# U-Boot

**u-boot的主要目的就是启动内核**

大部分的配置参数都是预先在include/configs/board\_name.h下定义的，因此如果我们要移植我们自己的板子的话，这个文件必不可少，它描述了我们板子的配置情况如CPU型号，RAM大小等

SRCTREE：根目录下，源文件

mkconfig中[ "${BOARD\_NAME}" ] || BOARD\_NAME="$1"的意思是如果BOARD\_NAME没有被定义那么就执行BOARD\_NAME="$1"否则不执行

$#:指令参数个数

在mkconfig中>表示新建一个文件，>>表示将该文件追加进去

脚本中if——fi，fi表示if的结束语句

start.S中主要做了什么工作：

1. 设置CPU模式进入SVC32模式
2. 关闭看门狗、屏蔽中断
3. 设置栈指针
4. 将代码复制到RAM中去
5. 清除bss段
6. 中断异常处理

u-boot第二部分代码是从\lib\_arm\board.c中start\_armboot()函数开始

主要是干了以下事情：

1. cpu, borad, interrupt的初始化，包括cache等，这些都于特定板子的配置有关。
2. 环境变量的初始化
3. 串口、控制台、RAM的初始化
4. 在控制台上实时的显示系统配置等相关参数。
5. 最后进入main\_loop();函数（引导内核以及控制台）

u-boot命令的实现：

命令的实现主要是依靠main\_loop中的run\_commd函数来实现

run\_command的主要流程：

1. 把传进来的指令先对\;解析,分割出一个个命令
2. 提取参数（parse\_line）
3. 查找命令（find\_cmd）
4. 检测其参数
5. 找到匹配的命令后，调用cmd\_tbl\_t->cmd执行

U-boot启动内核：

u-boot有个主函数main\_loop()，它是通过发送命令来实现对内核的启动，启动过程如下：

1、向内核传递参数

2、从nand-flash读出内核放到SDRAM中kernel分区中运行

注意：u-boot的分区有4个：bootloader、param、kernel、root，这四个分区不是硬件分的，而是在u-boot程序中写死的。

上面说到mainloop函数是发送指令来启动内核的，没错，发送的指令为：

**nand read.jffs2 0x30007FC0 kernel** 将0x3007fc0中的数据复制到kernel分区中

**bootm 0x30007fc0** 从该地址启动内核

**注：这里的read.jffs2并不是指定要什么特定的格式，而是用read.jffs2不需要块/页对齐,所以这个kernel的分区大小可以**参数中有两个值需要考虑，一个是加载地址，一个是入口地址

内核是uImage，它主要有头部和内核

头部主要是用于存放一些启动参数的比如说上面说到的加载地址和入口地址，执行命令需要先检测加载地址和内核真正存放的地址是否一样，如果不一样u-boot会把内核移动到加载地址所对应的地址

所以，加载地址不一定是程序中写的，也可以是0x31000000等等，只要它不影响到u-boot所用的空间就可以了（u-boot在SDRAM地址为0x33f4df74-0x34000000）

但为什么程序要写0x30007fc0，是因为当我们的内核正好处于头部指定的加载地址的话，那么就不用uboot的do\_bootm函数来帮我们搬运内核了，这样可以节省启动时间(uImage的头部大小为64K)

bootm一共做了以下工作：

1. 读取头部  
2. 将内核移动到加载地址  
3. 启动内核  
启动内核的关键函数是do\_bootm中**do\_bootm\_linux()**这个函数调用thekernel函数

参数是通过标记列表tag来完成的，在程序中表示为：

**setup\_start\_tag (bd);  
setup\_revision\_tag (&params);  
setup\_memory\_tags (bd);  
setup\_commandline\_tag (bd, commandline);  
setup\_initrd\_tag (bd, images->rd\_start, images->rd\_end);  
setup\_videolfb\_tag ((gd\_t \*) gd);  
setup\_end\_tag (bd);**这些函数是在do\_bootm\_linux中调用传递参数的

怎么传的呢？

内核跟u-boot设置好一个地址然后u-boot把数据保存在那里，然后内核过来取

Tag（标记列表）

存放的地址是在**setup\_start\_tag (bd);** 中params = (struct tag \*) bd->bi\_boot\_params;这句话里bi\_boot\_params的值为0x30000100这句话在board\_init中有所体现

