
网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

一、 内核编译.....	2
内核打补丁，编译，烧写，试验：	2
1. 打补丁：	2
2. 配置：3 种方法。	2
3. 编译：make	3
二、 内核分析配置.....	4
1. 配置过程：	4
2. 内核子目录 Makefile:	5
三、 内核 MAKEFILE.....	6
1. 从子目录的 Makefile 看起。	7
2. 架构相关的 Makefile。（arch/\${ARCH}/Makefile）	7
3. 顶层目录的 Makefile.	8
4. vmlinux 如何编译	11
四、 机器 ID，启动参数.....	14
1. 建立 SI 工程：先是添加所有的代码，再去除不相关的代码。	14
2. 分析内核源代码：	16
3. 内核启动：最终目标是就运行应用程序。对于 Linux 来说应用程序在 根文件 系统 中。要挂接。	17
4. 内核：	18
<3> 创建页表。	26
.使能 MMU。	26
（5）.跳转到 start_kernel (它就是内核的第一个 C 函数)	27
5. 命令行参数：	32

一、 内核编译

内核打补丁，编译，烧写，试验：

1. 打补丁：

用 patch 命令， -p 是指需要忽略的目录层数。如果打补丁，在源代码里有显示。

```
1: diff -urN linux-2.6.22.6/arch/arm/configs/s3c2410_defconfig linux-2.6.22.6_jz2440/arch/arm/configs/s3c2
2: --- linux-2.6.22.6/arch/arm/configs/s3c2410_defconfig    2007-08-31 14:21:01.000000000 +0800
3: +++ linux-2.6.22.6_jz2440/arch/arm/configs/s3c2410_defconfig    2010-11-26 08:40:21.318071654 +0800
```

--- 是指源文件。

+++ 是指修改过的文件。

-p 1 是指忽略一层目录：linux-2.6.22.6 这层目录。

-p 2 是指忽略两层目录：linux-2.6.22.6/arch 这两层目录。

```
patch -p1 < ../linux-2.6.22.6_jz2440.patch
```

2. 配置：3 种方法。

(1): make menuconfig（成千上万配置项要配置，很复杂，不可取）

(2):使用默认的配置，在此基础上修改。（但现在不知道默认的配置有哪些。可以搜索下）

(3):使用厂家提供的配置文件。（将厂家提供的 config_厂家 直接复制一份为 .config，然后再 make menuconfig）

使用默认的配置，在此基础上修改。

但现在不知道默认的配置有哪些。可以搜索下：

a.加了“通配符”

结果是 /arch/ 目录下各种架构下的 configs 目录下的 "****_defconfig" 文件。

```
[root@localhost configs]# pwd
/work/arm920t/kernel/linux-2.6.22.6/arch/arm/configs
[root@localhost configs]# ls
assabet_defconfig      ebsa110_defconfig      jornada720_defconfig   omap_h2_1610_defconfig
at91rm9200dk_defconfig edb7211_defconfig      kafa_defconfig         onearm_defconfig
at91rm9200ek_defconfig ep93xx_defconfig       kb9202_defconfig       picotux200_defconfig
at91sam9260ek_defconfig footbridge_defconfig   ks8695_defconfig       pleb_defconfig
at91sam9261ek_defconfig fortune_defconfig      lart_defconfig         pnx4008_defconfig
at91sam9263ek_defconfig h3600_defconfig        lpd270_defconfig       pxa255-idp_defconfig
at91sam9r1ek_defconfig h7201_defconfig        lpd7a400_defconfig     realview-smp_defconfig
ateb9200_defconfig     h7202_defconfig        lpd7a404_defconfig     realview_defconfig
badge4_defconfig       hackkit_defconfig      lubbock_defconfig      rpc_defconfig
carmeva_defconfig      integrator_defconfig   lus17200_defconfig     s3c2410_defconfig
cerfcube_defconfig     iop13xx_defconfig      mainstone_defconfig    shannon_defconfig
clps7500_defconfig     iop32x_defconfig       mx1ads_defconfig       shark_defconfig
collie_defconfig       iop33x_defconfig       neponset_defconfig     simpad_defconfig
corgi_defconfig        ixp2000_defconfig      netwinder_defconfig    spitz_defconfig
csb337_defconfig       ixp23xx_defconfig      netx_defconfig         trizeps4_defconfig
csb637_defconfig       ixp4xx_defconfig       ns9xxx_defconfig       versatile_defconfig
[root@localhost configs]#
```

在这个 /arch/arm 目录下找，看有没有和我们的单板相似的架构配置。

这里看到 s3c2410_defconfig 这个文件和我们的 2440 单板相像。则可：

b. `make s3c2410_defconfig`以 2410 作为默认配置。接着 `Make menuconfig` 在 2410 的配置基础上修改

执行完 `make s3c2410_defconfig` 后，显示“配置文件”写到了 “.config” 文件中。

```
CRC32c (Castagnoli, et al) Cyclic Redundancy-Check (LIBCRC32C) [N/m/y/?] n
#
# configuration written to .config
```

c. `make menuconfig`

在 `s3c2410_defconfig` 的基础上可以修改完善成我们需要的。

这里直接用开发板厂家提供的配置文件 `config_ok`，将它改成 `.config`

```
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search.
Legend: [*] built-in [ ] excluded <M> module < > module capable
```

如何操作 `make menuconfig` 的说明文件。

是指“高亮”的首字母。在键盘上按下这样的字母可以直接跳到相应的目录上。

```
Device Drivers --->
```

对于这个目录，直接按下“D”键盘键就可以跳到这一条配置目录上。

搜索是按下“/”会出现搜索框，输入要搜索的内容即可搜索。

3. 编译：make

要生成 `ulmage`：`make ulmage`

`ulmage`：头部+真正的内核，我们想编译一个内核给 UBOOT 用，则用 `make ulmage`。

上电开发板，接上 USB 线（装驱动）在 UBOOT 命令中选：[K]

```
[k] Download Linux kernel uImage
```

之后会等待通过 USB 上传文件。

用工具“dnw.exe”

查看下源代码，看看[k]对应于哪个命令：在 `Cmd_menu.c` 中查找。

```
00193:         case 'k':
00194:         {
00195:             strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase kernel; nand write.jffs2 0x30000000 kernel $(filesize)");
00196:             run_command(cmd_buf, 0);
00197:             break;
00198:         }
```

首先上传数据：usbslave 1 0x30000000 来接收 dnw.exe 发出来的数据，放到 0x3000 0000 地址处。

然后擦除内核分区：nand erase kernel 收到数据后，接着擦除这个 kernel 内核分区。

最后写到内核分区：nand write.jffs2 0x30000000 kernel \$(filesize)

将原来接收到 0x3000 0000 处的文件烧到 kernel 分区去。

烧多大\$(filesize)由这个宏定义，表示接收到的文件大小。

烧写完后，用 [b] 命令启动。

进入 UBOOT 后，可以用 `nand erase root` 是删除文件系统。然后 `boot` 是 UBOOT 启动内核。

二、 内核分析配置

想了解内核的结构，就要了解它的配置过程和编译过程。

1. 配置过程：

<1>配置后产生 `.config` 文件。

里面有很多配置项，等于 `y`、`m` 或某些数值。等于 `y` 表示会编译成内核中去。若是编译成了模块，便可以动态加载。

(2).举例说明：

配置项

```
root@book-desktop:/work/system/linux-2.6.22.6# grep "CONFIG_DM9000" * -nr
arch/mips/configs/db1550_defconfig:729:# CONFIG_DM9000 is not set
arch/mips/configs/ip27_defconfig:649:# CONFIG_DM9000 is not set
arch/mips/configs/sb1250-swarm_defconfig:557:# CONFIG_DM9000 is not set
arch/mips/configs/pb1550_defconfig:723:# CONFIG_DM9000 is not set
arch/mips/configs/yosemite_defconfig:484:# CONFIG_DM9000 is not set
arch/mips/configs/pb1100_defconfig:641:# CONFIG_DM9000 is not set
arch/mips/configs/capcella_defconfig:518:# CONFIG_DM9000 is not set
arch/mips/configs/atlas_defconfig:895:# CONFIG_DM9000 is not set
arch/mips/configs/lasat200_defconfig:630:# CONFIG_DM9000 is not set
arch/mips/configs/jazz_defconfig:853:# CONFIG_DM9000 is not set
arch/mips/configs/malta_defconfig:1022:# CONFIG_DM9000 is not set
arch/mips/configs/ocelot_defconfig:478:# CONFIG_DM9000 is not set
arch/mips/configs/jmr3927_defconfig:454:# CONFIG_DM9000 is not set
arch/mips/configs/pb1500_defconfig:722:# CONFIG_DM9000 is not set
arch/mips/configs/ocelot_c_defconfig:499:# CONFIG_DM9000 is not set
arch/mips/configs/emma2rh_defconfig:786:# CONFIG_DM9000 is not set
arch/mips/configs/ddb5477_defconfig:492:# CONFIG_DM9000 is not set
arch/mips/configs/db1200_defconfig:694:# CONFIG_DM9000 is not set
arch/mips/configs/tb0287_defconfig:651:# CONFIG_DM9000 is not set
arch/mips/configs/db1500_defconfig:692:# CONFIG_DM9000 is not set
arch/mips/configs/ocelot_3_defconfig:670:# CONFIG_DM9000 is not set
arch/mips/configs/ip32_defconfig:577:# CONFIG_DM9000 is not set
```

`include/linux/autoconf.h`

这些都是不同架构的默认配置下的（`config` 是配置文件目录）不用看。

还可以看到源代码中有这个值：

```
arch/arm/plat-s3c24xx/common-smdk.c:46:#if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
arch/arm/plat-s3c24xx/common-smdk.c:162:#if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
arch/arm/plat-s3c24xx/common-smdk.c:200:#endif /* CONFIG_DM9000 */
arch/arm/plat-s3c24xx/common-smdk.c:250:#if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
```

还有 Makefile 中有：

`drivers/net/Makefile`

```
#obj-$(CONFIG_DM9000) += dm9000.o
```

```
#obj-$(CONFIG_DM9000) += dm9ks.o
```

则：

a. C 源代码中用到 CONFIG_DM9000.从 C 语言语法看肯定是个宏。宏只能在 C 文件或是头文件中定义。

依照这里的情况，应该是在“include/linux/autoconf.h”这个头文件中定义。

b. 子目录下的 Makefile 中有 CONFIG_DM9000 配置项（如 drivers/net/Makefile）

c. include/config/auto.conf 中有。

d. include/linux/autoconf.h 中有。从 autoconf.h 可猜测这个文件是自动生成的。它里面的内容来源于

make 内核时，make 机制会自动根据生成的“.config”配置文件，生成 autoconf.h 这个文件。

在 autoconf.h 中搜索 DM9000 得到：#define CONFIG_DM9000 1

可见 CONFIG_DM9000 被定义成一个宏，为“1”。

整个文件 autoconf.h 中的宏基本都是定义为“1”，就是说不管在 .config 配置文件中，配置项=y，还是=m，

在这个由 .config 生成的头文件 autoconf.h 中都被定义成“1”。

C 语言源码中就只使用这些宏了。这些等于 y 等于 m 的在 autoconf.h 中定义宏时都定义为“1”，它们的区别则在

使用这些宏的 C 语言中体现不出来了。这些区别是在 Makefile 中定义。看子目录下的 Makefile.

