# Linux的启动流程

Linux的启动流程总共分几个部分

一、u-boot引导

1、开机首先关闭看门狗、设置内存、开启MPLL、启动icache、开启中断等等

2、对板子上的一些初始化，串口、nandflash、norflash等等

3、设置相应参数（TAG）

4、引导内核、将机器ID、环境参数之类的放进寄存器R0 R1 R2中

二、内核部分

1、被启动之后首先要将判断这个板子上的处理器是否支持（支持则继续不支持则卡死）

2、识别好处理器后然后要解析u-boot传递过来的参数（tag）

3、装载驱动程序（nandflash、网卡、。。。。。。）

4、挂接根文件系统

5、启动应用程序

# Linux内核

挂接根文件系统（linux的应用程序都是放在这里）

内核的最终目的是运行应用程序

相分析内核但是我们不知道从哪里下手，那么我们就先看它的Makefile，我们先打开根目录下的Makefile，因为我们编译出来的内核名字是uImage，所以我们先搜索uImage，但是没有这个词，那么我们就打开arch\arm\Makefile，再搜索，就出来了，“

**zImage Image xipImage bootpImage uImage: vmlinux**

**$(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@”**

可以看出uImage是依靠vmlinux的，那么我们在根目录下面的Makefile搜索vmlinux，搜索出“

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

可以看到vmlinux是依凭后面的几个东西而得出来的，第一个我们应该很清楚，是链接文件，我们暂且不看，我们先搜索vmlinux-init，我们就可以搜索出来如下

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

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

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

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

这么一段，我们就发现我们需要的全都在这里，我们都先搜索出来列下来

**init-y := init/**

**drivers-y := drivers/ sound/**

**net-y := net/**

**libs-y := lib/**

**core-y := usr/**

**head-y := arch/arm/kernel/head$(MMUEXT).oarch/arm/kernel/init\_task.o**

**（这个是在arch\arm\Makefile中搜索出来的根目录下的Makefile是没有的）**

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

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

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

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

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

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

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

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

head-y中有个MMUEXT，这个在该文件的前面有定义，意思是如果这个芯片有MMU的话就执行head，没有的话就执行另外一个文件，

以init-y为例，先给init-y一个初值，init/，然后执行

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

这段话的意思在韦东山的书上有详解，这里就简单阐述一下，就是给init-y赋值成 init/built-in.o，就是在init/中找有没有符合init/的字，如果有就替换成init/built-in.o

下面那几句也是如此，

但是那么多的同名文件Makefile是怎么分辨出来的，就是.config的作用，在我们编译之前我们要先编译config文件，确定我们的板子型号，然后再进行编译，.config的作用就是把定义该型号的宏定义出来，这样Makefile就能识别都是什么文件了。

makefile是根据include/config/auto.conf文件来编译的，其实auto.conf就是根据.confog生成的文件，它去掉了.config的注释并加入一些新的配置信息

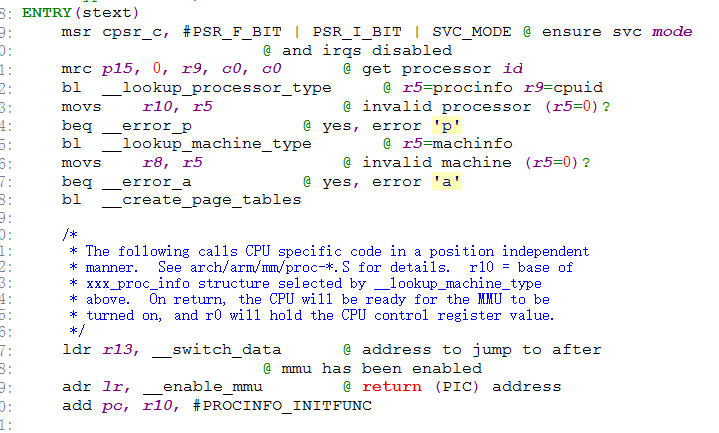
我们先编译一下，可以看到程序的入口是head.S所以我们先从head.S来分析

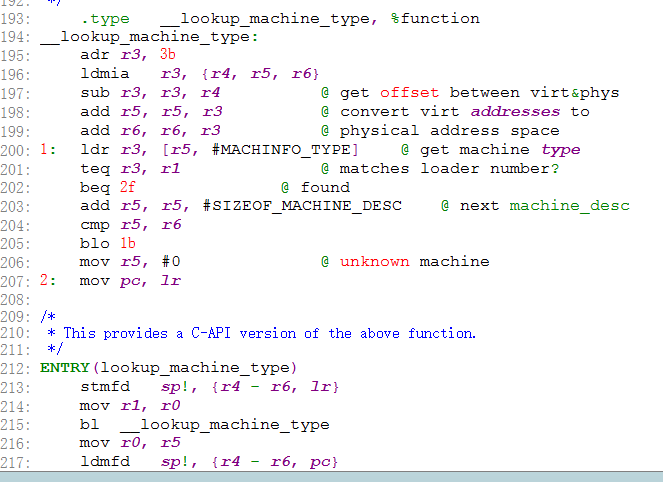
**Linux启动过程：**

我们先建立source insight工程，打开head.S,

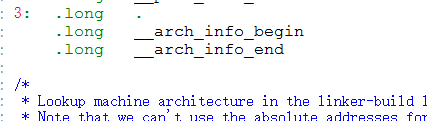
内核首先要处理的是u-boot传入的参数

我们先看一下head.S的前段代码

我们可以看到最先要获取机器ID，这个机器ID是u-boot运行的时候传递给内核的参数，获取完之后我们要先进去\_\_lookup\_processor\_type函数先判断是否支持这个架构，如果不支持就报错。判断完了之后就是判断是否支持这块板子，我们进入这个函数里面看看



首先，要把3b的地址给r3，那我们看看这个3b是什么



就是这段话，先把它的地址给到r3，接下来要把3中“.”给到r4，\_\_arch\_info\_begin给r5，\_\_arch\_info\_end给r6

这里第一句话r3里面的地址是真实的3b物理地址，

而r4里面是3b的当前虚拟地址

接下来是把r3和r4相减算出偏移地址

然后r5和r6分别加上这个偏移地址，这样就算出\_\_arch\_info\_begin和\_\_arch\_info\_end的物理地址来了，因为这时候MMU还没有启动，所以我们不能使用虚拟地址，需要转换为物理地址，那么为什么要用这两个需要虚拟地址的东西呢

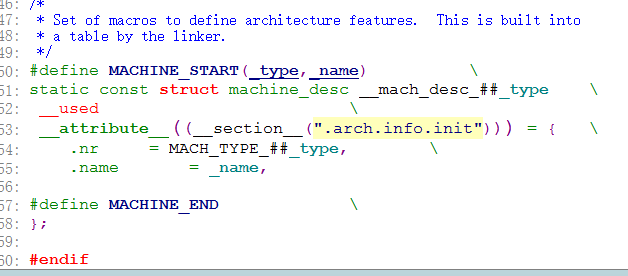
\_\_arch\_info\_begin和\_\_arch\_info\_end在vmlinux.lds文件中有定义

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\{MTF1]Z[YV(U3)O0D)OED81.png