每个标记列表最开始都会有一个头部大小，也就是这个列表的大小，然后是tag，之后是tag中的各种参数

# U-BOOT移植过程笔记（重定位版本）

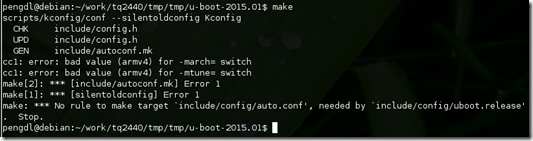
先下载最新u-boot，目前最新的是u-boot-2016.01

1. 先让u-boot下载到2440上有输出

先把u-boot放到虚拟机上编译，在编译之前要先配置处理器的型号

make smdk2410\_defconfig

配置好后运行make，但是这时候出现错误



不要慌，这是后我们在根目录下的Makefile中的前面添加

ARCH=arm

CROSS\_COMPILE=arm-linux-

然后再次编译就能编译通过

接下来把u-boot下载到mini2440但是这时候2440没有什么反应

接下来开始修改2440

1. **添加2440型号的板子**

先在u-boot-2016.01\board\samsung下复制smdk2410文件夹并改名为smdk2440

进入该文件夹，把文件名从2410都改成2440

打开Makefile，把Makefile里的文件从2410改为2440

打开Kconfig把文件内容从2410都改为2440

进入u-boot-2016.01\configs复制smdk2410\_config为smdk2440\_defconfig并打开修改为

CONFIG\_ARM=y

CONFIG\_TARGET\_SMDK2440=y

CONFIG\_SYS\_PROMPT="SMDK2440 # "

# CONFIG\_CMD\_SETEXPR is not set

再进入u-boot-2016.01\include\configs复制smdk2014.h为smdk2440.h

进入u-boot-2016.01\arch\arm\Kconfig中

添加

config TARGET\_SMDK2440

bool "Support smdk2440"

select CPU\_ARM920T

source "board/samsung/smdk2440/Kconfig"

这样就算添加了一个2440的板子

1. **让u-boot在2440中能正确的运行**

现在我们的u-boot只能编译通过，但下载到2440中去的时候是没有任何输出的

现在打开u-boot穿件source insight工程

打开arch\arm\lib\vectors.S文件，\_start就在这个文件里，u-boot.bin就是从这里开始的，它的第一句话是“b reset“然而reset并不在这个文件里，我们就追踪一下，发现是在u-boot-2016.01\arch\arm\cpu\arm920t\start.S中，一路看下来我发现并没有对MPLL进行初始化也就是设置时钟，而是在board\_init\_f中board\_early\_init\_f初始化的但是之前是按照100MHz设置的但是现在的2440只工作在12MHz，这样无法正常工作所以把board\_early\_init\_f中

/\* to reduce PLL lock time, adjust the LOCKTIME register \*/

writel(0xFFFFFF, &clk\_power->locktime);

/\* configure MPLL \*/

writel((M\_MDIV << 12) + (M\_PDIV << 4) + M\_SDIV,

&clk\_power->mpllcon);

注释掉，并在start.S中关闭看门狗之后加入

#define S3C2440\_MPLL\_400MHz ((0X5c<<12)|(0x01<<4)|(0x01))

/\*

\* mask all IRQs by setting all bits in the INTMR - default

\*/

/\* 设置时钟 \*/

ldr r0, =0x4c000014

mov r1, #0x05

str r1, [r0]

mrc p15, 0, r1, c1, c0, 0 /\* 读出控制寄存器 \*/

orr r1, r1, #0xc0000000 /\* 设置为“asynchronous bus mode” \*/

mcr p15, 0, r1, c1, c0, 0 /\* 写入控制寄存器 \*/

/\* MPLLCON = S3C2440\_MPLL\_200MHZ \*/

ldr r0, =0x4c000004

ldr r1, =S3C2440\_MPLL\_400MHz

str r1, [r0]

/\* 启动Icache \*/

mrc p15, 0, r0, c1, c0, 0 @ read control reg

orr r0, r0, #(1<<12)

mcr p15, 0, r0, c1, c0, 0 @ write it back

这时再编译会发现串口打印出来一堆乱码，这表示2440工作了，这时候出现乱码应该是串口方面的问题那么我们就继续追踪代码在board\_init\_f函数中调用了init\_sequence\_f数组里面有一个串口初始化serial\_init，点进去，里面有个get\_current函数，再进里面default\_serial\_console函数，再进，在这个函数所在的c文件（serial\_s3c24x0.c）中有个\_serial\_setbrg函数中里面有个get\_PCLK函数进去该函数中有这么一句”#ifdef CONFIG\_S3C2440”但是这个宏没有被定义，这个代表定义它就支持了2440，我们就在include\configs\smdk2440.h中定义一下就可以了