看子目录 device/net/Makefile 文件。搜索其中 CONFIG_DM9000 文件。

对于内核的 Makefile,它的子目录的 Makefile 很简单。

2. 内核子目录 Makefile:

1.格式比较简单：内核子目录下的 Makefile 中

obj-y += xxx.o

xxx.c 的文件最后会被编译进内核中去。

obj-m += yyy.o

yyy.c 文件最后会编译成 可加载 的模块 yyy.ko 。

这两种格式。

如下示例：

obj-\$(CONFIG_DM9000) += dm9000.o

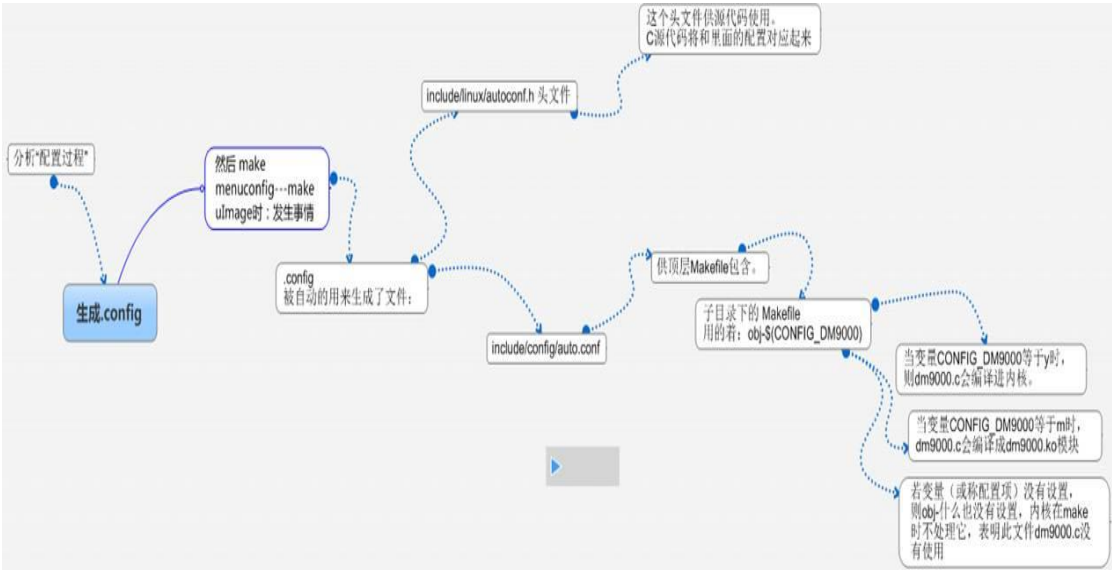
意思：若 CONFIG_DM9000 这个变量被定义为 y 的话，这个 dm9000.c 就会被编译进内核中去。

若这个配置项 CONFIG_DM9000 被定义为 m 的话，这个 dm9000.c 会被编译成 dm9000.ko 模块

y 和 m 的区别就是在内核的子目录下的 Makefile 中体现的。子目录下的 Makefile 中的 CONFIG_DM9000 是谁来定义的呢？

也是由其他来定义的。由“include/config/auto.conf”来定义（这个文件也是来源于 make

内核时的 .config)。从 auto.conf 可知它也是自动生成的。里面的内核和 .config 文件很像。其中的项要么等于 y，要么等于 m，或其他值。
显然这个 include/config/auto.conf 文件是会被别人包含进去。（被顶层的 Makefile 包含）



配置内核时生成了 .config 文件。然后 make menuconfig 或是 make ulmage 时：
1，.config 被自动来创建生成了一个 include/linuxautoconfig.h 文件 这个头文件被 C 源代码去对应里面的配置。
2，.config 也被自动来生成一个 include/config/auto.conf 文件。这个文件由顶层 Makefile 来包含。由子目录下的 Makefile 来用它。

三、 内核 Makefile

分析 Makefile:找到第一个目标文件和链接文件。
第一个目标文件（一路跟踪启动过程）
链接文件：它表示内核应该放在哪里，里面的代码是如何排布的。

表 16.3 Linux 内核 Makefile 文件分类	
名 称	描 述
顶层 Makefile	它是所有 Makefile 文件的核心，从总体上控制着内核的编译、连接
.config	配置文件，在配置内核时生成。所有 Makefile 文件（包括顶层目录及各级子目录）都是根据.config 来决定使用哪些文件
arch/S(ARCH)/Makefile	对应体系结构的 Makefile，它用来决定哪些体系结构相关的文件参与内核的生成，并提供一些规则来生成特定格式的内核映像
scripts/Makefile.*	Makefile 共用的通用规则、脚本等
kbuild Makefiles	各级子目录下的 Makefile，它们相对简单，被上一层 Makefile 调用来编译当前目录下的文件

对于 Makefile 的文档在 Documentation\kbuild 下的 makefiles.txt 对内核的 makefile 讲的很

透彻。

1. 从子目录的 Makefile 看起。

每个子目录下都会有个 Makefile 文件。

```
obj-$(CONFIG_A2232) += ser_a2232.o generic_serial.o
```

若这个变量 CONFIG_A2232 在配置文件中被定义成 y 时, 则 ser_a2232.c 和 generic_serial.c 文件会编译成如上图的 .o 文件。

最后会链接到内核中去。若这个变量定义成 m 时, 则 ser_a2232.c 和 generic_serial.c 文件会编译成 .ko 模块文件。

举例两个文件: a.c 和 b.c

(1). 要编译进内核:

则: obj-y += a.o b.o

(2). a.c 和 b.c 两者要组成一个模块。

可见 makefiles.txt 说明文档有举例:

```
189      Example:
190      #drivers/isdn/i4l/Makefile
191      obj-$(CONFIG_ISDN) += isdn.o
192      isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
193
```

则可以写成:

obj-m += ab.o (这名字无关紧要)

ab-objs := a.o b.o

编译时, a.c 编译成 a.o;

b.c 编译成 b.o;

a.o 和 b.o 会一起链接成 ab.ko 模块。

总结: 子目录下的 Makefile 很简单, 就只有几条格式:

obj-y += a.o b.o

obj-m += a.o

2. 架构相关的 Makefile 。(arch/\$(ARCH)/Makefile)

分析一个 Makefile 时, 从它的命令开始分析。编译内核时是直接 make 或 make ulmage 从顶层 Makefile 一直往下走时会涉及到所有的东西。

<1> make ulmage 时这个目标 ulmage 不在顶层的 Makefile 中, 在 arch/arm/Makefile 中定义了这个目标。

我们是在顶层目录 make ulmage 的, 则可知顶层 Makefile 会包含 arch/arm/Makefile 。

在顶层目录的 Makefile 中搜索 “include”：

```
413 include $(srctree)/arch/$(ARCH)/Makefile
```

srctree:源码树 ARCH 是 arm.

```
185 #ARCH ?= $(subst ARCH,
186 ARCH ?= arm
187 CROSS_COMPILE ?= arm-linux-
188
```

打上补丁后，我们直接在内核的顶层目录下的 Makefile 。这里 ARCH ?= arm 是修改过的。

(2) 顶层的 .config 最终会生成 include/linux/autoconf.h 头文件给源码用，另一个是 include/config auto.conf 文件。

```
443 -include include/config/auto.conf
444
```

可见配置文件也被包含到了顶层 Makefile 中。

可见，配置文件，子目录下的 Makefile 都会被包含进顶层的 Makefile 中去。则重点分析顶层 Makefile.

3. 顶层目录的 Makefile.

从 make ulmage 命令往下分析。

<1> 目标 ulmage 定义在 arch/arm/Makefile 中，找到 ulmage 目标所在行，查看它相关的依赖。

```
224: # Convert bzImage to zImage
225: bzImage: zImage
226:
227: zImage Image xipImage bootpImage uImage: vmlinux
228:     $(Q) $(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
229:
230: zinstall install: vmlinux
231:     $(Q) $(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $@
232:
233: CLEAN_FILES += include/asm-arm/mach-types.h \
234:     include/asm-arm/arch include/asm-arm/.arch
235:
```

可见 ulmage 依赖于 vmlinux.

ulmage 是一个 头部 + 真正的内核。所以制作这个 ulmage 时需要编译出真正的内核。这个真正的内核显然就是 vmlinux 。

vmlinux 的依赖在顶层目录的 Makefile 中。:

```
484: all: vmlinux
```


在顶层目录直接输入 `make`，默认就是执行第一个目标，"all"就是第一个目标。这个目标也是依赖于 `vmlinux`。即都是要先生成 `vmlinux`。

<2> `vmlinux` 的依赖：

```
744: # vmlinux image - including updated kernel symbols
745: vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE
746: ifdef CONFIG_HEADERS_CHECK
747:     $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
748: endif
749:     $(call if_changed_rule,vmlinux__)
750:     $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost $@
751:     $(Q)rm -f .old_version
752:
```

`vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE`

从名字上可以猜到：

`vmlinux-lds`：链接脚本。

`vmlinux-init`：一些初始化代码。

`vmlinux-main`：一些主要的代码（与内核核心相关的）。

<3> 分别分析这些变量：在顶层 `Makefile` 中。

```
608: vmlinux-init := $(head-y) | $(init-y)
609: vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
610: vmlinux-all := $(vmlinux-init) $(vmlinux-main)
611: vmlinux-lds := arch/$(ARCH)/kernel/vmlinux.lds
```

