfi_flash.h>
#include <watchdog.h>

#define DEBUG 1
#define _DEBUG 1
```

这里之所以定义了 DEBUG 和 _DEBUG 是因为不愿意分析了, 两个肯定有一个可以. 所以都定义了

使得 uboot 界面打印调试信息 得到芯片的厂家 ID: 2249

```
fwc addr (null) cmd f0 00f0 16bit x 16 bit
JEDEC PROBE: ID c2, 2249 0 信息 (比如银行帐户)
fwc addr (null) cmd ff 00ff 16bit x 16 bit
fwc addr (null) cmd 90 0090 16bit x 16 bit
fwc addr (null) cmd ff 00ff 16bit x 16 bit
```

进入 flash_size = flash_init(); 去分析为什么 size 为 0

```
flash_init() // Cfi_flash.c (drivers\mtd)
```

```
>> if (!flash_detect_legacy(cfi_flash_bank_addr(i), i)) //用老旧的方法检测 norflash
```

```
>> if (jedec_flash_match(info, info->start[0])) //用 jedec 的方法来匹配看能否找到我们的
```

norflash

```
for (i = 0; i < ARRAY_SIZE(jedec_table); i++) //在 jedec_table 数组里去一个匹配
```

(4) 添加代码, 支持 nor flash 芯片

在\drivers\mtd\jedec_flash.c 中增加代码, 让 uboot 支持我们的 nor flash 芯片

```
00369: #endif
00370: /* JZ2440v2 使用的 MT29LV160DB */
00371: {
00372:     .mfr_id      = (u16)MX_MANUFACT, /* 厂家 ID */ //串口读出来的是 c2
00373:     .dev_id      = 0x2249, /* 设备 ID */
00374:     .name        = "MXIC MT29LV160DB", //这个可以随便写
00375:     .uaddr       = { /* NOR flash 看到解锁地址 */
00376:         [1] = MTD_UADDR_0x0555_0x02AA /* x16 */ //这个是解锁地址, 每发一个命令前都要发送 0x555 和 0x2aa
00377:     },
00378:     .DevSize     = SIZE_2MiB, /* 总大小 */
00379:     .CmdSet      = P_ID_AMD_STD,
00380:     .NumEraseRegions = 4, /* 擦除的类型个数 */
00381:     .regions     = {
00382:         ERASEINFO(16*1024, 1), //16K 的 有 1 个
00383:         ERASEINFO(8*1024, 2),
00384:         ERASEINFO(32*1024, 1),
00385:         ERASEINFO(64*1024, 31),
00386:     },
00387: },
00388: };
00389:
00390: static inline void fill_info(flash_info_t *info, const struct amd_flash_info *jedec_entry, ulong base)
00391: {
```

参数解析:

厂家 ID 设备 ID 都是根据 UB00T 界面打印信息去确定的

芯片名称可以随便取

解锁地址就是 (每发一个命令前先发送 0x555 和 0x2AA)

Command Sequence		Data	Bus Cycles									
			1 st	2 nd	3 rd	4 th	5 th	6 th	7 th	8 th	9 th	10 th
Read	Reset		1	xxx	FD							
Manufacturer ID	Word	4	555	AA	2AA	55	555	90	1000	7F		
	Byte	4	AAA	AA	555	55	AAA		100	1C		
	Word	4	555	AA	2AA	55	555	90	1000	7F		
	Byte	4	555	AA	555	55	AAA		200	1C		
	Word	4	555	AA	2AA	55	555	90	x01	2249		
	Byte	4	555	AA	555	55	AAA		x02	C4		
Device ID	Word	4	555	AA	2AA	55	555	90	x01	2249		
	Byte	4	AAA	AA	555	55	AAA		(SA)	XX00		
Bottom Boot	Word	4	555	AA	2AA	55	555	90	X02	XX01		
	Byte	4	AAA	AA	555	55	AAA		(SA)	05		
Sector Protect	Word	4	555	AA	2AA	55	555	90	X04	01		
	Byte	4	AAA	AA	555	55	AAA					

里面对应 MTD_UADDR_0x0555_0x02AA

555 2AA

.NumEraseRegions= 4 就是表示擦除结构有 8K 16K 32K 64K 几类

```
.regions = {
```

ERASEINFO(0x04000, 1), 16K 1 个

ERASEINFO(0x02000, 2), 8K 2 个

ERASEINFO(0x08000, 1), 32K 1 个

ERASEINFO(0x10000, 31, 64K 31 个

注意: 这里的 8K 16K 32K 64K 对应的是芯片手册上 SIZE 的 (Kbytes

如果看的时候看的是 (Kwords) 则要乘以 2, 就像芯片图中的 4 我们在程序里写的就是 8K

Sector Size (Kbytes/ Kwords)
16/8
8/4
8/4
32/16

具体 8K 16K 32K 64K 分别有多少个, 数一下表中的红色框 就可以了。

Eg:

我们这里的 8K 就看 (Kbytes 下的 8, 得出有 2 个

我们这里的 16K 就看 (Kbytes 下的 8, 得出有 1 个

我们这里的 32K 就看 (Kbytes 下的 8, 得出有 1 个

我们这里的 64K 就看 (Kbytes 下的 8, 得出有 31 个

Table 3. Bottom Boot Sector Address Tables (EN29LV160AB)

Sector	A19	A18	A17	A16	A15	A14	A13	A12	Sector Size (Kbytes/ Kwords)	Address Range (in hexadecimal)		
										Byte mode (x8)	Word Mode (x16)	
SA0	0	0	0	0	0	0	0	X	16/8	000000-003FFF	00000-01FFF	
SA1	0	0	0	0	0	0	0	1	0	8/4	004000-005FFF	02000-02FFF
SA2	0	0	0	0	0	0	0	1	1	8/4	006000-007FFF	03000-03FFF
SA3	0	0	0	0	0	1	X	X	32/16	008000-00FFFF	04000-07FFF	
SA4	0	0	0	0	1	X	X	X	64/32	010000-01FFFF	08000-0FFFF	
SA5	0	0	0	1	0	X	X	X	64/32	020000-02FFFF	10000-17FFF	
SA6	0	0	0	1	1	X	X	X	64/32	030000-03FFFF	18000-1FFFF	
SA7	0	0	1	0	0	X	X	X	64/32	040000-04FFFF	20000-27FFF	
SA8	0	0	1	0	1	X	X	X	64/32	050000-05FFFF	28000-2FFFF	
SA9	0	0	1	1	0	X	X	X	64/32	060000-06FFFF	30000-37FFF	
SA10	0	0	1	1	1	X	X	X	64/32	070000-07FFFF	38000-3FFFF	
SA11	0	1	0	0	0	X	X	X	64/32	080000-08FFFF	40000-47FFF	
SA12	0	1	0	0	1	X	X	X	64/32	090000-09FFFF	48000-4FFFF	
SA13	0	1	0	1	0	X	X	X	64/32	0A0000-0AFFFF	50000-57FFF	
SA14	0	1	0	1	1	X	X	X	64/32	0B0000-0BFFFF	58000-5FFFF	
SA15	0	1	1	0	0	X	X	X	64/32	0C0000-0CFFFF	60000-67FFF	
SA16	0	1	1	0	1	X	X	X	64/32	0D0000-0DFFFF	68000-6FFFF	
SA17	0	1	1	1	0	X	X	X	64/32	0E0000-0EFFFF	70000-77FFF	
SA18	0	1	1	1	1	X	X	X	64/32	0F0000-0FFFFF	78000-7FFFF	
SA19	1	0	0	0	0	X	X	X	64/32	100000-10FFFF	80000-87FFF	
SA20	1	0	0	0	1	X	X	X	64/32	110000-11FFFF	88000-8FFFF	
SA21	1	0	0	1	0	X	X	X	64/32	120000-12FFFF	90000-97FFF	
SA22	1	0	0	1	1	X	X	X	64/32	130000-13FFFF	98000-9FFFF	
SA23	1	0	1	0	0	X	X	X	64/32	140000-14FFFF	A0000-A7FFF	
SA24	1	0	1	0	1	X	X	X	64/32	150000-15FFFF	A8000-AFFFF	
SA25	1	0	1	1	0	X	X	X	64/32	160000-16FFFF	B0000-B7FFF	
SA26	1	0	1	1	1	X	X	X	64/32	170000-17FFFF	B8000-BFFFF	
SA27	1	1	0	0	0	X	X	X	64/32	180000-18FFFF	C0000-C7FFF	
SA28	1	1	0	0	1	X	X	X	64/32	190000-19FFFF	C8000-CFFFF	
SA29	1	1	0	1	0	X	X	X	64/32	1A0000-1AFFFF	D0000-D7FFF	
SA30	1	1	0	1	1	X	X	X	64/32	1B0000-1BFFFF	D8000-DFFFF	
SA31	1	1	1	0	0	X	X	X	64/32	1C0000-1CFFFF	E0000-E7FFF	
SA32	1	1	1	0	1	X	X	X	64/32	1D0000-1DFFFF	E8000-EFFFF	
SA33	1	1	1	1	0	X	X	X	64/32	1E0000-1EFFFF	F0000-F7FFF	
SA34	1	1	1	1	1	X	X	X	64/32	1F0000-1FFFFF	F8000-FFFFF	

(5) 去除扇区太多提示信息

修改\include\configs\smk2440.h 190 行

把#define CONFIG_SYS_MAX_FLASH_SECT (19)

```
#define CONFIG_SYS_MAX_FLASH_SECT (128)
```

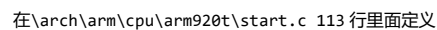
```
Flash: fwc addr (null) cmd f0 00f0 16bit x 16 bit
fwc addr 0000aaaa cmd aa 00aa 16bit x 16 bit
fwc addr 00005554 cmd 55 0055 16bit x 16 bit
fwc addr 0000aaaa cmd 50 0090 16bit x 16 bit
fwc addr (null) cmd f0 00f0 16bit x 16 bit
JEDEC: PROBE_ID c2 2249 0
ERROR: too many flash sectors
2 MiB
```

修改\drivers\mtd\cfi_flash.c 49 去掉 debug 定义

修改完成之后重新编译,然后 nor 启动,因为如果是 nand 启动的话,norflash 不可用,所以用 nor 启动来测试 norflash 是否可以用

经过测试发现:代码不同,查找原因:

此时还要设置把调用第二阶段的 SP 设到到内存高地址上面去



同时还要跳转，重新设置栈

在\arch\arm\lib\board.c 265 里面声明


```

static inline void memcpy_to_user(
    gd_t *id;
    ulong addr, addr_sp;
    extern ulong base_sp;
#ifdef CONFIG_PRAM

```

在\arch\arm\lib\board.c 439行 里面使用

```

gd->reloc_off = addr - TEXT_BASE;
debug("relocation Offset is: %08lx\n", gd->reloc
memcpy(id, (void *)gd, sizeof(gd_t));
base_sp=addr_sp;
//relocate code(addr_sp, id, addr);

```

修改完之后再次测试:成功

毕业班第2课第3.4节_移植最新 u-boot 之修改代码支持 NandFlash

之前在 3.1 课的时候,由于我们在设置时钟的时候把 CONFIG_S3C2410 这个宏关掉了,打开了 CONFIG_S3C2440 这个宏,之后发现 nand 相关的文件报错,然后我们把整个 nand 模块都屏蔽了,当时屏蔽的是 CONFIG_CMD_NAND 这个宏

(1) 添加 NAND 定义

修改: include/configs/smdk2440.h: 102 加上 #define CONFIG_CMD_NAND

```

#define CONFIG_CMD_DUP
#define CONFIG_CMD_ELF
#define CONFIG_CMD_NAND
#define CONFIG_CMD_PING
#define CONFIG_CMD_REGINFO

```

编译后报错:

```

s3c2410_nand.c: In function 's3c2410_hwcontrol':
s3c2410_nand.c:57: warning: implicit declaration of function 's3c2410_get_base_nand'
s3c2410_nand.c:57: warning: initialization makes pointer from integer without a cast
s3c2410_nand.c:72: error: dereferencing pointer to incomplete type
s3c2410_nand.c:72: error: dereferencing pointer to incomplete type
s3c2410_nand.c:75: error: dereferencing pointer to incomplete type
s3c2410_nand.c:75: error: dereferencing pointer to incomplete type
s3c2410_nand.c: In function 's3c2410_dev_ready':

```

原因我们之前在 3.1 分析过,是应为他调用的是 s3c2410 的 nandflash 的结构体,和 s3c2440 的 nandflash 的寄存器有差异

措施:使用 s3c2440 nandflash 寄存器的结构体,

我们直接把 drivers\mtd\nand\s3c2410_nand.c 复制为 s3c2440_nand.c,这样我们接下来就要修改对应目录下的 makefile 让他编译时使用

s3c2440_nand.c 而不是 s3c2410_nand.c

(2) 添加 NAND 的定义及修改对应 Makefile 加选项

在\driver\mtd\nand\Makefile 里面添加 2440 使得编译的时候 s3c2440_nand.o 可以被编译进 UBOOT 里面去

```

COBJS-$(CONFIG_NAND_NOMADIK) += nomadik.o
COBJS-$(CONFIG_NAND_S3C2410) += s3c2410_nand.o
COBJS-$(CONFIG_NAND_S3C2440) += s3c2440_nand.o
COBJS-$(CONFIG_NAND_S3C64XX) += s3c64xx.o

```

这里我们要添加 COBJS-\$(CONFIG_NAND_S3C2440) += s3c2440_nand.o

但是这句生效的前提是 CONFIG_NAND_S3C2440 有定义,并且 CONFIG_NAND_S3C2410 这个宏没有定义,我们可以猜测这个宏肯定是在每个单板的配置文件里面定义的,即 configs/smdk2440.h,我们可以搜索整个 uboot:

```

[root@redora 0 uboot-2012-04-01]# grep -nr CONFIG_NAND_S3C2410
drivers/mtd/nand/Makefile:61:COBJS-$(CONFIG_NAND_S3C2410) += s3c2410_nand.o
include/autoconf.mk:43:CONFIG_NAND_S3C2410=y
include/configs/smdk2410.h:210:#define CONFIG_NAND_S3C2410
include/configs/VCMA9.h:236:#define CONFIG_NAND_S3C2410
include/configs/smdk2440.h:211:#define CONFIG_NAND_S3C2410

```

只有上面几个定义,更加肯定了上面的猜测.

所以接下来为了让 CONFIG_NAND_S3C2410 不生效,而 CONFIG_NAND_S3C2440 生效,我们要修改 configs/smdk2440.h:

```

/*
 * NAND configuration
 */
#ifndef CONFIG_CMD_NAND
#define CONFIG_NAND_S3C2410
#define CONFIG_SYS_S3C2410_NAND_HWECC
#else
#define CONFIG_NAND_S3C2440
#define CONFIG_SYS_S3C2440_NAND_HWECC
#endif

#define CONFIG_SYS_MAX_NAND_DEVICE 1
#define CONFIG_SYS_NAND_BASE 0x4E000000
#endif

```

上面这张图貌似有问题,因为这样的话生效的是 CONFIG_NAND_S3C2410,需要重新看视频确认下,我通过查看老师最终的补丁文件是:

```

+#ifdef CONFIG_CMD_NAND
+
+#ifdef CONFIG_S3C2410
+#define CONFIG_NAND_S3C2410
+#define CONFIG_SYS_S3C2410_NAND_HWECC
+#else
+#define CONFIG_NAND_S3C2440
+#define CONFIG_SYS_S3C2440_NAND_HWECC
+#endif

```

同时在 source insign 添加 s3c2440_nand.c 文件方便阅读 去掉 s3c2410_nand.c

(3) 修改代码

Uboot 关于 nand 驱动的代码其实就是移植内核的,步骤也是设置 nand_chip 结构体,然后让 nand_scan 来调用产生 mtd_info 结构体

从第二阶段代码开始分析:

board_init_r

```

>> nand_init();

#ifdef CONFIG_SYS_NAND_SELF_INIT    //未定义

    board_nand_init();

#else

    int i;

    for (i = 0; i < CONFIG_SYS_MAX_NAND_DEVICE; i++)

        nand_init_chip(i);

#endif

>> nand_init_chip(i);

>>board_nand_init(nand)

    struct s3c2410_nand *nand_reg = s3c2410_get_base_nand(); //这里需要修改为 s3c2440,对应步骤 3.1
    ...

    cfg |= S3C2410_NFCONF_TACLS(tac1s - 1); //修改 nandflash 时序 对应步骤 3.2

    cfg |= S3C2410_NFCONF_TWRPH0(twrph0 - 1);

    cfg |= S3C2410_NFCONF_TWRPH1(twrph1 - 1);

    ...

    //这里下面都是 nandchip 的函数的初始化,这里上面是 nandflash 的初始化,而我们只初始化了时钟,对于 2440
    //来说还不够,还要增加 nfcont 寄存器的初始化,对应步骤 3.3

#ifdef CONFIG_S3C2410_NAND_HWECC

    nand->ecc.hwctl = s3c2410_nand_enable_hwecc;//这下面 3 个函数都是 ecc 代码,可以都删除,对应步骤 3.4
    nand->ecc.calculate = s3c2410_nand_calculate_ecc;
    nand->ecc.correct = s3c2410_nand_correct_data;

```

```

nand->select_chip = NULL; //这里没有设置片选的函数,那么进入后面的 nand_scan 会分配一个函数,
//我们这就直接自己写一个,nand_scan 和我们的 2440 也不能匹配 步骤 3.5

nand->cmd_ctrl = s3c2410_hwcontrol; //这个函数仿照之前的 nandflash 驱动,
//只要实现发送命令或者地址即可 步骤 3.6

nand->dev_ready = s3c2410_dev_ready; //修改读取 nand 状态函数 步骤 3.7

>>nand_scan(mtd, maxchips)

>>nand_scan_ident

>>nand_set_defaults

    chip->select_chip = nand_select_chip;

    chip->cmdfunc = nand_command;

    chip->read_byte = busw ? nand_read_byte16 : nand_read_byte;

>>nand_get_flash_type

    chip->select_chip

    chip->cmdfunc(mtd, NAND_CMD_RESET, -1, -1);

    nand_command() // 可以用来发命令,也可以用来发列地址(页内地址)、行地址(哪一页)

    chip->cmd_ctrl

        s3c2440_hwcontrol

    chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1);

    *maf_id = chip->read_byte(mtd);

    *dev_id = chip->read_byte(mtd);

```

总结:要修改的函数主要就是看 nand_chip 结构体里面主要的函数,如果没有定义,就要看 nand_scan 里面给的默认函数是否合适,不然就要自己写

3.1:修改名称

修改 drivers\mtd\nand\s3c2440_nand.c 129 行 2410 改成 2440

```

int board_nand_init(struct nand_chip *nand)
{
    u_int32_t cfg;
    u_int8_t tacls, twrph0, twrph1;
    struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();
    struct s3c2440_nand *nand_reg = s3c2440_get_base_nand();

    debug("board_nand_init()\n");
}

```

这里的修改后的 `struct s3c2440_nand *nand_reg = s3c2440_get_base_nand();` 是有定义的,不需要重新添加代码

3.2 初始化 Nand Flash 时序

初始化 nand 时序

修改 s3c2440_nand.c 143 行

```

    twrph1 = 8;
#endif
#if 0 //这段代码适合 2410 不适合 2440
    cfg = S3C2410_NFCONF_EN;
    cfg |= S3C2410_NFCONF_TACLS(tacls - 1);
    cfg |= S3C2410_NFCONF_TWRPH0(twrph0 - 1);
    cfg |= S3C2410_NFCONF_TWRPH1(twrph1 - 1);
#endif
//根据 2440手册和 init.c添加下面代码,设置时序
cfg = ((tacls-1)<<12)|((twrph0-1)<<8)|((twrph1-1)<<4);
writel(cfg, &nand_reg->nfconf);

/* initialize nand chip data structure */

```

蓝色框部分代码适合 2410 不适合 2440, 因为 2410 对时序寄存器对应的 bit 位和 s3c2440 有差异, 所以要修改, 自己添加红色框代码来初始化时序。

```
tacls = 4;
twrph0 = 8;
twrph1 = 8;
```

此处的 `twrph1 = 8;`, 可能偏大要是不行就修改为

```
tacls = 0;
twrph0 = 1;
twrph1 = 0;
```

具体大小跟自己板子搭载的 NAND FLASH 有关 若不知道就取 4 8 8 保留原来值

为什么 `cfg = ((tacls-1)<<12)|((twrph0-1)<<8)|((twrph1-1)<<4);` 都有减一?

因为 2440 手册上面

TWRPH0	[10:8]	TWRPH0 duration setting value (0~7) Duration = HCLK x (TWRPH0 + 1)
TWRPH1	[6:4]	TWRPH1 duration setting value (0~7) Duration = HCLK x (TWRPH1 + 1)

都有+1 了。比如要设置 4 个时钟。代码里面减一 公式里面加一 刚好就是 4

3.3 继续添加初始化代码

在前面初始化时序之后, 还要初始化 NFCONF 寄存器里面的值。

```
cfg |= S3C2410_NFCONF_TWRPH1(TWRPH1 - 1);
#endif
//根据 2440手册和 init.c添加下面代码, 设置时序
cfg = ((tacls-1)<<12)|((twrph0-1)<<8)|((twrph1-1)<<4);
writel(cfg, &nand_reg->nfconf);

/*使能 NAND Flash控制器, 初始化 ECC, 禁止片选*/
writel ((1<<4)|(1<<1)|(1<<0), &nand_reg->nfcont);
```

3.4 删除 ECC

再把 ECC 的代码全部删掉

```
#ifdef CONFIG_S3C2410_NAND_HWECCECC
void s3c2410_nand_enable_hwecc(struct mtd_info *mtd, int mode)
{
    struct s3c2410_nand *nand = s3c2410_get_base_nand();
    debug("s3c2410_nand_enable_hwecc(%p, %d)\n", mtd, mode);
    writel(readl(&nand->nfconf) | S3C2410_NFCONF_INITECC, &nand->nfconf);
}

static int s3c2410_nand_calculate_ecc(struct mtd_info *mtd, const u_char *dat,
u_char *ecc_code)
{
    struct s3c2410_nand *nand = s3c2410_get_base_nand();
    ecc_code[0] = readb(&nand->nfeccl);
    ecc_code[1] = readb(&nand->nfeccl + 1);
    ecc_code[2] = readb(&nand->nfeccl + 2);
    debug("s3c2410_nand_calculate_hwecc(%p,); 0x%02x 0x%02x 0x%02x\n",
mtd, ecc_code[0], ecc_code[1], ecc_code[2]);
    return 0;
}

static int s3c2410_nand_correct_data(struct mtd_info *mtd, u_char *dat,
u_char *read_ecc, u_char *calc_ecc)
{
    if (read_ecc[0] == calc_ecc[0] &&
read_ecc[1] == calc_ecc[1] &&
read_ecc[2] == calc_ecc[2])
return 0;

    printf("s3c2410_nand_correct_data: not implemented\n");
    return -1;
}
#endif
```

3.5 自己添加选中函数 s3c2440_nand_select

修改 s3c2440_nand.c 125 行 去掉 NULL 加上

```
s3c2440_nand_select;
```

```

/* initialize nand_chip data structure */
nand->IO_ADDR_R = (void *)&nand_reg->nfdma;
nand->IO_ADDR_W = (void *)&nand_reg->nfdma;

nand->select_chip = s3c2440_nand_select;

/* read_buf and write_buf are default */
/* read_byte and write_byte are default */
#ifdef CONFIG_NAND_SPL
nand->read_buf = nand_read_buf;
#endif

```

我们要自己来写选中函数

下面自己写选中函数 s3c2440_nand_select

参考 nand_base.c 206 行的 nand_select_chip 函数,也就是 nand_scan 里面给我们分配的默认函数

```

static void nand_select_chip(struct mtd_info *mtd, int chipnr)
{
    struct nand_chip *chip = mtd->priv;

    switch (chipnr) {
    case -1:
        chip->cmd_ctrl(mtd, NAND_CMD_NONE, 0 | NAND_CTRL_CHANGE);
        break;
    case 0:
        break;
    default:
        BUG();
    }
}

```

拷贝 复制到 s3c2440_nand.c 83 行 s3c2440_dev_ready 函数下面 添加拷贝代码 然后再修改成

```

static void s3c2440_nand_select(struct mtd_info *mtd, int chipnr)
{
    struct s3c2440_nand *nand = s3c2440_get_base_nand();

    switch (chipnr) {
    case -1: /* 取消选中 */
        nand->nfcnt |= (1<<1);
        break;
    case 0: /* 选中 */
        nand->nfcnt &= ~(1<<1);
        break;
    default:
        BUG();
    }
}

```

3.6 修改 s3c2440_hwcontrol 函数

修改 s3c2440_nand.c 里面的 s3c2440_hwcontrol 函数,把里面原来的大部分代码删除掉,自己添加代码,使其有注释上面的功能

(注意把参数上面的 cmd 改成 dat)

Nandchip 中的函数原型是 void (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);

```

static void s3c2440_hwcontrol(struct mtd_info *mtd, int dat, unsigned int ctrl)
{
    struct s3c2440_nand *nand = s3c2440_get_base_nand();

    if (ctrl & NAND_CLE)
    {
        /* 发命令 */
        writb(dat, &nand->nfcmd); // writb 后面的 b 表示只写寄存器后面的 8 位
    }
    else if (ctrl & NAND_ALE)
    {
        /* 发地址 */
        writb(ctrl, &nand->nfaddr);
    }
}

```

3.7 修改 s3c2440_dev_ready 函数

修改函数 s3c2440_dev_ready 为 s3c2440_dev_ready,并修改内容如下:

```
static int s3c2440_dev_ready(struct mtd_info *mtd)
{
    struct s3c2440_nand *nand = s3c2440_get_base_nand();
    debug("dev_ready\n");
    return readl(&nand->nfst) & 0x01;
}
```

解析:

其中, 原来的 s3c2440.h 代码里面, 已经帮我们定义好了 s3c2440_get_base_nand 返回值的是 S3C2440_NAND_BASE

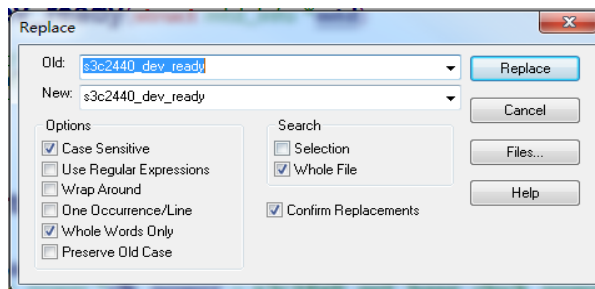
虽然和 2410 的一样, 但还是修改了好看点