这中间夹了一句话，那么应该就是调用这句话里的东西了

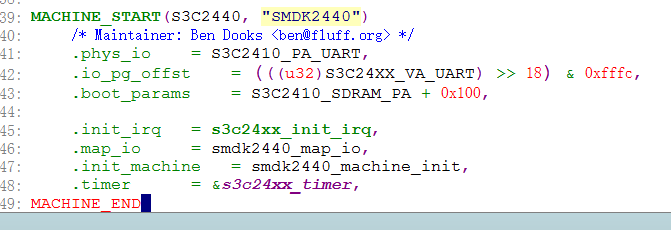
我们在source insight中搜索“.arch.info.init”

发现这个东西除了在lds中还在一个头文件里面（include\asm-arm\mach\arch.h）



这里定义了一个结构体，那么我们看看是哪里调用了这个结构体

我们搜索这个宏，因为我目前用的是2440所以查看关于2440的c文件所以这个宏在arch/arm/mach-s3c2440/mach-smdk2440.c



我们把这个宏展开

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,

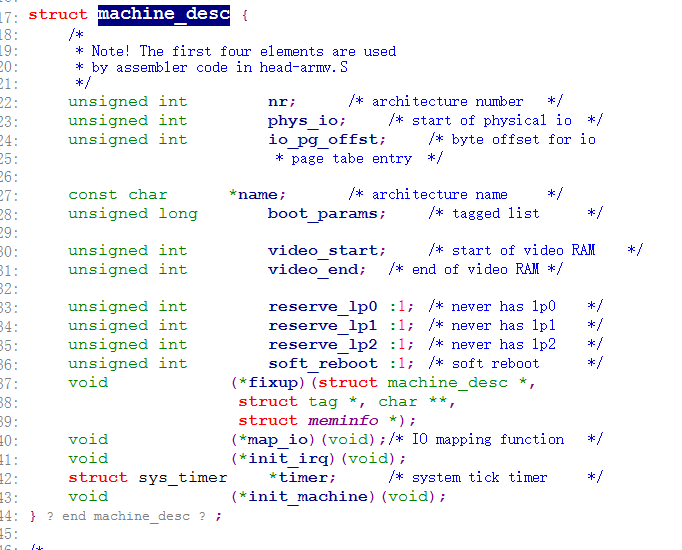
.init\_machine = smdk2440\_machine\_init,

.timer = &s3c24xx\_timer,

};

这个结构体中\_\_attribute\_\_((\_\_section\_\_(".arch.info.init")))的意思是把这个结构体的属性强制转换为.arch.info.init，编译的时候链接文件就会自动地把这段代码放到.arch.info.init段里面

我们看一下machine\_desc结构体



这个结构体就是存储单板信息的结构体了，所有的单板信息都存放在.arch.info.init段里面，linux内核会先获取u-boot传过来的机器ID，然后再与这些结构体一个一个地比较，所以\_\_lookup\_processor\_type函数的处理过程我们就很清晰明了了，

**现在我们就很清楚\_\_lookup\_machine\_type要做的事情了，就是从\_\_arch\_info\_begin与\_\_arch\_info\_end之间取出struct machine\_desc结构体，并将其机器码与r1寄存器值（就是u-boot传进来的机器码）相比较，如果机器码相同的话，就说明内核支持相应的单板，如果不相同就取下一个struct machine\_desc结构体，如果比较完还没有发现匹配项，就说明内核不支持相应的单板，出错返回。**

\_\_lookup\_processor\_type下面标着1、2的那些代码就是判断是否支持单板的代码，这样\_\_lookup\_processor\_type工作完毕回到head.S中

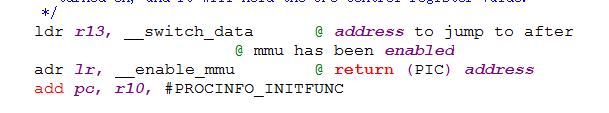
接下来程序运行到“\_\_create\_page\_tables”函数，这个函数是创建页表

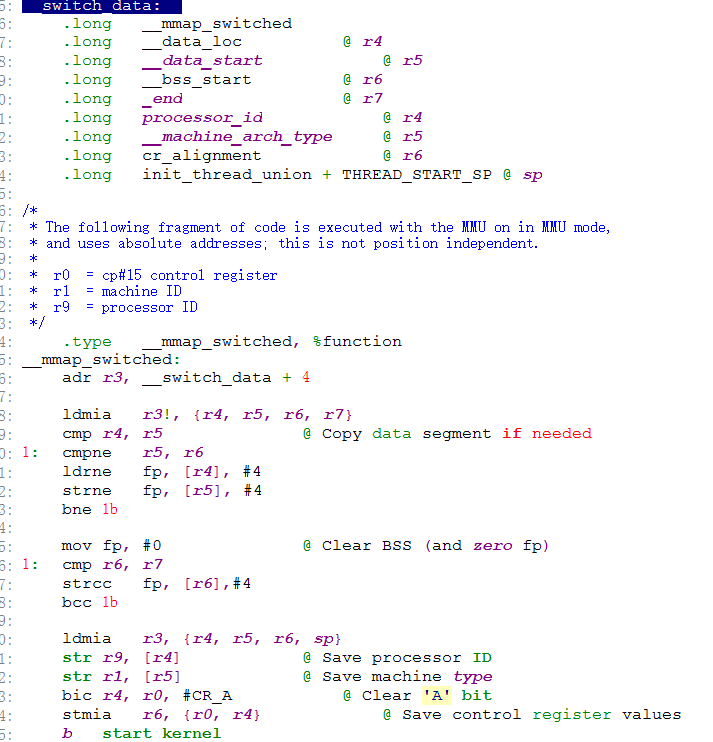
因为链接文件里面的开始地址是虚拟地址，

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\$P(8H$G2CU8T4YW9TW}U(}T.png

需要开启MMU，这个函数就是为开启MMU做准备

接下来就是使能MMU



这两个函数就是使能MMU，当使能MMU之后会跳到\_\_switch\_data里面去，我们进去看一看

这个就是\_\_switch\_data函数，我们可以看到这里一路走下来就会跳转到start\_kernel函数里面去，这个函数是内核的第一个c函数，u-boot传过来的有机器ID在刚才那几个函数里面处理掉了，但是u-boot还设置了一些启动参数内核到现在还是没有处理，所以应该是在start\_kernel里面处理，我们进去看看

总结一下head.S前部分代码主要都干了什么

1、判断是否支持CPU

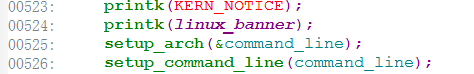
2、判断是否支持单板

3、创建页表

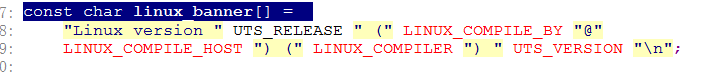
4、使能MMU

5、跳转到start\_kernel函数

进入到start\_kernel函数中首先要做各种初始化，然后就是打印出来内核信息



前面两句话是打印内核信息，



后面两句就开始要处理u-boot传过来的启动参数了

我们进去setup\_arch函数看看

首先它定义了一个machine\_desc结构体，这个就是之前说的那个结构体，它存放着各种参数，其中boot\_params变量是启动参数，在我们之前分析的那个定义单板的那个宏里面有这么一句

boot\_params = S3C2410\_SDRAM\_PA + 0x100 也就是0x30000100，这个就是当时分析u-boot时候说到的起始地址，和u-boot中是一样的，刚才那个宏就是设置各种启动参数的

从那个地址开始，然后一条一条地解析，C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\{[8ZE9`VW6XQOEF7K]Z5_2E.png