`vmlinux-init := $(head-y) $(init-y)`

`head-y` 在顶层目录的 `Makefile` 中没有定义，则会在架构目录下（`arch/arm/Makefile`）的 `Makefile` 中。

```
94: head-y := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
```

`MMUEXT` 若没人定义时，这个变量就变成空的（最终这里也没有定义它），就变成 `head.o`。

从依赖可知，

最初始的代码就是 `head-y` 的两个依赖代码（`head.o` 和 `init_task.o`）。

`init-y` 在顶层 `Makefile` 中。

```
434: init-y := init/
```

`init-y := init/`

```
573: init-y := $(patsubst %/, %/built-in.o, $(init-y))
```

```
init-y      := $(patsubst %, %/built-in.o, $(init-y))
```

这是一个 Makefile 的函数。

%/ 代表的是 init/目录下的所有文件。

%/built-in.o 相当于在 init/下的文件全部编译成 built-in.o 。

这个函数的意思是：init-y := \$(patsubst %, %/built-in.o, \$(init-y)) = init/built-in.o

即 init-y 等于 init 目录下所有涉及的那些文件，这些文件会被编译成一个 built-in.o

patsubst：替换通配符。

在\$(patsubst %.c, %.o, \$(dir))中，patsubst 把\$(dir)中的变量符合后缀是.c 的全部替换成.o，任何输出。

或者可以使用

```
obj=$(dir:%.c=%.o)
```

效果也是一样的。

```
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
```

core-y：核心 libs-y：库 drivers-y：驱动 net-y：网络

在 Makefile 中搜索 core-y 有如下依赖：

```
core-y      := usr/
```

```
core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

```
core-y      := $(patsubst %, %/built-in.o, $(core-y))
```

意思是最后 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
```

就是将这些目录（usr、kernel、mm、fs、ipc、security、crypto、block）下涉及的文件分别编译成 built-in.o

不是所有文件，而是涉及到的文件。

libs-y：依赖

```
libs-y1     := $(patsubst %, %/lib.a, $(libs-y))
```

```
libs-y2     := $(patsubst %, %/built-in.o, $(libs-y))
```

```
libs-y      := $(libs-y1) $(libs-y2)
```

最后 libs-y = lib/lib.a

```
+= lib/built-in.o
```

drivers-y：驱动

```
drivers-y   := drivers/ sound/ （依赖了这两个目录）
```

```
drivers-y   := $(patsubst %, %/built-in.o, $(drivers-y))
```

意思是最后 drivers-y = drivers/built-in.o (将 drivers 目录下所有涉及的文件编译成 built-in.o 文件)

```
+= sound/built-in.o    (将 sound 目录下所有涉及的编译成 built-in.o 文件)
```

net-y : 网络

$$\text{net-}y \quad := \text{net-}y$$

```
net-y      := $(patsubst %/, %/built-in.o, $(net-y))
```

意思是最后，将 net/目录下的所有涉及到的文件编译 built-in.o 这个文件。

从面的依赖文件展开来看，源材料就是上面这一大堆东西。这些东西如何组合成一个内核（链接成在一块），要看 `vmlinux` 如何编译的。

4. vmlinux 如何编译

```
745: vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE
746: ifdef CONFIG_HEADERS_CHECK
747:     $(Q)$ (MAKE) -f $(srctree)/Makefile headers_check
748: endif
749:     $(call if_changed_rule,vmlinux__)
750:     $(Q)$ (MAKE) -f $(srctree)/scripts/Makefile.modpost $@
751:     $(Q)rm -f .old version
```

编译时是通过这些命令来编译的。这些命令最终会生成什么东西？可以通过这里一一分析下去。这里涉及的脚本、函数太庞大了。没精力去做。

想知道上在的源材料如何编译成内核:

方法 1: 分析 Makefile .

方法 2：直接编译内核。看编译过程。

a.rm vmlinux 先删除原来编译得到的内核。

b.make ulmage V=1 (V=1 是更加详细的列出那些命令。)

我们关心详细命令的最后一条。

```
echo 'cmd vmlinux := arm-linux-ld -EL -p --no-undefined -X -o vmlinux -T arch/arm/kernel/vmlinux.lds arch/arm/kernel/head.o arch/arm/kernel/
t/built-in.o --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-s3c
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
3/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 s
.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/bu
ilt-in.o --end-group tmp kallsyms2.o' > ./vmlinux.cmd
```

arm-linux-ld -EL -p --no-undefined -X -o vmlinux (-o 这里生成 vmlinux 了)

-T arch/arm/kernel/vmlinux.lds 链接脚本

arch/arm/kernel/head.o

```
arch/arm/kernel/init task.o
```

init/built-in.o

--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/nwfpe/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

arm-linux-ld -EL 是链接。

-o vmlinux: -o 是输入 vmlinux.

arch/arm/kernel/vmlinux.lds 是链接脚本。它决定所有文件链接成一个内核时文件是如何分布的。

下面那些 .o 文件为原材料: (分别与下面的 Makefile 中的定义对应)

arch/arm/kernel/head.o

arch/arm/kernel/init_task.o

```
vmlinux-init := $(head-y) $(init-y)
head-y       := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
```

init/built-in.o

```
init-y       := init/
init-y       := $(patsubst %/, %/built-in.o, $(init-y)) = init/built-in.o
```

接着是一大堆 “--start-group” 到 “--end-group”:

```
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))
            = 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

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
```

分析后确定两个方面:

第一个文件是谁: 从上面 make ulmage 最后一条命令可知, 内核第一个文件是 arch/arm/kernel/head.o (head.S)

链接脚本: arch/arm/kernel/vmlinux.lds (决定内核如何排布).

链接脚本 vmlinux.lds 是由 vmlinux.lds.S 文件生成的。

```
root@book-desktop:/work/system/linux-2.6.22.6# ls arch/arm/kernel/vmlinux.lds.S
arch/arm/kernel/vmlinux.lds.S
```

查看最终生成的 vmlinux.lds 可见其中如下内容:

```
. = (0xc0000000) + 0x00008000;
```

这里一开始便指定了内核放在哪里。这显然是虚拟的地址。

```
.text.head : {
    _stext = .;
    _sinittext = .;
    *(.text.head)
}
```

一开始是放 “*” (指所有文件) 的 “.text.head” 段。

```
.init : { /* Init code and data          */
    *(.init.text)
```

再接着是放所有文件的 “.init.text” 段。

等等...

这些所有文件排在相应的 “段” 中, 排放的顺序就是如下 “链接脚本” 后面 “.o” 文件的排布顺序:

```
echo 'cmd vmlinux := arm-linux-ld -EL -p --no-undefined -X -o vmlinux -T arch/arm/kernel/vmlinux.lds arch/arm/kernel/head.o arch/arm/kernel/in
t/built-in.o --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-s3c24
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/a
3/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 sec
.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/bui
ilt-in.o --end-group tmp_kallsyms2.o' > ./vmlinux.cmd
```

如上首先放 head.o 的, 等等。文件的顺序由上面这些 “.o” 文件出现的顺序为准。里面的代码段等等其他段的排放由 “vmlinux.lds” 决定, 首先放 “.text.head” 段, 其次是 “.init.text” 段等依次往下排 (参见 “vmlinux.lds” 内容)。

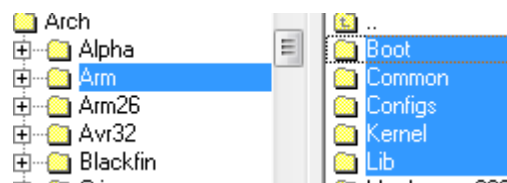
四、 机器 ID，启动参数

1. 建立 SI 工程：先是添加所有的代码，再去除不相关的代码。

添加完所有的文件代码后，再移除 ARCH 目录（因为里面有不需要的代码），移除后再进到 ARCH 目录重新

添加 ARM 相关的代码（因为这里处理的是 ARM 平台）

<1> 选择 ARCH 目录下与 24x0 相关的源代码：

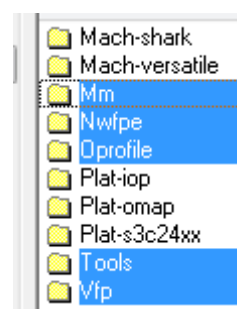


添加完所有目录后，去掉 Arch 目录，因为是 ARM，所以只需要保留 arm 目录相关文件就好。

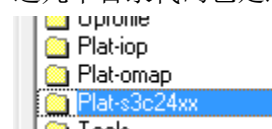
如上几个目录是 arm 目录下通用的。加进工程。



再加上 2410 和 2440 两个平台的源码。



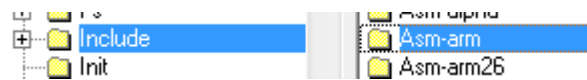
这几个目录代码也是通用的。加进工程。



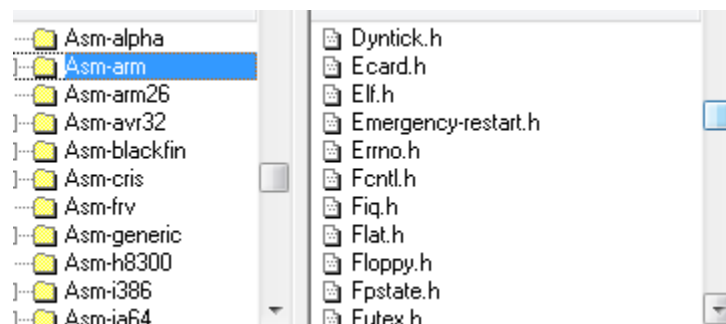
最后添加上平台：Plat-s3c24xx 平台，其他平台不需要加。

<2> include 目录。里面也有很多东西，先“Remove Tree”后，再挑选我们需要的加进工程。

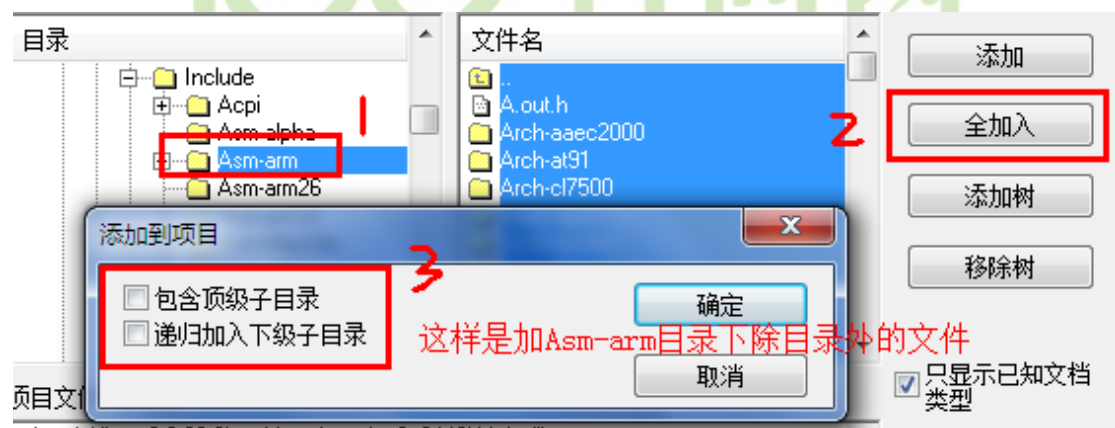
在 include 目录中，asm 开头的目录显然是 架构 相关的头文件。我们只关心 Asm-arm arm 架构的头文件。