```
static inline struct s3c2440_nand *s3c2440_get_base_nand(void)
{
    return (struct s3c2440_nand *)S3C2440_NAND_BASE;
}

#define S3C2440_NAND_BASE 0x4E000000
```

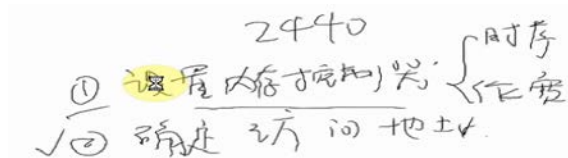
它是 0x4E000000 是 NFCONF 控制器的基地址

再把文件中的 s3c2410_dev_ready 全部替换成 s3c2440_dev_ready, 由于是 static 函数, 只要修改本文件即可



毕业班第 2 课第 3.5 节_移植最新 u-boot 之修改代码支持 DM9000 网卡

因为网卡是内存接口类 所以我们需要设置:



我们还是从入口函数开始分析:

```
eth_initialize
    >>board_eth_init
        >>cs8900_initialize
```

```
3:
4: #ifdef CONFIG_CMD_NET
5: int board_eth_init(bd_t *bis)
6: {
7:     int rc = 0;
8: #ifdef CONFIG_CS8900
9:     rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
10: #endif
11:     return rc;
12: }
13: #endif
```

我们发现这个代码里面是 CS8900 的网卡, 并非我们的 DM9000, 而我们的 u-boot 中里面其实已经实现了 DM9000 的网卡驱动, 只不过现在没有使用

现在我们先让他们使用我们的 DM9000 的驱动, 最后再将上图中的 init 函数换成我们 DM9000 中的初始化函数

之前我们在移植网卡驱动的时候发现网卡的驱动是在 driver/net 中, 我们就直接进去看看 net 目录下有没有 dm9000 的驱动, 如果有, 再改该层的 makefile

(1) 在 makefile 里面添加网卡定义宏。

查看\driver\net\Makefile 文件 发现里面已经定义了 所以不用再添加:

```
COBJS- $(CONFIG_DESIGNWARE_ETH) += designware.o
COBJS- $(CONFIG_DRIVER_DM9000) += dm9000x.o
COBJS- $(CONFIG_DNET) += dnet.o
```

解析这里宏的作用:

意思就是 只要定义了宏:CONFIG_DRIVER_DM9000

Dm9000x.o 就会被编译进 UB00T 里面去

(2) 确定访问地址

A. 添加 DM9000 的宏

在配置文件 \include\configs\smdk2440.h 里面 我们发现有了 DM8900 的网卡开关,去掉 DM8900 的宏加上 DM9000 的宏

以及参考其他配置文件对 DM9000 宏的定义

比如\ include\configs\davinci_dm355evm.h

里面的

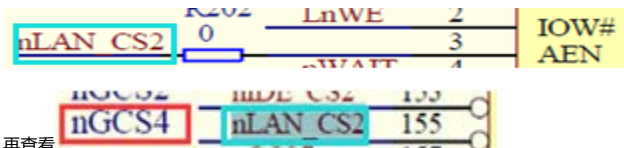
```
/* Ethernet: external DM9000 */
#define CONFIG_DRIVER_DM9000 1
#define CONFIG_DM9000_BASE 0x04014000
#define DM9000_IO CONFIG_DM9000_BASE
#define DM9000_DATA (CONFIG_DM9000_BASE + 2)
```

CONFIG_DM9000_BASE 表示的是基地址, 网卡是内存接口

查看网卡 DM9000 接在第几个 BANK 上面

开发板使用的是 DM9000

查看开发板原理图 DM9000 部分



再查看

得出 DM9000 接在内存 BANK4 处

再看 2440 芯片手册



得出 BANK4 的基地址就是 0x20000000

所以

```
#define CONFIG_DM9000_BASE 0x20000000
#define DM9000_IO CONFIG_DM9000_BASE
```

DM9000_DATA 对应开发板原理图的

就是 bit 2 ,1,0

即 1 0 0 就是 4

所以

```
#define DM9000_DATA (CONFIG_DM9000_BASE + 4)
```

所以使用宏开关“#if0 #else”去掉 DM8900 的宏定义部分之后,在\include\configs\smdk2440.h 里对 DM9000 的设置和定义为:


```

~/
#define CONFIG_CS8900 /* we have a CS8900 on-board */
#define CONFIG_CS8900_BASE 0x19000300
#define CONFIG_CS8900_BUS16 /* the Linux driver does accesses as short:
#else
#define CONFIG_DRIVER_DM9000
#define CONFIG_DM9000_BASE 0x20000000
#define DM9000_IO CONFIG_DM9000_BASE
#define DM9000_DATA (CONFIG_DM9000_BASE + 4)
#endif

```

B. 确定位宽和时序...

确定位宽:

对应的是 start.S 文件里面跳转的 lowlevel_init: 设置的

lowlevel_init: 函数在 \board\samsung\smdk2440\lowlevel_init.S 里面设置的 BWSCON

```

SMRDATA:
    .long 0x22011110 //BWSCON
    .long 0x00000700 //BANKCON0
    .long 0x00000700 //BANKCON1

```

此时设置位宽就是去设置 BWSCON 里面的

DW4	[17:16]	Determine data bus width for bank 4. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved
-----	---------	---

因为 [17:16] 对面的是 BANK4 的位宽设置 也就是我们的 DM9000 网卡位宽, 设置为 16Bit

设置时序, 就是去设置 BANKCON4 寄存器, 此处先采用 DM9000 驱动视频里面的设置值

```

    .long 0x00000700 //BANKCON2
    .long 0x00000700 //BANKCON3
    .long 0x00000740 //BANKCON4
    .long 0x00000700 //BANKCON5

```

直接设置为

(3) 初始化网卡 DM9000

进入 \board\samsung\smdk2440\smdk2410.c 141 行(这里的 2410 名字暂时还没有修改)

增加代码

```

#ifdef CONFIG_CS8900
    rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
#endif
#ifdef CONFIG_DRIVER_DM9000
    rc = dm9000_initialize(bis);
#endif
return rc;
}

```

意思是, 如果定义了宏 CONFIG_DRIVER_DM9000

就会调用 dm9000_initialize 实现 DM9000 的初始化

(注意: 开发板设置 IP 地址 网关 子网掩码之类参数之后 先不要设置 SAVE 命令 因为我们还没有指定参数存放在哪里, 由于我们的 flash 的空间被被分为了 uboot, kernel, env, root 区域, 我们还没指定只写区域地址, 所以 save 的时候可能会崩溃)

毕业班第2课第4.1节_移植最新u-boot之裁剪和修改默认参数

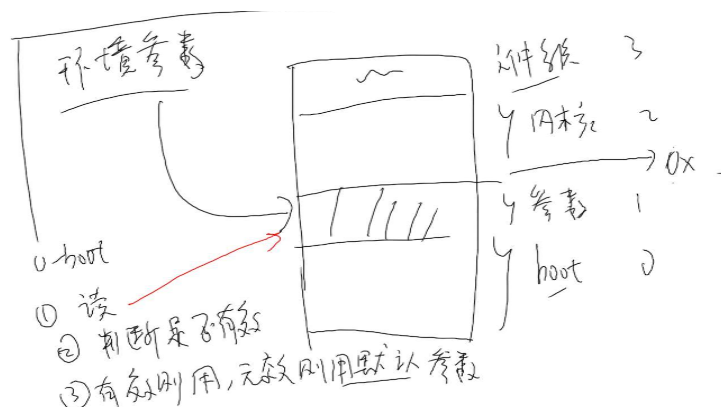
(1) 设置启动参数:

在\comomn\env_common.c 里面有各种可以设置的参数

```
const uchar default_environment[] = {
#ifdef CONFIG_BOOTARGS
    "bootargs=" CONFIG_BOOTARGS "\0"
#endif
#ifdef CONFIG_BOOTCOMMAND
    "bootcmd=" CONFIG_BOOTCOMMAND "\0"
#endif
#ifdef CONFIG_RAMBOOTCOMMAND
    "ramboot=" CONFIG_RAMBOOTCOMMAND "\0"
#endif
#ifdef CONFIG_NFSBOOTCOMMAND
    "nfsboot=" CONFIG_NFSBOOTCOMMAND "\0"
#endif
#ifdef CONFIG_BOOTDELAY && (CONFIG_BOOTDELAY >= 0)
    "bootdelay=" MK_STR(CONFIG_BOOTDELAY) "\0"
#endif
#ifdef CONFIG_BAUDRATE && (CONFIG_BAUDRATE >= 0)
    "baudrate=" MK_STR(CONFIG_BAUDRATE) "\0"
#endif
#ifdef CONFIG_LOADS_ECHO
    "loads_echo=" MK_STR(CONFIG_LOADS_ECHO) "\0"
#endif
#ifdef CONFIG_ETHADDR
    "ethaddr=" MK_STR(CONFIG_ETHADDR) "\0"
#endif
#ifdef CONFIG_ETH1ADDR
```

但是具体的设置 我们放到 smdk2440.h 里面去设置

对于 FLASH, 规划大致是这样的



```
#define CONFIG_BOOTARGS "console=ttySAC0 root=/dev/mtdblocks3 " //这里的 mtdblocks3 就是指给文件系统留出来的位置
```

这句代码 BOOTARGS 定义了传给内核的参数:

console=ttySAC0 内核打印信息从串口 0 里面输出出来显示

root=/dev/mtdblocks3 文件系统在第四个分区 (文件系统 3) 那里

这里是不是书写有问题 mtblock 没有 s 好像

继续在 smdk2440.h 添加代码

```
#define CONFIG_BOOTARGS "console=ttySAC0 root=/dev/mtdblocks3 "
#define CONFIG_BOOTCOMMAND "nand read 30000000 0xabc 0x200000 ;bootm 30000000 "
```

解析:

```
#define CONFIG_BOOTCOMMAND "nand read 30000000 0xabc 0x200000 ;bootm 30000000 "
```

CONFIG_BOOTCOMMAND 表示的是 内核启动命令, 意思就是内核的启动是通过这条命令启动的。

nand read 30000000 : 读到哪里去---》读到 30000000 地址去

(30000000 是 SDRAM 的起始地址)

0xabc : 从哪里读--》NAND FLASH 上的地址

0x200000 : 读多大--》读 2M

bootm 30000000 : 从 30000000 启动内核

(注意的是, 由于我们的地址还没有规划好, 所以这里的参数 都是先假定的值 0xabc, 先暂时这样写, 即使是有问题的地址, 但是格式是正确, 我们在后面设置了分区之后 再来修改值)

(2) 设置波特率和倒数计时:

在 env_common.c 里面

```
#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    "bootdelay=" MK_STR(CONFIG_BOOTDELAY) "\0"
#endif
#if defined(CONFIG_BAUDRATE) && (CONFIG_BAUDRATE >= 0)
    "baudrate=" MK_STR(CONFIG_BAUDRATE) "\0"
```

这些都可以设置的, 这里暂时先不设置

设置网络 IP:

在 smdk2440.h 126 行

```
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.0.107
#define CONFIG_SERVERIP     192.168.0.6
#define CONFIG_ETHADDR      00:0c:29:4d:e4:f4
```

解析:

CONFIG_NETMASK : 子网掩码 一般都是 255.255.255.0

CONFIG_IPADDR : 一般设置板子 IP, 必须与服务器 IP 同一网段 (只能最后的数不同)

CONFIG_SERVERIP: 服务器 IP -> PC 机 TFTP 服务器或者 LINUX 系统的 IP

CONFIG_ETHADDR: 网卡 IP 一般在 Linux 系统找个类似的就可以了

(3) 裁剪 UB00T, 去掉一些暂时不需要的功能

去掉 USB

Smdk244.h 80 行

```
/*
 * USB support (currently only works with D-cache off)
 */
// #define CONFIG_USB_OHCI
// #define CONFIG_USB_KEYBOARD
// #define CONFIG_USB_STORAGE
// #define CONFIG_DOS_PARTITION
```

去掉 RTC 87 行

```
// #define CONFIG_RTC_S3C24X0
```

由于 CONFIG_CMD_DATE 函数要用到 RTC

所以把 CONFIG_CMD_DATE 107 行也去掉

```
#define CONFIG_CMD_BSP
#define CONFIG_CMD_CACHE
// #define CONFIG_CMD_DATE
```

去掉 BOOTP 94 行

```
/*
 * BOOTP options
 */
// #define CONFIG_BOOTP_BOOTFILESIZE
// #define CONFIG_BOOTP_BOOTPATH
// #define CONFIG_BOOTP_GATEWAY
// #define CONFIG_BOOTP_HOSTNAME

/*
 * Command line configuration.
```

去掉 DHCP USB 108 行

```

#define CONFIG_CMD_DATE
// #define CONFIG_CMD_DHCP
#define CONFIG_CMD_ELF
#define CONFIG_CMD_NAND
#define CONFIG_CMD_PING
#define CONFIG_CMD_REGINFO
// #define CONFIG_CMD_USB

```

使用 if0 宏开关去掉下面这一段 236 行

```

/*
 * if 0
 */
#define CONFIG_CMD_FAT
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_UBI
#define CONFIG_CMD_UBIFS
#define CONFIG_CMD_MTDPARTS
#define CONFIG_MTD_DEVICE
#define CONFIG_MTD_PARTITIONS
#define CONFIG_YAFFS2
#define CONFIG_RBTREE
#endif
/* additions for raw relocation code */

```

(4) 设置参数分区位置

接下来，我们要设置这些参数存放在哪里了，首先我们的划分根据内核划分框架来划分就可以了，所以先启动内核，打印出内核的划分：

(我们这里划分的分区是不会影响到内核里面的分区，我们只是根据内核的划分结构来划分而已)

```

内核打印出来的分区信息
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"

```

接下来我们要在代码里面设置

首先使用 IF0 宏开关去掉 UBOOT 本身设置好的参数存放环境变量，我们要使用我们自己设置的。

Smdk2440.h 202 行

```

/*
 * if 0
 */
#define CONFIG_ENV_ADDR (CONFIG_SYS_FLASH_BASE + 0x070000)
#define CONFIG_ENV_IS_IN_FLASH
#define CONFIG_ENV_SIZE 0x10000
/* allow to overwrite serial and ethaddr */
#define CONFIG_ENV_OVERWRITE
#endif

```

而上面的 CONFIG_ENV_IS_IN_FLASH 这里是指 norflash 里面保存环境变量而如果要保存在 nand 上应该怎么写呢？下面有分析

再在下面添加我们自己的设置：

```

/*
 * if 0
 */
#define CONFIG_ENV_IS_IN_NAND
#define CONFIG_ENV_OFFSET 0x00040000
#define CONFIG_ENV_SIZE 0x20000
#define CONFIG_ENV_RANGE CONFIG_ENV_SIZE

```

```

#define CONFIG_ENV_OFFSET 0x00040000

```

：设置参数存放地址为 0x00040000（这个地址是在 NAND FLASH 上的）

```

#define CONFIG_ENV_SIZE 0x20000

```

：设置大小

必须是 128K 的几倍 因为 NAND FLASH 的操作是以块为单位的 我们这里使用 NAND FLASH 是 128K 一块

```

#define CONFIG_ENV_RANGE CONFIG_ENV_SIZE

```

: 擦除大小

在设置之前要先擦除 这个大小就看我们参数的大小了 也就是 CONFIG_ENV_SIZE

:怎么找到这个宏: #define CONFIG_ENV_IS_IN_NAND?

首先在 uboot 控制台查看 SAVE 命令的帮助信息

再根据 SAVE 的帮助信息在 source insignd 找到 uboot 源码里面对应的文件 env_nand.c

查看这个文件下面的 Makefile

可得: 

意思就是设置了这个宏 env_nand.c 就会被编译进 UB00T

所以#define CONFIG_ENV_IS_IN_NAND

(5) 开始分配分区还有命名 (在 UB00T 命令操作的时候 用分区名字代替地址)

原来我们是使用命令:

```
nand erase 60000 200000
```

```
nand write 30000000 60000 200000
```

来实现下载内核之类的命令

现在为了方便 我们要用命令:

```
tftp 30000000 uImage
```

```
nand erase.part kernel
```

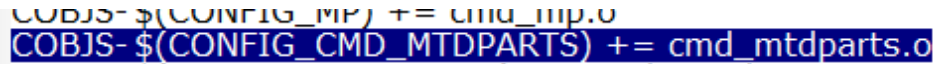
```
nand write 30000000 kernel
```

来实现下载内核 这样比较方便

解决方法就是, 使用 mtdparts 命令

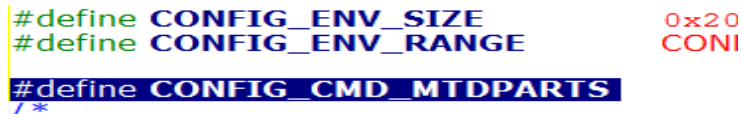
我们知道所有的 uboot 的命令都在 common 文件夹里面,我们直接搜索该目录下是否有 mtdparts 文件

由 common/Makefile 得:



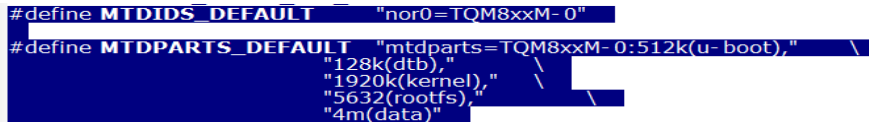
所以我们要定义了这个宏, mtdparts 命令才会变编译进 UB00T 里面去

在 smdk2440.h 215 行 里面定义这个宏

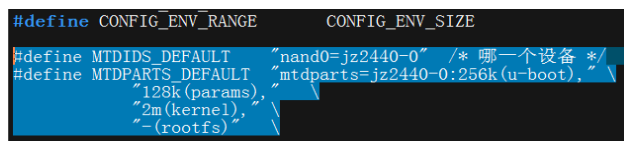


分区设置:

参考\include\configs\omap3_logic 273 行



拷贝到 smdk2440.h 215 行 修改为:



由于 mtdparts 命令要使用到 mtdcore.c 这个文件里面的函数

由 \drivers\mtd\Makefile 28 行得知要编译 mtdcore.c 进 uboot

要定义宏 CONFIG_MTD_DEVICE 才可以

```
COBJS-$(CONFIG_MTD_DEVICE) += mtdcore.o
COBJS-$(CONFIG_MTD_PARTITIONS) += mtdpart.o
```

所以我们要在 smdk2440.h 215 行里面定义

```
#define CONFIG_CMD_MTDPARTS
#define CONFIG_MTD_DEVICE
#define MTDIDS_DEFAULT "nand"
#define MTDPARTS_DEFAULT "mtdparts=0x00000000:32768@0x00000000"
```

(5) 初始化分区：

我们要让开发板 uboot 一开始就帮我们设置好 我们规划好的分区

在 \arch\arm\lib\board.c 656 里面初始化

添加：

```
#endif
run_command("mtdparts default",0);
/* main loop() can return to retrv autoboot.
```

最后我们可以设置我们的 bootcmd 让 kernel 分区读我们的内核了。

在 smdk2440.h 修改代码

```
#define CONFIG_BOOTARGS "console=ttySAC0 root=/dev/mtdblocks3"
#define CONFIG_BOOTCOMMAND "nand read 30000000 0xabc 0x200000 ;bootm 30000000"
/*-----
```

修改为

```
#define CONFIG_BOOTCOMMAND "nand read 30000000 kernel ;bootm 30000000 "
```

```
#define CONFIG_BOOTCOMMAND "nand read 30000000 0xabc 0x200000 ;bootm 30000000 "
```

CONFIG_BOOTCOMMAND 表示的是 内核启动命令，意思就是内核的启动是通过这条命令启动的。

nand read 30000000 : 读到哪里去---》读到 30000000 地址 () 去

0xabc : 从哪里读--》kernel 分区

bootm 30000000 : 从 30000000 启动内核

为什么不用设置大小？因为我们设置 kernel 分区的时候 我们已经指定了 kernel 分区的大小了

毕业班第 2 课第 4.2 节_移植最新 u-boot 支持烧写 yaffs 映象及制作补丁

1、先看 u-boot 能否支持烧写 jffs 文件系统

(1) 先用 tftp 命令下载文件系统到 SDRAM ,然后擦除 nand flash 的 rootfs 分区，然后把文件系统烧写到 0x00260000 里面，0x00260000 不能直接用 rootfs 替换。因为下载的文件肯定会小于 rootfs 分区，如果用 rootfs 写肯定会把不相干的内容都写进去

(2) 先下载查看文件的大小

(3) 查看代码

在此文件中搜索.yaffs,需要定义宏 CONFIG_CMD_NAND_YAFFS

所以修改配置文 `include\configs\smdk2440.h`

```
#define CONFIG_CMD_NAND_YAFFS|
```

各命令间用分号隔开，这样可以全部执行所有的命令。下载 u-boot.bin 到 nor flash

重启后下载文件系统

擦除然后烧写

启动后出现以下错误，虽然可以挂载上去，但说找不到 init 程序。

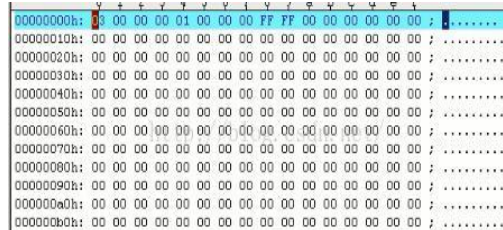
```

block 184 is bad
VFS: Mounted root (yaffs filesystem) .
Freeing init memory: 140K
Warning: unable to open an initial console.
Kernel panic - not syncing: No init found. Try passing init= option to kernel.

```

3. 对比文件 fs_mini_mdev.yaffs2

(1) 用 UE 工具打开 fs_mini_mdev.yaffs2 文件

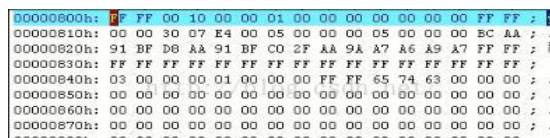


Hex editor view of fs_mini_mdev.yaffs2 file. The address range is from 00000000h to 0000000bh. The data is mostly 00 bytes, with some FF bytes at the end of each line.

文件里面的 OOB, 因为每一页是 2048, 按下 Ctrl+H 转到 2048 处



但发现与在 nand flash 地址 260000 的地方读到的 OOB 内容不一样



Hex editor view of nand flash address 260000. The address range is from 00000000h to 0000000bh. The data is mostly 00 bytes, with some FF bytes at the end of each line.

(2) 用 nand dump 命令查看烧写到 nand flash 的 260000 的地方的文件内容

看页数据

```

SMDK2410 # nand dump 260000
Page 00260000 dump:
03 00 00 00 01 00 00 00 ff ff 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

看 OOB

```

OOB:
ff ff ff ff 00 10 00 00
01 00 00 00 00 00 00 00
ff ff 00 00 30 07 e4 00
05 00 00 00 05 00 00 00
bc aa 91 bf d8 aa 91 bf
aa 9a a7 a6 a9 a7 ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff

```

(3) 查看 cmd_nand.c

如果是 yaffs 文件的话, 调到函数 nand_write_skip_bad 烧写

```

else if (!strcmp(s, ".yaffs")) {
    if (read) {
        printf("Unknown nand command suffix '%s'.\n", s);
        return 1;
    }
    ret = nand_write_skip_bad(nand, off, &rwsize,
        (u_char *)addr, WITH_YAFFS_OOB);
}

```

函数 `nand_write_skip_bad` 中 `rval` 参数有问题, 如果返回 0 是成功的, 返回非零值是失败的。第二个问题是我们的 OOB 要用原始的。(在 `Nand_util.c` 里面), 这两点是老师直接告诉我们的, 并没有讲述过程, 步骤也是和上面一样通过对比内容来找问题的

```

ops.len = pagesize;
ops.ooblen = nand->oobsize;
ops.mode = MTD_OOB_AUTO;
ops.ooboffs = 0;

pages = write_size / pagesize_oob;
for (page = 0; page < pages; page++) {
    WATCHDOG_RESET();

    ops.datbuf = p_buffer;
    ops.oobbuf = ops.datbuf + pagesize;

    rval = nand->write_oob(nand, offset, &ops);
    if (rval)
        break;
}

```

因而更改如下

```

ops.len = pagesize;
ops.ooblen = nand->oobsize;
ops.mode = MTD_OOB_RAW;
ops.ooboffs = 0;

pages = write_size / pagesize_oob;
for (page = 0; page < pages; page++) {
    WATCHDOG_RESET();

    ops.datbuf = p_buffer;
    ops.oobbuf = ops.datbuf + pagesize;

    rval = nand->write_oob(nand, offset, &ops);
    if (rval) /* weidongshan@qq.com */
        break;
}

```

(4) 重新编译, 下载烧写

```

SMDK2410 # tftp 30000000 u-boot_new.bin; protect off all; erase 0 3ffff; cp.b 30000
000 0 40000

```

(5) 用 `reset` 命令重启 u-boot, 然后烧写文件系统

下载、擦除、烧写、启动

```

SMDK2410 # tftp 30000000 fs_mini_mdev.yaffs2
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:0c:29:4d:e4:f4
could not establish link
Using dm9000 device
TFTP from server 192.168.1.3; our IP address is 192.168.1.17
Filename 'fs_mini_mdev.yaffs2'.
Load address: 0x30000000
Loading: #####
#####
#####
#####
#####
#####
#####
#####
#####
#####
done
Bytes transferred = 8952768 (889bc0 hex)

```

```

SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Skipping bad block at 0x00a60000
Skipping bad block at 0x01940000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.yaffs 30000000 260000 889bc0

NAND write: device 0 offset 0x260000, size 0x889bc0
Skip bad block 0x00a60000
8952768 bytes written: OK

```

4. 制作 u-boot 补丁

(1) 先 make distclean 清除掉生成的目标文件

```

book@book-desktop:/work/system/u-boot-2012.04.01$ make distclean
book@book-desktop:/work/system/u-boot-2012.04.01$ ls
\      config.mk  dts      MAKEALL  post      u-boot.dis
api    COPYING     examples Makefile  README
arch   CREDITS      fs       mkconfig rules.mk
board  disk        include  nand_spl snapshot.commit
boards.cfg doc        lib      net      spl
common drivers    MAINTAINERS onenand ipl tools
book@book-desktop:/work/system/u-boot-2012.04.01$ rm u-boot.dis

```

(2) 把 u-boot 重新命名, 以防重复

```

book@book-desktop:/work/system$ mv u-boot-2012.04.01 u-boot-2012.04.01_100ask
book@book-desktop:/work/system$

```

(3) 解压未经修改的 u-boot, 用 diff 命令制作补丁, 前面的是未经修改的 u-boot, 后面是修改后的 u-boot, 生成的是补丁文件。

```

book@book-desktop:/work/system$ diff -uRn u-boot-2012.04.01 u-boot-2012.04.01_100ask
k > u-boot-2012.04.01_100ask.patch

```

(4) 打补丁、配置、编译 (make)

我们再将自己制作的补丁烧到原始的文件里, 从小配置编译, 看看是否效果和修改的一致

```

book@book-desktop:/work/system$ cd u-boot-2012.04.01
book@book-desktop:/work/system/u-boot-2012.04.01$ patch -p1 < ../u-boot-2012.04.01_100ask.patch
book@book-desktop:/work/system/u-boot-2012.04.01$ make smdk2440_config
Configuring for smdk2440 board.