传入的命令行参数需要单独处理

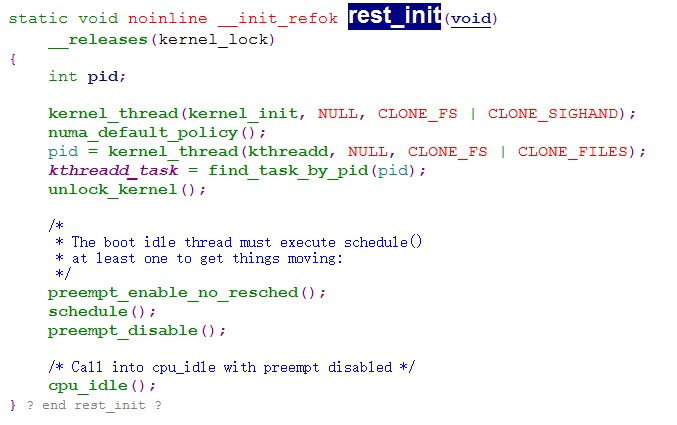
在setup\_arch函数中有两句

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\RMO07A5ZA5R2Z@1$GTOG`Q4.pngC:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\W0AZ]SM9)WEXBW5GRW]6BT6.png

这就表示如果u-boot没有传过来命令参数的话，那就使用默认的命令行参数，如果传进来了就解析

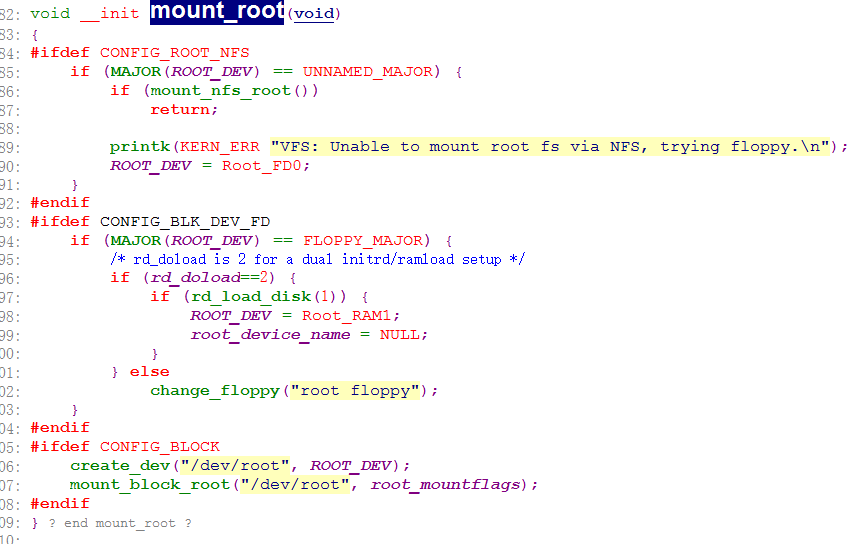
接下来就是各种的初始化，到了最后有个rest\_init函数

我们的内核和u-boot工作了那么多目的就是为了能够运行应用程序，所以rest\_init函数的作用就是挂接根文件系统调用第一个应用程序

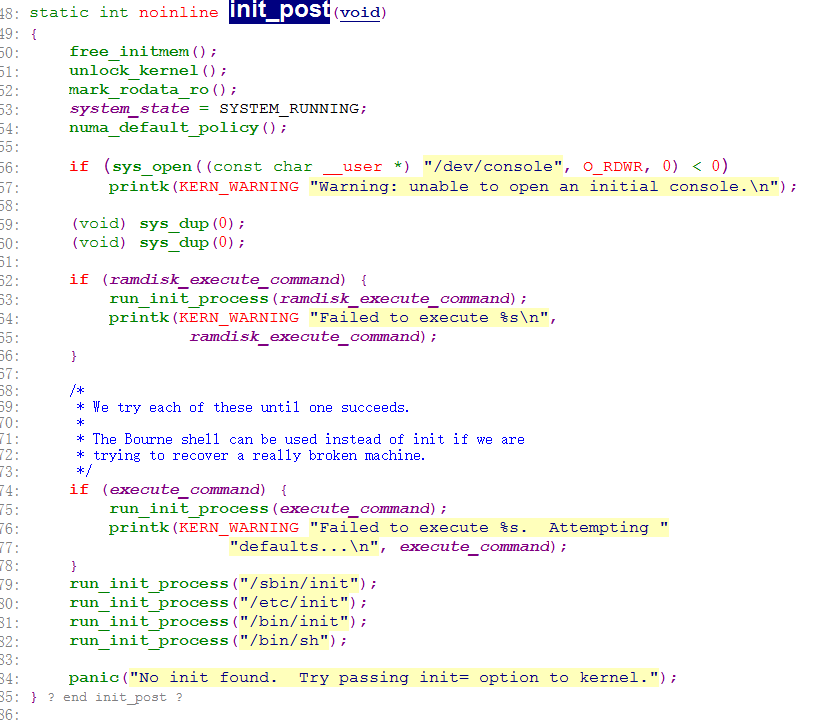
我们点进去看一看

这里面有个kernel\_init，首先这里有个prepare\_namespace函数，点进去

最后面有个mount\_root函数，字面上的意思就是挂载根文系统，所以这个函数的作用就是挂载根文件系统，



我们挂载完根文件系统之后回到kernel\_init函数中，最后有个init\_post函数



从这个函数可以看到，这个函数就是运行第一个应用程序

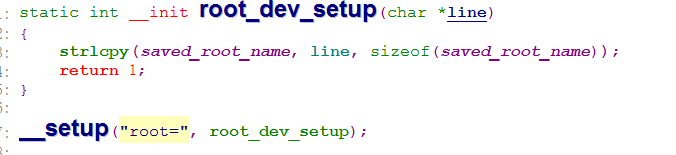
我们现在来看看命令行是怎么解析的

首先内核是挂接根文件系统，但是内核不知道要挂接哪个根文件系统，这就是需要u-boot传过来的命令行参数了

我们先从prepare\_namespace函数入手



这里面有个ROOT\_DEV，我们看看这个，在它的前面有一个saved\_root\_name我们搜索一下



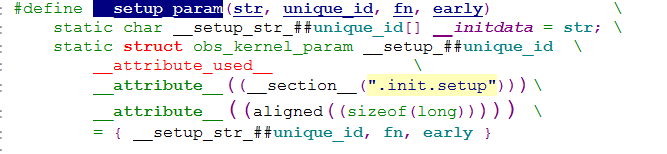
这里我们猜想一下，先找到root=这几个字符然后把后面的句子保存之后开始处理

在这有个\_\_setup，我们点进去看看

在include\linux\init.h中有这个东西的定义

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\~~WV6}8YCKONZJ8MH4Z82VG.png

我们在继续追踪



这里就不展开定义宏了，简单说明一下

这里先把这个结构体的属性定义为.init.setup段里面，我们搜索一下，结果就在vmlinux.lds文件里面，

\_\_setup\_start = .;

\*(.init.setup)

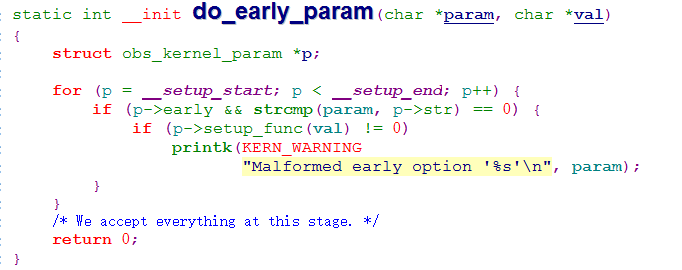
\_\_setup\_end = .;