进入 Asm-arm 目录，因为还有其他 CPU 的 arm 头文件，所以进去 Asm-arm 目录后要选择。

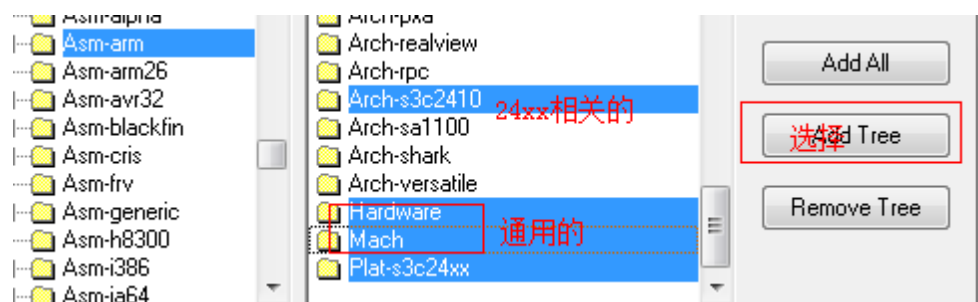


这是 Asm-arm 目录最上层中的文件，先将它们加进工程，一会再选择 Asm-arm 下的目录添加。

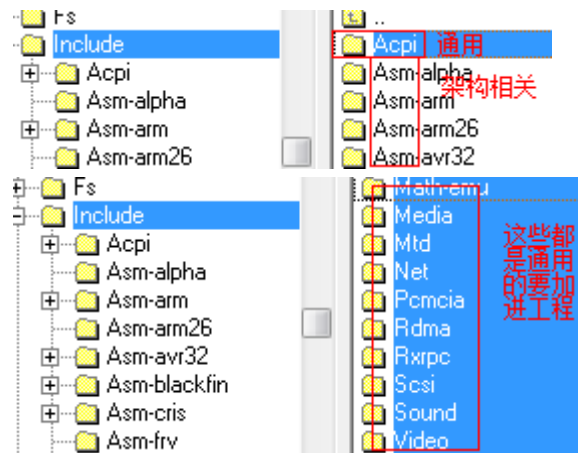


这样就添加了文件，没有添加里面的目录。

接着选择通用的（不以 Arch-开头的目录），和 24xx 相关的。



退出到 include 目录后，还有很多不是 Asm 开头标明是架构相关的目录和文件。它们是通用要加进工程。



至此，这个工程建立好了。然后要同步下工程下的源代码。

head.S 做的事情：

- (0) .判断是否支持此 CPU
- (1) .如何比较 机器 ID 是：（判断是否支持单板）
- (3) .创建页表。
- (4) .使能 MMU。
- (5) .跳转到 start_kernel (它就是内核的第一个 C 函数)

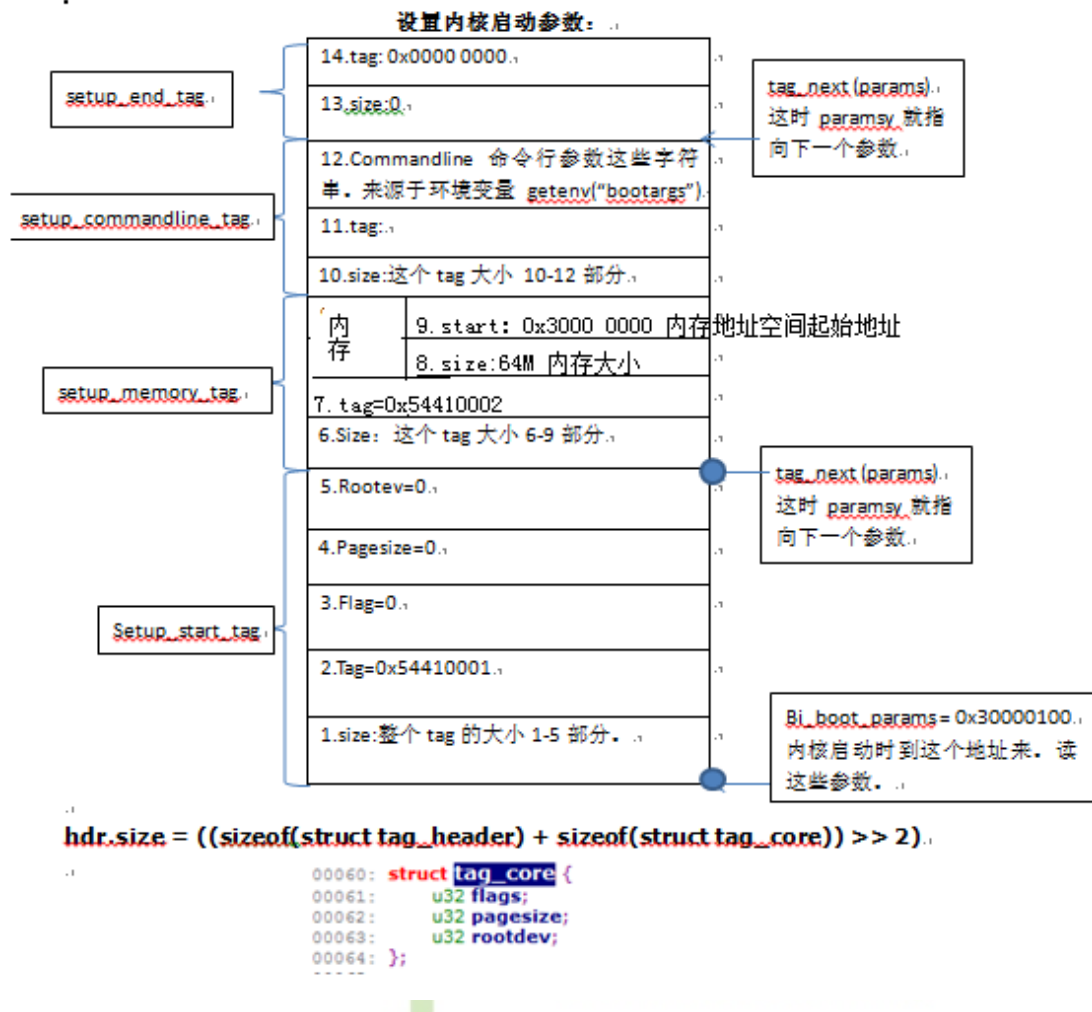
2. 分析内核源代码：

<1> 通过 make ulmage V=1 详细查看内核编译时的最后一条命令可知。

内核中排布的第一个文件是：arch/arm/kernel/head.S

链接脚本： arch/arm/kernel/vmlinux.lds

UBOOT 启动时首先在内存里设置了一大堆参数。



接着启动内核：

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

theKernel 就是内核的入口地址。有 3 个参数。参 1 为 0，参 2 为机器 ID，参 3 是上面那些参数所存放的地址。

所以，内核一上来肯定要处理这些参数。

3. 内核启动：最终目标是就运行应用程序。对于 Linux 来说应用程序在 根文件系统中。要挂接。

(1) 处理 UBOOT 传入的参数。

内核中排布的第一个文件是： arch/arm/kernel/head.S

File Name
head.S (arch\arm\boot\compressed)
head.S (arch\arm\kernel)

内核编译出来后比较大，可以压缩很小。在压缩过的内核前部加一段代码“自解压代码”。这样内核运行时，先运行“自解压代码”。然后再执行解压缩后的内核。

(2) 我们看不用解压的 head.S 文件。

```
00081:      mrc p15, 0, r9, c0, c0      @ get processor id
00082:      bl  __lookup_processor_type    @ r5=procinfo r9=cputid
```

__lookup_processor_type 查找处理器类型

内核能够支持哪些处理器，是在编译内核时定义下来的。内核启动时去读寄存器：获取 ID。

```
00081:      mrc p15, 0, r9, c0, c0      @ get processor id 获取ID
```

看处理器 ID 后，看内核是否可以支持这个处理器。若能支持则继续运行，不支持则跳到“_error_p”中去：

```
00084:      beq __error_p      @ yes, error 'p'
```

这是个死循环。

```
00085:      bl  __lookup_machine_type    @ r5=machinfo
00086:      movs r8, r5      @ invalid machine (r5=0)?
00087:      beq __error_a      @ yes, error 'a'
00088:      bl  __create_page_tables
```