然后再重新编译，编译出来的u-boot.bin下载到2440中我们就可以看到串口打印出信息了

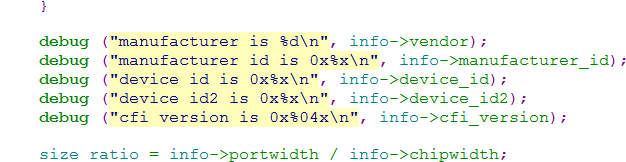
1. **支持norflash**

这时候串口打印出来的信息中flash和nand的大小都是0K，这就代表u-boot现在还不支持这款mini2440上的norflash和nandflash，那么就要修改它

在start.S中有” bl \_main “这么一句，\_main函数在arch\arm\cet0.c中这个函数主要作用是对板子的一些初始化、重定位和引导第二部分代码，主要调用了board\_init\_f（初始化）、relocate\_code（重定位）、board\_init\_r（引导第二部分代码）我们进入board\_init\_r函数看看，进去之后有个init\_sequence\_r数组中有个initr\_flash，我们进去，里面有个flash\_init，点进去，这里有这么一个判断C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\HMI5WP}B`T`51}C9043TG{7.png

If中的函数是老的识别norflash的方法底下那个是新的，这句话代表了如果老方法识别不出来，那就使用新的方法，既然我们都没识别出来那么我们直接就进入新的方法的这个函数中

我们进去之后能看到有很多debug函数打印出很多调试信息



那么我们就打开debug，在这个c文件中写入

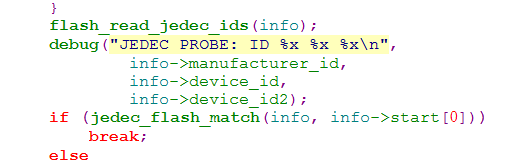
#define \_DEBUG 1

#define DEBUG 1

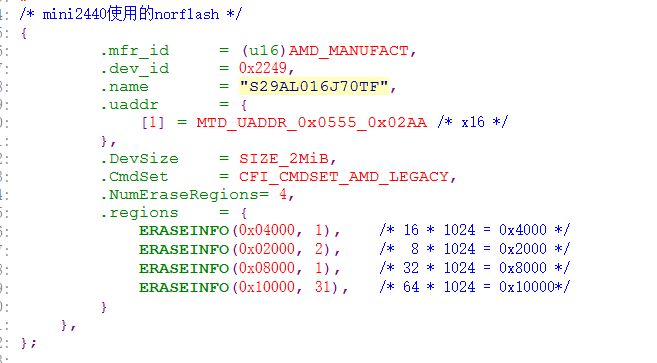
这时候再编译烧写我们就能看到很多打印信息

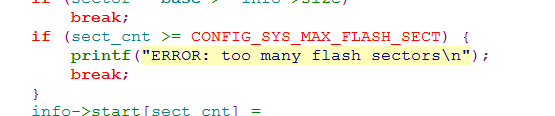
中间打印的ID号 1 2249

这时候我们返回去看flash\_detect\_legacy函数看看为什么读不出来，该函数中



这里是打印出ID的地方，我们进入jedec\_flash\_match函数进入jedec\_table数组，在这个数组里我们可以看到这里是写norflash型号的地方，经过查找这里根本没有这块板子的norflash，所以我们需要自己手动添加，这块板子的norflash型号是S29AL016J，我们一边根据这个型号的数据手册一边写代码