在\_\_setup\_start和\_\_setup\_end之间，我们搜索一下\_\_setup\_start这样我们就知道是怎么使用命令行的了

搜索到在init\main.c文件obsolete\_checksetup函数和do\_early\_param函数中有所使用

我们再搜索一下do\_early\_param函数，有可能跟setup宏中的early有关

在它的底下有个parse\_early\_param函数，那里调用了do\_early\_param函数，那么parse\_early\_param又是在start\_kernel函数里面调用的，也就是解析param



在这里可以看到p->early也就是定义了early的函数但是我们刚才看到setup那个宏的early是0，所以不是这个函数调用的setup宏，那么就是obsolete\_checksetup函数调用的

我们搜索这个函数，结果发现是被unknown\_bootoption调用，我们再搜索unknown\_bootoption，发现也是在start\_kernel函数中的，所以我们总结一下

parse\_early\_param是调用early参数

unknown\_bootoption是调用非early参数的

它们都被编译在.init.setup段中，然后内核一个一个对比找到之后执行就可以了

我们可以先打开开发板看u-boot打印出来的命令行参数

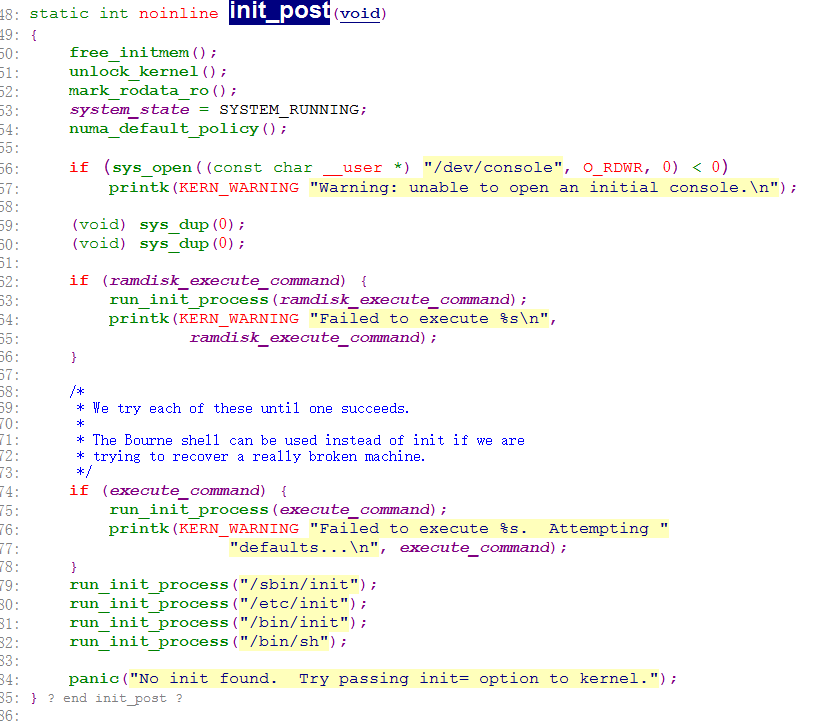
root=/dev/mtdblock3这个就是根文件系统的分区位置（第四分区）

# 根文件系统

当我们内核启动时进入init\_post函数，内核的目的就是执行应用程序，但是在执行应用程序之前需要启动一个init进程，这个进程会根据配置文件确定执行那些应用程序，init进程是后续所有进程的发起者，如果没有这个进程系统是跑步起来的

在我们的嵌入式系统中用的是busybox集成的init程序

我们先看在之前分析到的内核启动应用程序是在init\_post函数中实现运行第一个应用程序



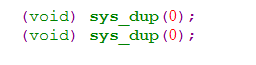
我们先看一下代码，

在linux中最先要打开标准输入、标准输出、标准错误这三个文件，他们的文件描述符分别为0、1、2

标准输入的大致意思为从哪个文件获取数据，标准输出标准错误一个是printf，一个是fprintf

在我们代码中，sys\_open在尝试打开/dev/console文件，如果打开成功那么这个文件就是标准输入文件，文件描述符为0

接下来的两句



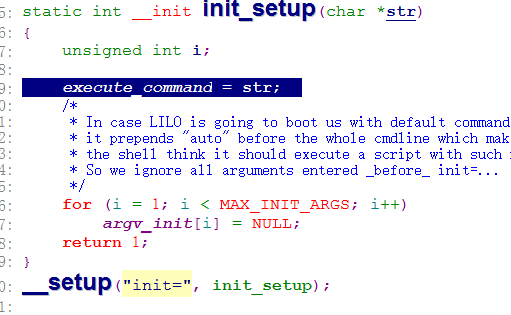
的意思是将文件描述符0分别复制到文件描述符1和2中，这样标准输入、输出、错误共用一个文件

也就说/dev/console文件是作为了控制台

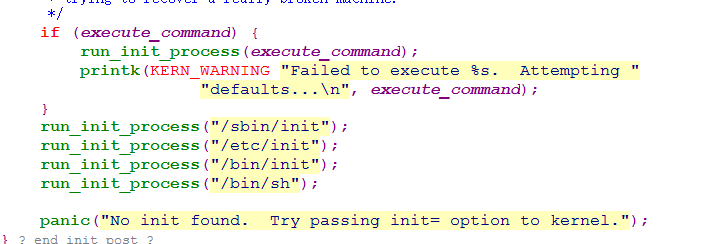
如果没有这个文件linux直接挂死

接着看，这里有个execute\_command

我们搜索看一下

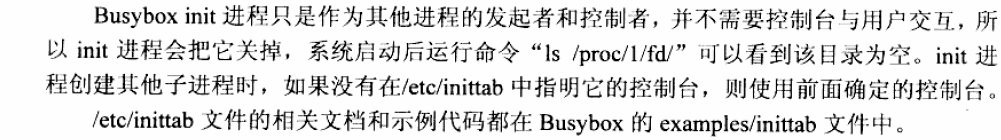


这样我们就清楚了，那句话就表示如果传入的参数有定义init的位置的话就执行那个位置的init，没有就执行下面的，注意这里执行之后就不会返回了，我们看一下后续的代码



这里的这些个init并不是都执行而是只要有一个位置的init存在就执行过去了不再返回

我们的根文件系统没有定义参数所以执行的是/sbin/init



既然是这个init进程决定用什么应用程序，用什么文件那么肯定会有一个配置文件，inittab就是这个配置文件，在busybox中example\inittab中有详细说明