```

毕业班第 3 课移植 3.4.2 内核及制作文件系统

毕业班第 3 课第 1 节_移植 3.4.2 内核之框架介绍及简单修改

下载新版本内核:

内核的源代码可在 <http://www.kernel.org> 中下载

Protocol	Location
HTTP	https://www.kernel.org/pub/
Git	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

打开这个网址, 然后点击

, 然后 linux->kernel->v3.x 就可以找到 3 开头的版本的内核

1. 启动过程分析

1.1: 从毕业班移植新内核里面分析是如何启动内核的 (新的 u-boot 的启动内核的流程和老的 u-boot 有些差异, 这里分别分析)

我们 u-boot 启动内核的时候都是调用命令 bootm 命令

我们直接搜索文件 cmd_bootm.c 看这个命令是如何执行来启动 kernel 的

uboot 下使用 bootm 命令启动内核镜像文件 uImage, uImage 是在 zImage 头添加了 64 字节的镜像信息供 uboot 解析使用, 具体这 64 字节头的内容, 我们在分析 bootm 命令的时候就会——说到, 那直接来看 bootm 命令。

在 common/cmd_bootm.c 中

```
int do_bootm(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
{
    boot_fn = boot_os(images.os.os); // 进入这个boot_os的函数数据
    >>> do_bootm_linux // bootm.c (larch/arm/lib)
    boot_jump_linux(images):
        kernel_entry(v, machid, rz); // 执行内核
}
```

1.2: 从毕业班自己写的 uboot 里面分析是如何启动内核的

(1) bootloader 启动内核

把内核从 flash 里面读到内存里面去, 还要设置很多参数, 设置好参数后, 才跳过去执行内核, 怎么跳过去看源代码

把内核从 flash 里面读到内存里

```
/* 1. 从NAND FLASH里把内核读入内存 */
puts("Copy kernel from nand\nr");
nand_read(0x60000+64, (unsigned char *)0x30008000, 0x200000);
```

上面是把内核从 nandflash 读到内存: 参数分别为, src, dst, len : 内核源地址, 内存的目标地址, 读的长度

然后设置传给内核的参数

```
/* 2. 设置参数 */
puts("Set boot params\nr");
setup_start_tag();
setup_memory_tags();
setup_commandline_tag("noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0");
setup_end_tag();
```

调用内核函数执行内核

第一个参数是 0, 第二个参数是机器 ID (板子属于哪一类) (参考 u-boot 程序), 第 3 个参数是存放传给内核的参数的位置 0x30000100

在汇编中就是这三个参数分别保存存在了 r0, r1, r2 这三个寄存器中

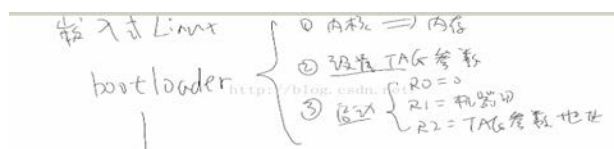
```
/* 3. 跳转执行 */
puts("Boot kernel\nr");
theKernel = (void (*)(int, int, unsigned int))0x30008000;
theKernel(0, 362, 0x30000100);
/*
 * mov r0, #0
 * ldr r1, #362
 * ldr r2, #0x30000100
 * mov pc, #0x30008000
 */
```

(2) bootloader 目的是为了引导内核

把内核读入内存

设置 TAG 参数

启动

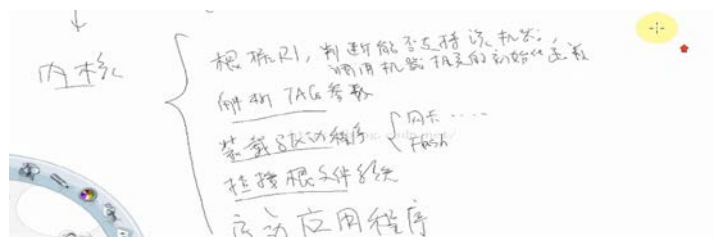


(3) 内核的目的 (启动应用程序)

应用程序在 flash 上或硬盘上, 所以需要挂载根文件系统, 假设在 flash 上还要有 flash 的驱动程序, 因而需要装载驱动程序。

flash 那么大, 根文件系统在 flash 的哪里, 在 u-boot 中传来的 TAG 参数中知道, 所以要解析 TAG 参数。根据 R1 寄存器

的值判断能否支持该机器, 如果能够支持, 调用机器相关的初始化函数。



(4) 根据 R1 判断能否支持该机器, 如果能够支持, 调用机器相关的初始化函数。

我们编译一个内核的时候, 不同单板用的芯片都是 s3c2440, 但是所接的外设不一样, 导致初始化函数不一样, 怎么分辨不同的开发板, 开发板在内核代码中分配一个 ID, 编译内核的时候, 内核怎么知道所用开发板是哪一种, 是由 bootloader 来确定是哪一种开发板, 通过 R1 传给内核, 内核根据传进来的机器 ID 跟自己已有的机器 ID 比较, 如果存在相等的就调用相关的初始化函数, 如果没有相等的就卡死在哪里。

2. 下载、配置、编译、烧写内核

(1) 解压内核并修改 Makefile

```
book@book-desktop:/work/system$ tar xjf linux-3.4.2.tar.bz2
book@book-desktop:/work/system$ cd linux-3.4.2/
book@book-desktop:/work/system/linux-3.4.2$ vi Makefile
```

我们是 arm 架构, 我们的交叉编译工具链是 arm-linux-

```
ARCH             = arm
CROSS_COMPILE    = arm-linux-
```

编译新版本的内核需要用的编译器版本比较高, 我们用的 4.3.2

所以我们之前的 linux-2.6.22.6 编译的时候并没有修改 makefile 的编译环境, 是因为在补丁文件里完成了

(2) 使用默认的配置

看一下有哪些默认的配置 这里要学习用 ls 来查看带关键字的文件

```
book@book-desktop:/work/system/linux-3.4.2$ find -name "*defconfig"
```

我们的是 arm 架构, 进入 arm 架构里面, 里面有一大堆配置项

```
book@book-desktop:/work/system/linux-3.4.2$ cd arch/arm/configs/
book@book-desktop:/work/system/linux-3.4.2/arch/arm/configs$ ls
```

搜索 2440

```
book@book-desktop:/work/system/linux-3.4.2/arch/arm/configs$ ls *2440*
mini2440_defconfig
```

搜索 2410

```
book@book-desktop:/work/system/linux-3.4.2/arch/arm/configs$ ls *2410*
s3c2410_defconfig
```

配置 make s3c2410_defconfig, 生成.config 文件

```
book@book-desktop:/work/system/linux-3.4.2$ make s3c2410_defconfig
```

查看.config

里面有 smdk2440 和 mini2440, 我们的 jz2440 跟 smdk2440 很像, 天下 2440 开发板基本一样, 都来源于 samsung 的公板 smdk2440

```
#
# S3C2440 Boards
#
CONFIG_MACH_ANUBIS=y
CONFIG_MACH_AT2440EVB=y
CONFIG_MACH_MINI2440=y
CONFIG_MACH_NEXCODER_2440=y
CONFIG_MACH_OSIRIS=y
CONFIG_MACH_OSIRIS_DVS=m
CONFIG_MACH_RX3715=y
CONFIG_ARCH_S3C2440=y
CONFIG_SMDK2440_CPU2440=y
```

(3) 编译内核

make uImage

(4) 把编译好的内核拷贝到网络文件系统里面去

```
book@book-desktop:/work/system/linux-3.4.2$ cp arch/arm/boot/uImage /work/nfs_root/
uImage_new
```

(5) 启动 u-boot 用 nfs 协议下载内核到 0x3200,0000

(6) 启动内核

3、启动内核出现乱码

- 找到在哪里启动，搜索 bootm,调用 do_bootm 函数

在 do_bootm 函数中的 boo_os 结构体数组

调用 `do_bootm_linux` 函数 (在 `bootm.c` 中)

调用 `boot_jump_linux` 函数，函数又调用 `kernel_entry` 函数

里面的参数 machid 根据搜索

可以从环境变量里面得到，如果环境变量里面没有设置，就使用默认值

`unsigned long machid = gd->bd->bi_arch_number;` , 搜索 `bi_arch_number` 有

`gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;` 由 `MACH_TYPE_SMDK2410` 的宏定义为 193.

因而机器 ID 默认值为 193, 内核启动时根据 193 找到对应的机器, 调用它的初始化函数

```
s = getenv("machid");
if (s) {
    strict_strtol(s, 16, &machid);
    printf("Using machid 0x%x from environment\n", machid);
}

debug("## Transferring control to Linux (at address %08lx)" \
    "...", (ulong) kernel_entry);
bootstage_mark(BOOTSTAGE_ID_RUN_OS);
announce_and_cleanup();
```

(3) 怎么找到对应的机器

机器在哪里, 进入 arch/arm 目录, 找到机器相关的文件。看后缀为 .o 的文件, 看哪些文件被编译进去。有我们内核支持的单板

```
book@book-desktop:~/work/system/linux-3.4.2/arch/arm$ find -name "mach*.o"
```

4. 实验

(1) 设置机器 ID, 下载并启动内核, 我们随便输入的机器 ID 肯定不对, 看内核会打印出可用的机器 ID

```
SMDK2410 # set machid 33245
SMDK2410 # save
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x40000 -- 100% complete.
Writing to Nand... done
SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage_new
ERROR: resetting DM9000 -> not responding
```

启动

```
SMDK2410 # bootm 32000000
## Booting kernel from Legacy Image at 32000000 ...
Image Name: Linux-3.4.2
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 2399640 Bytes = 2.3 MiB
Load Address: 30108000
Entry Point: 30108000
Verifying Checksum ... OK
Loading Kernel Image ... OK
```

打印出可用的机器 ID

```
000007cf      MINI2440
000002a9      NexVision - Nexcoder 2440
0000034a      Simtec-OSIRIS
00000250      IPAQ-RX3715
0000016a      SMDK2440
```

(2) 设置机器 ID 为 16a, 对应的文件是 Mach-smdk2440.c

根据 16a (10 进制是 362) 这个数值来找到结构体, 调用里面的初始化函数

```
MACHINE_START(S3C2440, "SMDK2440")
/* Maintainer: Ben Dooks <ben-linux@fluff.org> */
.atag offset = 0x100,

.init_irq = s3c24xx_init_irq,
.map_io = smdk2440_map_io,
.init_machine = smdk2440_machine_init,
.timer = &s3c24xx_timer,
.restart = s3c244x_restart,
MACHINE_END
```

上面这个结构体可以展开成下面的那一种结构体

```
static const struct machine_desc __mach_desc_S3C2440
__used
__attribute__((__section__(".arch.info.init"))) = {
    .nr = MACH_TYPE_S3C2440,
    .name = "SMDK2440",
    ...
};
```

其中的 MACH_TYPE_S3C2440 在 arch/arm/tools/mach-types 中定义, 它最后会被转换成一个头文件 include/asm-arm/mach-types.h 供其他文件包含。machine_desc 结构在 include/asm-arm/mach/arch.h 文件中定义, 所有的 machine_desc 结构都处于 ".arch.info.init" 段中, 在连接内核时, 它们被组织在一起, 开始地址为 _arch_info_begin, 结束地址为 _arch_info_end。这可以从连接脚本文件 arch/arm/kernel/vmlinux.lds 中看出来:

arch_info_end。这可以从连接脚本文件 arch/arm/kernel/vmlinux.lds 中看出来：

```
305 __arch_info_begin = .; /* machine_desc 结构的开始地址 */
306 *(.arch.info.init)
307 __arch_info_end = .; /* machine_desc 结构的结束地址 */
```

(3) include/asm-arm/mach/arch.h 这个文件是编译进内核时从文件 arch/arm/tools/mach-types 这个文件生成的

查看 mach-types.h 文件在哪里

```
book@book-desktop:/work/system/linux-3.4.2$ find -name mach-types.h
./include/generated/mach-types.h
./arch/arm/include/asm/mach-types.h
book@book-desktop:/work/system/linux-3.4.2$
```

查看一下

```
book@book-desktop:/work/system/linux-3.4.2$ vi ./arch/arm/include/asm/mach-types.h
```

里面有，它又包含一个被生成的文件，generated 是说这些文件是临时生成的。

```
#include <generated/mach-types.h>
```

查看这个临时生成的文件

```
book@book-desktop:/work/system/linux-3.4.2$ vi include/generated/mach-types.h
```

里面有 s3c2440

```
-----
#define MACH_TYPE ENP2611          356
#define MACH_TYPE S3C2440          362
#define MACH_TYPE GUMSTIX          373
#define MACH_TYPE OMAP H2          382
#define MACH_TYPE E740             384
-----
```

(4) 用 smdk2440 的机器 ID，设置机器 ID 为 16a 重新实验，下载后用 bootm 3200,0000 启动还是出现乱码

```
-----
SMDK2410 # set machid 16a
SMDK2410 # save
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x40000 -- 100% complete.
Writing to Nand... done
SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage_new
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
-----
```

(5) 用 mini2440 的机器 ID，设置机器 ID 为 7cf 重新实验，下载后用 bootm 3200,0000

启动后还是出现乱码，那就不是机器 ID 的问题，是不是启动参数里面没有设置波特率

(6) 设置启动参数，再来下载，用 bootm 3200,0000 启动，为什么设置机器 ID 为 7cf 有输出，设置机器 ID 为 16a 没输出

```
-----
SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3
SMDK2410 # save
Saving Environment to NAND...
Erasing Nand...
-----
```

这应该是时钟没有设置正确

我们就要知道时钟怎么修改

我们就像知道设置时钟的切入点在哪，我们之前有分析 linux2.6 的启动过程，如下图：

但是 linux3.6 有部分实现的方式不一样，但总体流程是一样的，可以参考：

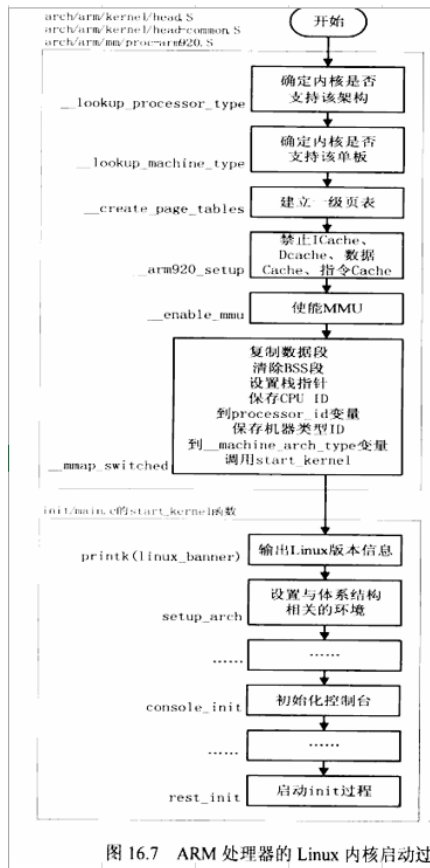


图 16.7 ARM 处理器的 Linux 内核启动过程

Linux 内核启动过程概述

与移植 U-Boot 的过程相似，在移植 Linux 之前，先了解它的启动过程。Linux 的启动过程可以分为两部分：架构/开发板相关的引导过程、后续的通用启动过程。如图 16.7 所示是 ARM 架构处理器上 Linux 内核 vmlinux 的启动过程。之所以强调是 vmlinux，是因为其他格式的内核在进行与 vmlinux 相同的流程之前会有一些独特的操作。比如对于压缩格式的内核 zImage，它首先进行自解压得到 vmlinux，然后执行 vmlinux 开始“正常的”启动流程。

引导阶段通常使用汇编语言编写，它首先检查内核是否支持当前架构的处理器，然后检查是否支持当前开发板。通过检查后，就为调用下一阶段的 start_kernel 函数作准备了。这主要分为如下两个步骤。

- ① 连接内核时使用的虚拟地址，所以要设置页表、使能 MMU。
- ② 调用 C 函数 start_kernel 之前的常规工作，包括复制数据段、清除 BSS 段、调用 start_kernel 函数。

第二阶段的关键代码主要使用 C 语言编写。它进行内核初始化的全部工作，最后调用 rest_init 函数启动 init 过程，创建系统第一个进程：init 进程。在第二阶段，仍有部分架构/开发板相关的代码，比如图 16.7 中的 setup_arch 函数用于进行架构/开发板相关的设置（比如重新设置页表、设置系统时钟、初始化串口等）。

按照上面的图片我们在 linux3.6 内核中可以找到 head.s 中调用 __mmap_switched

```

: ldr r13, __mmap_switched @ address to jump to after
: @ mmu has been enabled
: adr lr, BSYM(1) @ return (PIC) address
: mov r8, r4 @ set TTR1 to swapper_pg_dir
: ARM( add pc, r10, #PROCINFO_INITFUNC )
: THUMB( add r12, r10, #PROCINFO_INITFUNC )
: THUMB( mov pc, r12 )

```

然后进入 head-common.S，找到 __mmap_switched，发现它下面就是进入内核调用的第一个 C 函数代码 start_kernel

```

: __mmap_switched:
: adr r3, __mmap_switched_data
:
: ldmia r3!, {r4, r5, r6, r7}
: cmp r4, r5 @ Copy data segment if needed
: 1: cmpne r5, r6
: ldne fp, [r4], #4
: strne fp, [r5], #4
: bne 1b
:
: mov fp, #0 @ Clear BSS (and zero fp)
: 1: cmp r6, r7
: strccfp, [r6], #4
: bcc 1b
:
: ARM( ldmia r3, {r4, r5, r6, r7, sp} )
: THUMB( ldmia r3, {r4, r5, r6, r7} )
: THUMB( ldr sp, [r3, #16] )
: str r9, [r4] @ Save processor ID
: str r1, [r5] @ Save machine type
: str r2, [r6] @ Save atags pointer
: bic r4, r0, #CR_A @ Clear 'A' bit
: stmia r7, {r0, r4} @ Save control register values
: b start_kernel
: ENDP __mmap_switched

```

接下来我们分析内核是如何通过 start_kernel 一步步来调用设置我们的时钟的

start_kernel

```

→ setup_arch(&command_line);
→ paging_init(mdesc);
→ devicemaps_init(mdesc);
→ mdesc->map_io();

```

这里最终是调用了 mdesc 结构体指针中的 map_io

而我们在内核中每种类型的开发板都会定义一个 mdesc 的结构体，具体的过程我猜测应该是内核根据 uboot 传入的机器 id 来查找我们内核中对应的开发板，从而来获得它的 mdesc 结构体。

对于 S3C2440 开发板，就是调用 smdk2440_map_io 函数，它也是在 arch/arm/mach-s3c2440/mach-smdk2440.c 中定义，如下所示：

```
177 static void __init smdk2440_map_io(void)
178 {
179     s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
180     s3c24xx_init_clocks(16934400);
181     s3c24xx_init_ufs(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
182 }
```

具体实现调用 mach-smdk2440.c 中的 mdesc 结构体的原理要分析 setup_arch 函数：

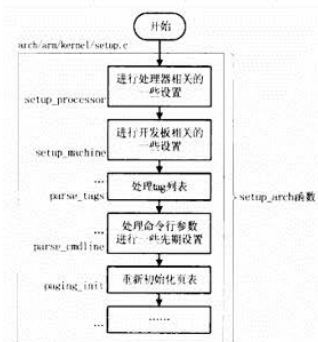


图 16.9 setup_arch 函数流程

setup_arch 函数分析。

进入 start kernel 函数（在 init/main.c 中）之后，如果在串口上没有看到内核的启动信息，一般而言有两个原因：Bootloader 传入的命令行参数不对，或者 setup_arch 函数（在 arch/arm/kernel/setup.c 中）针对开发板的设置不正确。

从图 16.7 可知，在调用 setup_arch 函数之前已经调用“printk(linux_banner)”了，但是这个时候 printk 函数只是将打印信息放在缓冲区中，并没有打印到控制台上（比如串口、LCD 屏等），因为这个时候控制台还没有初始化。当读者阅读 start kernel 函数的代码时，请注意，printk 函数打印的内容在 console_init 函数注册、初始化控制台之后才真正输出。

移植 U-Boot 时，U-Boot 传给内核的参数有两类：预先存在某个地址的 tag 列表和调用内核时在 r1 寄存器中指定的机器类型 ID。后者在引导阶段的 lookup_machine_type 函数已经用到，而 tag 列表将在 setup_arch 函数中进行初步处理。本节将重点介绍 setup_arch 函数、console_init 函数，以 tag 列表的处理（内存 tag、命令行 tag）、串口控制台的初始化为主线。

770 void __init setup_arch(char **cmdline_p) (1) setup_arch 函数分析。

771 { 先看 setup_arch 函数，它在 arch/arm/kernel/setup.c 中定义，其部分代码如下，图 16.9 是它的流程图。

```
776 setup_processor(); // 进行处理器相关的一些设置
777 mdesc = setup_machine(machine_arch_type); // 获得开发板的 machine_desc 结构
...
783 if (mdesc->boot_params) // 定义了 Bootloader 传入参数的地址？
784     tags = phys_to_virt(mdesc->boot_params); // 这个地址就是 tag 列表的首地址
...
798 if (tags->hdr.tag == ATAG_CORE) {
799     if (meminfo.nr_banks != 0) // 如果已经在内核中定义了 meminfo 结构
800         squash_mem_tags(tags); // 则忽略内存 tag
801     parse_tags(tags); // 解释每个 tag
802 }
...
809 memcpy(boot_command_line, from, COMMAND_LINE_SIZE);
810 boot_command_line[COMMAND_LINE_SIZE-1] = '\0';
811 parse_cmdline(cmdline_p, from); // 对命令行进行一些先期的处理
812 paging_init(&meminfo, mdesc); // 重新初始化列表
...
```

首先，第 776 行的 setup_processor 函数被用来进行处理器相关的一些设置，它会调用引导阶段的 lookup_processor_type 函数（它的主体是前面分析过的 lookup_processor_type 函数）以获得该处理器的 proc_info list 结构。

接下来，第 777 行的 setup_machine 函数被用来获得开发板的 machine_desc 结构，这通过调用引导阶段 lookup_machine_type 函数（它的主体是前面分析过的 lookup_machine_type 函数）来实现。以后，就会根据开发板的 machine_desc 结构来进行一些开发板相关的操作。

第 783~784 行用来确定 Bootloader 传入的启动参数的地址，它在开发板的 machine_desc 结构中指定，第 784 行将它转换为虚拟地址。比如对于 S3C2440 开发板，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中有如下定义。启动参数的地址就是 (S3C2410_SDRAM_PA + 0x100)，即 0x30000100。uboot 传入的启动参数的存放地址是固定的。

```
192 MACHINE_START(S3C2440, "SMDK2440")
...
196 .boot_params = S3C2410_SDRAM_PA + 0x100,
```

通过这里我们就知道了我们这里设置的时钟不对，我们开发板是 12M 的，所以要改为 12000000

而老师是用偷懒的办法来找到这个乱码问题的，看步骤 7

(7) 看看对应的文件

```
set machid 16a // smdk2440 mach-smdk2440.c
set machid 7cf // mini2440 mach-mini2440.c
```

看看这两个文件在初始化函数方面有什么差别，Mach-smdk2440.c 中 map_io 函数在初始化时钟的参数是 16934400，而我们板子的晶振时钟是

12M 啊

```
static void __init smdk2440_map_io(void)
{
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    s3c24xx_init_clocks(16934400); // log.csdn.net/
    s3c24xx_init_ufs(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}
```

因而需要修改晶振时钟为 12M

```
static void __init smdk2440_map_io(void)
{
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    s3c24xx_init_clocks(12000000); // log.csdn.net/
    s3c24xx_init_ufs(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}
```

(8) 重新编译 uImage，下载，启动

5. 总结


```

步骤1:
在UBOOT里:
set machid 16a // smdk2440 mach-smdk2440.c
或
set machid 7CF // mini2440 mach-mini2440.c

步骤2:
arch\arm\mach-s3c24xx\mach-smdk2440.c
s3c24xx_init_clocks(16934400);
改为
s3c24xx_init_clocks(12000000);

步骤3:
配置/编译: make s3c2410_defconfig, make mini2440_defconfig
make uImage

```

在 u-boot 指定波特率

```

1
2 步骤4:
3 在uboot里: set bootargs console=ttySAC0,115200 .....
4

```

现在总结一下上面分析的内核启动第二阶段的函数调用过程（以 S3C2440 开发板为例），相同缩进的函数表示它们是在同一个函数中被调用：

```

start_kernel ->
  setup_arch ->
    setup_processor
    setup_machine
    ...
    parse_tags
    ...
    parse_cmdline
    paging_init ->
      devicemaps_init ->
        mdesc->map_io() ->
          s3c24xx_init_io
          s3c24xx_init_clocks
          s3c24xx_init_ucts
        ...
      console_init ->
        s3c24xx_serial_initconsole
        register_console(&s3c24xx_serial_console)
      ...
    ...

```

待整理的问题：

Q1：开发板的启动方式和 uboot 是烧些在 norflash 或者 nandflash 上有关么

A1：想使用 NOR 启动，那么 NOR 上就应该有 UBOOT；想使用 NAND 启动，那么 NAND 上就应该有 UBOOT

Q1：我们想使用 NOR 启动，那么 NOR 上就应该有 UBOOT；想使用 NAND 启动，那么 NAND 上就应该有 UBOOT，

1:那么 NOR 启动的话，内核应该烧写到 nandflash 还是 norflash 上有要求么？Nand 启动的？

2:我们内核为什么一般都要先烧到 nandflash 上再复制到 sdram 上呢？不能直接烧到 sdram 上么？是不是因为 sdram 掉电数据丢失，而 nandflash 不会？

3:书上有写：

将上面生成的 uImage 放入 tftp 服务器目录或 nfs 目录（/work/nfs_root），然后在 U-Boot 中执行以下命令下载、烧写。

```

tftp 0x32000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
nand erase 0 0x2000000 /* 擦除 NAND Flash 前 2MB */
nand write.jffs2 0x32000000 0 $(filesize) /* 烧写 uImage */

```

如果用 tftp 和 nfs 是把内核烧到 0x3200 0000 和 0x3000 0000 上么，这不是 sdram 的地址么？

还要 nand write，这是写到 0x3200 0000 还是 nandflash 的 0 地址，如果是前者，不是多此一举么，已经都到 sdram 了

Q1：我们之前说当应用程序调用驱动程序，在驱动程序里发生了错误的时候会打印出很多信息，比如发生错误的时候寄存器的值，回溯信息，栈信息，而当应用程序出错时，信息比较少，那么当我调用应用程序的时候，开发板是怎么知道是在内核发生的错误，还是在应用程序发生的错误？

A1：当发生错误时，会进入到内核；内核会根据出错的地址来判断是“内核导致”还是“应用导致”，应用程序和内核的地址空间是不一样的，

比如 APP 的地址一般从 0 到 xxx，而内核是从 0x8.....开始

Q2:我们怎么知道一个.c或.h是应用层的还是驱动层的,是看什么来区分的?