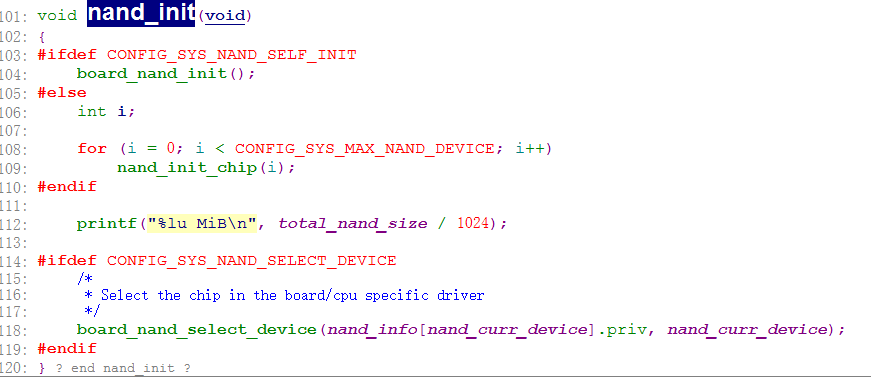
在这个数组末尾加上

这个段就OK了，这样重新编译就可以支持norflash了，别忘了把debug调试信息关掉，但是烧写进去看打印信息有这么一行“Flash: ERROR: too many flash sectors”说我们的扇区太多了，那么我们就在source insight中搜索这句话，

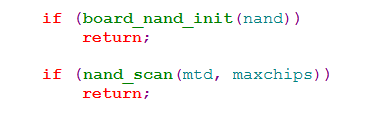
发现这里有一个宏，我们点进去看一下在smdk2440.h中有定义，它只有19，而我们的早就超过了这个数，所以把它改成一个比我们扇区大的数就大功告成了，之后的u-boot就完全支持mini2440板子上的norflash了

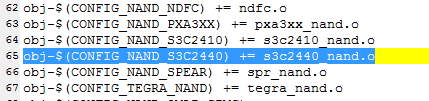
1. **支持nandflash**

当我们打开开发板，它打印出来的信息上面写着nand:0b这就代表u-boot目前还不支持这个板子上的nandflash，所以我们看board\_r.c文件中的board\_init\_r函数中init\_sequence\_r数组里有个initr\_nand函数，我们点进去有个nand\_init，再进，可以看到这个函数

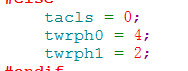


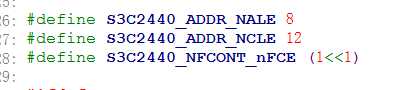
它执行的是nand\_init\_chip(i)函数，“CONFIG\_SYS\_NAND\_SELF\_INIT”发现是未定义所以不执行board\_nand\_init追踪进去我们可以看到



这两个函数，第一个是对nandflash进行初始化的，第二个是分辨出nandflash的型号以及大小的我们先进去第一个函数看看，进去后可以看到这个函数是对nand进行初始化，设置时序以及发送命令，这个函数只是针对2410的，所以我们要先复制s3c2410.c命名为s3c2440.c，再进入drivers\mtd\nand\Makefile中添加这个文件

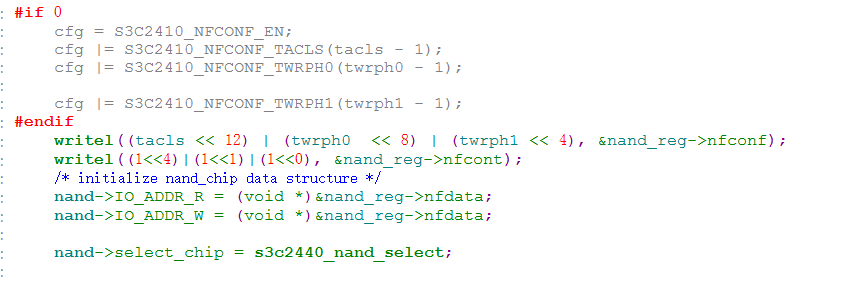
我们再进去2440的这个文件，进入刚才的函数，这个函数现在是针对2410的所以我们要对其进行改动先把时序参数改为如下