配置文件的格式：<id>:<runlevels>:<action>:<process>

各个参数的说明

# <id>: WARNING: This field has a non-traditional meaning for BusyBox init!

#

# 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. There is no need for this field to

# be unique, although if it isn't you may have strange results. If this

# field is left blank, it is completely ignored. Also note that if

# BusyBox detects that a serial console is in use, then all entries

# containing non-empty id fields will be ignored. BusyBox init does

# nothing with utmp. We don't need no stinkin' utmp.

#

# <runlevels>: The runlevels field is completely ignored.

#

# <action>: Valid actions include: sysinit, respawn, askfirst, wait, once,

# restart, ctrlaltdel, and shutdown.

#

# Note: askfirst acts just like respawn, but before running the specified

# process it displays the line "Please press Enter to activate this

# console." and then waits for the user to press enter before starting

# the specified process.

#

# Note: unrecognised actions (like initdefault) will cause init to emit

# an error message, and then go along with its business.

#

# <process>: Specifies the process to be executed and it's command line.

id表示用的什么控制台

runlevels在busybox中是被忽略掉的

action是执行的时机，也就是怎么执行

process要执行的程序

我们打开busybox创建source insight工程

我们找到init文件打开里面有个init\_main函数就是主函数了

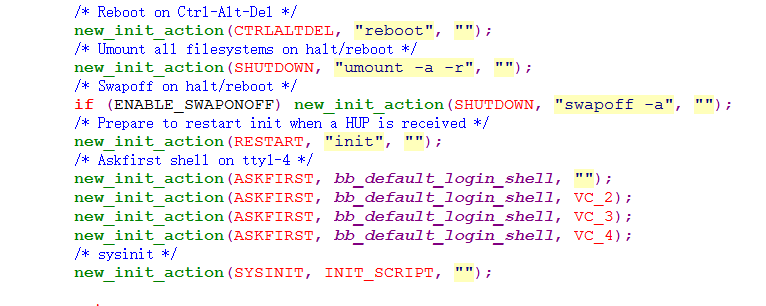
这个函数里面有个parse\_inittab函数，这个函数就是解析inittab配置文件的

我们点开来看看

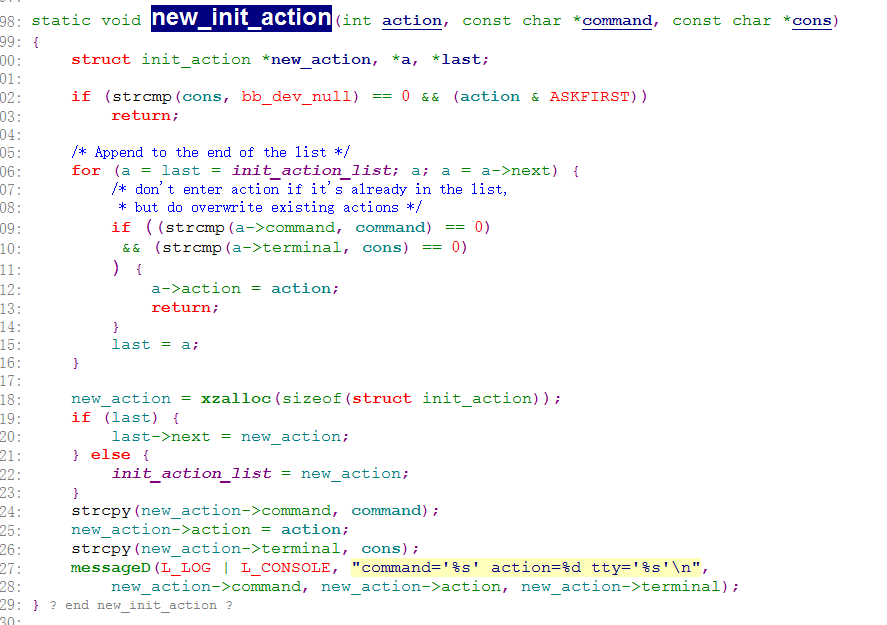
在函数里面前面有

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\O_TW(2NSCSDEJWI{PLIISC7.png

意思是如果有这个文件就打开没有就用底下的默认数据



我们打开new\_init\_action函数看一看

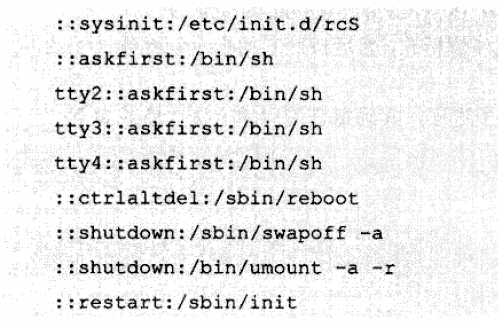


这里定义了一个链表来填充这个配置参数，我们看一下这个函数的形参

int action, const char \*command, const char \*cons

分别对应action process id

这样我们就可以看出来了



默认参数就是这样的

所以我们总结出来一个最小的根文件系统需要如下几个东西：

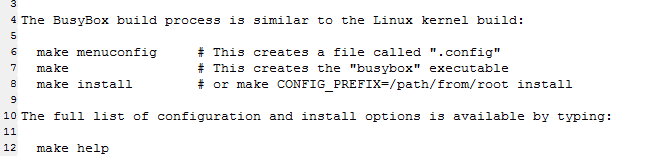
1. 控制台/dev/console或者/dev/null
2. Init进程
3. 配置文件inittab
4. 配置文件中指定的那些应用程序
5. C库

**构建最小的根文件系统：**

既然我们已经知道最小根文件系统都需要什么那么我们自己就构造一个出来

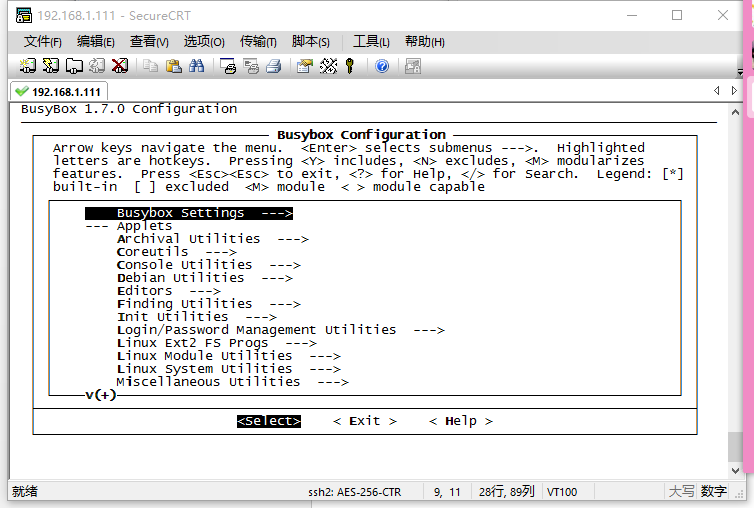
我们要先编译安装busybox

首先我们打开根目录下的INSTALL文件看一下是怎么编译的



在文件的开头有说明

我们先make menuconfig，这样会出来一个选择界面



，我们暂时先用默认值退出来

然后再make，make的时候有可能会有错误，到时候就自己一条一条地百度就可以了，肯定有人遇到过这样的问题

最后一步，这里需要注意，千万不要执行make install，因为这样的话这个busybox就会安装你的PC上了，这样会导致系统崩溃，

我们要用提示后面的那句话make CONFIG\_PREFIX=/path/from/root install

我们先自己找一个位置穿件目录，然后执行make 目录 install就可以了

这样我们cd到那个目录里面去ls看看

![C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\](UEO4TB76SVJR5(`)~P455.png](data:image/png;base64,)

现在只有这四个文件然而我们现在只是满足了最小根文件系统的第二条拥有init进程

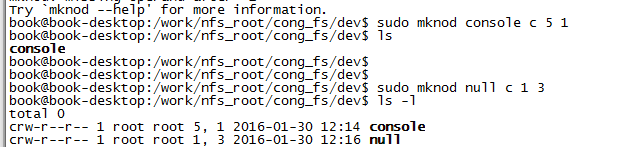
那么我们需要自己创建文件，首先创建控制台

我们先看看PC机上面的控制台的详细信息

ls -l /dev/console /dev/null

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\YQL8CPI8{_I{%LN][AL0S1J.png