A2: APP 跟驱动的代码,两者是完全无关的;一个.c或一个.h是属于驱动还是APP,这不是问题。

Q3:我们说我们读取内核的时候,可以随便设置读取内核到某处的地址,就是因为 uImage 有这个头部,他知道内核的加载的地址和入口地址,如果他发现真正的内核不在他的加载地址的话,他就会将内核移动到他的加载地址,最后再跳到入口地址,而对于我们的开发板的内核地址是 0x30008000,这个值是怎么确定的呢?从哪里可以看出来的

Q4:我们启动内核的时候,用的是 bootm xxxx,这个 xxx 的地址应该是除去头部之后内核实际应该位于 sdram 的只把,应该是 0x30008000-64K

为什么书上是:

3. 启动内核

可以使用以下命令启动 NAND Flash 上的内核。

```
nboot 0x32000000 0 0
bootm 0x32000000
```

Q1:我们拿到一个内核固件,如何知道他是什么版本的呢?

A1:在顶层 Makefile 里面搜索 VERSION

Linux 内核的版本号可以从源代码的顶层目录下的 Makefile 中看到,比如下面几行它们构成了 Linux 的版本号: 2.6.22.6。

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 22
EXTRAVERSION = .6
```

其中的“VERSION”和“PATCHLEVEL”组成主版本号,比如 2.4、2.5、2.6 等,稳定版本的主版本号用偶数表示(比如 2.4、2.6),每隔 2~3 年出现一个稳定版本。开发中的版本号用奇数来表示(比如 2.3、2.5),它是下一个稳定版本的前身。

“SUBLEVEL”称为次版本号,它不分奇偶,顺序递增。每隔 1~2 个月发布一个稳定版本。

“EXTRAVERSION”称为扩展版本号,它不分奇偶,顺序递增。每周发布几次扩展版本号,修正最新的稳定版本的问题。值得注意的是,“EXTRAVERSION”也可以不是数字,而是类似“-rc6”的字样,表示这是一个测试版本。在新的稳定版本发布之前,会先发布几个测试版本用于测试。

上面的截图只是个例子,并不是我们移植的内核的版本号

Q2:在 linux2.6 的内核中我们知道内核读取 uboot 传入的 cpu 类型和机器 id 是分别通过 head.S 中的 __lookup_processor_type 和 __lookup_machine_type 作为入口点来进行和自己内核中的 cpu 架构列表和机器 id 列表来匹配的,但是在 linux3.4 内核中并没有找到 __lookup_machine_type,那么内核是怎么匹配 uboot 传入的机器 id 的呢?

A2: setup_arch

→ mdesc = setup_machine_tags(machine_arch_type);

毕业班第 3 课第 2 节_移植 3.4.2 内核之修改分区及制作根文件系统

1、制作分区

(1) 分析

在上节实验中出现以下问题:无法挂载根文件系统

```

List of all partitions:
1f00          16 mtdblock0  (driver?)
1f01          2048 mtdblock1 (driver?)
1f02          4096 mtdblock2 (driver?)
1f03          2048 mtdblock3 (driver?)
1f04          4096 mtdblock4 (driver?)
1f05          10240 mtdblock5 (driver?) /6410裸板/u-boot/内核/文件系统/驱动入门
1f06          24576 mtdblock6 (driver?) 边写边讲
1f07          212992 mtdblock7 (driver?)
No filesystem could mount root, tried: ext3 ext2 cramfs vfat mdsos iso9660 romfs
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,3)
Backtrace:
[<c0118cc0>] (dump_backtrace+0x0/0x10c) from [<c045c768>] (dump_stack+0x18/0x1c)
r6:c38a9000 r5:fffff000 r4:c381df2c r3:18406000
[<c045c750>] (dump_stack+0x0/0x1c) from [<c045c7e8>] (panic+0x7c/0x1d0)
[<c045c76c>] (panic+0x0/0x1d0) from [<c054af5c>] (mount_block_root+0x1e0/0x23c)
r3:00000003 r2:c381df2c r1:c381df2c r0:c04e4c20
[<c054ad7c>] (mount_block_root+0x0/0x23c) from [<c054b040>] (mount_root+0x88/0x108)
[<c054af88>] (mount_root+0x0/0x108) from [<c054b1d0>] (prepare_namespace+0x110/0x1d0)
r7:c05ac540 r6:c056e764 r5:c056e764 r4:c05ac5a0
[<c054b0c0>] (prepare_namespace+0x0/0x1d0) from [<c054a3e8>] (kernel_init+0x164/0x1b8)
r6:00000008 r5:c056df38 r4:c0574b34
[<c054a284>] (kernel_init+0x0/0x1b8) from [<c012b8b0>] (do_exit+0x0/0x76c)

```

我们通过上面的打印出来的信息发现,我们是用之前移植的新的 u-boot 来引导内核的,之前的 u-boot 中只有四个分区,而这里有很多个分区在以前

移植新的 u-boot 烧写文件系统时是烧写到某个地方, 以前的分区是下面那样划分的, 划分为 4 个分区, 但是上面有 8 个分区

个人理解是由于内核中分配的分区和我们 u-boot 的分区不匹配,相应的地址存放的文件不对,导致读取不到正确的文件

```

0. 内核打印出来的分区信息
1 0x00000000-0x00040000 : "bootloader"
2 0x00040000-0x00060000 : "params"
3 0x00060000-0x00260000 : "kernel"
4 0x00260000-0x10000000 : "root"

```

分区修改参考 (分区是在代码里面写死的, u-boot 怎么设置不会影响内核)

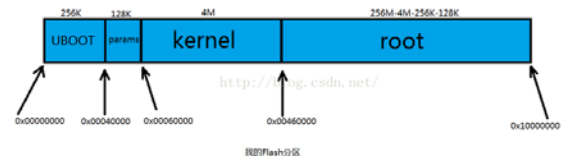
下面我们来修改内核的分区:

1、先解释一下什么是 MTD

Memory Technology Device,内存技术设备, 如 nand flash

2、为什么要修改 MTD 分区

用于存放不同大小的文件 (如 u-boot,params, uImage, root filesystem)



3、怎么知道在哪里修改分区

(1) 看 u-boot 启动后出现的分区信息

```

Creating 8 MTD partitions on "NAND":
0x00000000-0x00000000040000 : "Boot Agent"
mtd: partition "Boot Agent" doesn't end on an erase block -- force read-only
0x00000000-0x00000002000000 : "S3C2410 flash partition 1"
0x00000004000000-0x00000008000000 : "S3C2410 flash partition 2"
0x00000008000000-0x0000000a000000 : "S3C2410 flash partition 3"
0x0000000a000000-0x0000000e000000 : "S3C2410 flash partition 4"
0x0000000e000000-0x00000018000000 : "S3C2410 flash partition 5"
0x00000018000000-0x00000030000000 : "S3C2410 flash partition 6"
0x00000030000000-0x00000010000000 : "S3C2410 flash partition 7"

```

(2) 在内核中查找分区名字, 如“Boot Agent”, 查看代码所在的地方, 下面的 grep 命令中,其中反斜杠\是转移的意思, 只用双引号的本来含义, 用空格的本来含义, 下面的 common-smdk.c 是通用的意思。所以我们可以猜测,我们应该是 common-smdk.c 文件中修改,其他的文件的名字和我们肯定不匹配

```

book@book-desktop: /work/system/linux-3.4.2$ grep "\"Boot Agent\"" * -nR
arch/arm/mach-s3c24xx/common-smdk.c:113: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-at2440evb.c:85: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-osiris.c:140: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-osiris.c:163: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-anubis.c:129: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-anubis.c:152: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-bast.c:224: .name = "Boot Agent",
arch/arm/mach-s3c24xx/mach-vstms.c:81: .name = "Boot Agent",

```

4、如何修改 MTD 分区

在 linux 内核中的 arch/arm/mach-s3c24xx/common-smdk.c 中修改

static struct mtd_partition smdk_default_nand_part[] = {} 结构体数组里的内容替换成


```
static struct mtd_partition smdk_default_nand_part[] = {

[0] = {

.name = "bootloader",

.size = SZ_256K,

.offset = 0,

},

// MTDPART_OFS_APPEND 表示分区开始的偏移地址紧接着上一个分区

[1] = {

.name = "params",

.offset = MTDPART_OFS_APPEND,

.size = SZ_128K,

},

[2] = {

.name = "kernel",

.offset = MTDPART_OFS_APPEND,

.size = SZ_4M,

},

//MTDPART_SIZ_FULL 表示取剩余下的容量

[3] = {

.name = "rootfs",

.offset = MTDPART_OFS_APPEND,

.size = MTDPART_SIZ_FULL,

}

};
```

(2) 实验

修改完之后,下载并启动内核,分区已改变,是不是分区上没有烧写文件系统。

```
List of all partitions:
1f00      256 mtdblock0  (driver?)
1f01      128 mtdblock1  (driver?)
1f02      2048 mtdblock2  (driver?)
1f03     259712 mtdblock3  (driver?)
No filesystem could mount root, tried: ext3 ext2 cramfs vfat msdos iso9660 romfs
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,3)
Backtrace:
[<c0118cc0>] (dump_backtrace+0x0/0x10c) from [<c045c768>] (dump_stack+0x18/0x1c)
x6:c38a9000 x5:ffffffea r4:c381df2c
[<c045c750>] (dump_stack+0x0/0x1c) from [<c045c7e8>] (panic+0x7c/0x1d0)
[<c045c76c>] (panic+0x0/0x1d0) from [<c054af5c>] (mount_block_root+0x1e0/0x23c)
x3:00000003 x2:c381df2c x1:c381df2c x0:c04e4c20
[<c054ad7c>] (mount_block_root+0x0/0x23c) from [<c054b040>] (mount_root+0x88/0x108)
[<c054afb8>] (mount_root+0x0/0x108) from [<c054b1d0>] (prepare_namespace+0x110/0x1d0)
x7:c05ac4c0 x6:c056e764 x5:c056e764 x4:c05ac520
[<c054b0c0>] (prepare_namespace+0x0/0x1d0) from [<c054a3e8>] (kernel_init+0x164/0x1b8)
x6:00000008 x5:c056df38 x4:c0574b34
[<c054a284>] (kernel_init+0x0/0x1b8) from [<c012b8b0>] (do_exit+0x0/0x76c)
```

我们发现对应的 linux 中的分区确实已经按照我们的发生了改变,但是提示还是没有成功挂接文件系统

这是由于我们还没有烧些文件系统映像

(3) 烧写文件系统

下载

```
U-Boot> run key to stop autoboot: y
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev.yaffs2
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
```

```
-----
Bytes transferred = 8952768 (889bc0 hex)
SMDK2410 # nand erase.part rootfs
```

擦除

烧写

```
Bytes transferred = 8952768 (889bc0 hex)
SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.yaffs 30000000 260000 889bc0

NAND write: device 0 offset 0x260000, size 0x889bc0
8952768 bytes written: OK
```

下载内核并启动

```
SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage_new
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
```

```
Bytes transferred = 2399712 (249de0 hex)
SMDK2410 # bootm 32000000
## Booting kernel from Legacy Image at 32000000 ...
Image Name: Linux-3.4.2
Image Type: ARM Linux Kernel Image (uncompressed)
```

还是出现错误，因为不支持 yaffs 文件系统

```
List of all partitions:
1f00          16 mtdblock0 (driver?)
1f01         2048 mtdblock1 (driver?)
1f02         4096 mtdblock2 (driver?)
1f03         2048 mtdblock3 (driver?)
1f04         4096 mtdblock4 (driver?)
1f05        10240 mtdblock5 (driver?)
1f06        24576 mtdblock6 (driver?)
1f07       212992 mtdblock7 (driver?)
No filesystem could mount root, tried: ext3 ext2 cramfs vfat msdos iso9660 romfs
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,3)
Backtrace:
[<0118cc0>] (dump_backtrace+0x0/0x10c) from [<045c768>] (dump_stack+0x18/0x1c)
r6:c38aa000 r5:ffffffea r4:c381df2c
[<045c750>] (dump_stack+0x0/0x1c) from [<045c7e8>] (panic+0x7c/0x1d0)
[<045c76c>] (panic+0x0/0x1d0) from [<054af5c>] (mount_block_root+0x1e0/0x23c)
r3:00000003 r2:c381df2c r1:c381df2c r0:c04e4c20
[<054ad7c>] (mount_block_root+0x0/0x23c) from [<054b040>] (mount_root+0x88/0x108)
[<054afb8>] (mount_root+0x0/0x108) from [<054b1d0>] (prepare_namespace+0x110/0x1d0)
r7:c05ac540 r6:c056e764 r5:c056e764 r4:c05ac5a0
[<054b0c0>] (prepare_namespace+0x0/0x1d0) from [<054a3e8>] (kernel_init+0x164/0x1b8)
r6:00000008 r5:c056df38 r4:c0574b34
[<054a284>] (kernel_init+0x0/0x1b8) from [<012b8b0>] (do_exit+0x0/0x76c)
```

(4) 看内核支持的文件系统

查看 vi .config

```
book@book-desktop:/work/system/linux-3.4.2$ vi .config
```

搜索不出 yaffs，但是有 jffs2

```
# CONFIG_JFFS2_FS is not set
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_DEBUG=0
CONFIG_JFFS2_FS_WRITEBUFFER=y
# CONFIG_JFFS2_FS_WBUF_VERIFY is not set
CONFIG_JFFS2_SUMMARY=y
# CONFIG_JFFS2_FS_XATTR is not set
# CONFIG_JFFS2_COMPRESSION_OPTIONS is not set
CONFIG_JFFS2_ZLIB=y
# CONFIG_JFFS2_LZO is not set
CONFIG_JFFS2_RTREE=y
# CONFIG_JFFS2_RUBIN is not set
```

(5) 烧写 jffs2 文件系统

实际下载文件的时候，会把大小存在一个所谓的环境变量里面，可以通过 \$filesize 或 \$ (filesize) 这样引用。

下载

```

hit any key to stop autoboot: 0
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev.jffs2
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46

```

擦除、烧写

```

Bytes transferred = 5999016 (5b89a8 hex)
SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.jffs2 30000000 260000 $filesize

NAND write: device 0 offset 0x260000, size 0x5b89a8
5999016 bytes written: OK

```

(6) 设置启动参数 (在启动参数中指明内核在哪里挂载根文件系统)

并下载内核

```

SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2
SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage_new
ERROR: resetting DM9000 -> not responding

```

用 bootm 3200,0000 启动

挂载上去了, 但还是发现找不着 init 文件

```

VFS: Mounted root (jffs2 filesystem) on device 31:3.
Freeing init memory: 172K
Kernel panic - not syncing: No init found. Try passing init= option to kernel. See
Linux Documentation/init.txt for guidance.
Backtrace:
[<c0118cc0>] (dump_backtrace+0x0/0x10c) from [<c045c768>] (dump_stack+0x18/0x1c)
r6:00000008 r5:c05ac540 r4:c0580f68
[<c045c750>] (dump_stack+0x0/0x1c) from [<c045c7e8>] (panic+0x7c/0x1d0)
[<c045c76c>] (panic+0x0/0x1d0) from [<c0108988>] (init_post+0xc8/0x140)
r3:0000000a r2:00000007 r1:00000006 r0:c04e4850
[<c01088c0>] (init_post+0x0/0x140) from [<c054a3ec>] (kernel_init+0x168/0x1b8)
r5:c056df38 r4:c0574b34
[<c054a284>] (kernel_init+0x0/0x1b8) from [<c012b8b0>] (do_exit+0x0/0x76c)

```

2. FHS (Filesystem Hierarchy Standard) 标准介绍

当我们在 linux 下输入 `ls /` 的时候, 见到的目录结构以及这些目录下的内容都大同小异, 这是因为所有的 linux 发行版在对根文件系统布局上都遵循 FHS 标准的建议规定。

该标准规定了根目录下各个子目录的名称及其存放的内容:

目录名	存放的内容
/bin	必备的用户命令, 例如 <code>ls</code> 、 <code>cp</code> 等
/sbin	必备的系统管理员命令, 例如 <code>ifconfig</code> 、 <code>reboot</code> 等
/dev	设备文件, 例如 <code>mtdblock0</code> 、 <code>tty1</code> 等
/etc	系统配置文件, 包括启动文件, 例如 <code>inittab</code> 等
/lib	必要的链接库, 例如 C 链接库、内核模块
/home	普通用户主目录
/root	root 用户主目录
/usr/bin	非必备的用户程序, 例如 <code>find</code> 、 <code>du</code> 等
/usr/sbin	非必备的管理员程序, 例如 <code>chroot</code> 、 <code>inetd</code> 等
/usr/lib	库文件
/var	守护程序和工具程序所存放的可变, 例如日志文件
/proc	用来提供内核与进程信息的虚拟文件系统, 由内核自动生成目录下的内容

/sys	用来提供内核与设备信息的虚拟文件系统，由内核自动生成目录下的内容
/mnt	文件系统挂载点，用于临时安装文件系统
/tmp	临时性的文件，重启后将自动清除

3. 制作根文件系统

构造文件系统

内核启动后，挂载根文件系统，最终启动应用程序。有那么多应用程序，到底启动哪一个，内核只会启动第一个应用程序，第一个应用程序就会根据配置文件去决定启动哪一家的公司的哪一个应用程序。

制作根文件系统就是要建立以上的目录，并在其中建立完整目录内容。其过程大体包括：

- 编译 / 安装 busybox, 生成/bin、/sbin、/usr/bin、/usr/sbin 目录
- 利用交叉编译工具链，构建/lib 目录
- 手工构建/etc 目录
- 手工构建最简化的/dev 目录
- 创建其它空目录
- 配置系统自动生成/proc 目录
- 利用 udev 构建完整的/dev 目录
- 制作根文件系统的 jffs2 映像文件

(1) 编译 / 安装 busybox, 生成/bin、/sbin、/usr/bin、/usr/sbin 目录

(编译内核用哪一个编译器，就用这个编译器去编译应用程序)

这些目录下存储的主要是常用命令的二进制文件。

1、从 <http://www.busybox.net/> 下载 busybox-1.20.0.tar.bz2



2、tar xjvf busybox-1.20.0.tar.bz2 解包

3、修改交叉编译环境：

方法一：修改 Makefile 文件

ARCH ?= arm

CROSS_COMPILE ?= arm-linux-

方法二：

通过修改 menuconfig 中的选项

make menuconfig 配置 busybox

我们要交叉编译，需要指定编译器



```
General Configuration --->
Build Options --->
Debugging Options --->
Installation Options ("make install" behavior) --->
Busybox Library Tuning --->

[ ] Build BusyBox as a static binary (no shared libs) (NEW)
[ ] Build BusyBox as a position independent executable (NEW)
[ ] Force NOMMU build (NEW)
[ ] Build shared libbusybox (NEW)
[*] Build with Large File Support (for accessing files > 2 GB) (NEW)
(0) Cross Compiler prefix (NEW)
( ) Path to sysroot (NEW) http://blog.csdn.net/
( ) Additional CFLAGS (NEW)
( ) Additional LDFLAGS (NEW)
( ) Additional LDLIBS (NEW)
```

编译选项交叉编译前缀

———— Cross Compiler prefix ————

Please enter a string value. Use the <TAB> key to move from the input field to the buttons below it.

arm-linux- <http://blog.csdn.net/>

< Ok > < Help >

4. 编译 busybox

make

5. 安装 busybox

在 nfs_root 中创建目录: mkdir fs_mini_mdev_new

```
book@book-desktop:/work/system/busybox-1.20.0$ make install CONFIG_PREFIX=/work/nfs_root/fs_mini_mdev_new
```

最终,在我们自己定义的文件夹内会自动生成 4 个文件夹

```
指定路径下安装
make install CONFIG_PREFIX=/work/nfs_root/fs_mini_mdev_new/
book@book-desktop:/work/system/busybox-1.20.0$ make install CONFIG_PREFIX=/work/nfs_root/fs_mini_mdev_new
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ ls
bin  linuxrc  sbin  usr
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ ls
bin  linuxrc  sbin  usr
```

(2) 安装 glibc 库

2.1: 看环境变量, 我们的交叉编译工具链在哪里

```
book@book-desktop:/work/system/busybox-1.20.0$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/arm/4.3.2/bin
book@book-desktop:/work/system/busybox-1.20.0$ cd /usr/local/arm/4.3.2/bin
book@book-desktop:/usr/local/arm/4.3.2/bin$ cd ..
book@book-desktop:/usr/local/arm/4.3.2$ ls
arm-none-linux-gnueabi bin lib libexec share
book@book-desktop:/usr/local/arm/4.3.2$ find -name lib
./arm-none-linux-gnueabi/libc/thumb2/lib
./arm-none-linux-gnueabi/libc/thumb2/usr/lib
./arm-none-linux-gnueabi/libc/armv4t/lib
./arm-none-linux-gnueabi/libc/armv4t/usr/lib
./arm-none-linux-gnueabi/libc/lib
./arm-none-linux-gnueabi/libc/usr/lib
./arm-none-linux-gnueabi/lib
./lib
```


进工具链中查找相关的库, 需要注意的是, 如果我们直接上面的文件中 ls 是找不到的, 因为 arm-none-linux-gnueabi 这个目录是个隐藏文件, 上面的 thumb2 是 thumb 指令集用到的库, 我们用到的是

```
./arm-none-linux-gnueabi/libc/armv4t/lib
./arm-none-linux-gnueabi/libc/armv4t/usr/lib
```

这两个库。

2.2: 在根文件系统中建立一个 lib 目录

```
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ mkdir lib
```

2.3: 把交叉编译工具链下的动态链接库文件拷贝到跟文件系统下的 lib 目录, -d 是链接文件在拷贝后仍是链接文件, 否则是拷贝实际的文件。

```
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ cp /usr/local/arm/4.3.2/arm-none
-linux-gnueabi/libc/armv4t/lib/*so* /work/nfs_root/fs_mini_mdev_new/lib -d
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ ls lib
```

2.4: 在根文件系统中建立一个 usr/lib 目录, 不加 -p 要先创建目录 usr, 后创建目录 lib.-p 可以一次性创建。把交叉编译工具链下的动态链接库文件拷贝到跟文件系统下的 usr/lib 目录

```
book@book-desktop:/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/lib$ mki
r /work/nfs_root/fs_mini_mdev_new/usr/lib -p
book@book-desktop:/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/lib$ cp /
usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/usr/lib/*so* /work/nfs_root
/fs_mini_mdev_new/usr/lib -d
book@book-desktop:/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/lib$
```

(3) 构造 etc 目录

把之前的根文件系统的 etc 目录下的内容拷贝过来并修改

```
book@book-desktop:/work/nfs_root$ cp fs_mini_mdev/etc fs_mini_mdev_new -rf
book@book-desktop:/work/nfs_root$ cd fs_mini_mdev_new
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ cd etc/
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ ls
fstab init.d inittab
```

我们内核启动的第一个进程就会打开 inittab, 根据文件指示去做某些事情

会执行脚本/etc/init.d/rcS, 执行完脚本后, 启动 shell, 这个 shell 从串口 0 的设备名, console 中得到输入

下图为 inittab 的内容

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
console::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

看脚本/etc/init.d/rcS, mount -a 是根据 fstab 文件的内容来挂接各种根文件系统

```
#!/bin/sh
ifconfig eth0 192.168.1.17

mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

fstab 文件的内容

```
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ cat fstab
# device mount-point type options dump fsck order
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
```

```
#####
#####
```

第一列: 设备名或者设备卷标名, (/dev/sda10 或者 LABEL=)

第二列: 设备挂载目录 (例如上面的“/”或者“/mnt/D/”)

第三列: 设备文件系统 (例如上面的“ext3”或者“vfat”)

第四列: 挂载参数 (看帮助 man mount)

对于已经挂载好的设备，例如上面的/dev/sda2，现在要改变挂载参数，这时可以不用卸载该设备，而可以使用下面的命令（没有挂载的设备，remount 这个参数无效）

```
#mount /mnt/D/ -o remount,ro (改 defaults 为 ro)
```

为了安全起见，可以指明其他挂载参数，例如：

noexec（不许可可执行文件可执行，但千万不要把根分区挂为 noexec，那就无法使用系统了，连 mount 命令都无法使用了，这时只有重新做系统了！

nodev（不允许挂载设备文件）

nosuid,nosgid（不允许有 suid 和 sgid 属性）

nouser（不允许普通用户挂载）

第五列：指明是否要备份，（0 为不备份，1 为要备份，一般根分区要备份）

第六列：指明自检顺序。（0 为不自检，1 或者 2 为要自检，如果是根分区要设为 1，其他分区只能是 2）如果想开机就自动挂载(mount)上，可以在/etc/fstab 上添加如下几行：

```
/dev/hda5 /mnt/d vfat exec,dev,suid,rw,umask=0,ioccharset=gb2312,codepage=936 0 0
```

同时也解决了中文文件名的乱码问题，我的 fstab 文件如下，仅供参考：

```
#####
#####
```

(4) 构造 dev 目录

我们上电后执行，里面会根据 inittab 这个文件来决定执行脚本/etc/init.d/rcS，这个脚本会执行 mdev -s 创建 dev 目录下的内容，但是在 mdev -s 执行前，dev 目录下需要有东西。创建两个设备节点。下面的目录需要修改一下，在 fs_mini_mdev_new 目录下的 dev 目录下创建两个设备节点。

```
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ mkdir dev
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ ls -l /dev/console /dev/null

crw----- 1 root root 5, 1 2012-06-17 07:10 /dev/console
crw-rw-rw- 1 root root 1, 3 2012-06-17 07:10 /dev/null
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ sudo mknod dev/console c 5 1
[sudo] password for book:
book@book-desktop:/work/nfs_root/fs_mini_mdev_new/etc$ sudo mknod dev/null c 1 3
```

(5) 创建其他目录（挂载虚拟的文件系统）

```
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ mkdir proc tmp mnt sys root
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$ ls
bin dev etc lib linuxrc mnt proc root sbin sys tmp usr
book@book-desktop:/work/nfs_root/fs_mini_mdev_new$
```