__lookup_machine_type 机器 ID。

一个编译好的内核能支持哪些单板，都是定下来的。内核上电后会检测下看是否支持当前的单板。若可以支持则继续往下跑，不支持则 __error_a 跳到死循环。

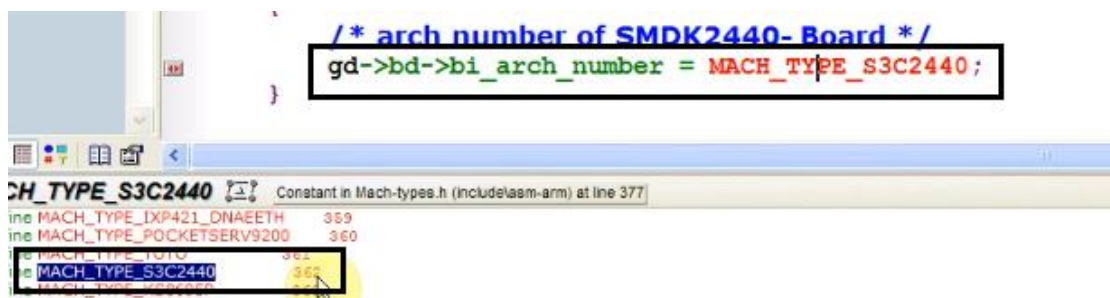
4. 内核：

处理 UBOOT 传入的参数

1. 首先判断是否支持这个 CPU。

2. 判断是否支持这个单板。(UBOOT 启动内核时传进来的：机器 ID bd->bi_arch_number)

查 UBOOT 代码，对于这块开发板是：gd->bd->bi_arch_number = MACH_TYPE_SMDK2410



这个 362 这个值存在哪里？从汇编的 C 语言交互规则知道。这个参数 bi_arch_number 是

存在 r1 寄存器中的。

```
00201:      teq r3, r1      @ matches loader number?
```

r1 传进来的就是。r1 存放的就是 UBOOT 传进来的 362（机器 ID）。

（0）.判断是否支持此 CPU

（1）.如何比较 机器 ID 是：（判断是否支持单板）

```
00194:      lookup_machine_type:
00195:      adr r3, 3b
00196:      ldmbia r3, {r4, r5, r6}
00197:      sub r3, r3, r4      @ get offset between virt&phys
00198:      add r5, r5, r3      @ convert virt addresses to
00199:      add r6, r6, r3      @ physical address space
00200: 1:    ldr r3, [r5, #MACHINE_TYPE] @ get machine type
00201:      teq r3, r1      @ matches loader number?
00202:      beq 2f          @ found
00203:      add r5, r5, #SIZEOF_MACHINE_DESC @ next machine_desc
00204:      cmp r5, r6
00205:      blo 1b
00206:      mov r5, #0      @ unknown machine
00207: 2:    mov pc, lr
```

__lookup_machine_type:

```
adr    r3, 3b
```

首先 r3 等于 3b 的地址。这是实际存在的地址。UBOOT 启动内核时，MMU 还没启动。所以这是物理地址。

3b 就是

```
00178: 3: .long .
```

这个的地址。

则 r3 就等于这段代码

```
00178: 3: .long .
```

的地址。

```
ldmbia r3, {r4, r5, r6}
```

r4 等于

```
00178: 3: .long .
```

这里的 “.”（当前地址的意思）。

点 是一个“虚拟地址”。是编译到

```
00178: 3: .long .
```

这里时，此处指令

这句话的虚拟地址。“.”代表虚拟地址。是标号为“3”的指令的虚拟地址。

r5 等于

```
00179: .long __arch_info_begin
```

中的 “__arch_info_begin”

r6 等于

```
00180:      .long      __arch_info_end
```

中的 “__arch_info_end”

```
sub    r3, r3, r4          @ get offset between virt&phys
```

这两相地址相减就得到了偏移量 offset,虚拟地址和物理地址间的偏差。

r4 等于 “.” 虚拟地址; r3 等于 3b 这个实际存在的地址。

```
add    r5, r5, r3          @ convert virt addresses to
```

```
add    r6, r6, r3          @ physical address space
```

r5,r6 是加上这个偏差值。原来 r5=__arch_info_begin;

r6=__arch_info_end.它们加上这个偏差值后,就变成了__arch_info_begin

和__arch_info_end 的真正物理地址了。

__arch_info_begin

__arch_info_end

没有在内核源码中定义,是在 链接脚本 中定义的 (vmlinux.lds)。

```
305     __arch_info_begin = .;
306     *(.arch.info.init)
307     __arch_info_end = .;
```

中间夹着 *(.arch.info.init)

* 表示所有文件。这里是指所有文件的 .arch \.info \.init 段。

(架构、信息、初始化)即架构相关的初始化信息全放在这里。它开始地址是 __arch_info_begin, 结束地址是 __arch_info_end。这两个地址上从

```
291     . = (0xc0000000) + 0x00008000;
```

(虚拟地址)一路的增涨下来的。

```

291 . = (0xc0000000) + 0x00008000;
292
293 .text.head : {
294     _stext = .;
295     _sinittext = .;
296     *(.text.head)
297 }
298
299 .init : { /* Init code and data */
300     *(.init.text)
301     _einittext = .;
302     __proc_info_begin = .;
303     *(.proc.info.init)
304     __proc_info_end = .;
305     __arch_info_begin = .;
306     *(.arch.info.init)
307     __arch_info_end = .;
308     __tagtable_begin = .;
309     *(.taglist.init)
310     __tagtable_end = .;
311     . = ALIGN(16);
312     __setup_start = .;
313     *(.init.setup)
314     __setup_end = .;
315     __early_begin = .;
316     *(.early_param.init)
317     __early_end = .;
318     __initcall_start = .;

```

知道上面这个关系后，我们则一定要知道在代码里面谁定义了.arch.info.init 这些东西。在内核中搜索它们。

```

book@book-desktop:/work/system/linux-2.6.22.6# grep ".arch.info.init" * -nR
Binary file arch/arm/mach-s3c2443/mach-s3c2443.o matches
Binary file arch/arm/mach-s3c2443/built-in.o matches
Binary file arch/arm/mach-s3c2410/mach-s3c2410.o matches
Binary file arch/arm/mach-s3c2410/mach-qt2410.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-s3c2440.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-s3c2412.o matches
Binary file arch/arm/mach-s3c2412/built-in.o matches
Binary file arch/arm/mach-s3c2412/mach-vstas.o matches
include/asm-mach/arch.h:53: __attribute__((__section__(".arch.info.init"))) = { \
include/asm-mach/arch.h:53: __attribute__((__section__(".arch.info.init"))) = { \
book@book-desktop:/work/system/linux-2.6.22.6#
book@book-desktop:/work/system/linux-2.6.22.6#

```

由此可以猜测支持这些单板

查它们的定义：include/asm-arm/mach/Arch.h 53 行。

```

00050: #define MACHINE_START(_type, _name) \
00051: static const struct machine_desc __mach_desc_##_type \
00052: __used \
00053: __attribute__((__section__(".arch.info.init"))) = { \
00054:     .nr = MACH_TYPE_##_type, \
00055:     .name = _name, \
00056: \
00057: #define MACHINE_END \
00058: };

```

```

#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 \
};

```

找下这个宏 MACHINE_START 谁在用。基本上 arch/arm 架构下都在用这个宏。

```

---- MACHINE_START Matches (14 in 14 files) ----
Arch.h (include/asm-arm/mach):#define MACHINE_START(_type,_name) \
Mach-amlm5900.c (arch/arm/mach-s3c2410):MACHINE_START(AML_M5900, "AML_M5900")
Mach-anubis.c (arch/arm/mach-s3c2440):MACHINE_START(ANUBIS, "Simtec-Anubis")
Mach-bast.c (arch/arm/mach-s3c2440):MACHINE_START(BAST, "Simtec-BAST")
Mach-h1940.c (arch/arm/mach-s3c2410):MACHINE_START(H1940, "IPAQ-H1940")
Mach-n30.c (arch/arm/mach-s3c2410):MACHINE_START(N30, "Acer-N30")
Mach-nexcoder.c (arch/arm/mach-s3c2440):MACHINE_START(NEXCODER_2440, "NexVision - Nexcoder 2440")
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 a new identifier and switch
Mach-smdk2440.c (arch/arm/mach-s3c2440):MACHINE_START(S3C2440, "SMDK2440")
Mach-vr1000.c (arch/arm/mach-s3c2410):MACHINE_START(VR1000, "Thorcom-VR1000")

00339: MACHINE_START(S3C2440, "SMDK2440")
00340: /* Maintainer: Ben Dooks <ben@fluff.org> */
00341: .phys_io = S3C2410_PA_UART,
00342: .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xffff,
00343: .boot_params = S3C2410_SDRAM_PA + 0x100,
00344:
00345: .init_irq = s3c24xx_init_irq,
00346: .map_io = smdk2440_map_io,
00347: .init_machine = smdk2440_machine_init,
00348: .timer = &s3c24xx_timer,
00349: MACHINE_END

```

上面一段代码使用了这个宏，将宏展开：

```

MACHINE_START(S3C2440, "SMDK2440")
/* Maintainer: Ben Dooks <ben@fluff.org> */
.phys_io = S3C2410_PA_UART,
.io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 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
（这是个结构体）

```

根据以前分析 UBOOT 的命令实现来猜测，这两个宏 “MACHINE_START”，“MACHINE_END

”就是定义一个结构体。

这个结构体的特殊是被强制的设置一个属性 “__attribute__((__section__(“.arch.info.init”)))” 将它的段设置成 “.arch.info.init”。

如果哪个文件中要是有这个结构体的话，就会被这个属性和链接脚本 vmlinux.lds(

```
305  __arch_info_begin = .;
306  *(.arch.info.init)
307  __arch_info_end = .;
308  )
```