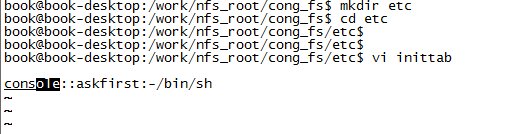
然后我们在自己的文件系统内先创建dev目录然后用mknod创建这两个文件



创建完控制台我们还要创建配置文件inittab

这个文件是在etc目录下，所以我们先创建etc，然后新建inittab文件

我们在配置文件就先暂时放一个最简单的程序吧



然后保存退出

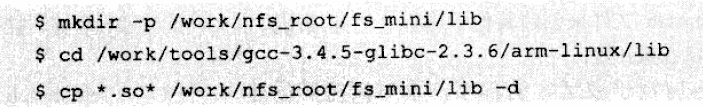
这样还差应用程序和C库了，由于现在要创建最小根文件系统所以我们先不管它，我们来安装C库

这里的c库就是连接gcc编译器上面的C库，所以我们要找到编译器的位置之后查看，我用的是韦东山的Ubuntu，之后更新过gcc编译器，他原来的编译器版本是3.4.5，太老了，我就更新成4.4.3了，这里用哪个库都可以，因为现在用的是busybox-1.7.0版本后面可能会用新的，因为现在是学习阶段所以就先用韦东山老的版本，所以我们就暂时先用他的那个C库以免出现不必要的问题，因为目的只是学习这个过程

我们就切换到那个目录下复制所有.so文件，so是动态链接，就是使用的时候才去调用那个文件，那里面还有一个.a文件那个是静态链接，如果我们用静态的话那么文件就会很大，所以我们用动态的

首先在我们的文件系统中创建lib目录，然后切换到3.4.5的lib目录里面去

类似于这样



最后那个就是复制了，-d就代表建立动态链接

**制作yaffs2：**

这样我们的最小根文件系统就完成了，那么我们要烧写进开发板就要制作yaffs2文件

在韦东山给的光盘文件中有一个system目录下有个yaffs\_source\_util\_larger\_small\_page\_nand.tar.bz2文件

我们解压那个文件，解压后里面有两个文件夹打开yaffs2文件夹

里面有个util文件夹打开它

然后make一下

之后出来一个mkyaffs2image文件

然后将它复制到usr/local/bin/目录下

加上可执行属性“sudo chmod +x mkyaffs2image”就可以了

这样我们就可以执行mkyaffs2image命令制作yaffs2映像文件了

mkyaffs2image 目录名 目录名.yaffs2

**完善根文件系统：**

我们现在只是完成了最小的根文件系统并没有什么功能，比如我们输入ps命令的时候它输出找不到，因为我们还没有创建proc目录

在linux内核中有一个虚拟文件系统proc，如果把它挂接到proc目录,就可以使用ps命令查看内存中发生的一些事情

我们先在板子的跟文件系统创建个proc的目录

然后我们输入命令mount –t proc none /proc

这个命令是把proc挂载到proc目录里面去的命令

这样我们再输入ps就会有反应了

# 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

  778 0          3096 R   ps

输入ls –l /proc/1/fd

lrwx------    1 0        0              64 Jan  1 00:21 0 -> /dev/console

lrwx------    1 0        0              64 Jan  1 00:21 1 -> /dev/console

lrwx------    1 0        0              64 Jan  1 00:21 2 -> /dev/console

但是我们这样太麻烦了，我们就在根文件系统中改好脚本然后放在里面

我们回到linux中去，首先创建proc目录然后在inittab中添加::sysinit:/etc/init.d/rcS

也就是添加一个rcS脚本

我们创建etc/init.d目录然后在该目录下创建rcS文件

在里面写上mount –t proc none /proc

或者mount –a，这个命令的意思是在etc/fstab中获取文件系统种类

那么我们就用这个参数，创建fstab文件

这个文件的格式如下

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\NR@JK[0L[)CB}W%}8L@C}N8.png

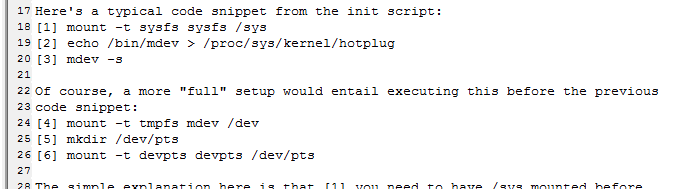
我们就按照这个格式来写就可以了

这下我们再编译就没有问题了

然后我们来创建mdev，所谓mdev是busybox简化的udev得出的

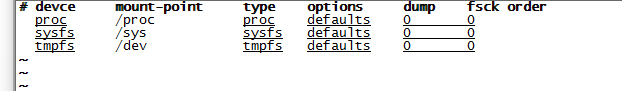
因为如果手工一个一个把设备驱动程序加上是非常费力的，所以linux就退出了udev这个东西，这样他就自动地把需要用的设备文件链接到dev中去了

我们在busybox中搜索一下mdev，搜索出来会有一个txt文件打开，这个文件是对mdev的说明里面有这么一段



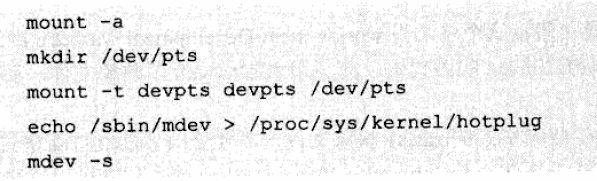
我们看1和4

这两个也是文件系统



我们这么写

然后我们在rcS脚本里面写上



然后制作yaffs2就可以了，这样根文件系统就制作的差不多了

关于如何制作jffs2在韦东山的书上说明的很详细了在此就不再赘述

如果u-boot找不到jffs2的话就要添加一个环境变量，在init=/linuxrc之前加上rootfstype=jffs2强制指定这个文件就好了

# Linux内核移植

## 下载配置内核

移植内核之前我们要把内核下载下来，在[www.kernel.org](http://www.kernel.org)网站下载就可以了

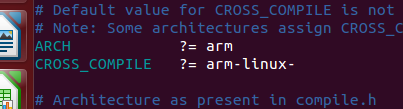
我们下载下来之后要先在linux环境下解压

tar xjf xxx.tar.bz2

解压完了之后cd进去

然后我们在编译之前需要修改一下Makefile让它使用我们的交叉编译工具链

vim Makefile



把这段话改成如图所示的样子

我们编译内核的流程是

使用默认配置

make s3c410\_defconfig

make uImage

这些配置文件在arch/arm/configs/目录下

这里面我们搜索一下是有mini2440\_defconfig和s3c2410\_defconfig这两个文件

由于我们用的u-boot中给内核传递过来的参数是362也就是16A这符合s3c2410\_defconfig中的配置所以我们目前用这个配置文件当然我们也可以修改u-boot传递过来的机器ID来支持mini2440