(6) 制作映像文件

-s 是扇区的大小，我们的 nand flash 的一个扇区是 2048，-e 是可擦除块是 128K，-d 指明目录是 fs_mini_mdev_new,后面-o 跟输出文件 fs_mini_mdev_new.jffs2

```
book@book-desktop:/work/nfs_root$ mkfs.jffs2 -n -s 2048 -e 128KiB -d fs_mini_mdev_n
ew -o fs_mini_mdev_new.jffs2 http://blog.csdn.net/
```

4、验证制作的跟文件系统

(1) 重启开发板、下载跟文件系统，擦除 nand flash,然后烧写

```
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev_new.jffs2
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
```



```

Bytes transferred = 3903024 (3b8e30 hex)
SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.jffs2 30000000 260000 $filesize

NAND write: device 0 offset 0x260000, size 0x3b8e30
3903024 bytes written: OK

```

(3) 设置启动参数、下载新内核、用 bootm 32000000 启动

```

SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2

SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage_new
ERROR: resetting DM9000 -> not responding

```

(4) 看启动信息，挂接上去了，并找到了 init 程序，但是这个程序有点问题

```

jffs2: Empty flash at 0x003b8e30 ends at 0x003b9000
VFS: Mounted root (jffs2 filesystem) on device 31:3.
Freeing init memory: 172K
Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

Backtrace:
[<c0118cc0>] (dump_backtrace+0x0/0x10c) from [<c045c768>] (dump_stack+0x18/0x1c)
r6:c3819d60 r5:c058eb14 r4:c058eb14
[<c045c750>] (dump_stack+0x0/0x1c) from [<c045c7e8>] (panic+0x7c/0x1d0)
[<c045c76c>] (panic+0x0/0x1d0) from [<c012bef4>] (do_exit+0x644/0x76c)
r3:c381fc40 r2:c3819e80 r1:00000004 r0:c04ea054
[<c012b8b0>] (do_exit+0x0/0x76c) from [<c012c06c>] (do_group_exit+0x50/0xb8)
[<c012c01c>] (do_group_exit+0x0/0xb8) from [<c0137888>] (get_signal_to_deliver+0x1e0/0x4f4)
r4:00000000
[<c01376a8>] (get_signal_to_deliver+0x0/0x4f4) from [<c0117dec>] (do_notify_resume+0xd0/0x520)
[<c0117d1c>] (do_notify_resume+0x0/0x520) from [<c0115c34>] (work_pending+0x24/0x28)
)

```

(5) 在内核中搜索 exitcode,看 exitcode=0x00000004 是什么意思。

在 Exit.c 的 find_new_reaper 函数中

```

if (unlikely(pid_ns->child_reaper == father)) {
    write_unlock_irq(&tasklist_lock);
    if (unlikely(pid_ns == &init_pid_ns)) {
        panic("Attempted to kill init! exitcode=0x%08x\n",
            father->signal->group_exit_code ?:
            father->exit_code);
    }
}

```

看参数是什么意思

find_new_reaper 函数被 forget_original_parent 函数调用

forget_original_parent 函数被 exit_notify 函数调用

exit_notify 函数被 do_exit 函数调用

```

void do_exit(long code)
{
    http://blog.csdn.net/
}

```

后面的 code 就是退出代码，搜索 do_exit 函数，看有哪些退出代码

```

do_exit(SIGKILL);
看宏 SIGKILL,这里 4 代表非法指令的意思，可以在网上搜索宏 SIGKILL

#define SIGHUP 1
#define SIGINT 2
#define SIGQUIT 3
#define SIGTTL 4
#define SIGTRAP 5

```

(5) 怎么会有非法指令

看看我们编译应用程序的时候用的是所谓的 eabi 接口，我们内核也要支持 eabi 接口才行，eabi 是可执行层应用程序的二进制接口，应用程序支持这个接口，是用编译器 arm-none-linux-gnueabi 编译出来的符合 eabi 接口的应用程序，那么内核应该支持这种接口。

```
book@book-desktop:/work/system/linux-3.4.2$ arm-linux-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabi
Configured with: /scratch/julian/lite-respin/linux/src/gcc-4.3/configure --build=i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi --enable-t
hreaded --disable-libmudflap --disable-libssp --disable-libstdc++-pch --with-gnu-as
--with-gnu-ld --enable-languages=c,c++ --enable-shared --enable-symvers=gnu --enabl
e-_cxa_atexit --with-pkgversion='Sourcery G++ Lite 2008q3-72' --with-bugurl=https:
//support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery -
-with-sysroot=/opt/codesourcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/s
cratch/julian/lite-respin/linux/install/arm-none-linux-gnueabi/libc --with-gmp=/scr
atch/julian/lite-respin/linux/obj/host-libs-2008q3-72-arm-none-linux-gnueabi-i686-p
c-linux-gnu/usr --with-mpfr=/scratch/julian/lite-respin/linux/obj/host-libs-2008q3-
72-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --disable-libgomp --enable-poison-s
ystem-directories --with-build-time-tools=/scratch/julian/lite-respin/linux/install
/arm-none-linux-gnueabi/bin --with-build-time-tools=/scratch/julian/lite-respin/lin
ux/install/arm-none-linux-gnueabi/bin
Thread model: posix
gcc version 4.3.2 (Sourcery G++ Lite 2008q3-72)
```

(6) 配置内核 (以支持 eabi)

make menuconfig

搜索 eabi

```

Search Results
Symbol: AEABI [=n]
Type : boolean
Prompt: Use the ARM EABI to compile the kernel
Defined at arch/arm/Kconfig:1678
Location:
-> Kernel Features
Selected by: ARCH_OMAP2PLUS_TYPICAL [=n] && ARCH_OMAP2PLUS [=n] || THUMB

http://blog.csdn.net/

(100%)

```

```

Memory split (3G/1G user/kernel split) --->
Preemption Model (No Forced Preemption (Server)) --->
[*] Use the ARM EABI to compile the kernel
[*] Allow old ABI binaries to run with this kernel (EXPERIMENTAL)
[ ] High Memory Support
Memory model (Flat Memory) --->
[ ] Allow for memory compaction
[ ] Enable KSM for page merging
(4096) Low address space to protect from user allocation
[ ] Enable cleancache driver to cache clean pages if tmem is present
[ ] Use kernel mem{cpy,set}() for {copy_to,clear}_user() (EXPERIMENT
[ ] Enable seccomp to safely compute untrusted bytecode

```

重新编译内核, 下载、烧写、启动

```

14 重新配置内核支持EABI
15 set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2
16 nfs 32000000 192.168.1.123:/work/nfs_root/ulmage_eabi
17 bootm 32000000

```

Q1:我们在修改 linux 中的 MTD 分区的时候是参照 uboot 里面的分区来修改的,而我们之前在写 uboot 的 MTD 的分区的时候是按照 linux-2.2.26 版本的内核 kernel 的分区修改的,我们说 uboot 的分区不会影响内核,因为他是在代码中写死的,我疑惑的是我们为什么要修改按照 uboot 的分区来修改 kernel 里面的分区,是不是 uboot 的分区一定要和 kernel 的分区信息完全保持一致才对?

毕业班第 3 课第 3 节_移植 3.4.2 内核之支持 yaffs 文件系统

yaffs 文件系统是专门为 nand flash 定义的一种文件系统格式,比 jffs 更适用于 nand flash, yaffs 文件系统源码并没有合并到内核里面去,

1. yaffs 文件系统

(1) 源码在 <http://www.yaffs.net/> 下载

进里面我们可以看到在 GIT 中我们可以获得 yaffs

[Home](#) > [Get Yaffs](#)

Get Yaffs

Yaffs is available under the GNU General Public Licence, v2, and under commercial terms.

For testing and experiment you can grab Yaffs under the GPL free of charge right now [from our GIT repository](#) and we strongly recommend you also read [How Yaffs Works](#).

The GIT summary page shows **the changes that have been made over time** in the base code.

用 git 命令来获得，格式如下

If you want to use GIT directly from the command line then public read-only access is available, using the (bash) command:

```
git clone git://www.aleph1.co.uk/yaffs2
```

If you are a registered developer with a server account you can get read/write access using:

```
git clone ssh://www.aleph1.co.uk/home/aleph1/git/yaffs2
```

(2) 如果系统没有装 git 工具，需要先安装 git

```
book@book-desktop:/work/system$ sudo apt-install git
```

安装后下载 yaffs 源码，

```
book@book-desktop:/work/system$ git clone git://www.aleph1.co.uk/yaffs2
Initialized empty Git repository in /work/system/yaffs2/.git/
```

下载后得到一个 yaffs2 目录，里面包含有官方提供的所有源码

```
book@book-desktop:/work/system$ cd yaffs/yaffs2/
book@book-desktop:/work/system/yaffs/yaffs2$ ls
direct          yaffs_checkpoint.h      yaffs_packedtags1.h
Kconfig_multi   yaffs_ecc.c: S3C2440/    yaffs_packedtags2.c
Kconfig_single  yaffs_ecc.h             yaffs_packedtags2.h
linux-tests     yaffs_getblockinfo.h    yaffs_summary.c
Makefile        yaffs_guts.c            yaffs_summary.h
Makefile.kernel yaffs_guts.h             yaffs_tagscompat.c
moduleconfig.h  yaffs_linux.h           yaffs_tagscompat.h
```

如何使用看 readme 文件

查看里面的例子

```
eg.

cd yaffs-dir
./patch-ker.sh c m linux-tree

You will now have to do "make menuconfig" or similar in the Linux tree to
set up the yaffs2 configs. The configs are found under:
File systems/Miscellaneous file systems/ yaffs2
```

(2) 打补丁

进入 yaffs 目录，打补丁

```
book@book-desktop:/work/system/yaffs/yaffs2$ ./patch-ker.sh c m /work/system/linux-3.4.2
Updating /work/system/linux-3.4.2/fs/Kconfig
Updating /work/system/linux-3.4.2/fs/Makefile
```

(3) 配置内核

上面的命令把下载到的 yaffs 源码拷贝到内核里面

```

book@book-desktop:/work/system/yaffs/yaffs2$ cd /work/system/linux-3.4.2
book@book-desktop:/work/system/linux-3.4.2$ ls fs/yaffs2/
Kconfig          yaffs_ecc.h      yaffs_nameval.c  yaffs_tagscompat.h
Makefile         yaffs_getblockinfo.h yaffs_nameval.h  yaffs_trace.h
yaffs_allocator.c yaffs_guts.c     yaffs_nand.c     yaffs_verify.c
yaffs_allocator.h yaffs_guts.h     yaffs_nand.h     yaffs_verify.h
yaffs_attribs.c  yaffs_linux.h    yaffs_packedtags1.c yaffs_vfs.c

```

然后进行配置，搜索 yaffs，选上 yaffs2 file system support

```

Symbol: YAFFS_DOES_ECC [=n]
Type : boolean
Prompt: Lets yaffs do its own ECC
Defined at fs/yaffs2/Kconfig:51
Depends on: MISC_FILESYSTEMS [=y] && YAFFS_FS [=n] && YAFFS_YAFFS1 [=n]
Location:
-> File systems
-> Miscellaneous filesystems (MISC_FILESYSTEMS [=y])
-> yaffs2 file system support (YAFFS_FS [=n])
-> 512 byte / page devices (YAFFS_YAFFS1 [=n])
-> Use older-style on-NAND data format with pageStatus byte (Y

```

(4) 编译时出现错误

结构体没有某个成员

```

fs/yaffs2/yaffs_vfs.c:2514: error: 'struct mtd_info' has no member named 'sync'
fs/yaffs2/yaffs_vfs.c:2515: error: 'struct mtd_info' has no member named 'sync'

```

我们通过查找 mtd_info 的结构体定义，发现里面的成员都已经存在只不过成员的名字有变化，需要添加前缀_，因此我们把编译提示没有的成员的名字全部添加前缀_即可

没有 d_alloc_root 函数，在新的 yaffs 里面用到这个函数，但是我们内核根本没有实现这个函数，应该用成什么函数，

```

fs/yaffs2/yaffs_vfs.c:2967: error: implicit declaration of function 'd_alloc_root'
fs/yaffs2/yaffs_vfs.c:2967: warning: assignment makes pointer from integer without a cast

```

看看 root 在其他文件中怎么用，s_root 是 dentry 结构体

```

sb->s_root = root;

```

搜索 s_root，看看在别的文件系统里面怎么用，用的是 d_make_root 函数，而不是我们的 d_alloc_root 函数，因而把

```

sb->s_root = d_make_root(inode);

```

因而修改函数

```

root = d_make_root(inode); // d_alloc_root(inode);

```

(5) 编译和拷贝

```

Entry Point: 0x30108000
Image arch/arm/boot/uImage is ready
book@book-desktop:/work/system/linux-3.4.2$ cp arch/arm/boot/uImage /work/nfs_root/
uImage_yaffs

```

(6) 制作 yaffs 文件映像

```

book@book-desktop:/work/system/linux-3.4.2$ cd /work/nfs_root/
book@book-desktop:/work/nfs_root$ mkyaffs2image fs_mini_mdev_new fs_mini_mdev_new.y
aaffs2

```

2. 实验

(1) 下载 yaffs 文件系统

```

Hit any key to stop autoboot: 0
SMDFK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev_new.yaffs2
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46

```

(2) 擦除、烧写


```
done
Bytes transferred = 7934784 (791340 hex)
SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.yaffs 30000000 260000 $filesize

NAND write: device 0 offset 0x260000, size 0x791340
7934784 bytes written: OK
```

(3) 设置参数、下载内核

```
7934784 bytes written: OK
SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3
SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage.yaffs
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x20000000, id: 0x90000a46
```

(4) 用 bootm 32000000 启动内核

出现问题，挂载了上去。

```
yaffs: passed flags ""
VFS: Mounted root (yaffs filesystem) on device 31:3.
Freeing init memory: 164K
Kernel panic - not syncing: No init found. Try passing init= option to kernel. See
Linux Documentation/init.txt for guidance.
[<c011abb8>] (unwind_backtrace+0x0/0xf8) from [<c044bfa8>] (panic+0x74/0x1c8)
[<c044bfa8>] (panic+0x74/0x1c8) from [<c0108958>] (init_post+0xc0/0x138)
[<c0108958>] (init_post+0xc0/0x138) from [<c055f384>] (kernel_init+0x16c/0x1bc)
```

用替代法查找问题

```
107
108 7. 没成功，用替代法查找问题
109 7.1 uboot有问题：换上1.1.6的uboot
110 7.2 YAFFS映象有问题
111
```

下载 u-boot

```
SMDK2410 # tftp 30000000 u-boot.bin
ERROR: resetting DM9000 -> not responding
```

擦除、烧写

```
Bytes transferred = 262144 (256K hex)
SMDK2410 # nand erase.part u-boot

NAND erase.part: device 0 offset 0x0, size 0x40000
Erasing at 0x20000 -- 100% complete.
OK
SMDK2410 # nand write 30000000 u-boot

NAND write: device 0 offset 0x0, size 0x40000
262144 bytes written: OK
```

用 reset 命令重启、然后重新下载 yaffs

```
OpenJTAG> nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev_new.yaffs2
ERROR: resetting DM9000 -> not responding
```

1.1.6 版的 u-boot 擦除命令没有之前的 .part 后缀，烧写命令引用长度要加上括号

设置环境变量，下载新内核，然后启动

```

OpenJTAG> nand erase rootfs

NAND erase: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
OpenJTAG> nand write.yaffs 30000000 260000 $(filesize)

NAND write: device 0 offset 0x260000, size 0x791340

Writing data at 0x9d6000 -- 100% complete.
7934784 bytes written: OK
OpenJTAG> set bootargs console=ttySAC0,115200 root=/dev/mtdblock3
OpenJTAG> nfs 32000000 192.168.1.123:/work/nfs_root/uImage_yaffs
ERROR: resetting DM9000 -> not responding

```

(5) 用旧的 u-boot 启动没有问题，所以新的 u-boot 存在问题

查看 cmd_nand.c

如果我们加了 .yaffs 的话，就调用函数 nand_write_skip_bad. 会传入一个参数 flag(WITH_YAFFS_OOB)

```

    } else if (!strcmp(s, ".yaffs")) {
        if (read) {
            printf("Unknown nand command suffix '%s'.\n", s);
            return 1;
        }
        ret = nand_write_skip_bad(nand, off, &rwsize,
                                (u_char *)addr, WITH_YAFFS_OOB);
    }
}

```

函数 nand_write_skip_bad 里面

check_skip_len 函数判断里面有没有坏块，有坏块就忽略并跳过。

```

need_skip = check_skip_len(nand, offset, *length);
if (need_skip < 0) {
    printf("Attempt to write outside the flash area\n");
    *length = 0;
    return -EINVAL;
}

```

如果 flash 没有坏块，且标志项不为 WITH_DROP_FFS 的话，也没有 yaffs 标志，走下面分支。

```

if (!need_skip && !(flags & WITH_DROP_FFS)) {
    rval = nand_write(nand, offset, length, buffer);
    if (rval == 0)
        return 0;

    *length = 0;
    printf("NAND write to offset %llx failed %d\n",
           offset, rval);
}

```

所以要添加 flag&WITH_YAFFS_OOB 这一项 (在 Nand_util.c)

```

if (!need_skip && !(flags & WITH_DROP_FFS) && !(flags & WITH_YAFFS_OOB))
    rval = nand_write(nand, offset, length, buffer);
if (rval == 0)
    return 0;

```

但是我们走的分支是下面，如果上面还需加上一项

```

#ifdef CONFIG_CMD_NAND_YAFFS
    if (flags & WITH_YAFFS_OOB) {
        int page, pages;
        size_t pagesize = nand->writesize;
        size_t pagesize_oob = pagesize + nand->oobsize;
        struct mtd_oob_ops ops;

        ops.len = pagesize;
        ops.ooblen = nand->oobsize;
        ops.mode = MTD_OOB_RAW;
        ops.ooboffs = 0;

        pages = write_size / pagesize_oob;
        for (page = 0; page < pages; page++) {
            WATCHDOG_RESET();
        }
    }
}

```

(6)重新下载烧写 u-boot

```

OpenJTAG> nfs 30000000 192.168.1.123:/work/nfs_root/u-boot_new.bin
ERROR: resetting DM9000 -> not responding

```

```

Bytes transferred = 217492 (35194 hex)
OpenJTAG> nand erase u-boot

NAND erase: device 0 offset 0x0, size 0x40000
Erasing at 0x20000 -- 100% complete.
OK
OpenJTAG> nand write 30000000 u-boot

NAND write: device 0 offset 0x0, size 0x40000
262144 bytes written: OK
? enJTAG> reset
U-Boot 2012.04.01 (Dec 18 2012 - 03:19:19)

```

(7) 下载yaffs 映像文件，擦除分区

```

SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/fs_mini_mdev_new.yaffs2
ERROR: resetting DM9000 -> not responding

```

```

SMDK2410 # nand erase.part rootfs

NAND erase.part: device 0 offset 0x260000, size 0xfda0000
Erasing at 0xffe0000 -- 100% complete.
OK
SMDK2410 # nand write.yaffs 30000000 260000 $filesize

NAND write: device 0 offset 0x260000, size 0x791340
7031784 bytes written: OK

```

(8) 下载内核、烧写启动内核成功

```

SMDK2410 # nfs 32000000 192.168.1.123:/work/nfs_root/uImage.yaffs
ERROR: resetting DM9000 -> not responding

SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3
SMDK2410 # bootm 32000000
## Booting kernel from Legacy Image at 32000000 ...

```

毕业班第3课第4节_移植 3.4.2 内核之裁剪及 ECC 简介及制作补丁

1、裁剪内核

(1) 查看内核大小

```

book@book-desktop:/work/system/linux-3.4.2$ ls arch/arm/boot/uImage -l
-rw-r--r-- 1 book book 4486192 2012-12-18 03:12 arch/arm/boot/uImage
book@book-desktop:/work/system/linux-3.4.2$ make menuconfig

```

(2) 裁剪 (通过 make menuconfig 去掉不需要的项)

把一些单板文件去掉

```

book@book-desktop:~$ cd /work/system/linux-3.4.2
book@book-desktop:/work/system/linux-3.4.2$ vi .config

```

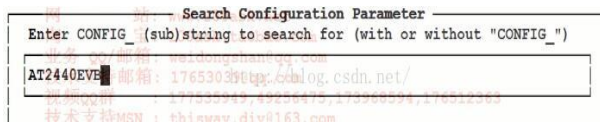
看.config 文件，看一下有哪些东西，如单板：搜索 mini2440 的单板，

```

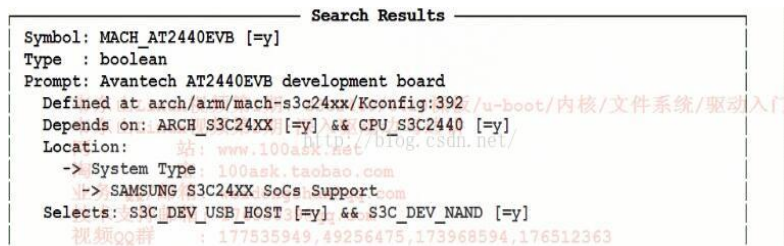
#
# S3C2440 Boards
#
CONFIG_MACH_ANUBIS=y
CONFIG_MACH_AT2440EVB=y
CONFIG_MACH_MINI2440=y
CONFIG_MACH_NEXCO_2440=y
CONFIG_MACH_OSIRIS=y
CONFIG_MACH_OSIRIS_DVS=m
CONFIG_MACH_RX3715=y
CONFIG_ARCH_S3C2440=y
CONFIG_SMDK2440_CPU2440=y

```

如单板 AT2440EVB 没见过可以在搜索栏中搜索并去掉 (输入斜杠进入搜索界面)



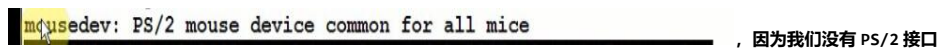
搜索结果如下，到相应位置去掉不必要项



把一些不需要的文件系统去掉

前面带 M 的是作为模块的不用管，只需理会前面带*的，因为是要编译进内核的。

(3) 看输出信息，看一下有哪些我们不需要

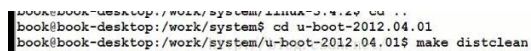


再看一下临时生成的文件系统，看一下不需要的文件系统

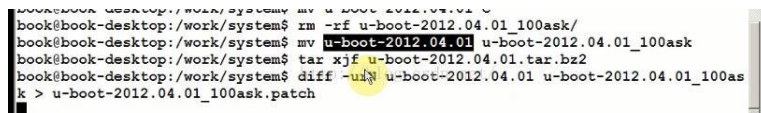


2、重新制作 u-boot，因为之前有一个 bug

(1) 先清除掉生成的目标文件

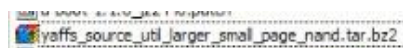


(2) 重命名改好的 u-boot，解压原版的 u-boot，从而制作补丁

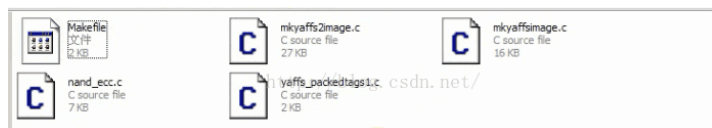


3、mkyaffs2image 工具

(1) 解压文件



(2) 解压后是 Development_util_ok\yaffs2\utils 目录下

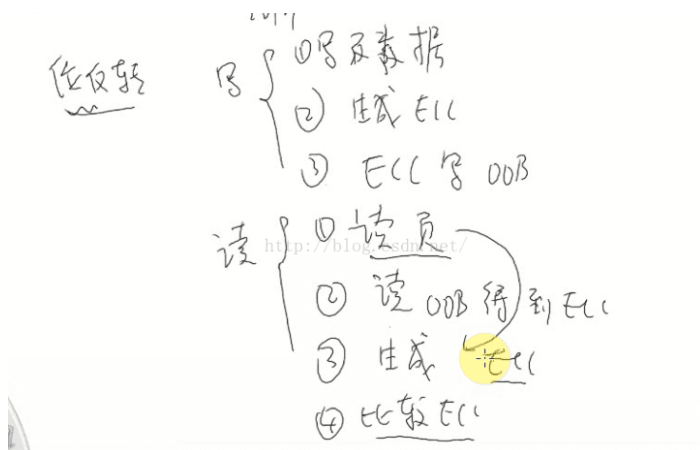


(3) 在 utils 目录下执行 make 命令可以生成 mkyaffs2image 工具

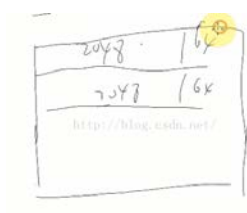
我们用 git 命令下载的 yaffs 里面有这个工具，但是不完整，编译时有问题的

4、ECC 简介

(1) 介绍

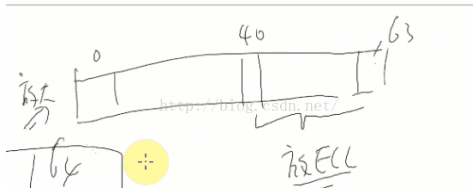


(2) 我们制作出来的 yaffs 映像, mkyaffs2image 工具生成的映像文件既包含一页数据, 也包含 OOB 数据

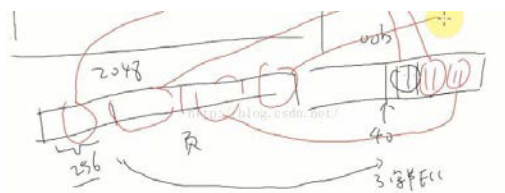


(3) OOB 里面的数据如何排放 (看 Mkyaffs2image.c)

useecc 表示使用 ecc, eccpos 表示 ecc 的位置, OOB 是 64 字节,



校验码是每 256 个字节生成 3 字节的 ECC 码, ECC 码总共 24 字节



obbfree 表示 oob 除了一部分放 ECC, 其他可以让你使用, 这里 2 是表示从哪个字节开始, 总共有 38 个字节 (39-2+1), yaffs 把它用作 yaffs_tag

```
static struct nand_oobinfo nand_oob_64 = {
    .useecc = MTD_NANDECC_AUTOPLACE,
    .eccbytes = 24,
    .eccpos = {
        40, 41, 42, 43, 44, 45, 46, 47,
        48, 49, 50, 51, 52, 53, 54, 55,
        56, 57, 58, 59, 60, 61, 62, 63,
    },
    .oobfree = { (2, 38) }
};
```

第 0 位表示是否坏块, 0xFF 表示为好, 0 表示是坏的。

5. 实验

制作了 u-boot_new.bin, uImage_new, fs_mini_mdev_new.yaffs

重烧整个系统:

使用 jtag 工具烧 u-boot_new.bin

或使用 uboot 来更新自己: tftp 30000000 u-boot_new.bin; nand erase.part u-boot; nand write 30000000 u-boot

启动 uboot, 用它来烧写内核、FS

```
tftp 30000000 uImage_new; nand erase.part kernel; nand write 30000000 kernel

tftp 30000000 fs_mini_mdev_new.yaffs2; nand erase.part rootfs; nand write.yaffs 30000000 260000 $filesize
```

设置参数

```
set 'nand read 30000000 kernel;bootm 30000000'

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3

set machid 16a

save
```

6. 制作内核补丁

(1) 制作补丁前把.config 拷贝为 config_ok,保存内核的配置

```
book@book-desktop:/work/system/linux-3.4.2$ cp .config config_ok
```

(2) 清除生成的目标文件

```
book@book-desktop:/work/system/linux-3.4.2$ make distclean
```

(3) 更改修改后的内核的名字, 解压原版内核, diff 命令生成补丁文件

```
book@book-desktop:/work/system$ mv linux-3.4.2 linux-3.4.2_100ask
book@book-desktop:/work/system$ tar xjf linux-3.4.2.tar.bz2
book@book-desktop:/work/system$ diff -uzN linux-3.4.2 linux-3.4.2_100ask > linux-3.4.2_100ask.patch
```

(3) 给内核打补丁, 然后 cp config_ok .config, 然后进行 make uImage 编译内核

毕业班第 4 课第 1 节_移植驱动到 3.4.2 内核

毕业班第 4 课第 1 节_移植驱动到 3.4.2 内核之 DM9000

这里的移植网卡部分主要有两部分内容:

- 1: 移植 linux3.4.2 内核自带的网卡驱动让他支持我们的 s3c2440
- 2: 由于我们 linux3.4.2 内核自带的网卡驱动比较老, 我们希望移植之前第二期的修改好的网卡到 linux3.4.2 中

移植过程中基本上会遇到下面的问题:

```
1 移植:
2 1. 编译
3 2. 解决错误
4 2.1 头文件不对: 去掉或改名
5 2.2 头文件不对: 改名使用新头文件
6 2.3 有些函数没有了: 改名使用新函数
考东山Linux移植第1期: S3C2440/
```

移植网卡是因为以后用到的其他驱动程序用 nfs 比较好用

- 1: 移植 linux3.4.2 内核自带的网卡驱动让他支持 smdk2440

我们先打开 dm9000.c 文件, 发现人家是用平台驱动来的结构来实现驱动的

```
static struct platform_driver dm9000_driver = {
    .driver = {
        .name = "dm9000",
        .owner = THIS_MODULE,
        .pm = &dm9000_drv_pm_ops,
    },
    .probe = dm9000_probe,
    .remove = __devexit_p(dm9000_drv_remove),
};
```

也就是说如果内核中有同名的 dm9000 的设备就会调用上面的探测函数, 搜索时候有 dm9000 的设备

```
[root@fedora Linux-3.4.2]# grep -nr "\"dm9000\""
Documentation/networking/dm9000.txt:54: .name
arch/arm/mach-s3c24xx/mach-vr1000.c:245: .name = "dm9000",
arch/arm/mach-s3c24xx/mach-bast.c:340: .name = "dm9000",
arch/arm/mach-s3c24xx/mach-mini2440.c:321: .name = "dm9000",
arch/arm/mach-at91/board-sam9261ek.c:98: .name = "dm9000",
arch/arm/mach-davinci/board-dm355-legend.c:164: .name = "dm9000"
```

我们发现 mini2440 里面有 dm9000 设备, 而 smdk2440 并没有, 也就说 mini2440 可能已经支持了 dm9000

1. 实验 (内核对 mini2440 已经支持网卡)

(1) 设置机器 ID 为 Mini2440, 启动

```
Hit any key to stop autoboot: 期0 S3C2440/64
SMDK2410 # set machid 7CF 第2期:深入驱动边
SMDK2410 # boot: //blog.csdn.net/
```

(2) 配置 IP, ping 一下服务器

```
/ # ifconfig eth0 192.168.1.17 2449.59286475,173568594,176512363
dm9000 dm9000: eth0: link down
/ # dm9000 dm9000: eth0: link up, 100Mbps, full-duplex, lpa 0xC1E1
ping 192.168.1.123
PING 192.168.1.123 (192.168.1.123): 56 data bytes
64 bytes from 192.168.1.123: seq=0 ttl=64 time=2.338 ms 看密码一个
64 bytes from 192.168.1.123: seq=1 ttl=64 time=0.887 ms 能3000元
64 bytes from 192.168.1.123: seq=2 ttl=64 time=0.887 ms
```

(3) 挂载文件系统

```
/ # mount -t nfs -o nolock,vers=2 192.168.1.123:/work/nfs_root/fs_mini_mdev_new
/mnt
/ # ls /mnt
bin etc linuxrc proc http://blog.csdn.net/
dev lib mnt root sys usr
```

2. 内核对 mini2440 已经支持了网卡, 对 smdk2440 不支持。

下面我们来看看为什么 smdk2440 对网卡驱动不支持

在 DM9000.c 驱动程序中, 先看入口函数, 是注册一个 platform_driver 结构体

```
dm9000_init(void)
{
    printk(KERN_INFO "%s Ethernet Driver, V%s\n", CARDNAME, DRV_VERSION);
    return platform_driver_register(&dm9000_driver);
}
```

platform_driver 结构体在内核里面有同名的 DM9000 时, probe 函数才被调用, 然后在 probe 函数中实现如下内容:

根据输入参数平台设备结构体确定访问基地址, 去基地址里面看看 DM9000 是否在基地址上。读某个寄存器看看能否读取它的 ID

获得内存类资源的第 0 个和第 1 个资源还有中断类资源的第 0 个。

```
db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
db->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

```
static struct platform_driver dm9000_driver = {
    .driver = {
        .name = "dm9000",
        .owner = THIS_MODULE,
        .pm = &dm9000_drv_pm_ops,
    },
    .probe = dm9000_probe,
    .remove = __devexit_p(dm9000_drv_remove),
};
```

在内核中搜索 DM9000, 在 Mach-mini2440.c 里面有平台设备结构体, 她的名字是 DM9000

```
static struct platform_device mini2440_device_eth = {
    .name = "dm9000",
    .id = -1,
    .num_resources = ARRAY_SIZE(mini2440_dm9k_resource),
    .resource = mini2440_dm9k_resource,
    .dev = {P http://blog.csdn.net/},
    .platform_data = &mini2440_dm9k_pdata,
};
```

我们来看看为什么 mini2440 可以支持了 dm9000, 我们已经知道 mini2440 有定义了 platform_device mini2440_device_eth, 那么他要能够支持 dm9000 就必须把它这个 device 结构体注册到平台设备里面去, 如何实现呢?

先放在平台设备数组里面。

```
static struct platform_device *mini2440_devices[] __initdata = {
    &s3c_device_ohci,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_rtc, http://blog.csdn.net/
    &s3c_device_usb gadget,
    &mini2440_device_eth,
    &mini2440_led1,
};
```

将数组注册进内核

```
platform_add_devices(mini2440_devices, ARRAY_SIZE(mini2440_devices));
```

这样 mini2440 就能支持 dm9000 了, 接下来我们看看 smdk2440 对应的文件有没有对应的平台 device 结构体

而我们的 Mach-smdk2440.c 里面没有那个平台设备, 所支持的平台设备只有下面几项

```
static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_ohci,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
};
```

想内核在使用机器 ID 的时候能支持网卡的话, 需要有网卡的平台设备结构体

3、修改 (Mach-smdk2440.c)

(1) 参考 Mach-mini2440.c 里面的平台设备结构体

```
static struct dm9000_plat_data smdk2440_dm9k_pdata = {
    .flags = (DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM),
};

static struct platform_device smdk2440_device_eth = {
    .name = "dm9000",
    .id = -1,
    .num_resources = ARRAY_SIZE(smdk2440_dm9k_resource),
    .resource = smdk2440_dm9k_resource,
    .dev = {
        .platform_data = &smdk2440_dm9k_pdata,
    },
};
```

(2) 参考所谓的资源, 网卡是内存接口的芯片, 要指定它的访问地址。所以资源数组中有它的访问基址和中断号

```
#define MACH_SMDK2440_DM9K_BASE (S3C2410_CS4 + 0x300)

static struct resource smdk2440_dm9k_resource[] = {
    [0] = {
        .start = MACH_SMDK2440_DM9K_BASE,
        .end = MACH_SMDK2440_DM9K_BASE + 3,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = MACH_SMDK2440_DM9K_BASE + 4,
        .end = MACH_SMDK2440_DM9K_BASE + 7,
        .flags = IORESOURCE_MEM,
    },
    [2] = {
        .start = IRQ_EINT7,
        .end = IRQ_EINT7,
        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHEDGE,
    },
};
```

(3) 在平台设备数组中加入我们的平台设备

```
static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_ohci,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    &smdk2440_device_eth,
};
```

(4) 驱动里应该设置内存控制器, 没有设置也能够用, 因为没有设置可能 u-boot 里面已经帮我们设置啦

4、实验 1

(1) 编译修改后的内核

(2) 出现问题及解决

第一个错误是缺乏某些头文件

```
CC arch/arm/mach-s3c24xx/mach-smdk2440.o
arch/arm/mach-s3c24xx/mach-smdk2440.c:180: error: variable 'smdk2440_dm9k_pdata' has
s initializer but incomplete type
arch/arm/mach-s3c24xx/mach-smdk2440.c:181: error: unknown field 'flags' specified in
initializer
arch/arm/mach-s3c24xx/mach-smdk2440.c:181: error: 'DM9000_PLATF_16BITONLY' undeclar
ed here (not in a function)
arch/arm/mach-s3c24xx/mach-smdk2440.c:181: error: 'DM9000_PLATF_NO_EEPROM' undeclar
ed here (not in a function)
```

(3) 内核自带的内核驱动版本过低, 在网卡移植中是从厂家获得最新的驱动程序

```
#define CARDNAME "dm9000"
#define DRV_VERSION "1.31"
```


(4) 下载, 启动内核, 设置 smdk2440 的机器 ID0x16a

```
bytes transferred - 2000000 (1100000 hex)
SMDK2410 # set machid 16a
SMDK2410 # bootm 32000000
Wrong Image Format for bootm command
ERROR: can't get kernel image!
SMDK2410 # bootm 32000000
## Booting kernel from Legacy Image at 32000000 ...
Image Name: Linux-3.4.2
```

(5) 配置 IP, 挂载文件系统

```
/ #
/ # ifconfig eth0 192.168.1.17
dm9000 dm9000: eth0: link down
/ # dm9000 dm9000: eth0: link up, 100Mbps, full-duplex, lpa 0xC1E1
/ # mount -t nfs -o nolock,vers=2 192.168.1.123:/work/nfs root/fs mini mdev new
/mnt
```

2: 移植第二期视频中的厂家提供的新的网卡驱动程序

(1) 把新的网卡驱动程序和相应的 Makefile 放到我们新建的目录

```
book@book-desktop:/work/drivers_and_test_new$ ls
17th_dm9000c
book@book-desktop:/work/drivers_and_test_new$ cd 17th_dm9000c/
book@book-desktop:/work/drivers_and_test_new/17th_dm9000c$ ls
dm9dev9000c.c Makefile
```

(2) 修改 Makefile

把内核目录修改成我们新的内核目录 (里面的内核必须配置好, 编译过)

```
KERN_DIR = /work/system/linux-3.4.2
all:
    make -C $(KERN_DIR) M=`pwd` modules
clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order
obj-m += dm9dev9000c.o
~
```

(2) 进行编译出现一大堆错误, 进行修改

```
book@book-desktop:/work/drivers_and_test_new/17th_dm9000c$ make
make -C /work/system/linux-3.4.2 M=`pwd` modules
```

没有某个头文件, 先注释掉, 出错再说, 把我们的网卡驱动加入内核的 sourceinsight 中方便查阅。出错时找不出原因可参考类似的驱动程序如

DM9000.c

编译报错修改步骤:

```
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:97:39: error: asm/arch-s3c2410/regs-
A: s-mem.h: No such file or directory
```

```
00096: #include <asm/IO.h>
00097: // #include <asm/arch-s3c2410/regs-mem.h>
00098:
```

```
CC [M] /root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.o
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:302: error: expected '=', ',', ';',
B: 'asm' or '__attribute__' before 'dmfe_interrupt'
```

→这是由于 irqreturn_t 未定义, 在 irqreturn.h 中定义的, 我们参照 dm9000.c 文件中没看到他有包含这个.h, 但是有包含一个 interrupt.h

发现这个 interrupt.h 中有包含 irqreturn.h, 所以我们直接添加 interrupt.h

```
00086: #include <asm/uaccess.h>
00087: #include <linux/interrupt.h>
00088:
00089: #ifdef CONFIG_ARCH_MAINSTONE
```

```
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:358: error: implicit declaration of
C: f function 'SET_MODULE_OWNER'
```

→在 sourceinsight 中看到 SET_MODULE_OWNER 这个是灰色的, 说明没有定义, 所以我们直接屏蔽好了


```

00357:
00358: //          SET_MODULE_OWNER(dev);
00359: err = dmfe_probe1(dev);

/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:409: error: 'struct net_device' has no member named 'priv'
D:

```

我们看对应的 dm9000.c 中的函数

```

2: static int __devinit
3: dm9000_probe(struct platform_device *pdev)
4: {
5:     struct dm9000_plat_data *pdata = pdev->dev.platform_data;
6:     struct board_info *db; /* Point a board information structure */
7:     struct net_device *ndev;
8:     const unsigned char *mac_src;
9:     int ret = 0;
10:    int i;
11:    int i;
12:    u32 id_val;
13:
14:    /* Init network device */
15:    ndev = alloc_etherdev(sizeof(struct board_info));
16:    if (!ndev)
17:        return -ENOMEM;
18:
19:    SET_NETDEV_DEV(ndev, &pdev->dev);
20:
21:    dev_dbg(&pdev->dev, "dm9000_probe()\n");
22:
23:    /* setup board info structure */
24:    db = netdev_priv(ndev);
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48:
49:
50:
51:
52:
53:
54:
55:
56:
57:
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
70:
71:
72:
73:
74:
75:
76:
77:
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
89:
90:
91:
92:
93:
94:
95:
96:
97:
98:
99:
100:

```

将二期中的网卡的驱动修改为

```

23:
24:     /* Allocated board information structure */
25:     db=netdev_priv(dev);
26:     memset(db, 0, sizeof(struct board_info));
27:     // db = (board_info_t *)dev->priv;
28:     dmfe dev = dev;
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48:
49:
50:
51:
52:
53:
54:
55:
56:
57:
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
70:
71:
72:
73:
74:
75:
76:
77:
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
89:
90:
91:
92:
93:
94:
95:
96:
97:
98:
99:
100:

```

```

/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_probe1':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:425: error: 'struct net_device' has no member named 'open'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:426: error: 'struct net_device' has no member named 'hard_start_xmit'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:428: error: 'struct net_device' has no member named 'tx_timeout'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:429: error: 'struct net_device' has no member named 'stop'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:430: error: 'struct net_device' has no member named 'get_stats'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:431: error: 'struct net_device' has no member named 'set_multicast_list'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:432: error: 'struct net_device' has no member named 'do_ioctl'
E: /root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_open':

```

→这些都是在 linux3.4.2 中 net_device 结构体中没有这些成员造成的,我们的做法应该是去参照 dm9000.c 中找到这些成员,看他们是怎么使用的,发现这些成员在 dm9000.c 中用了一个其他的结构体定义:

```

:
: static const struct net_device_ops dm9000_netdev_ops = {
:     .ndo_open          = dm9000_open,
:     .ndo_stop          = dm9000_stop,
:     .ndo_start_xmit     = dm9000_start_xmit,
:     .ndo_tx_timeout     = dm9000_timeout,
:     .ndo_set_rx_mode    = dm9000_hash_table,
:     .ndo_do_ioctl       = dm9000_ioctl,
:     .ndo_change_mtu     = eth_change_mtu,
:     .ndo_set_features   = dm9000_set_features,
:     .ndo_validate_addr  = eth_validate_addr,
:     .ndo_set_mac_address = eth_mac_addr,
: #ifdef CONFIG_NET_POLL_CONTROLLER
:     .ndo_poll_controller = dm9000_poll_controller,
: #endif
: };

```

我们在二期的网卡驱动中添加这个结构体并且

将我们的二期中网卡驱动中的提示未定义的成员用这些来替换

E.1:在二期网卡驱动中添加 net_device_ops 结构体

```

00379: static const struct net_device_ops dm9000_netdev_ops = {
00380:     .ndo_open          = dm9000_open,
00381:     .ndo_stop          = dm9000_stop,
00382:     .ndo_start_xmit     = dm9000_start_xmit,
00383:     .ndo_tx_timeout     = dm9000_timeout,
00384:     .ndo_set_rx_mode    = dm9000_hash_table,
00385:     .ndo_do_ioctl       = dm9000_ioctl,
00386:     .ndo_change_mtu     = eth_change_mtu,
00387:     .ndo_set_features   = dm9000_set_features,
00388:     .ndo_validate_addr  = eth_validate_addr,
00389:     .ndo_set_mac_address = eth_mac_addr,
00390: #ifdef CONFIG_NET_POLL_CONTROLLER
00391:     .ndo_poll_controller = dm9000_poll_controller,
00392: #endif
00393: };
00394:

```

E.2: 替换上面的结构体中的成员名字替换成编译时提示未定义的成员,而不是把编译时提示未定义的成员替换成上面的结构体的名字,因为那样的话还需要把每个函数的定义的地方和声明的地方都要改名字,比较麻烦,替换后如下:

```

378:
379: static const struct net_device_ops dm9k_netdev_ops = {
380:     .ndo_open      = dmfe_open,
381:     .ndo_stop      = dmfe_stop,
382:     .ndo_start_xmit = dmfe_start_xmit,
383:     .ndo_tx_timeout = dmfe_timeout,
384:     .ndo_set_rx_mode = dm9000_hash_table,
385:     .ndo_do_ioctl   = dmfe_do_ioctl,
386:     .ndo_change_mtu = eth_change_mtu,
387:     .ndo_get_stats  = dmfe_get_stats,
388:     // .ndo_set_features = dm9000_set_features,
389:     .ndo_validate_addr = eth_validate_addr,
390:     .ndo_set_mac_address = eth_mac_addr,
391:     #ifdef CONFIG_NET_POLL_CONTROLLER
392:     .ndo_poll_controller = dm9000_poll_controller,
393:     #endif
394: };

```

将编译是替换成上面的之后,还有几个问题上标注了,

1:要改个结构体的名字,因为担心和 dm9000.c 冲突

2:编译是提示

```

/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:430: error: 'struct net_device' has no member named 'get_stats'

```

但是我们从 dm9000.c 中拷贝的这个 net_device_ops 也没有找到这个成员,我们就自己添加了这个成员

3:从 dm9000.c 拷贝的 net_device_ops 中的 dm9000_set_features 成员在我们这并没有被使用,所以屏蔽

E.3:编译报错的地方代码如下:

```

dev->base_addr      = iobase;
dev->irq             = irq;
dev->open             = &dmfe_open;
dev->hard_start_xmit = &dmfe_start_xmit;
dev->watchdog_timeo  = 5*HZ;
dev->tx_timeout       = dmfe_timeout;
dev->stop             = &dmfe_stop;
dev->get_stats        = &dmfe_get_stats;
dev->set_multicast_list = &dm9000_hash_table;
dev->do_ioctl         = &dmfe_do_ioctl;
Linux_VERSION_CODE >= KERNEL_VERSION(2,4,28)
dev->ethtool_ops = &dmfe_ethtool_ops;

```

这样我们直接就把报错的成员屏蔽掉,然后让 dev 指向我们的 E.2 中结构体即可:

```

/* driver system function */
dev->netdev_ops = &dm9k_netdev_ops;

dev->base_addr      = iobase;
dev->irq             = irq;
// dev->open         = &dmfe_open;
// dev->hard_start_xmit = &dmfe_start_xmit;
dev->watchdog_timeo  = 5*HZ;
// dev->tx_timeout    = dmfe_timeout;
// dev->stop          = &dmfe_stop;
// dev->get_stats      = &dmfe_get_stats;
// dev->set_multicast_list = &dm9000_hash_table;
// dev->do_ioctl       = &dmfe_do_ioctl;
Linux_VERSION_CODE >= KERNEL_VERSION(2,4,28)

```

F:再次编译报错提示:

```

CC [R] /root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.o
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_open':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:501: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_init_dm9000':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:685: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_start_xmit':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:768: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_stop':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:879: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_tx_done':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:912: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_interrupt':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:961: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_get_stats':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1028: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_do_ioctl':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1063: error: 'struct net_device' has no member named 'priv'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dmfe_timeout':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1092: error: 'struct net_device' has no member named 'priv'

```

→都是提示我们刚刚的那些函数中没有 priv 成员,把这些全部替换成下面的作图所示

```

static int dmfe_open(struct net_device *dev)
{
    board_info_t *db = netdev_priv(dev);
    u8 reg_nsr;
}

static int dmfe_open(struct net_device *dev)
{
    board_info_t *db = (board_info_t *)dev->priv;
    u8 reg_nsr;
}

```

```

CC [R] /root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.o
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c: In function 'dm9000_hash_table':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1402: error: 'struct net_device' has no member named 'mc_list'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1403: error: 'struct net_device' has no member named 'mc_count'
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1440: warning: dereferencing pointer

```

G:再次编译报错:

这里我们直接把 dm9000_hash_table 函数用 dm9000.c 中的函数替换

H:再次编译

```
locked':
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1466: error: 'RCR_DIS_LONG' undeclared (first use in this function)
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1466: warning: (Each undeclared identifier is used only once; this warning can be removed by adding 'extern'.)
```

发现是几个宏没定义,添加 dm9000.h,要注意的是如果仅仅是 include 这个头文件并没有用,还要把这个文件复制到我们当前的文件夹内

I:再次编译:

```
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1694: error: unknown field 'get_rx_csum' specified in initializer
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1695: error: unknown field 'set_rx_csum' specified in initializer
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1695: warning: initialization from incompatible pointer type
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1696: error: unknown field 'get_tx_csum' specified in initializer
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1696: warning: initialization from incompatible pointer type
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1697: error: unknown field 'set_tx_csum' specified in initializer
/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.c:1697: warning: initialization from incompatible pointer type
scripts/Makefile.build:313: recipe for target '/root/Dev/work/oschina_git/drivers_and_test_new/17th_dm9000c/dm9dev9000c.o' failed
```

→我们发现这几个成员都没有被人使用,所以我们直接屏蔽

至此,编译成功

(3) 编译后编进内核里面用一下 (make)

把 dm9dev9000c.c 中拷贝到相应的网卡目录中, 然后去修改它的 Makefile

```
book@book-desktop:/work/drivers_and_test_new/17th_dm9000c$ cp dm9dev9000c.c /work/system/linux-3.4.2/drivers/net/ethernet/davicom/
book@book-desktop:/work/drivers_and_test_new/17th_dm9000c$ cd /work/system/linux-3.4.2/drivers/net/ethernet/davicom/
book@book-desktop:/work/system/linux-3.4.2/drivers/net/ethernet/davicom$ vi Makefile
```

替换成我们的文件

```
192.168.1.123 | com-7 | 192.168.1.123 (1) |
# Makefile for the Davicom device drivers.
#
#obj-$(CONFIG_DM9000) += dm9000.o
obj-$(CONFIG_DM9000) += dm9dev9000c.o
```

然后在内核目录中编译内核

```
book@book-desktop:/work/system/linux-3.4.2/drivers/net/ethernet/davicom$ make uImage
```

(3) 使用新内核启动

用 nfs 下载内核, 启动内核

```
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/uImage_net_new
dm9000 i/o: 0x20000000, id: 0x90000a46
```

看看网卡是否能用, 配置网卡 IP, ping 别的 IP

```
/ # ifconfig eth0 192.168.1.17
/ # ifconfig
/ # ping 192.168.1.123
PING 192.168.1.123 (192.168.1.123): 56 data bytes
64 bytes from 192.168.1.123: seq=0 ttl=64 time=1.954 ms
64 bytes from 192.168.1.123: seq=1 ttl=64 time=0.911 ms
```

挂载根文件系统

```
/ # mount -t nfs -o nolock,vers=2 192.168.1.123:/work/nfs_root/fs_mini_mdev_new /mnt
/ # ls /mnt
bin      etc      linuxrc  proc     sbin     tmp
dev      lib      mnt      root     sys      usr
```

6. nfs 作为根文件系统使用

(1) 设置启动参数

最后的 ip 是单板自己的 ip(在内核中搜索一个文档 nfsroot.txt)

怎么设置参数

```
SMDK2410 #
SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/nfs nfsroot=192.168.1.123:/work/nfs_root/fs_mini_mdev_new ip=192.168.1.17:192.168.1.123:192.168.1.1:255.255.255.0::eth0:off
```

save 保存后启动，分号隔开可以写多个命令

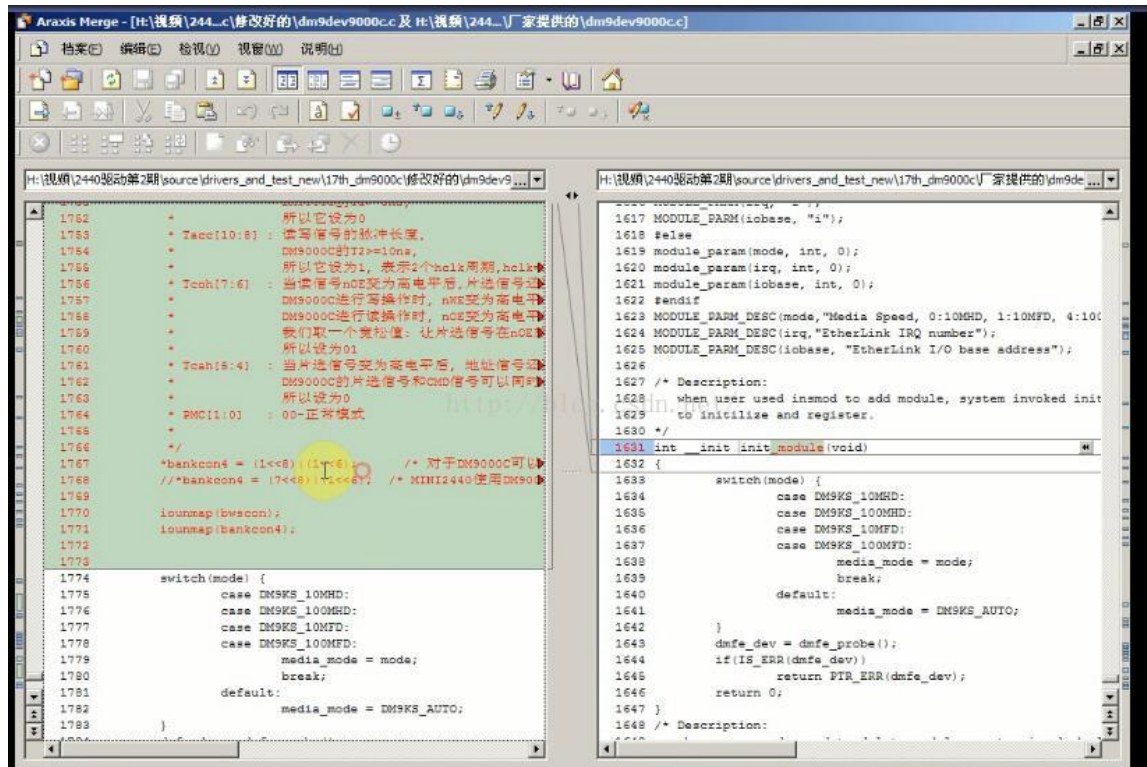
```
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/uImage_net_new; bootm 30000000
```

出现如下错误，虽然挂载上去啦