组合在一块。

以下展开宏看看结果：

```
1: MACHINE_START(S3C2440, "SMDK2440")
2:     /* Maintainer: Ben Dooks <ben@fluff.org> */
3:     .phys_io      = S3C2410_PA_UART,
4:     .io_pg_offst  = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
5:     .boot_params  = S3C2410_SDRAM_PA + 0x100,
6:
7:     .init_irq     = s3c24xx_init_irq,
8:     .map_io       = smdk2440_map_io,
9:     .init_machine = smdk2440_machine_init,
10:    .timer         = &s3c24xx_timer,
11: MACHINE_END
12: -----
13: #define MACHINE_START(_type, _name) \
14: static const struct machine_desc __mach_desc_##_type \
15:     __used \
16:     __attribute__((__section__(".arch.info.init"))) = { \
17:     .nr      = MACH_TYPE_##_type, \
18:     .name    = _name, \
19:
20: #define MACHINE_END    }; /*MACHINE_END 这个宏被定义成 " }; " */
21: -----
```

```
23: MACHINE_START(_type, _name)
24: MACHINE_START(S3C2440, "SMDK2440")
25: 1.这个宏的有两个参数，参1: type; 参2: name, 将它们带入下面。
26: 2.__mach_desc_##typeKH :##两个#表示连接符号。
27: 3.可知：
28:     _type = S3C2440
29:     _name = "SMDK2440"
30: 将实参带到宏定义中去：
31: static const struct machine_desc __mach_desc_S3C2440 \
32: /*定义了一个静态的结构体 machine_desc, desc是描述的意思，即“机器描述”结构体。*/
33: /*这个结构体的名字: __mach_desc_S3C2440 */
34:     __used \
35:     __attribute__((__section__(".arch.info.init"))) = { \
36: /*它的属性 (__attribute__) 是 (__section__) 段被强制设置为 (".arch.info.init") */
37:     .nr      = MACH_TYPE_S3C2440, \
38:     .name    = "SMDK2440", \
39:     /*以上有参宏MACHINE_START(S3C2440, "SMDK2440")就是被展开了。看行号1处*/
40:     /*接着添加要加上 行2~10 处的代码*/
41:     /* Maintainer: Ben Dooks <ben@fluff.org> */
42:     .phys_io      = S3C2410_PA_UART,
43:     .io_pg_offst  = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
44:     .boot_params  = S3C2410_SDRAM_PA + 0x100,
45:
46:     .init_irq     = s3c24xx_init_irq,
47:     .map_io       = smdk2440_map_io,
48:     .init_machine = smdk2440_machine_init,
49:     .timer         = &s3c24xx_timer,
50: }; /*MACHINE_END 这个宏被定义成 " }; " */
```


最后展开的结果是：

```
最后：
MACHINE_START(S3C2440, "SMDK2440")
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params = S3C2410_SDRAM_PA + 0x100,
    .init_irq    = s3c24xx_init_irq,
    .map_io      = smdk2440_map_io,
    .init_machine = smdk2440_machine_init,
    .timer       = &s3c24xx_timer,
MACHINE_END

变展开成：
static const struct machine_desc __mach_desc_S3C2440 \
    __used \
    __attribute__((__section__(".arch.info.init"))) = { \
    .nr          = MACH_TYPE_S3C2440, \
    .name        = "SMDK2440",
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params = S3C2410_SDRAM_PA + 0x100,
    .init_irq    = s3c24xx_init_irq,
    .map_io      = smdk2440_map_io,
    .init_machine = smdk2440_machine_init,
    .timer       = &s3c24xx_timer,
};
```

看看结构体：machine_desc

```
00017: struct machine_desc {
00018:     /*
00019:      * Note! The first four elements are used
00020:      * by assembler code in head-armv.S
00021:      */
00022:     unsigned int nr; /* nr: 机器ID */
00023:     unsigned int phys_io; /* start of physical io */
00024:     unsigned int io_pg_offst; /* byte offset for io
00025:                               * page table entry */
00026:
00027:     const char *name; /* architecture name */
00028:     unsigned long boot_params; /* tagged list */
00029:     /* UBOOT传进来的参数 */
00030:     unsigned int video_start; /* start of video RAM */
00031:     unsigned int video_end; /* end of video RAM */
00032:
00033:     unsigned int reserve_lp0 :1; /* never has lp0 */
00034:     unsigned int reserve_lp1 :1; /* never has lp1 */
00035:     unsigned int reserve_lp2 :1; /* never has lp2 */
00036:     unsigned int soft_reboot :1; /* soft reboot */
00037:     void (*fixup)(struct machine_desc *,
00038:                  struct tag *, char **);
00039:
00040:     void (*map_io)(void); /* IO mapping function */
00041:     void (*init_irq)(void);
00042:     struct sys_timer *timer; /* system tick timer */
00043:     void (*init_machine)(void);
00044: } ? end machine_desc ? ;
```

内核支持多少单板，就有多少个这种以 “MACHINE_START” 开头，以 “MACHINE_END” 定义起来

的代码。每种单板都有机器 ID（结构体中的 nr），机器 ID 是整数。

UBOOT 传来这个参数：

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

与内核的这部分刚好对应（如下部分）

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

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

内核将这部分代码编译进去了，就支持这段代码定义的 单板。上面这段代码中的结构体有一个属性，

它的段被强制设置到了

```
__attribute__((__section__(".arch.info.init"))) = {
```

，这段代码和结构

体被放在 vmlinux.lds 定义的：

```
305 __arch_info_begin = .;
306 *(.arch.info.init)
307 __arch_info_end = .;
```

2410，2440，qt2410 的单板代码都强制放在这个地方。

内核启动时，会从 __arch_info_begin = . 开始读，读到 __arch_info_end = . 一个一个的将单板

信息取出来。将里面的机器 ID 和 UBOOT 传进来的机器 ID 比较。相同则表示内核支持这个单板。

下面就是比较机器 ID 了。(内核中的和 UBOOT 传进来的)看 arch/arm/kernel/head-common.S 从 “__lookup_machine_type:” 中可知 r5=__arch_info_begin

```
1: ldr r3, [r5, #MACHINE_TYPE] @ get machine type
```

r5 是: __arch_info_begin

r1 是 UBOOT 传来的参数: bi_arch_number

```
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
```

最后比较成功后，会回到：head.S

```
00085:    bl    __lookup_machine_type    @ r5=machinfo
00086:    movs   r8, r5                  @ invalid machine (r5=0)?
```

以上 单板机器 ID 比较完成。

<3> 创建页表。

```
bl    __lookup_processor_type    @ r5=procinfo r9=cuid
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
```

内核的链接地址从虚拟地址

```
. = (0xc0000000) + 0x00008000;
```

开始。这个地址并不代表真实存在的

内存。我们的内存是从 0x3000 0000 开始的。

故这里面要建立一个页表，启动 MMU 。

.使能 MMU。

```
00097:    ldr    r13, __switch_data      @ address to jump to after
00098:                                @ mmu has been enabled
00099:    adr    lr, __enable_mmu      @ return (PIC) address
00100:    add    pc, r10, #PROCINFO_INITFUNC
```

@ mmu has been enabled 当 MMU 使能后，会跳到 __switch_data 中去。

如何跳到 __switch_data ，则看：__enable_mmu

```

00014:      .type    __switch_data, %object
00015:  switch_data:
00016:      .long    __mmap_switched
00017:      .long    __data_loc          @ r4
00018:      .long    __data_start        @ r5
00019:      .long    __bss_start         @ r6
00020:      .long    __end               @ r7
00021:      .long    processor_id        @ r4
00022:      .long    __machine_arch_type @ r5
00023:      .long    cr_alignment        @ r6
00024:      .long    init_thread_union + THREAD_START_SP @ sp
-----

```

在 head_common.S 文件中，__switch_data 后是：__mmap_switched

```

00035:  mmap_switched:
00036:      adr r3, __switch_data + 4
00037:
00038:      ldmia   r3!, {r4, r5, r6, r7}
00039:      cmp r4, r5                      @ Copy data segment if needed
00040:  1:  cmpne r5, r6
00041:      ldrne   fp, [r4], #4
00042:      strne   fp, [r5], #4
00043:      bne 1b
00044:
00045:      mov fp, #0                      @ Clear BSS (and zero fp)
00046:  1:  cmp r6, r7
00047:      strcc   fp, [r6], #4
00048:      bcc 1b
00049:
00050:      ldmia   r3, {r4, r5, r6, sp}
00051:      str r9, [r4]                    @ Save processor ID
00052:      str r1, [r5]                    @ Save machine type
00053:      bic r4, r0, #CR_A               @ Clear 'A' bit
00054:      stmia   r6, {r0, r4}           @ Save control register values
00055:      b start_kernel              跳到第一个C函数中处理 UBOOT 传来的“参数”

```

(5) 跳转到 start_kernel (它就是内核的第一个 C 函数)

UBOOT 传进来的启动参数，参 2：机器 ID 在 head.Skh 中会比较。

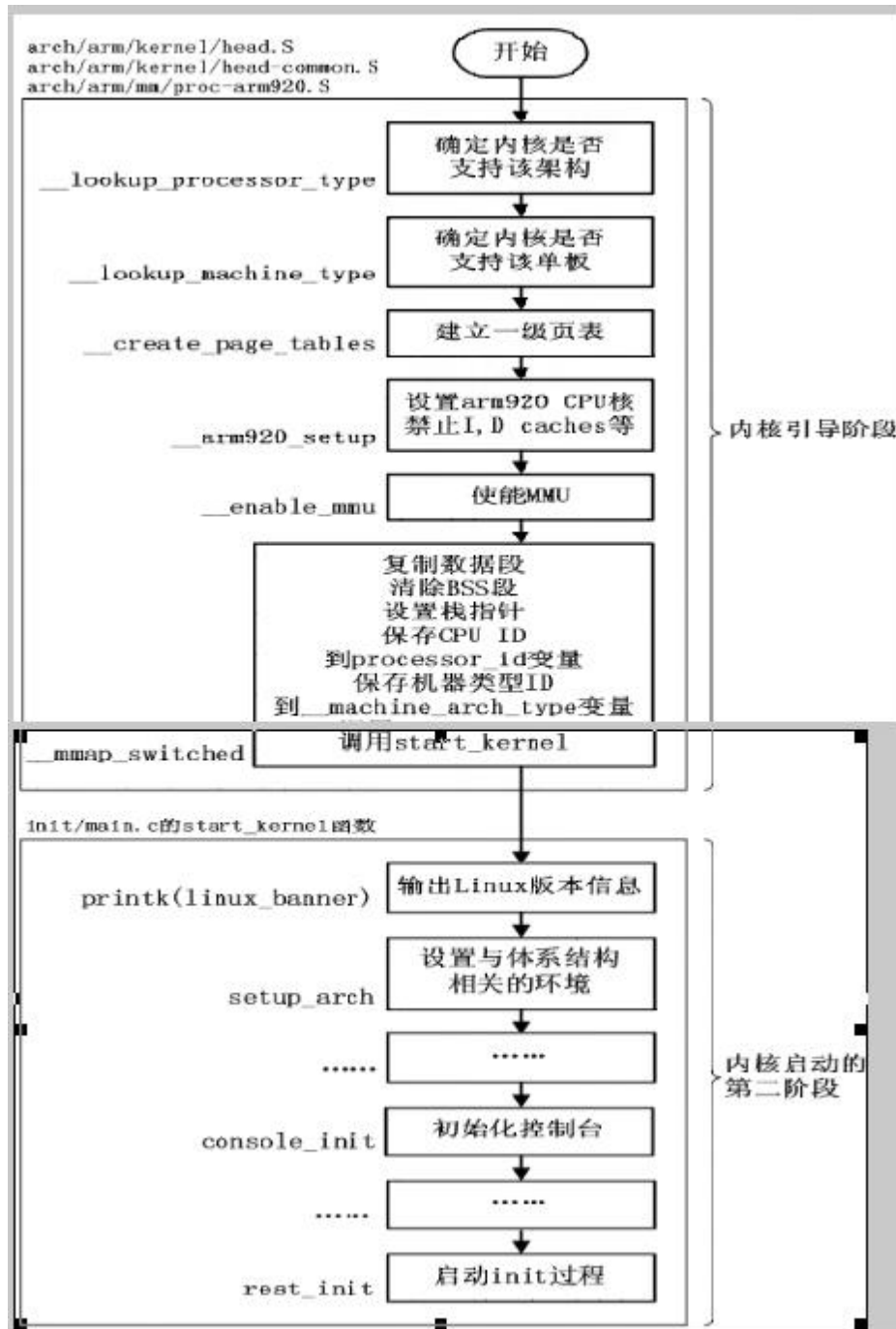
参 3：传进来的参数，就是在这个第一个 C 函数 start_kernel 中处理。