所以我们make s3c2410\_defconfig

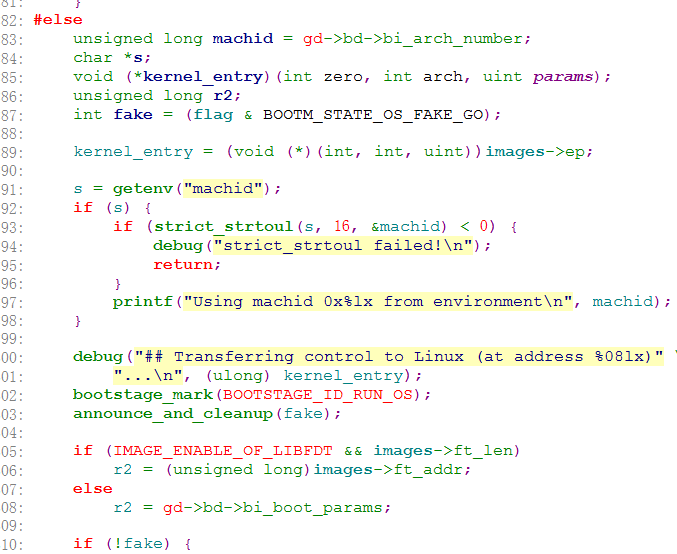
之后把uImage下载到我们的单板中，我这里用的是dnw直接下载到SDram中去了

usb 1 32000000

bootm 32000000

在这里我们看到有一堆乱码，所说有乱码但表示我们的内核还是可以启动的

我们在u-boot里面找一下bootm的c文件然后点开它，里面有一个do\_bootm\_linux函数，然后里面有我们点开jump函数



在这里有这么一段，这里有个machid，这个就是我们的机器ID，我们看到

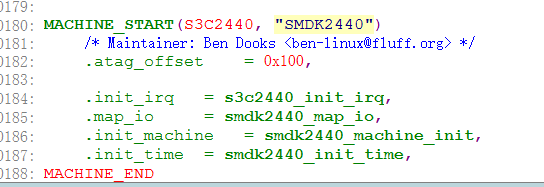
“s = getenv("machid");”这句话，这句话的意思就是从设置的环境变量里面读取machid，也就是说我们可以在u-boot的控制台界面是指机器ID也可以使用u-boot程序以及写好默认的ID，我们看一下这个默认的id是多少

我们看这里的第一句unsigned long machid = gd->bd->bi\_arch\_number;这句是定义machid的，我们看一下bi\_arch\_number的值是多少，我们搜索一下

可以搜索到gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410;我们点开MACH\_TYPE\_SMDK2410看看值是多少

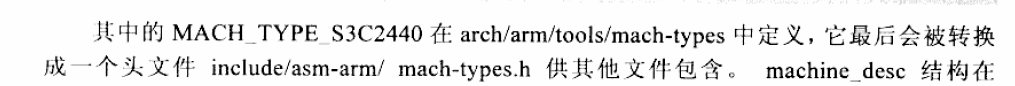
我们可以看到是193，我们在去内核里面看看这个机器ID应该是多少

我们打开内核中对应的芯片文件比如说我们的就是“arch/arm/mach-s3c24xx/mach-smdk2440.c”和“arch/arm/mach-s3c24xx/mach-mini2440.c”在这些文件的最后都会有一个定义的结构体类似下图



这个结构体是临时定义的

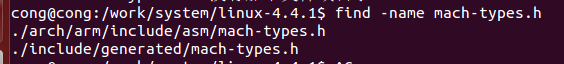
在韦东山的书里面有说，这些宏都是在



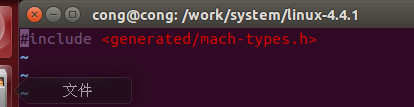
这里已经说的很清楚了，这是一个内核编译自动生成的文件，所以我们要在linux下查找这个文件

我们 find –name mach-types.h

搜索出来之后我们打开它



这里有两个文件，那么我们就先打开书上说的那个文件也就是第一个文件



这里就这么一句话我们看到include的文件刚好是我们搜索的第二个文件，所以我们就直接打开那个文件

我们在里面搜索2440





这里有S3C2440和mini2440

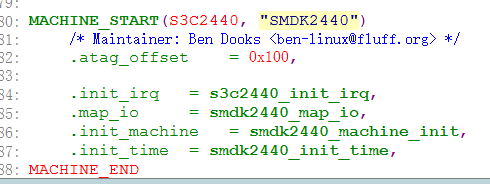
所以我们知道了机器ID，所以我们在u-boot中设置一下机器ID

set machid 16a（362）或者set machid 7cf（1999）

然后我们再bootm，发现还是有乱码，这时候我们想一下是不是波特率没有设置对，所以我们在u-boot中设置一下环境变量

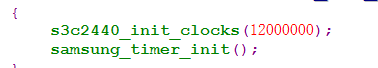
set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 init=/linuxrc

我们在console=ttySAC0后面加上115200，然后save，之后再启动内核

但是我们发现还是乱码，这时候我们就检查一下内核源码是不是哪里初始化出问题了，我们就把那个结构体展开一下看看，我们就发现

里面的init\_time函数中

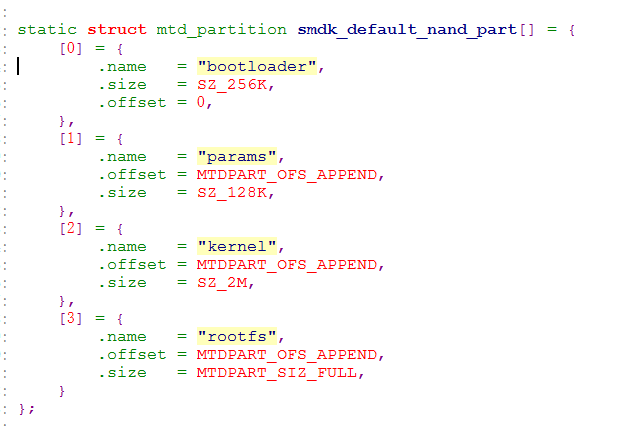
这个函数是设置晶振的，我们开发板的晶振是12M的，但是它设置的是16点多M，这样就可以解释了，由于晶振不同所以串口打印的波特率也出现了异常，那么我们就设置到12M看看，



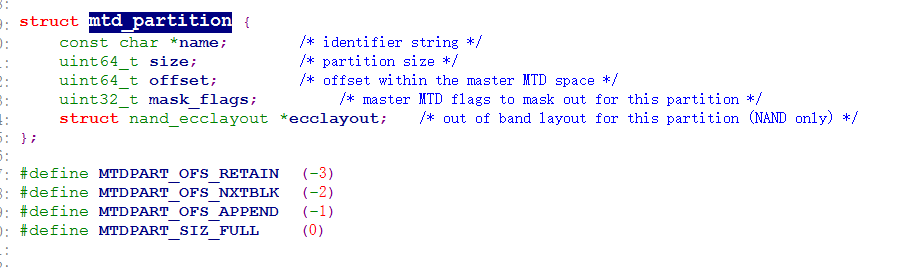
然后再make uImage 烧写，我们这时候就会发现可以打印出来信息并启动了

## 修改内核的分区以及制作最小的根文件系统

我们启动内核的时候观察一下内核的打印信息，在后面几行里面我们可以看到内核的分区，这里4.4.1的内核的分区是8个分区而我们u-boot给分的是4个，我们以前也都是四个，所以我们需要改一下分区，我们可以搜索一下任意一个分区的名字，搜索之后，我们发现有这个位置有一个文件，这个明显是我们三星的系列，所以我们点开它，里面就会发现我们的分区了

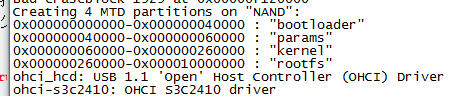
我们的分区是在下面的结构体里面

我们把它改成我们自己想要的分区就好了，然后这里面有个offset，这个就是偏移多少，也就是该分区的位置，我们点开mtd\_partition看看这个结构体有没有定义



在这里我们看到在这个结构体底下有明显定义，我们看这些英文单词，我们可以看到append是紧接着的意思，所以，我们第一个写0 ，接下来都是这个宏，然后到最后一个需要的“其余都是这个分区的”，所以最后一个我们选择最底下的FULL个宏