```
NO soundcard found.
VFS: Mounted root (nfs filesystem) on device 0:10.
Freeing init memory: 128K
dm9ks: Rxbyte error!
dm9ks: Rxbyte error! http://blog.csdn.net/
dm9ks: Rxbyte error!
dm9ks: Rxbyte error!
```

(2) 在内核中搜索 Rxbyte error!,跟第二期中原厂提供未修改的网卡驱动对比

看看哪里设置会有问题。可能时间参数有问题。



修改完时间参数之后就 ok 了

毕业班第 4 课第 2 节_移植驱动到 3.4.2 内核之 LED 和按键

本节视频是讲了之前第一期和第二期视频中关于 LED 和按键的全部驱动移植到新的 linux 内核中

包含了 first_drv 到 8th_drv

大致的流程和上面的网卡驱动一样，都是把之前的驱动复制到虚拟机，然后修改 makefile 中内核的目录

然后 make，解决编译错误，将驱动程序和测试程序复制到文件系统中，这里的应该是用了上节视频中的 nfs 作为文件系统，

然后装载驱动，测试驱动，再卸载该驱动，如果都成功，说明这个驱动程序移植成功了

1. 实验

(1) 把驱动拷贝到服务器去，然后修改 Makefile

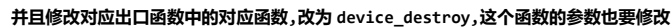
(2) 使用新内核的目录

(3) make 编译出现一大堆错误

这里只介绍几个复杂的错误修改

class_device_create 函数呈现灰色，说明新版本的内核没有该函数的实现，

另外一个小技巧:可以让 sourceinsight 在指定目录下搜索关键字,比如在字符设备目录下搜索



(4) 编译好的文件拷贝到网络文件系统中去

(5) 重新编译测试程序，把测试程序也拷贝到网络文件系统中

```

book@book-desktop:/work/drivers_and_test_new/first_drv$ ls
first_drv.c  first_drv.mod.c  first_drv.o  firstdrvtest.c  modules.order
first_drv.ko  first_drv.mod.o  firstdrvtest  Makefile  Module.symvers
book@book-desktop:/work/drivers_and_test_new/first_drv$ arm-linux-gcc -o firstdrvte
st firstdrvtest.c
book@book-desktop:/work/drivers_and_test_new/first_drv$ cp firstdrvte /work/nfs_ro
ot/fs_mini_mdev_new
cp: cannot stat `firstdrvte': No such file or directory
book@book-desktop:/work/drivers_and_test_new/first_drv$ cp firstdrvtest /work/nfs_
root/fs_mini_mdev_new

```

(5) 在单板上安装驱动，查看安装的驱动，和注册的设备节点

```

/ # insmod first_drv.ko
/ # lsmod
first_drv 1313 0 - Live 0xbf000000 (0)
/ # ls /dev/xyz -l
crw-rw---- 1 0 0 253, 0 Jan 1 00:17 /dev/xyz

```

在卸载驱动中出现问题，创建需要的目录

```

/ # lsmod
third_drv 1984 2 - Live 0xbf000000 (0)
/ # rmmod third_drv
rmmod: chdir(/lib/modules): No such file or directory
/ # mkdir /lib/modules
/ # rmmod third_drv
rmmod: chdir(3.4.2): No such file or directory
/ # mkdir /lib/modules/3.4.2
/ # rmmod third_drv
rmmod: remove 'third_drv': Resource temporarily unavailable
/ #

```

驱动程序在用，需要把应用程序杀掉

```

884 0 0:00 -/bin/sh
889 0 0:00 ./thirddrvtest
890 0 0:00 [flush-0:10]
898 0 0:00 ps
/ # kill -9 889
/ #
[1]+ Killed ./thirddrvtest
/ #

```

cat /proc/devices 查看设备，也有我们 253 这个设备

```

253 first_drv
254 rtc

Block devices:
1 ramdisk
259 blkext
7 loop
8 sd
31 mtdblock
65 sd

```

(6) 执行测试程序

on 的时候是点灯，off 的时候是灭灯

```

/ # ./firstdrvtest
Usage:
./firstdrvtest <on|off>
/ # ./firstdrvtest on
/ # ./firstdrvtest off
/ #

```

2. 卸载驱动出现问题

```

/ # rmmod third_drv
Unable to handle kernel paging request at virtual address ffffffff
pgd = c3ac4000
[ffffffffff] *pgd=33ffe831, *pte=00000000, *ppte=00000000

```

相关的函数可能有问题如 device_destroy 函数和 third_drv_exit 函数

```

Backtrace:
[<02ec074>] (klist_next+0x0/0xe0) from [<0199160>] (class_dev_iter_next+0x18/0x40)
r7:c0194814 r6:c3a51efc r5:c3a51ecc r4:c3a51ecc
[<0199148>] (class_dev_iter_next+0x0/0x40) from [<01993a4>] (class_find_device+0x
5c/0xc4)
r5:c3a51ecc r4:c3a2e598
[<0199348>] (class_find_device+0x0/0xc4) from [<0195d4c>] (device_destroy+0x24/0x
44)
r7:c3a50000 r6:bf000480 r5:00000013 r4:bf0005b4
[<0195d28>] (device_destroy+0x0/0x44) from [<bf000034>] (third_drv_exit+0x34/0x58
[third_drv])
r4:bf0005b4
[<bf000000>] (third_drv_exit+0x0/0x58 [third_drv]) from [<c004d670>] (sys_delete_mo
dule+0x1f8/0x258)
r5:00000013 r4:00000000
[<004d478>] (sys_delete_module+0x0/0x258) from [<c0009280>] (ret_fast_syscall+0x0/
0x2c)
r8:c0009404 r7:00000081 r6:bea6bf93 r5:00000000 r4:000ef008
Code: e3a03000 e1540002 e5853004 0a00000d (e5943000)
---[ end trace 71a61927cc1193bb ]---
Segmentation fault

```


这里是由于之前我们的对应出口函数里面卸载函数的参数没有修改只改了出口函数中卸载驱动的函数名

```
0074: static void first_drv_exit(void)
0075: {
0076:     unregister_chrdev(major, "first_drv"); // 卸载
0077:
0078:     device_destroy(firstdrv_class, MKDEV(major, 0));
0079:     class_destroy(firstdrv_class);
0080:     iounmap(gpicon);
0081: }
0082:
```

修改如上即可

Second_drv 的修改直接用对比工具和 1th 对比修改即可,没什么难点

3th 中

1:我们用到的 GPIO 的引脚宏定义的方式需要修改

2:中断的触发方式的宏定义也需要修改

6th 中

有用到互斥信号量,而新的内核的定义方式有变动,我们要修改两点:

1:右边为改动前,左边为改动后,要通过定义一个结构体来定义互斥锁

```
> //static DECLARE_MUTEX(button_lock); //定义互斥锁
   struct semaphore button_lock;

static DECLARE_MUTEX(button_lock); //定义互斥锁
```

2:定义了该结构体之后还要初始化,我们必须在驱动的入口函数添加结构体的初始化

```
static int sixth_drv_init(void)
{
    major = register_chrdev(0, "sixth_drv", &sixth_drv_fops);
    sixthdrv_class = class_create(THIS_MODULE, "sixth_drv");
    sixthdrv_class_dev = device_create(sixthdrv_class, NULL, MKDEV(major, 0), NULL, "button");
    gpicon = (volatile unsigned long *)ioremap(0x5000000, 1);
    gpfat = gpicon + 1;
    gpcon = (volatile unsigned long *)ioremap(0x5000000, 1);
    gpfat = gpcon + 1;
    sema_init(&button_lock, 1);
}

static int sixth_drv_init(void)
{
    major = register_chrdev(0, "sixth_drv", &sixth_drv_fops);
    sixthdrv_class = class_create(THIS_MODULE, "sixth_drv");
    sixthdrv_class_dev = device_create(sixthdrv_class, NULL, MKDEV(major, 0), NULL, "button");
    gpicon = (volatile unsigned long *)ioremap(0x5000000, 1);
    gpfat = gpicon + 1;
    gpcon = (volatile unsigned long *)ioremap(0x5000000, 1);
    gpfat = gpcon + 1;
}
```

其他的都没什么难点

毕业班第 4 课第 3 节_移植驱动到 3.4.2 内核之 LCD 及输入系统

```
1 移植:
2 1. 编译
3 2. 解决错误
4 2.1 头文件不对: 去掉或改名
5 2.2 宏不对: 改名使用新宏
6 2.3 有些函数没有了: 改名使用新函数
考东山Linux移植第1期: S3C2440/
```

9th 的驱动老师在资料盘中没有看到移植后的代码

1、把驱动程序放到服务器上, 修改 Makefile 并进行编译

```
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ mv 4th/* .
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ ls
4th  lcd.c  Makefile
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ rm -rf 4th/
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ ls
lcd.c  Makefile
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ vi Makefile
```

修改 Makefile 里面的内核目录

```
KERN_DIR = /work/system/linux-3.4.2

all:
    make -C $(KERN_DIR) M=pwd modules

clean:
    make -C $(KERN_DIR) M=pwd modules clean
    rm -rf modules.order
obj-m += lcd.o
```

进行编译

```
"Makefile" 10L, 169C written
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ make
make -C /work/system/linux-3.4.2 M=pwd modules
```

2、编译出现错误并解决

没有头文件就把它注释掉

```
/work/drivers_and_test_new/10th_lcd/lcd.c:22:31: error: asm/arch/regs-lcd.h: No such file or directory
/work/drivers_and_test_new/10th_lcd/lcd.c:23:32: error: asm/arch/regs-gpio.h: No such file or directory
/work/drivers_and_test_new/10th_lcd/lcd.c:24:25: error: asm/arch/fb.h: No such file or directory
/work/drivers_and_test_new/10th_lcd/lcd.c: In function 'lcd_init':
/work/drivers_and_test_new/10th_lcd/lcd.c:224: warning: passing argument 3 of 'dma_alloc_writecombine' from incompatible pointer type
make[2]: *** [/work/drivers_and_test_new/10th_lcd/lcd.o] Error 1
make[1]: *** [module_/work/drivers_and_test_new/10th_lcd] Error 2
make[1]: Leaving directory '/work/system/linux-3.4.2'
make: *** [all] Error 2
```

3、把生成的驱动拷贝到网络文件系统

```
book@book-desktop:/work/drivers_and_test_new/10th_lcd$ cp lcd.ko /work/nfs_root/fs_mini_mdev_new
```

配置内核把原来的 LCD 驱动程序去掉

```
book@book-desktop:/work/system/linux-3.4.2$ make menuconfig
HOSTCC scripts/kconfig/lxdialog/checklist.o
```

把 s3c2410 LCD framebuffer support 设置为 M

```
--- Support for frame buffer devices
[ ] Enable firmware EDID
[ ] Framebuffer foreign endianness support --->
[ ] Enable Video Mode Handling Helpers
[ ] Enable Tile Blitting Support
*** Frame buffer hardware drivers ***
< > Epson S1D13XXX framebuffer support
<M> S3C2410 LCD framebuffer support
[ ] S3C2410 lcd debug messages
< > SMSC UFX6000/7000 USB Framebuffer support
< > Displaylink USB Framebuffer support
< > Virtual Frame Buffer support (ONLY FOR TESTING!)
v(+)
```

最后编译内核

```
book@book-desktop:/work/system/linux-3.4.2$ make uImage
```

4、重新烧写、启动内核

```
Image arch/arm/boot/uImage is ready
book@book-desktop:/work/system/linux-3.4.2$ cp arch/arm/boot/uImage /work/nfs_root/uImage_nolcd
```

还需要 make modules, 编译所有的模块驱动

```
book@book-desktop:/work/system/linux-3.4.2$ make modules
CHK include/linux/version.h
CHK include/generated/utsrelease.h
```

因为 LCD 驱动程序依赖于其他三个函数, 那三个函数分别放在三个文件里面, 那三个文件也要编译成模块加载进去

```
static struct fb_ops s3c_lcd_fb_ops = {
    .owner = THIS_MODULE,
    .fb_setcolreg = s3c_lcd_fb_setcolreg,
    .fb_fillrect = cfb_fillrect,
    .fb_copyarea = cfb_copyarea,
    .fb_imageblit = cfb_imageblit,
};
```

下载内核并启动内核

```
SMDK2410 # nfs 30000000 192.168.1.123:/work/nfs_root/uImage_nolcd; bootm 30000000
ERROR: resetting DM9000 -> not responding
```

5、测试程序

(1) 把测试程序放到服务器编译

```
book@book-desktop:/work/drivers_and_test_new$ cd fb_test/
book@book-desktop:/work/drivers_and_test_new/fb_test$ ls
fb.h fb_test fb_test.c Makefile
book@book-desktop:/work/drivers_and_test_new/fb_test$ vi Makefile
```

(2) 修改 Makefile

```
CC=arm-linux-gcc
fb_test: fb_test.c
    $(CC) -o $@ $^
clean:
    rm -f *.o fb_test
~
```

(3) 进行编译

```
book@book-desktop:/work/drivers_and_test_new/fb_test$ make
make: 'fb_test' is up to date.
book@book-desktop:/work/drivers_and_test_new/fb_test$ make clean
rm -f *.o fb_test
book@book-desktop:/work/drivers_and_test_new/fb_test$ make
arm-linux-gcc -o fb_test fb_test.c
```

(4) 把测试程序拷贝到根文件系统

```
book@book-desktop:/work/drivers_and_test_new/fb_test$ cp fb_test /work/nfs_root/fs_
mini_mdev_new
```

6. 实验

(1) 装载 LCD 驱动，出现问题（某些函数没有定义）

```
/ # insmod lcd.ko
lcd: Unknown symbol cfb_copyarea (err 0)
lcd: Unknown symbol cfb_imageblit (err 0)
lcd: Unknown symbol cfb_fillrect (err 0)
insmod: can't insert 'lcd.ko': unknown symbol in module or invalid parameter
```

(2) 把 LCD 驱动的依赖拷贝到根文件系统那里去

```
book@book-desktop:/work/system/linux-3.4.2$ cd drivers/video/
book@book-desktop:/work/system/linux-3.4.2/drivers/video$ ls *.ko
cfbcopyarea.ko cfbfillrect.ko cfbimgblt.ko s3c2410fb.ko
book@book-desktop:/work/system/linux-3.4.2/drivers/video$ cp cfb*.ko /work/nfs_root/
fs_mini_mdev_new
book@book-desktop:/work/system/linux-3.4.2/drivers/video$
```

(3) 对各个驱动进行装载

```
/ #
/ # insmod cfbcopyarea.ko
/ # insmod cfbimgblt.ko
/ # insmod cfbfillrect.ko
/ # insmod lcd.ko
Console: switching to colour frame
/ #
```

(4) 测试（测试程序后面跟设备名称）

```
Usage (example): ./fb_test /dev/fb/0
./fb_test /dev/fb/0 set <width>
./fb_test /dev/fb/0 r8 <offset>
./fb_test /dev/fb/0 w8 <offset>
./fb_test /dev/fb/0 r16 <offset>
./fb_test /dev/fb/0 w16 <offset>
./fb_test /dev/fb/0 r32 <offset>
./fb_test /dev/fb/0 w32 <offset>
/ # ./fb_test /dev/fb0
```

(5) 吧文字写到 LCD 上

```
240x320, 16bpp
/ # echo hello > /dev/tty1
/ # echo hello > /dev/tty1
/ # echo hello > /dev/tty1
/ # echo hello > /dev/tty1
/ #
```

7. 按键驱动移植（键盘输入，显示器输出）

(1) 把驱动拷贝到服务器，并修改 Makefile

```
book@book-desktop:/work/drivers_and_test_new$ cd 8th_buttons_input/
book@book-desktop:/work/drivers_and_test_new/8th_buttons_input$ ls
buttons.c Makefile
book@book-desktop:/work/drivers_and_test_new/8th_buttons_input$ vi Makefile
```

(2) 修改 Makefile，把内核目录改成新的内核目录

```
KERN_DIR = /work/system/linux-3.4.2

all:
    make -C $(KERN_DIR) M=`pwd` modules

clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order

obj-m += buttons.o
```

(3) 编译出现一大堆错误 (根据不同情况修改)

(4) 编译修改后的文件, 并把驱动拷贝到网络文件系统上去

```
book@book-desktop:/work/drivers_and_test_new/8th_buttons_input$ cp buttons.ko /work/
nfs_root/fs_mini_mdev_new
```

(5) 装载驱动程序

```
/ # ls /dev/ev*
ls: /dev/ev*: No such file or directory
/ # ls /dev/input/ev*
ls: /dev/input/ev*: No such file or directory
/ # insmod buttons.ko
input: Unspecified device as /devices/virtual/input/input0
/ # ls /dev/event0 -l
crw-rw---- 1 0 0 13, 64 Jan 1 00:08 /dev/event0
```

(6) 按下按键, cat 是将某个文档中的内容显示出来

```
/ # cat /dev/event0
cat: /dev/event0: No such file or directory
/ # cat /dev/tty1
lls
sl
^~
```

(7) 修改/etc/inittable

加上一行 tty1, 表示重启系统时会启动一个 shell, 这个 shell 从 /dev/tty1 得到输入, 把输出信息输出到 /dev/tty1 里面去。

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
console::askfirst:/bin/sh
tty1::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

(8) 重启系统, 下载新内核启动

(9) 装载模块

```
/ #
/ # insmod cfbimgblt.ko
/ # insmod cfbcopyarea.ko
/ # insmod cfbfillrect.ko
/ # insmod lcd.ko
Console: switching to colour frame buffer dev:
/ #
Console: switching to colour frame buffer device 30x40
/ # insmod buttons.ko
input: Unspecified device as /devices/virtual/input/input0
```

(10) 按下按键, 在 LCD 上可以看到控制台一样的信息

8. 触摸屏驱动移植

(1) 把触摸屏驱动放到服务器上去, 修改 Makefile

```
book@book-desktop:/work/drivers_and_test_new$ cd 11th_ts/
book@book-desktop:/work/drivers_and_test_new/11th_ts$ ls
Makefile s3c_ts.c tslib-1.4.tar.gz
book@book-desktop:/work/drivers_and_test_new/11th_ts$ vi Makefile
```

(2) 修改 Makefile, 把内核目录更换为新内核目录, 然后进行 make 编译

(3) Make 编译后出现一大堆错误。(根据不同错误进行解决)

(4) 编译后把生成的驱动拷贝到网络文件系统上

(5) 加载触摸屏驱动, /dev/event1 对应触摸屏。


```

/ # ls /dev/event*
/dev/event0
/ # insmod s3c_ts.ko
input: Unspecified device as /devices/virtual/input/input1
/ # ls /dev/event*
/dev/event0 /dev/event1

```

(6) 触摸屏的使用

解压触摸屏库，autogen.sh 是检测你的编译工具和依赖关系包是否完整，自动生成 Makefile

```

book@book-desktop:/work/drivers_and_test_new/11th_ts$ ls
Makefile      Module.symvers  s3c_ts.ko      s3c_ts.mod.o   tslib-1.4.tar.gz
modules.order s3c_ts.c        s3c_ts.mod.c   s3c_ts.o
book@book-desktop:/work/drivers_and_test_new/11th_ts$ tar xzf tslib-1.4.tar.gz
book@book-desktop:/work/drivers_and_test_new/11th_ts$ cd tslib/
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ ls
acinclude.m4      autogen.sh      COPYING         INSTALL         NEWS           src
AUTHORS           ChangeLog       CVS             m4              plugins        tests
autogen-clean.sh  configure.ac     etc             Makefile.am     README         tslib.pc.in
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ ./autogen.sh
libtoolize: putting auxiliary files in `.'.
libtoolize: copying file `./config.guess'
libtoolize: copying file `./config.sub'
libtoolize: creating `stamp.h'

```

创建文件夹、向文件添加内容、配置文件

```

plugins/Makefile.am: installing `./depcomp'
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ mkdir tmp
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ echo "ac_cv_func_malloc_0_nonnull=yes" >arm-linux.cache
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ ./configure --host=arm-linux --cache-file=arm-linux.cache --prefix=$(pwd)/tmp
configure: WARNING: If you wanted to set the --build type, don't use --host.
If a cross compiler is detected then cross compile mode will be used.
configure: loading cache arm-linux.cache
checking for a BSD-compatible install... /usr/bin/install -c

```

编译和安装 (make 、 make install)

把编译安装好的文件拷贝到根文件系统上去

```

book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ cd tmp/
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ ls
bin  etc  include  lib
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ cp * -rf /work/nfs_root/fs_mini_mdev_new

```

修改配置文件

```

1.
修改 /etc/ts.conf 第1行 (去掉#号和第一个空格):
# module_raw input
改为: http://blog.csdn.net/
module_raw input

```

输入设备是 event1, 设置环境变量, 触摸屏程序会根据这些环境变量决定打开哪一个设备

```

/ # export TSLIB_TSDEVICE=/dev/event1
/ # export TSLIB_CALIBFILE=/etc/pointercal
/ # export TSLIB_CONFFILE=/etc/ts.conf
/ # export TSLIB_PLUGINDIR=/lib/ts
/ # export TSLIB_CONSOLEDVICE=/dev/tty
/ # export TSLIB_FBDEVICE=/dev/fb0
/ #

```

校准 (出现错误) (校准程序不能理解里面的数据)

```

/ # ts_calibrate
xres = 240, yres = 320
selected device is not a touchscreen I understand

```

tslib 中的输入系统和内核 input 系统版本不匹配 (在浏览器中搜索错误内容)

其中关键的就是 `version == EV_VERSION` 这个判断语句, 如果这两者不等, 那么就会打印出 `selected device is not a touchscreen I understand`

tslib 中的 input 版本号是在交叉编译的时候指定的, 赋值给 `version`, 而 `EV_VERSION` 是内核中的定义的, 我做 tslib 时, 用的是 2.6.39 内核, 这个 `EV_VERSION` 定义在 `include/linux/input.h` 中, 为 `0x010001`, 而 tslib 中的是和交叉编译器相同, 于是查看编译器, 在 `/usr/include/linux/input.h` 中, 指定为 `0x010000`, 这俩明显不相等, 当然不会满足 `version == EV_VERSION` 了, 由此可见问题就出现在内核的输入子系统的版本号不匹配的问题

tslib 用的版本是什么? 我们编译的应用程序它的头文件是在工具链里面的, 在交叉编译工具链里面搜索版本

```

book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/
arm/4.3.2/bin
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ cd /usr/local/arm/4
.3.2/
book@book-desktop:/usr/local/arm/4.3.2$ grep "EV_VERSION" * -nR
arm-none-linux-gnueabi/libc/usr/include/linux/input.h:32:#define EV_VERSION      0
x010000
arm-none-linux-gnueabi/libc/usr/include/linux/firewire-cdev.h:183:/* FW_CDEV_VERSIO
N History
arm-none-linux-gnueabi/libc/usr/include/linux/firewire-cdev.h:187:#define FW_CDEV_V
ERSION      1

```

修改文件的版本宏跟内核一样

提示下这个交叉编译下的 input.h 有可能你的路径跟我的不一样，你可以在你交叉编译路径下用如下命令搜索：

```
find -name "input.h"
```

```

12 tslib:
13 /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/usr/include/linux/input.h
14 #define EV_VERSION      0x010000
15 改为
16 #define EV_VERSION      0x010001
17

```

修改权限

```

book@book-desktop:/usr/local/arm/4.3.2$ sudo chmod +w /usr/local/arm/4.3.2/arm-none
-linux-gnueabi/libc/usr/include/linux/input.h
[sudo] password for book:

```

然后重新编译

```
make clean
```

```
make
```

再安装

```

make[1]: Leaving directory /work/drivers_and_test_new/11th_ts/tslib
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ make install

```

安装到 tmp 目录下，吧目录下的所有文件拷贝到文件系统里

```

book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib$ cd tmp/
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ ls
bin etc include lib
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$ cp * -rf /work/nfs_
root/fs_mini_mdev_new
book@book-desktop:/work/drivers_and_test_new/11th_ts/tslib/tmp$

```

(7) 重新执行校正程序

点

```

/ # ts_calibrate
No raw modules loaded.
ts config: No such file or directory

```

修改配置文件吧 module_raw input 这个宏前面的 # 去掉。

```
/ # vi /etc/ts.conf
```

再次实验（点击触摸屏上面的十字架）

```

/ # ts_calibrate
xres = 240, yres = 320

```

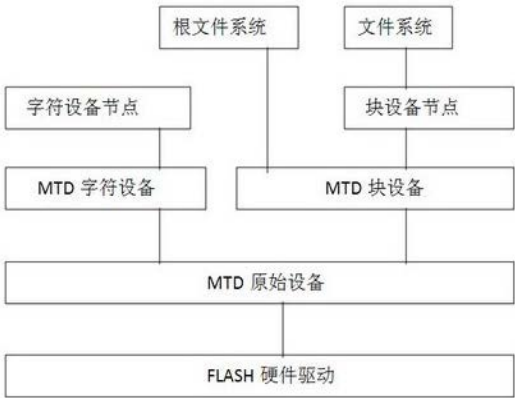
测试（执行测试程序后在触摸屏上画线）

