1. 首先将下载好的linux内核文件解压
2. 打补丁 ptach –p < 补丁文件 ==》 –p 文件的后面可以带数字表示忽略补丁文件的目录 ：--- linux-2.6.22.6/arch/arm/configs/s3c2410\_defconfig 2007-08-31 14:21:01.000000000 +0800 –p1 忽略linux-2.6.22.6/ –p2 忽略linux-2.6.22.6/arch/
3. 配置： a：make menuconfig b：使用默认的配置 C:使用厂家提供的配置

b:

用find –name “\*defconfig\*” 找到默认配置的目录i

在arch/arm/configs 找到相应的跟板子相关的配置

Make s3c2410\_defconfig ==》 结果保存在.config

Make menuconfig 读取.config

C:

Cp config\_厂家 .config

Make menuconfig

1. 编译 make UImage

Make s3c2410\_defconfig 失败解决方法：

是由于我的系统的make工具太新，make的旧版规则已经无法兼容新版。 1在makefile中将416行代码 config %config: scripts\_basic outputmakefile FORCE 改为 %config: scripts\_basic outputmakefile FORCE 2在makefile中将1449行代码 / %/: prepare scripts FORCE 改为 %/: prepare scripts FORCE

Make menuconfig 失败解决方法：

sudo apt-get install libncurs\*

20200623：

配置生成.Config:

假如有配置项：CONFIG\_DM9000（肯定是一个宏） 搜索：grep “CONFIG\_DM9000” -nwR

1源码 CONFIG\_DM9000

2. Makefile (子目录) drivers/net/Makefile:

3 .include/config/auto.conf

4 .include/linux/autoconf.h (自动由.Config:生成）

CONFIG\_DM9000这个宏在此文件中定义为1

CONFIG\_DM9000=y 或者m 或者其他的在哪里体现

20200625：

内核子目录makefile:

vi drivers/net/Makefile

Obj-y += xx.o 编译进内核

Obj-m += yyy.o 编译成一个可加载的模块 .ko

#obj-$(CONFIG\_DM9000) += dm9000.o

#obj-$(CONFIG\_DM9000) += dm9ks.o

会根据CONFIG\_DM9000/ CONFIG\_DM9000 是y 或者m

CONFIG\_DM9000/ CONFIG\_DM9000 怎末来？根据.include/config/auto.conf

生成.config后

Make UImage

自动生成：

.include/config/auto.conf （被顶层makefile包含）

.include/linux/autoconf.h (被源码使用)

#obj-$(CONFIG\_DM9000) += dm9000.o 会根据生成的CONFIG\_DM9000 将dm9000.c编译进内核或者编译成一个模块

20200626 makefile 分析

假如有两个文件：编译进内核

Obj-y += a.o b.o

假如两个要组成一个模块：

Obj-m += ab.o

ab-objs = a.o b.o

最后被链接成ab.ko

1.子目录下的makefile obj-y += obj-m +=

2.顶层目录下没有Image,而是位于arch/arm/Makefile 。所以肯定被包含在顶层的makefile中

uImage: vmlinux uImage （是一个头 + 真正的内核 ） vuImage 依赖vmlinux

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

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

在arch 下的makefile

head-y := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init\_task.o // MMUEXT为空就是head.o

init-y := init/

init-y := $(patsubst %/, %/built-in.o, $(init-y)) //是makefile下的函数 表示

init/ built-in.o

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/ fs/ built-in.o ipc/ built-in.o security/ built-in.o crypto/ built-in.o block/ built-in.o

libs-y := lib/

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

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

libs-y := $(libs-y1) $(libs-y2) // lib/ lib.a lib/ built-in.o

drivers-y := drivers/ sound/

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

net-y ：=net//built-in.o

vmlinux-all := $(vmlinux-init) $(vmlinux-main)

vmlinux-lds := arch/$(ARCH)/kernel/vmlinux.lds

查看文件怎末链接成最终的文件可以编译来看

删除 vmlinux

rm vmlinux

make uImage v=1 //更详细的展示过程

分析makefile工程要明白：

1.第一个文件：arch/arm/kernel/head.s

2链接脚本：文件是怎末排布的arch/arm/kernel/ vmlinux.lds

内核启动：

1 处理uboot传入的参数

1.0是否支持cpu

1.1判断是否支持这个单板 uboot启动时传入的机器id

1.2创建页表\_\_create\_page\_tables 因为链接文件中的地址并不是真实的内存的地址 启动mmu

1.3使能mmu

1.4 跳到start\_kernel 第一个函数 处理uboot传入的启动参数

内核的启动流程：

start\_kernel

setup\_arch(&command\_line);//解析uboot传入的参数

setup\_command\_line(command\_line); //解析uboot传入的参数

parse\_early\_param();

do\_early\_param// \_\_setup\_start; 到 \_\_setup\_end调用early函数

unknown\_bootoption

obsolete\_checksetup//\_\_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，根据命令行参数

init\_post();//执行应用程序

如何分析挂接根文件系统，根据命令行参数：

从prepare\_namespace()开始分析

找到Saved\_root\_name()

找到\_\_setup

root=/dev/mtdblock3

static int \_\_init root\_dev\_setup(char \*line)

{

strlcpy(saved\_root\_name, line, sizeof(saved\_root\_name));

return 1;

}

\_\_setup("root=", root\_dev\_setup);//在下面的结构体加上了一些属性

#define \_\_setup(str, fn) \

\_\_setup\_param(str, fn, fn, 0)// 0是非early

#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\_used\_\_ \

\_\_attribute\_\_((\_\_section\_\_(".init.setup"))) \

\_\_attribute\_\_((aligned((sizeof(long))))) \

= { \_\_setup\_str\_##unique\_id, fn, early }

\_\_setup\_start = .;

\*(.init.setup)//放在这里面

\_\_setup\_end = .;

root=/dev/mtdblock3 我们知道flash是没有分区的所以我们要认为的自己在地址写死位置：

怎末找？

Boot启动。

找到bootloader

查找 grep "\"bootloader\"" \* -nR

可以找到arch/arm/plat-s3c24xx/common-smdk.c:

最后运行应用程序

编译出来的内核有时会很大，所以加一段自解压代码,将代码解压缩出来。