然后再编译，我们就可以看到内核打印出来我们分区的信息了



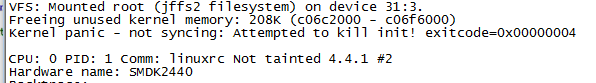
然后我们就要让它挂接根文件系统，我们在busybox网上下载最新的busybox，但是由于我用的交叉编译器是4.4.3版本的，它的glibc库版本与最新的busybox是不匹配的，编译会出错，所以我选择了比较低一点的版本1.22.1版本，我们跟上次制作根文件系统一样首先 make menuconfig，然后在

**Busybox Settings ---> Build Options --->Cross Compiler prefix**

点开然后加上arm-linux-就可以了，其它我们选择默认配置，然后我们make

最后我们新建一个文件夹然后把busybox安装到那个目录下

make CONFIG\_PREFIX=需要存放的目录 install

然后我们按照之前说过的过程做一遍完整的最小根文件系统，在此就不详细说明了，但是这时候内核还是不支持yaffs2的，所以我们就先把它制作成jffs2，然后烧写进去，之后就会看到

这个，这个代表已经挂接了根文件系统，但是init进程不对，被杀死了

这里涉及到一个erbi接口的问题，我们需要在编译配置kernel的时候要选上它，我们再次进入linux4.4.1，make menuconfig

我们**Kernel Features --->Use the ARM EABI to compile the kernel**选中这个选项就可以了，然后重新编译，烧写进去我们就可以运行busybox了

**修改内核支持yaffs2文件系统**

首先我先要说一下，在做到这里的时候我发现linux4.4.1中有些函数变动，自己尝试了很长时间，但是都没有成功把4.4.1做成支持yaffs2文件系统的内核，所以在这里我换成了3.18.26版本的内核，这个内核用之前一样的做法就可以修改到这一步，所以3.18.26版本之前的修改就不说了，照着之前写的做就可以实现了，4.4.1我也会继续修改，所以我们先来说说怎么把3.18.26改成支持yaffs的然后再说明为什么4.4.1不行

首先我们打开3.18.26的内核的fs文件夹，看一下，它是没有yaffs2这个目录的，我们要自己添加，我们先在网上搜索yaffs ，它有个官方网站，它告诉你了要怎么获取yaffs2源码，它是需要在linux中执行git clone git://www.aleph1.co.uk/yaffs2命令，在执行这个命令之前我们要先安装git，系统会有提示怎么安装git，

sudo apt-get git install就可以了，然后执行那个获取命令，这样我们所在的目录下就会有个yaffs的文件夹，我们打开进入yaffs2文件夹，里面有个README-linux文件，打开阅读一下里面告诉了怎么在内核中打补丁，

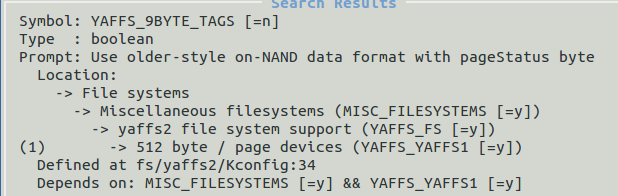
cd yaffs-dir

./patch-ker.sh c m linux-tree

所以，我们先进yaffs2目录下，然后执行./patch-ker.sh c m 内核所在路径就可以了，这样我们再进内核的时候就会多出来一个yaffs2文件夹

然后我们在内核中make menuconfig一下

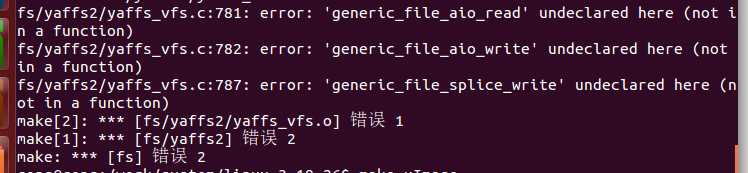
搜索一下yaffs，



可以看到，yaffs2的选项在File systems -> Miscellaneous filesystems (MISC\_FILESYSTEMS [=y])下，我们进去添加一下这个选项就可以了

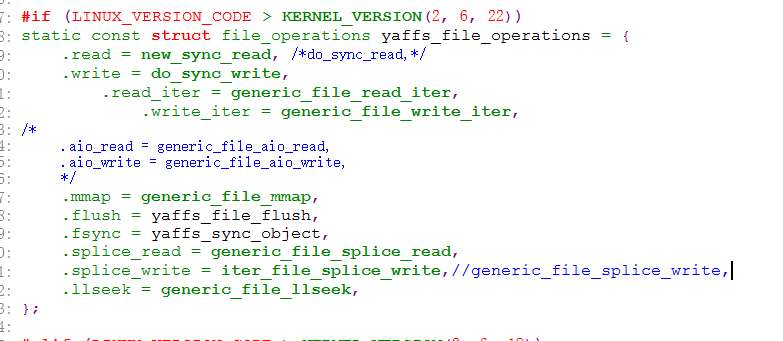
然后我们退出，保存然后我们make uImage

之后会发现如下错误



我们进去这个文件

在新的内核中有很多文件的文件名会被代替或者是改名字了，所以我们需要改动它，我们需要把下面的结构体改成如下



注释掉的是原来的代码

我们也可以搜索一下这个结构体，看看别的文件系统是怎么用的，然后我们再编译就可以了，这样就支持了yaffs2文件系统

我们再来看看为什么4.4.1版本的不行，4.4.1版本的按照上述方法做就可以，然后编译出来一大堆错误，经过修改发现除了上面那几个函数没有之外，连do\_sync\_read,do\_sync\_write，以及nd\_set\_link也是没有的，前面两个函数是可以找到别的函数代替的，但是nd\_set\_link函数就不行了，本来我还找到原来老内核里面的nd\_set\_link函数添加到4.4.1的源码当中，结果发现还是不行，我在网上看到有人说nameidata结构体已经变动了，原来是

struct nameidata {  
    struct path    path;  
    struct qstr    last;  
    struct path    root;  
    unsigned int    flags;  
    int        last\_type;  
    unsigned    depth;  
    char \*saved\_names[MAX\_NESTED\_LINKS + 1];  
  
    /\* Intent data \*/  
    union {  
        struct open\_intent open;  
    } intent;  
}

然而现在是

struct nameidata {  
    struct path    path;  
    struct qstr    last;  
    struct path    root;  
    struct inode    \*inode; /\* path.dentry.d\_inode \*/  
    unsigned int    flags;  
    unsigned    seq, m\_seq;  
    int        last\_type;  
    unsigned    depth;  
    int        total\_link\_count;  
    struct saved {  
        struct path link;  
        void \*cookie;  
        const char \*name;  
        struct inode \*inode;  
        unsigned seq;  
    } \*stack, internal[EMBEDDED\_LEVELS];  
    struct filename    \*name;  
    struct nameidata \*saved;  
    unsigned    root\_seq;  
    int        dfd;  
}

它用到了cookie变量来引用了，所以我也跟网上的那位一样不会了，所以我就选择用3.18.26版的linux内核，但是并不代表4.4.1的内核不可以用，只不过是目前我自己还没弄明白怎么支持yaffs2而已，但是它还是可以支持jffs2或者别的文件系统的。

## Linux内核的裁剪

Linux内核移植的最后一步，裁剪，没什么好说的，就一个思路，看menuconfig中哪个板子是不需要的，如果menuconfig中看的不够具体就直接打开.config直接看就可以了，如果裁剪不到预期大小的话那就只能改分区大小了