```
/ # ts_test
```


18.2 块设备驱动程序的框架

MTD 体系结构: Linux 内核提供 MTD 子系统来建立 FLASH 针对 Linux 的统一、抽象接口。MTD 将文件系统与底层的 FLASH 存储器进行隔离。

引入 MTD 后 Linux 系统中对 FLASH 的设备驱动分为 4 层



设备节点: 用户在/dev 目录下使用 mknod 命令建立 MTD 字符设备节点(主设备号为 90), 或者 MTD 块设备节点(主设备号为 31), 使用该设备节点即可访问 MTD 设备。

MTD 设备层: 基于 MTD 原始设备层, 系统将 MTD 设备可以定义为 MTD 字符(在/mtd/mtdchar.c 中实现, 设备号 90)和 MTD 块设备(在/mtd/mtdblock.c 中实现, 设备号 31)。

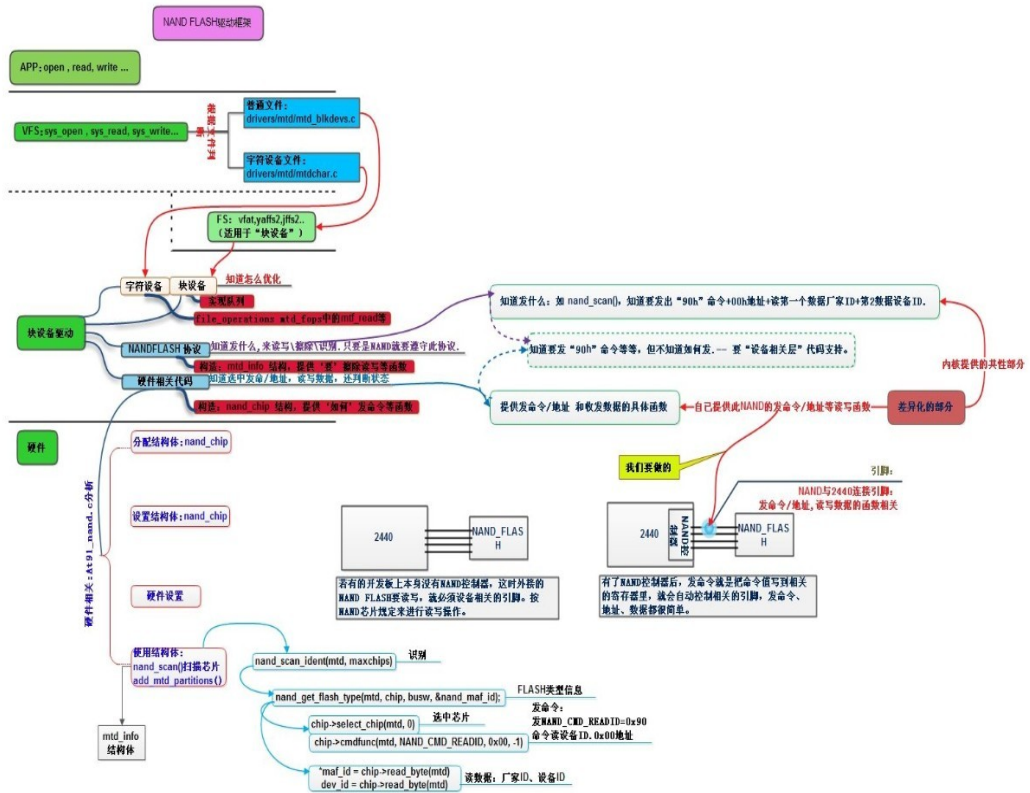
MTD 原始设备层: MTD 原始设备层由两部分构成, 一部分是 MTD 原始设备的通用代码, 另一部分是各个特定 Flash 的数据, 如分区。

主要构成的文件有:

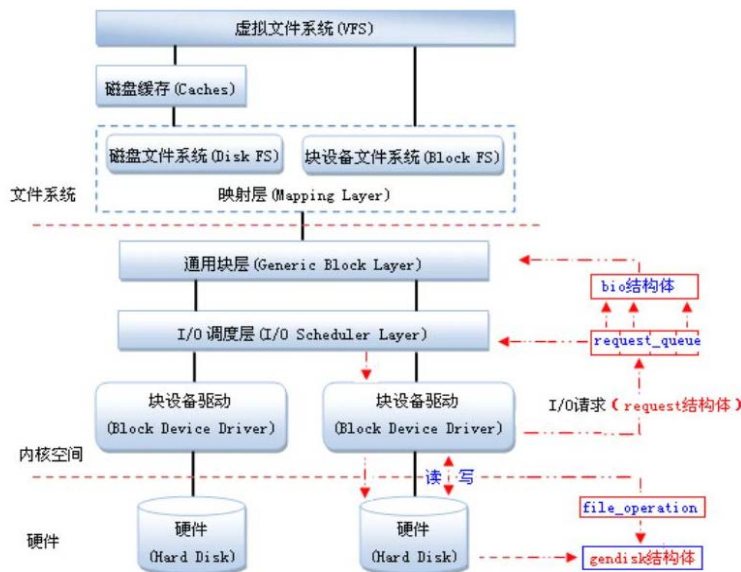
drivers/mtd/mtdcore.c 支持 mtd 字符设备

driver/mtd/mtdpart.c 支持 mtd 块设备

Flash 硬件驱动层: Flash 硬件驱动层负责对 Flash 硬件的读、写和擦除操作。MTD 设备的 Nor Flash 芯片驱动位于 drivers/mtd/chips/子目录下, Nand Flash 芯片的驱动则位于 drivers/mtd/nand/子目录下



在应用程序对文件的读写通过具体的文件系统被转换为对扇区的读写，对扇区的读写调用的是 ll_rw_block 这个函数，对扇区进行读写的时候会根据某种调度进行优化，之后通过块设备驱动程序来访问具体的硬件。



我们知道应用层最终会将块设备的请求通过 request_queue 下发到块设备驱动,也就是对扇区进行读写
所以我们的驱动层应该对扇区读写的操作函数,这个函数就是在

应用层对块设备的读写操作经过文件系统后转化为对扇区的读写操作,最终会调用如下函数:

从 ll_rw_block 函数开始分析:fs\buffer.c

```
ll_rw_block(int rw, int nr, struct buffer_head *bhs[])
{
    for (i = 0; i < nr; i++) {
        struct buffer_head *bh = bhs[i]; //将数据存进bh
        >> submit_bh(rw, bh); //提交数据
        struct bio *bio; // bio即block input/output, 接下来根据bh来构造bio
        bio->bi_size = bh->b_size;
        >> submit_bio(rw, bio); //提交bio
        >> generic_make_request(bio); // 通用的构造请求: 使用bio来构造请求(request)
        >> __generic_make_request(bio);
        q = bdev_get_queue(bio->bi_bdev); // 找到队列
        ret = q->make_request_fn(q, bio); //调用q中的构造请求函数
    }
}
```

结合上图可以知道:上面步骤经过了:将对扇区的读写操作通过构造 bio→利用 bio 构造请求 request_queue 结构体 q→最终调用请求队列结构体里面的 make_request_fn 来处理这个对块设备的请求

可以猜测我们的驱动应该干什么:

由于我们的对块设备的操作经过一系列转换是通过 request_queue 来实现的,而具体的动作是由这个结构体中的 make_request_fn 函数来实现的。

所以我们要构造一个请求队列 request_queue,并且里面要实现对块设备的具体操作: make_request_fn

而我们的驱动除了要有这个请求队列的结构体,肯定还要有主次设备号,容量等等关于块设备的信息,统一都在 gendisk 这个结构体中,

所以写块设备的驱动步骤:

1. 分配gendisk: alloc_disk
2. 设置
 - 2.1 分配/设置队列: request_queue_t // 它提供对块设备读写的操作
blk_init_queue
 - 2.2 设置gendisk 其他信息 // 它提供属性: 比如容量
3. 注册: add_disk 这个块设备到内核

```
struct gendisk
{
    int major; /* 主设备号 */
    int first_minor; /* 第 1 个次设备号 */
    int minors; /* 最大的次设备数, 如果不能分区, 则为 1 */
    char disk_name[32]; /* 设备名称 */
    struct hd_struct **part; /* 磁盘上的分区信息 */
    struct block_device_operations *fops; /* 块设备操作结构体 */
    struct request_queue *queue; /* 请求队列, queue 是内核用来管理这个设备的 I/O 请求队列的指针 */
    void *private_data; /* 私有数据 */
    sector_t capacity; /* 扇区数, 512 字节为 1 个扇区 */

    int flags;
    char devfs_name[64];
    int number;
    struct device *driverfs_dev;
    struct kobject kobj;

    struct timer_rand_state *random;
    int policy;

    atomic_t sync_io; /* RAID */
    unsigned long stamp;
    int in_flight;
#ifdef CONFIG_SMP
    struct disk_stats *dkstats;
#else
    struct disk_stats dkstats;
#endif
};
```

还剩下一个问题:最终调用 make_request_fn 来实现对块设备的读写操作,虽然我们在驱动里面写了对应的对块设备读写的函数,那么他又是怎么和我们的驱动中提供的函数联系起来的呢?

```

继续上面程序的分析：
>> q->make_request_fn(q, bio); //调用请求队列中的指针函数
回溯寻找在那里调用了make_request_fn
<回溯
blk_queue_make_request(request_queue_t * q, make_request_fn * mfn)
q->make_request_fn = mfn;
<回溯
blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock, int node_id)
>>blk_queue_make_request(q, __make_request);
q->make_request_fn = mfn(也就是__make_request);
<回溯
blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
现在就搞明白了，在使用blk_init_queue时最后调用的是__make_request函数
>> __make_request; //上面的那条代码实际上是执行这个函数
elv_merge(q, &req, bio); //先尝试将bio合并到请求队列中的某个请求中
req = get_request_wait(q, rw_flags, bio); //如果无法将bio合并到某一请求，就会构造一个请求
init_request_from_bio(req, bio); //根据bio初始化这个请求
add_request(q, req); //把请求放入队列，这里采用的电梯调度算法，详见注释2
__generic_unplug_device(q); //在适当的时候处理队列里面的请求，这个有内核决定
q->request_fn(q); //调用队列的“处理函数，我们猜想他应该在设备驱动程序里面被设置
/*
 * request_fn指向用户自定义的函数
 */
>> do_xxxdisk_request_rfn(request_queue_t *q) (=request_fn)
    struct request *req=elv_next_request(q); //取出电梯里一个请求
    >>hard_ware_transfer(req); //与具体硬件相关的读写函数
    /* 磁盘数据读写 */
    unsigned long offset = req->sector*512;
    /* 数据长度 */
    unsigned long len = req->current_nr_sectors*512;
    /* 缓冲区数据源:req->buffer */
    >> end_request(req,1); //成功处理完一次bio请求 */

```

通过上面可以知道这个是我们写驱动的时候初始化 request_queue_t 这个时候就建立了联系：

我们的驱动是：

```
static request_queue_t *ramblock_queue = blk_init_queue(do_ramblock_request, &ramblock_lock);
```

```
->> blk_init_queue_node (rfn, lock, -1);
```

q->request_fn = rfn; //这里的 rfn 就是传进来的第一个参数,也就是我么自己写的实际对扇区读写的函数

```
>> blk_queue_make_request(q, __make_request); //原型是 void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn)
```

q->make_request_fn = mfn; //这个 mfn 就是 blk_queue_make_request 传进来的第二个参数,也就是__make_request

也就是说到这里最终调用的是__make_request 函数，

```

>> __make_request; //上面的那条代码实际上是执行这个函数
elv_merge(q, &req, bio); //先尝试将bio合并到请求队列中的某个请求中
req = get_request_wait(q, rw_flags, bio); //如果无法将bio合并到某一请求，就会构造一个请求
init_request_from_bio(req, bio); //根据bio初始化这个请求
add_request(q, req); //把请求放入队列，这里采用的电梯调度算法，详见注释2
__generic_unplug_device(q); //在适当的时候处理队列里面的请求，这个有内核决定
q->request_fn(q); //调用队列的“处理函数，我们猜想他应该在设备驱动程序里面被设置
/*
 */

```

通过上面的截图发现__make_request 这个函数经过一些列算法，最终调用 request_fn 函数,这个已经在 blk_init_queue_node (rfn, lock, -1);函数中指向了我们自己写的读写扇区函数了

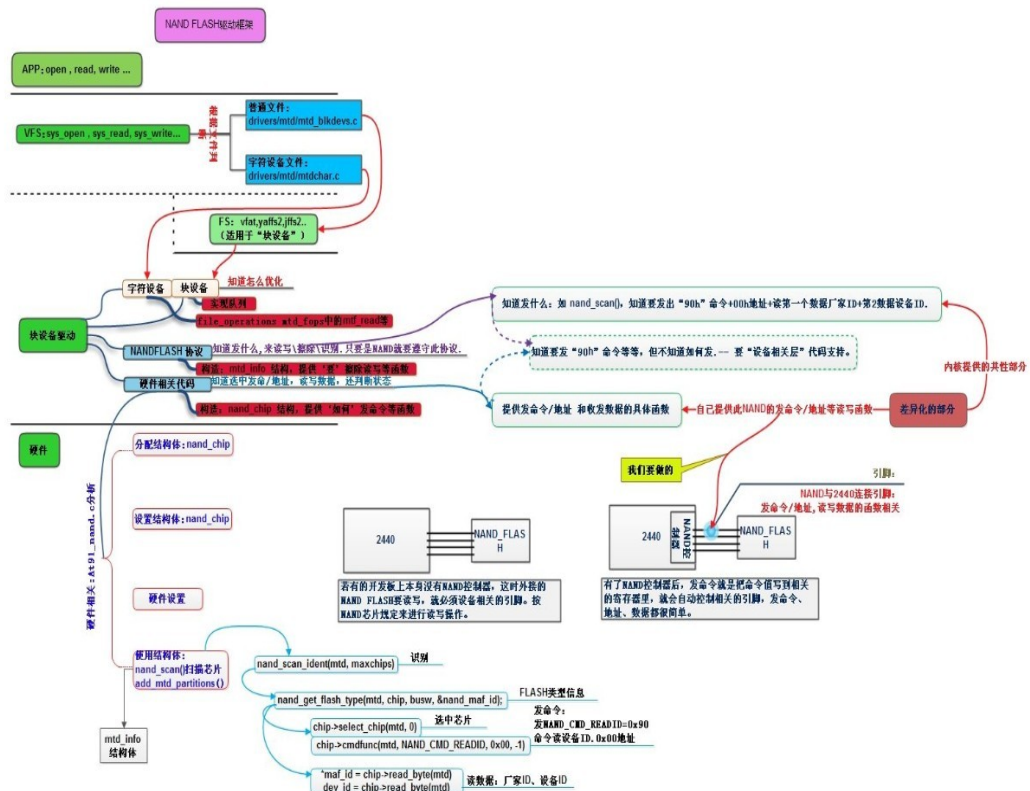
Q:在块设备驱动那节：将对扇区的读写操作通过构造 bio→利用 bio 构造请求 request_queue 结构体 q→最终调用请求队列结构体里面的 make_request_fn 来处理这个对块设备的请求

其中我们通过 q = bdev_get_queue(bio->bi_bdev);来查找队列中的请求队列 request_queue,我们是如何让这个请求队列和我们驱动中的 request_queue 来进行匹配的呢,进而调用我们自己的对块设备的读写函数的?

An: 通过 q = bdev_get_queue(bio->bi_bdev);来查找队列的时候,其实我们在写驱动的时候添加块设备时候就已经将 nandchip 结构体组装成了 bio,然后假想了链表中: add_disk > register_disk > bdev = bdget_disk(disk, 0);

19.2 NAND FLASH 驱动程序框架

学习《韦东山—嵌入式Linux开发完全手册》--第二期《深入Linux驱动》的第20课: NAND+FLASH 驱动
图片制作: 100ask.net, ian1900



 百问网
www.100ask.net

未完成:

- 1:移植 uboot 支持 norflash 的笔记,最后测试验证部分
- 2:uboot 支持 nand 关于删除 ECC 部分的笔记
- 3:nandflash 驱动的笔记

驱动没有设置内存控制器时可以用，因为内存控制器在 uboot 里面已经设置好了，DM9000 驱动程序能够运行依赖于 uboot 的设置，写 1 个不依赖 uboot 的驱动程序需要设置内存控制器。而我们之前的 norflash 的驱动时候，它也是内存类接口，之所以没有设置内存控制器中关于它的时序是因为用默认的参数就可以运行。

Q:虽然我们的 uboot 有设置内存控制器,但是引导完内核后,uboot 不是已经完全消失了么,为什么不用重新设置,之前的 uboot 里面的看门狗 sdram 等等都需要在内核中重新设置吧?

这里需要重新听下视频,感觉表达的意思有问题

关于 uboot 的命令行和启动参数的理解:

我们在开发板烧录完 uboot 之后,然后启动开发板,进入倒计时,然后按任意键,就可以进入命令

```
never reach
TRY_TIME
Hit any key to stop autoboot: 0
##### 100ask Bootloader for OpenJTAG #####
[n] Download u-boot to Nand Flash
[k] Download Linux kernel uImage
[j] Download root_jffs2 image
[y] Download root_yaffs image
[d] Download to SDRAM & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the Nand Flash
[s] Set the boot parameters
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection: q
OpenJTAG>
OpenJTAG>
Ready
```

要注意的是这个是韦老师自己实现的,自带的 uboot 并没有这个界面,并且是 uboot1.1.6 补丁里面实现的其他版本的 uboot 或者是 1.16 没有打补丁的 uboot 都是没有这个的

这个界面按 q 或者其他版本 uboot 倒计时的时候按空格进入 uboot 命令行模式

常用的命令是:

1:help 命令

2:下载命令

相关的命令有:loadb,loads,loady,tftpboot,nfs

3:内存操作命令

相关的命令有:md,mw,mb,mm,cp

4:Norflash 操作命令

Flinfo,protect,erase,由于 norflash 为内存类接口,所有一些内存类命令也可以操作 norflash

5:nandflash 命令

只有一个:nand,它根据不同的参数进行不同的操作

“nand info” 查看 NAND Flash 信息。

“nand erase [clean] [off size]” 擦除 NAND Flash。加上“clean”时,表示在每个块的第一个扇区的 OOB 区加入清除标记; off、size 表示要擦除的开始偏移地址的长度,如果省略 off 和 size,表示要擦除整个 NAND Flash。

“nand read[jffs2] addr off size” 从 NAND Flash 偏移地址 off 处读出 size 个字节的数据存放到开始地址为 addr 的内存中。是否加后缀“jffs”的差别只是读操作时的 ECC 校验方法不同。

“nand write[jffs2] addr off size” 把开始地址为 addr 的内存中的 size 个字节数据写到 NAND Flash 的偏移地址 off 处。是否加后缀“jffs”的差别只是写操作时的 ECC 校验方法不同。

“nand read.yaffs addr off size” 从 NAND Flash 偏移地址 off 处读出 size 个字节的数据(包括 OOB 区域),存放到开始地址为 addr 的内存中。

“nand write.yaffs addr off size” 把开始地址为 addr 的内存中的 size 个字节数据(其中有要写入 OOB 区域的数据)写到 NAND Flash 的偏移地址 off 处。

“nand dump off” 将 NAND Flash 偏移地址 off 的一个扇区的数据打印出来,包括 OOB 数据。

6:环境变量命令

“printenv” 命令打印全部环境变量,“printenv name1 name2...” 打印名字为 name1、name2、...的环境变量。

“setenv name value” 设置名字为 name 的环境变量的值为 value。

“setenv name” 删除名字为 name 的环境变量。

上面的设置、删除操作只是在内存中进行,“saveenv” 将更改后的所有环境变量写入 NOR Flash 中。

7:启动命令

不带参数的“boot”、“bootm”命令都是执行环境变量 bootcmd 所指定的命令。

“bootm [addr [arg...]]”命令启动存放在地址 addr 处的 U-Boot 格式的映像文件（使用 U-Boot 目录 tools 下的 mkimage 工具制作得到），[arg...]表示参数。如果 addr 参数省略，映像文件所在地址为配置文件中定义的宏 CFG_LOAD_ADDR。

“go addr [arg...]”与 bootm 命令类似，启动存放在地址 addr 处的二进制文件，[arg...]表示参数。

“nboot [[[loadAddr] dev] offset]”命令将 NAND Flash 设备 dev 上偏移地址 off 处的映像文件复制到内存 loadAddr 处，然后，如果环境变量 autostart 的值为“yes”，就启动这个映像。如果 loadAddr 参数省略，存放地址为配置文件中定义的宏 CFG_LOAD_ADDR；如果 dev 参数省略，则它的取值为环境变量 bootdevice 的值；如果 offset 参数省略，则默认为 0。

U-Boot 常用环境变量

ipaddr:

设置 DVS6446 平台的 IP 地址

说明: setenv ipaddr 192.168.253.53

serverip:

设置服务器 ip 地址，多用于 tftp 中

说明: setenv serverip 192.168.253.33

bootargs:

启动 OS 的启动参数

说明: setenv bootargs mem=120M console=ttyS0,115200n8 ...

解释: 设置传递参数，如内存大小，控制台，根文件系统等。

bootcmd:

设置自动及执行命令。启动延时根据 bootdelay 的值，若没有设置 bootdelay,默认延时 3 秒

说明: setenv bootcmd bootm 0x23450000,即设置启动后自动执行 0x23450000 处的代码。

bootdelay:

设置自启动延时时间，单位为秒。只有当 bootcmd 变量被设置后，该变量才有效。改变了值范围为大于等于-1 的整数，当设置为 -1 时，关闭自启动的功能。

说明: setenv bootdelay 4 ; 设置自启动延时 4 秒

说明: setenv bootdelay -1; 关闭自启动功能

提示: 在延时时间内可按任意键切换到命令行模式。注意，在产品开发调试阶段请勿设置延时时间为 0，若设置，可以在启动瞬间使用 CTRL+C 中断程序而进入命令行模式。

netmask:

设置子网掩码。

说明: setenv netmask 255.255.255.0

gatewayip:

设置网关。

说明: setenv gatewayip 192.168.253.1

ethaddr:

设置以太网网卡 MAC 地址。

说明: setenv ethaddr xx:xx:xx:xx:xx:xx

注意: 设置 ethaddr 后，应运行 saveenv,重启才能使 MAC 地址配置有效。

有几点注意:uboot 命令行里面支持 nfs 和 tftp 下载文件,其中 uboot 命令行中的 nfs 要和文件系统挂载中用到的 nfs 区分开

设备文件的操作：

什么是 I/O：

I/O 是 input/output 的缩写，即输入输出端口。每个设备都会有一个专用的 I/O 地址，用来处理自己的输入输出信息。CPU 与外部设备、存储器的连接和数据交换都需要通过接口设备来实现，前者被称为 I/O 接口，而后者则被称为存储器接口。存储器通常在 CPU 的同步控制下工作，接口电路比较简单；而 I/O 设备种类繁多，其相应的接口电路也各不相同，因此，习惯上说到接口只是指 I/O 接口。

1 接口分类编辑

I/O 接口的功能是负责实现 CPU 通过系统总线把 I/O 电路和外围设备联系在一起，按照电路和设备的复杂程度，I/O 接口的硬件主要分为两大类：

(1) I/O 接口芯片

这些芯片大都是集成电路，通过 CPU 输入不同的命令和参数，并控制相关的 I/O 电路和简单的外设作相应的操作，常见的接口芯片如定时计数器、中断控制器、DMA 控制器、并行接口等。

(2) I/O 接口控制卡

由若干个集成电路按一定的逻辑组成为一个部件，或者直接与 CPU 同在主板上，或是一个插件插在系统总线插槽上。

按照接口的连接对象来分，又可以将他们分为串行接口、并行接口、键盘接口和磁盘接口等。

2 接口功能编辑

由于计算机的外围设备种类繁多，几乎都采用了机电传动设备，因此，CPU 在与 I/O 设备进行数据交换时存在以下问题：

速度不匹配：I/O 设备的工作速度要比 CPU 慢许多，而且由于种类的不同，他们之间的速度差异也很大，例如硬盘的传输速度就要比打印机快出很多。

时序不匹配：各个 I/O 设备都有自己的定时控制电路，以自己的速度传输数据，无法与 CPU 的时序取得统一。

信息格式不匹配：不同的 I/O 设备存储和处理信息的格式不同，例如可以分为串行和并行两种；也可以分为二进制格式、ASCII 编码和 BCD 编码等。

信息类型不匹配：不同 I/O 设备采用的信号类型不同，有些是数字信号，而有些是模拟信号，因此所采用的处理方式也不同。

基于以上原因，CPU 与外设之间的数据交换必须通过接口来完成，通常接口有以下一些功能：

(1) 设置数据的寄存、缓冲逻辑，以适应 CPU 与外设之间的速度差异，接口通常由一些寄存器或 RAM 芯片组成，如果芯片足够大还可以实现批量数据的传输；

(2) 能够进行信息格式的转换，例如串行和并行的转换；

(3) 能够协调 CPU 和外设两者在信息的类型和电平的差异，如电平转换驱动器、数模或模数转换器等；

(4) 协调时序差异；

(5) 地址译码和设备选择功能；

(6) 设置中断和 DMA 控制逻辑，以保证在中断和 DMA 允许的情况下产生中断和 DMA 请求信号，并在接受到中断和 DMA 应答之后完成中断处理和 DMA 传输。

3 控制方式编辑

CPU 通过接口对外设进行控制的方式有以下几种：

(1) 程序查询方式

这种方式下，CPU 通过 I/O 指令询问指定外设当前的状态，如果外设准备就绪，则进行数据的输入或输出，否则 CPU 等待，循环查询。

这种方式的优点是结构简单，只需要少量的硬件电路即可，缺点是由于 CPU 的速度远远高于外设，因此通常处于等待状态，工作效率很低

(2) 中断处理方式

在这种方式下，CPU 不再被动等待，而是可以执行其他程序，一旦外设为数据交换准备就绪，可以向 CPU 提出服务请求，CPU 如果响应该请求，便暂时停止当前程序的执行，转去执行与该请求对应的服务程序，完成后，再继续执行原来被中断的程序。

中断处理方式的优点是显而易见的，它不但为 CPU 省去了查询外设状态和等待外设就绪所花费的时间，提高了 CPU 的工作效率，还满足了外设的实时要求。但需要为每个 I/O 设备分配一个中断请求号和相应的中断服务程序，此外还需要一个中断控制器（I/O 接口芯片）管理 I/O 设备提出的中断请求，例如设置中断屏蔽、中断请求优先级等。

此外，中断处理方式的缺点是每传送一个字符都要进行中断，启动中断控制器，还要保留和恢复现场以便能继续原程序的执行，花费的工作量很大，这样如果需要大量数据交换，系统的性能会很低。

(3) DMA (直接存储器存取) 传送方式

DMA 最明显的一个特点是它不是用软件而是采用一个专门的控制器来控制内存与外设之间的数据交流，无须 CPU 介入，大大提高 CPU 的工作效率。

在进行 DMA 数据传送之前，DMA 控制器会向 CPU 申请总线控制权，CPU 如果允许，则将控制权交出，因此，在数据交换时，总线控制权由 DMA 控制器掌握，在传输结束后，DMA 控制器将总线控制权交还给 CPU。