3.分析第一个 C 函数 start_kernel：在 Main.c 文件中

```

00497: asmlinkage void __init start_kernel(void)

```



(1). 进行一系列初始化后，打印“内核信息”：(init\Main.c-->start_kernel)

```
00524:     printk(linux_banner);
```

对应于如下开发板上电启动时内核打印的信息：
linux_banner:在 Version.c 中：

```
00036: /* FIXED STRINGS! Don't touch! */
00037: const char linux_banner[] =
00038:     "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
00039:     LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```



```
Uncompressing Linux.....done, booting the kernel.
Linux version 2.6.22.6 (book@book-desktop) (gcc version 3.4.5) #2 Fri Nov 26 08:54:25 CST 2010
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177
```

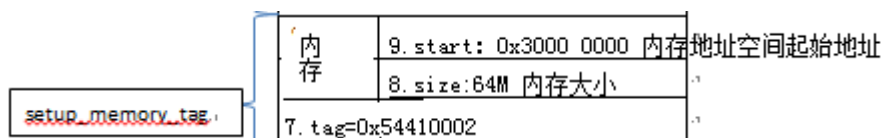
(2) UBOOT 传进来的启动参数。

```
00524:    printk(linux_banner);
00525:    setup_arch(&command_line);
00526:    setup_command_line(command_line);
```

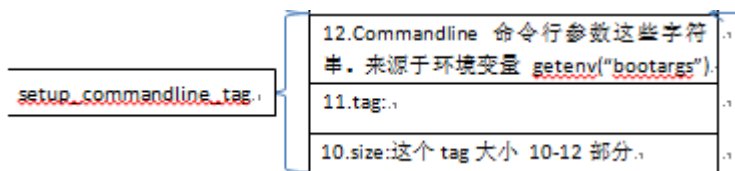
这两个函数就是处理 UBOOT 传进来的启动参数的。

UBOOT 传来的启动参数有：

内存有多大，内存的起始地址。



命令行参数（来源于 UBOOT 设置的“bootargs”环境变量：getenv("bootargs")）



这些 UBOOT 传进来的启动参数，就是在上面这两个函数来处理的。

static int __init customize_machine(void) 在 arch/arm/kernel/Setup.c 中。

static void __init setup_command_line(char *command_line) 在 Main.c 中。

```
00770: void __init setup_arch(char **cmdline_p)
00771: {
00772:     struct tag *tags = (struct tag *)&init_tags;
00773:     struct machine_desc *mdesc;
```

machine_desc UBOOT 传进来的 ID 可以找到如下的结构。里面有很多有用的信息。

```
26 static const struct machine_desc mach_desc_s3c2440 \
27     __used \
28     __attribute__((__section__(".arch.info.init"))) = { \
29     .nr = MACH_TYPE_S3C2440, \
30     .name = "SMDK2440", \
31     /* Maintainer: Ben Dooks <ben@fluff.org> */ \
32     .phys_io = S3C2410_PA_UART, \
33     .io_pg_offst = (((u32)S3C24XX_PA_UART) >> 18) & 0xfffc, \
34     .boot_params = S3C2410_SDRAM_PA + 0x100, \
35     \
36     .init_irq = s3c24xx_init_irq, \
37     .map_io = smdk2440_map_io, \
38     .init_machine = smdk2440_machine_init, \
39     .timer = &s3c24xx_timer, \
40 };
```

为物理地址加上 0x100，就是 0x30000100

```

00782: |
00783:     if (mdesc->boot_params)
00784:         tags = phys_to_virt(mdesc->boot_params);
00785:
100ask24x0.c (board\100ask24x0): gd->bd->bi_boot_params = 0x30000100;

```

0x3000 0100 就是存放启动参数的地址。

```

00801:     parse_tags(tags);

```

解析 tags。

可以看到 tags 上面就是将那些内核的标记 tag 一个个取出来。取出来后在这里解析它们。

(3) 解析命令行参数。

```

00811:     parse_cmdline(cmdline_p, from);

```

解析命令行参数。

这个命令行首先有一个默认的命令行：

```

00774:     char *from = default_command_line;

```

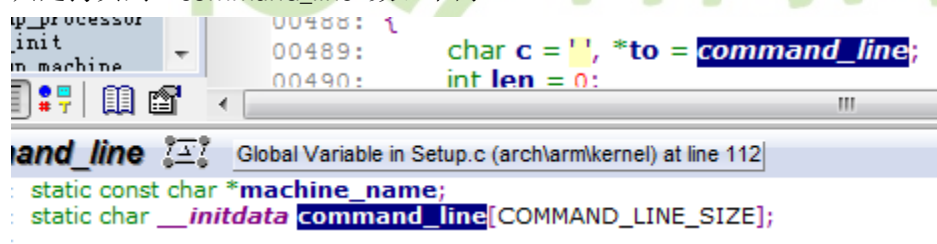
若 UBOOT 没有传入“命令行参数”(就是 getenv("bootargs")所对应的环境变量的参数)时，则内核启动时使用“默认的命令行参数”。如果有传进来命令行参数，则解析下命令行：

```

00811:     parse_cmdline(cmdline_p, from);

```

解析过程在：static void __init parse_cmdline(char **cmdline_p, char *from)函数中只是拷贝到：command_line 数组中而已。



```

00488: {
00489:     char c = '\0', *to = command_line;
00490:     int len = 0;

```

Global Variable in Setup.c (arch\arm\kernel) at line 112

```

static const char *machine_name;
static char __initdata command_line[COMMAND_LINE_SIZE];

```

(3) 最终的目的就是“挂载根文件系统” ----> “应用程序”。

```

00426: static void noinline __init_refok rest_init(void)
00427:     __releases(kernel_lock)
00428: {
00429:     int pid;
00430:
00431:     kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
00432:     numa_default_policy();
00433:     pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
00434:     kthreadd_task = find_task_by_pid(pid);
00435:     unlock_kernel();
00436:
00437:     /*
00438:      * The boot idle thread must execute schedule()
00439:      * at least one to get things moving:
00440:      */
00441:     preempt_enable_no_resched();
00442:     schedule();
00443:     preempt_disable();
00444:
00445:     /* Call into cpu_idle with preempt disabled */
00446:     cpu_idle();
00447: } ? end rest_init ?

```

kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);创建内核线程。暂且认为它是调用 kernel_init 这个函数。这个函数中又有一个：prepare_namespace()，它其中又有一个：mount_root();挂接根文件系统。

整个过程的函数包含：内核启动流程（arch/arm/kernel/head.S）

start_kernel-->

setup_arch 解析 UBOOT 传进来的启动参数

setup_command_line 解析 UBOOT 传进来的启动参数

rest_init-->

 kernel_init-->

 prepare_namespace-->

 mount_root 挂接根文件系统(识别根文件系统)

(假设挂接好了根文件系统)init_post-->

sys_open((const char __user *) "/dev/console", O_RDWR, 0) 打开/dev/console

再执行应用程序 run_init_process("/sbin/init");

run_init_process("/etc/init");

run_init_process("/bin/init");

run_init_process("/bin/sh");

缩进表示调用关系，prepare_namespace 中执行完了 挂接根文件系统 后，会执行 init_post 函数，

在这个函数中会打开 /dev/console 和 执行应用程序。

5. 命令行参数：

```
bootargs=ninitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
bootargc=ninitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0
setup_arch
```

setup_command_line 解析 UBOOT 传进来的启动参数

这里只是将 命令行参数 记录下来。

mount_root 挂接根文件系统，但具体是挂接到哪个根文件系统上去？

root=/dev/mtdblock3 表示根文件系统放在 第 4 个分区 上面。在函数 prepare_namespace 肯定要确定你要挂接哪个根文件系统。以这个函数为入口点来分析这些参数如何处理。

函数“prepare_namespace”在“init\Do_mounts.c”中

ROOT_DEV 从名字可见是“根文件设备”

saved_root_name 在哪里定义。

```
00027: static char __initdata saved_root_name[64];
00211: static int __init root_dev_setup(char *line)
00212: {
00213:     strcpy(saved_root_name, line, sizeof(saved_root_name));
00214:     return 1;
00215: }
00216:
00217: __setup("root=", root_dev_setup);
```

__setup (“root=”, root_dev_setup): 其中 __setup 是一个宏。

大概意思是发现在命令行参数： bootargc=ninitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0

中的 root= 时，就以这个 root= 来找到 “root_dev_setup”这个函数。然后调用这个函数。这个函数将：

/dev/mtdblock3 init=/linuxrc console=ttySAC0 保存到

“strcpy(saved_root_name, line, sizeof(saved_root_name));”中的变量 saved_root_name 中。

这个变量是个数组。