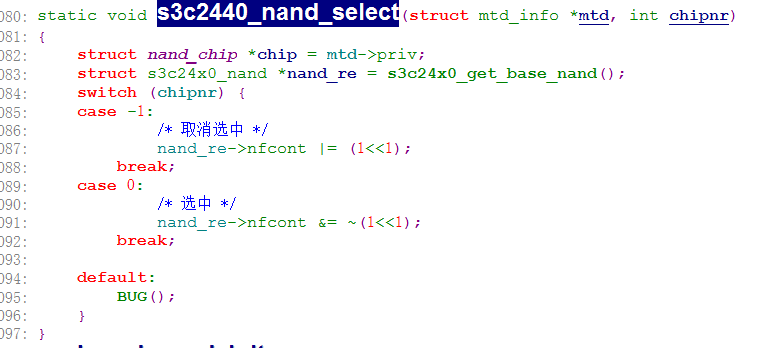
添加宏定义

这个宏是2440中ALE和CLE的偏移地址，和2410的略微有所不同

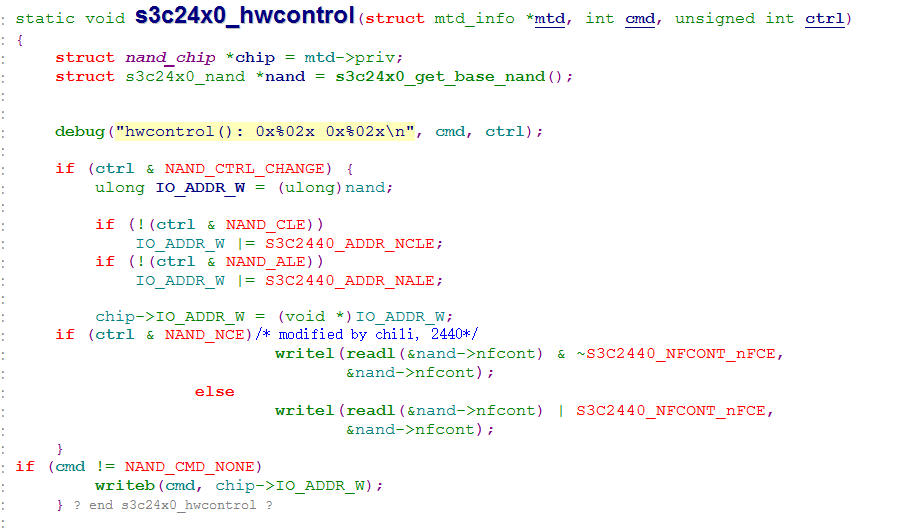
我们继续看，下面一段是对soc的nand控制器进行初始化，然而2410和2440的nand寄存器是不一样的所以底下的一段我们都不要了，改成我们自己的改成如下



这样我们才设置对nand控制器。这里有个“s3c2440\_nand\_select”函数，这个函数是我们自己写的，（其实可写可不写，其实对这种移植方法这个函数是没起到作用的）

s3c2440\_nand\_select函数内容如下

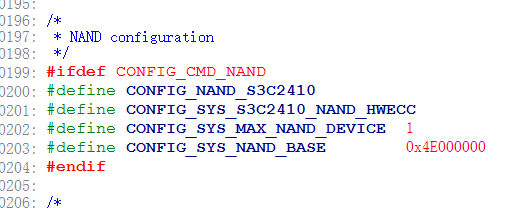
还有一个函数我们需要改动

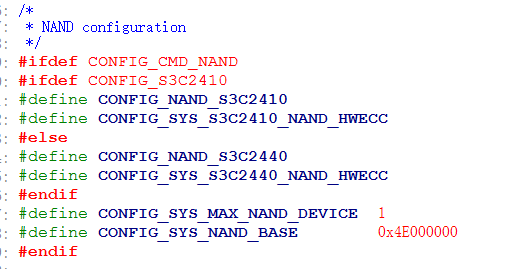


这个函数是能否启动nandflash的关键函数，改动完这个函数之后我们就大致改动完毕了

我们接着要把配置文件定义2440的nand，我们先看smdk2440.h，这个配置文件有一个“#define CONFIG\_CMD\_NAND”

底下还有一句



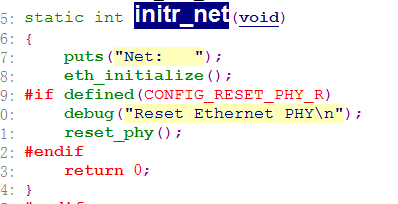
我们要改成

这样再编译就能支持我们的nandflash了

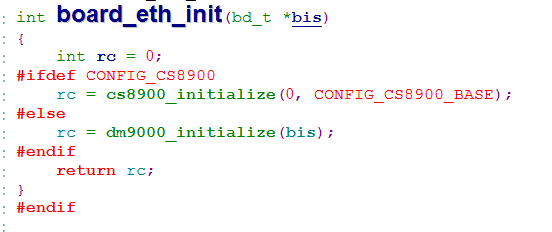
1. **支持dm9000网卡**

我们看到u-boot的打印信息中的Net:的型号所说是打印出来了但不是我们网卡的型号，这时候就需要我们自己改动网卡

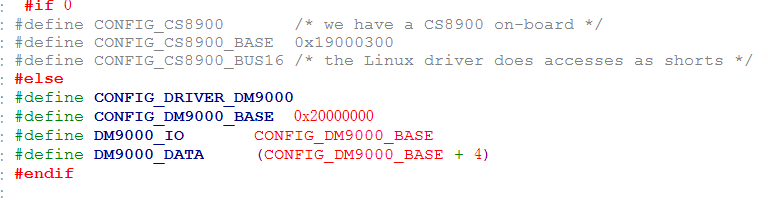
首先，我们在source insight搜索一下Net:，搜索出来是在common\board\_r.c中initr\_net函数里