```
static int __init root_dev_setup(char *line)
{
    strcpy(saved_root_name, line, sizeof(saved_root_name));
    return 1;
}
```

__setup (“root=”, root_dev_setup);

__setup 这也是个宏，这个宏也是定义一个结构体。经过分析 UBOOT 和内核可知，这个宏是个 结构体 。结构体里面有 “root= ”

和 root_dev_setup 函数指针（结构体中有 “名字” 和 “函数指针”）。

搜索 “__setup”:

```
Init.h (include\linux): static struct obs_kernel_param __setup_##unique_id \
Init.h (include\linux): = { __setup_str_##unique_id, fn, early } \
Init.h (include\linux): #define __setup_null_param(str, unique_id) \
Init.h (include\linux): __setup_param(str, unique_id, NULL, 0) \
Init.h (include\linux): #define __setup(str, fn) \
Init.h (include\linux): __setup_param(str, fn, fn, 0) \
Init.h (include\linux): __setup_null_param(str, __LINE__) \
Init.h (include\linux): /* NOTE: fn is as per module_param, not __setup! Emits warning if fn \
Init.h (include\linux): __setup_param(str, fn, fn, 1) \
Init.h (include\linux): #define __setup_param(str, unique_id, fn) /* nothing */
```

会有很多，则在这个页面用一般搜索 define 就可缩小范围。

```
00170:
00171: #define __setup(str, fn) \
00172:     __setup_param(str, fn, fn, 0)
00173:
```

再找其中的 __setup_param 如何定义。

```
00160: #define __setup_param(str, unique_id, fn, early) \
00161:     static char __setup_str_##unique_id[] __initdata = str; \
00162:     static struct obs_kernel_param __setup_##unique_id \
00163:         __attribute__((__section__(".init.setup"))) \
00164:         __attribute__((aligned(sizeof(long)))) \
00165:         = { __setup_str_##unique_id, fn, early } \
00166:
00167:
```

展开: __setup("root=", root_dev_setup);

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

```
#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 }
```

展开__setup 就相当于定义了 static char 字符串 和 static struct obs_kernel_param 结构体。它有一个段属性是 “.init.setup”。段属性强制放到 .init.setup 段。

里面的内容是:

一个字符串: static char __setup_str_##unique_id[] __initdata = str;

一个函数: fn <--> root_dev_setup

一个 early(先不管)

可知宏 __setup 定义的结构体 static struct obs_kernel_param 中定义了三个成员: 一个名字“root=”, 一个是函数 root_dev_setup, 一个是 early

这个结构体中的 段属性强制将其放一个段.init.setup 中, 这个段在链接脚本中。

```

312 I setup start = .;
313 *(.init.setup)
314 setup end = .;

```

这些特殊的结构体，会放在 `__setup_start =` 和 `__setup_end =` 里面。搜索下它们两者是被谁使用的。这样就知道这些命令行是如何使用的了。

在 Main.c 中：

```

00190: static int __init obsolete_checksetup(char *line)
00191: {
00192:     struct obs_kernel_param *p;
00193:     int had_early_param = 0;
00194:
00195:     p = __setup_start;
00196:     do {
00197:         int n = strlen(p->str);
00198:         if (!strcmp(line, p->str, n)) {
00199:             if (p->early) {
00200:                 /* Already done in parse_early_param?
00201:                  * (Needs exact match on param part).
00202:                  * Keep iterating, as we can have early
00203:                  * params and __setups of same names 8( */
00204:                 if (line[n] == '\0' || line[n] == '=')
00205:                     had_early_param = 1;
00206:             } else if (!p->setup_func) {
00207:                 printk(KERN_WARNING "Parameter %s is obsolete,"
00208:                     " ignored\n", p->str);
00209:                 return 1;
00210:             } else if (p->setup_func(line + n))
00211:                 return 1;
00212:             }
00213:             p++;
00214:         } while (p < __setup_end);
00215:
00216:         return had_early_param;
00217:     } ? end obsolete_checksetup ?

00450: static int __init do_early_param(char *param, char *val)
00451: {
00452:     struct obs_kernel_param *p;
00453:
00454:     for (p = __setup_start; p < __setup_end; p++) {
00455:         if (p->early && strcmp(param, p->str) == 0) {
00456:             if (p->setup_func(val) != 0)
00457:                 printk(KERN_WARNING
00458:                     "Malformed early option '%s'\n", param);
00459:             }
00460:         }
00461:         /* We accept everything at this stage. */
00462:         return 0;
00463:     }

```

`do_early_param` 从 `p = __setup_start` 到 `p < __setup_end`; 里面调用函数。

调用那些用 `early` 来标注的函数。我们这里：

```

#define __setup(str, fn)
__setup_param(str, fn, 0)

```


可见 early 为 0.则显然我们的: "root="和 root_dev_setup 并不在这里调用。(不在 do_early_param, 早期的参数初始化)可见__setup_start 和__setup_end 在“do_early_param”函数中用不着。

那么看另一个函数“ obsolete_checksetup”, 从上面的代码截图中可知, 它里面也用到了“__setup_start”各“__setup_end”。

do_early_param 又被如下的函数调用:

```
00466: void __init parse_early_param(void)
00467: {
00468:     static __initdata int done = 0;
00469:     static __initdata char tmp_cmdline[COMMAND_LINE_SIZE];
00470:
00471:     if (done)
00472:         return;
00473:
00474:     /* All fall through to do_early_param. */
00475:     strcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
00476:     parse_args("early options", tmp_cmdline, NULL, 0, do_early_param);
00477:     done = 1;
00478: }
```

内核启动流程:

arch/arm/kernel/head.S

start_kernel

setup_arch //解析 UBOOT 传入的启动参数

setup_command_line //解析 UBOOT 传入的启动参数

parse_early_param

do_early_param

从__setup_start 到__setup_end, 调用 early 函数

unknown_bootoption

obsolete_checksetup

从__setup_start 到__setup_end, 调用非 early 函数

rest_init

kernel_init

prepare_namespace

mount_root //挂接根文件系统

init_post

//执行应用程序

“early”“非 early”是:

__setup("root=",root_dev_setup)

#define __setup(str, fn)

__setup_param(str,fn,fn,0)中参数"0".

从代码里知道, 这里 early 这个成员为 0.则没“do_early_param” 和“parse_early_param”,

则调用的是“非 early 函数”：unknow_bootoption

```
__setup("root=", root_dev_setup);
#define __setup(str, fn) \
__setup_param(str, fn, fn, 0)
=====
=====
```

由上的分析可知：“mount_root”挂接到哪个分区上。是由“命令行参数”指定的。

UBOOT 传进来的命令行参数：bootargc=ninitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0

这些参数一开始是保存在一个字符串中，最后被“do_early_param”或“obsolete_checksetup”——来分析它

们。

这些函数如何分析？是内核里面有这样一些代码：

```
static int __init root_dev_setup(char *line)
{
    strcpy(saved_root_name, line, sizeof(saved_root_name));
    return 1;
}
```

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

对于不同的“root=”这样的字符串，如等于“root=/dev/mtdblock3”，它有一个处理函数“root_dev_setup”。

这个("root=", root_dev_setup);字符串和函数被定义在一个结构体中：

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

```
#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 }
```

这个结构体被加了一个段属性为“.init.setup”，则这些很多这样的结构体，被内核脚本vmlinux.lds

放到一块地方：

```
312 __setup_start = .;
313 *(.init.setup)
314 __setup_end = .;
```

当调用这些结构体（UBOOT 传来的参数）时，就从“__setup_start”一直搜到“__setup_end”。

现在是 root=/dev/mtdblock3，我们说过在 FLASH 中没有分区表。那这个分区 mtdblock3

体现在写死的

代码。和 UBOOT 一样：“bootloader|参数|内核|文件系统”在代码中写死。也是用这个分区，在代码里也要写死。

启动内核时，会打印出这些“分区信息”。

```
Creating 4 MTD partitions on "NAND 256MiB 3,3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
```

在源代码中搜索下分区名字：“bootloader”.看哪个文件比较像。

```
root@book-desktop:~/work/system/linux-2.6.22.6# grep "\"bootloader\"" * -nR
arch/blackfin/mach-bf537/boards/generic_board.c:252:         .name = "bootloader",
arch/blackfin/mach-bf537/boards/cm_bf537.c:52:             .name = "bootloader",
arch/blackfin/mach-bf537/boards/pnav10.c:237:         .name = "bootloader",
arch/blackfin/mach-bf537/boards/stamp.c:286:         .name = "bootloader",
arch/blackfin/mach-bf561/boards/cm_bf561.c:51:         .name = "bootloader",
arch/blackfin/mach-bf533/boards/ezkit.c:84:         .name = "bootloader",
arch/blackfin/mach-bf533/boards/cm_bf533.c:50:         .name = "bootloader",
arch/blackfin/mach-bf533/boards/stamp.c:107:         .name = "bootloader",
arch/ppc/syslib/m8xx_setup.c:70:         .name = "bootloader",
arch/arm/mach-davinci/board-evm.c:42:         .name = "bootloader",
arch/arm/mach-omap2/board-h4.c:83:         .name = "bootloader",
arch/arm/plat-s3c24xx/common-smdk.c:120:         .name = "bootloader",
arch/arm/mach-sa1100/assabet.c:110:         .name = "bootloader",
arch/arm/mach-sa1100/assabet.c:131:         .name = "bootloader",
```

```
00118: static struct mtd_partition smdk_default_nand_part[] = {
00119:     [0] = {
00120:         .name = "bootloader",
00121:         .size = 0x00040000,
00122:         .offset = 0,
00123:     },
00124:     [1] = {
00125:         .name = "params",
00126:         .offset = MTDPART_OFS_APPEND,
00127:         .size = 0x00020000,
00128:     },
00129:     [2] = {
00130:         .name = "kernel",
00131:         .offset = MTDPART_OFS_APPEND,
00132:         .size = 0x00200000,
00133:     },
00134:     [3] = {
00135:         .name = "root",
00136:         .offset = MTDPART_OFS_APPEND,
00137:         .size = MTDPART_SIZ_FULL,
00138:     }
00139: };
```

MTDPART_OFS_APPEND 这个 offset 意思是紧接着上面一个分区的意思。

```
asmlinkage long sys_poll(struct pollfd __user *ufds, unsigned int nfds,
                        long timeout_msecs)
{
    s64 timeout_jiffies;
```