这里有个eth\_initialize()函数进去里面eth\_common\_init再进board\_eth\_init继续追踪这个函数里面只有cs8900的初始化而没有dm9000的，所以，我们要自己添加如下图



然后我们在smdk2440.h中修改如下定义



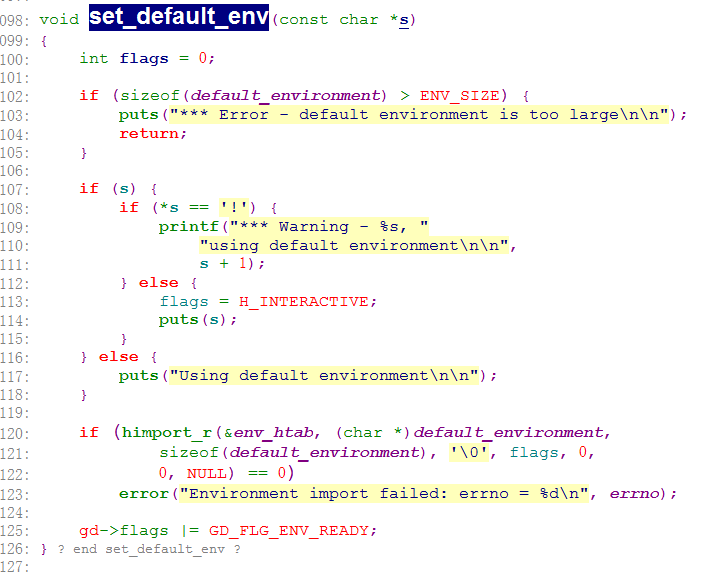
这样就完成了对dm9000的支持

1. **对u-boot的裁剪以及配置环境变量**

现在，我们的u-boot已经对全部的硬件支持了，接下来就是启动内核，在这之前需要先配置好环境参数，在u-boot的打印信息中我们可以看到

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\CL100O03T%{D_6L6IMN@1@9.png

我们就搜索这句话，结果是在common\env\_common.c中set\_default\_env函数里面



我们进default\_environment（默认的环境变量）里面看看首先我们看bootargs这个变量在我们的配置文件里面没有定义，那么我们就定义它

C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\VQ`U9DXS1(S5C9JDUWGPB4B.png

目前这个变量是干什么的我还不太清楚，估计是分区用的

接下来定义CONFIG\_BOOTCOMMAND

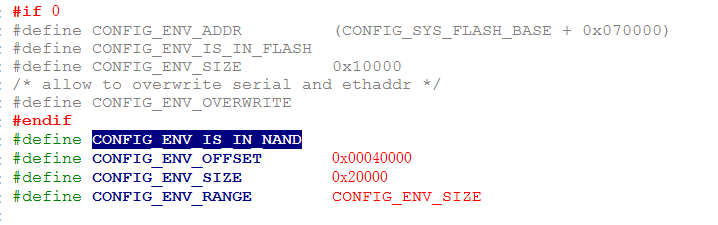
C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\VJKZS3Z__YE5KYODNB4ZQ}C.png

这是读取内核的命令

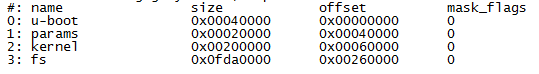
C:\Users\smart\AppData\Roaming\Tencent\Users\1043503402\QQ\WinTemp\RichOle\_M825Q[E7)`S)X`}B]VTL}R.png

我们看到nandflash里面的环境变量需要定义CONFIG\_ENV\_IS\_IN\_NAND，所以我们就定义它就可以了

因为我们是在nandflash里面分区和放置环境变量的下载内核到nandflash里去，所以要定义它，原来的配置文件中给的是在norflash里面的所以我们应该这么改



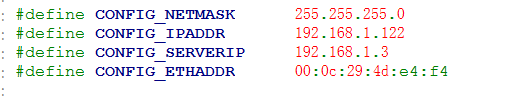
这里的地址是根据内核分区来决定的



所以环境变量的地址是0x40000大小是0x20000

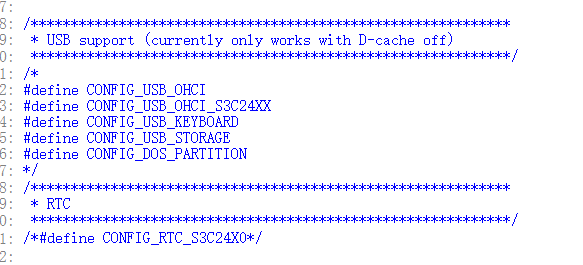
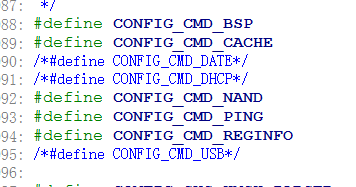
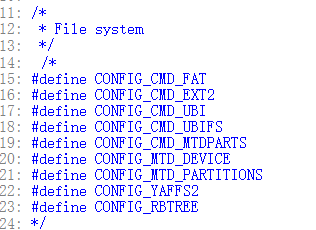
接下来我们需要配置IP地址

修改如下宏成为如下值

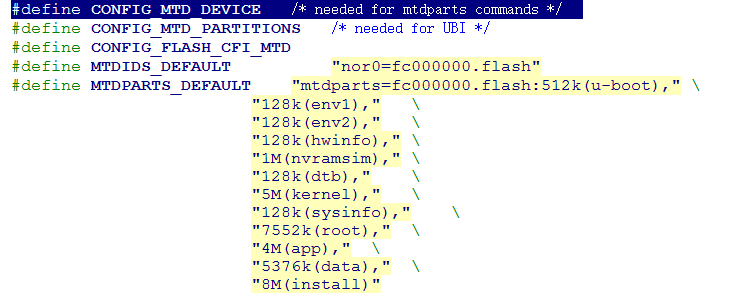


这时候我们的u-boot就开始有读秒了接下来我们要对它进行裁剪把没用的命令都给删掉

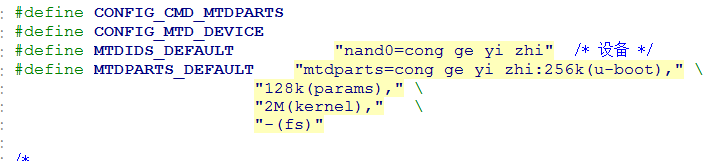
修改配置文件

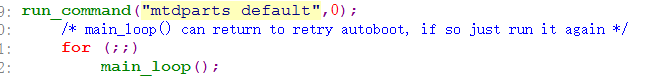


把配置文件里面的这个变量都给注释掉，然后编译就会发现u-boot已经在200K到300K之间了，接下来我们就让分区在u-boot中打印出来，我们搜索mtdparts命令然后我们随便点开一个头文件看看他们是怎么写的，



我们就可以仿照他们的来定义，定义成



这样就差不多了，最后一步我们把board\_r.c中在main\_loop前面加上

run\_command命令就可以了，然后我们预先把文件系统和内核下载到nandflash中，我们再把最新编译好的u-boot下载到norflash中，重启开发板u-boot就可以正常运行提取内核了

1. **支持yaffs以及制作补丁**

我们的u-boot是不支持烧写yaffs的，因为它没有关于识别yaffs的代码，我们需要将它补上由于字数太多直接上个链接

<http://wenku.baidu.com/link?url=BeZPoTfM5fPZfgIEI9KbgLGMSQ0rFlFp3KU4hb-g_9BxJUoxcyFsJ1D0YOOURpr2xYJ5u9guc90SdcsRKR2n0LQnGBApTdedejfjHyrqHvm>

或者直接百度搜索u-boot-2015支持yaffs烧写就可以了

打补丁：

先进入修改好的u-boot内，先make distclean一下，然后返回上一层目录cd ..

输入指令diff -urN 未曾修改的u-boot文件夹名字 已经修改的u-boot文件夹名字 > 补丁文件名.patch

之后再进入未曾修改的u-boot里，执行

patch -p1(这里的数字1是要看补丁文件里文件层数代表忽略几个文件夹，这里只要忽略一个就可以了) < ../补丁文件名.patch

这样就可以了，在这要说明一下有可能出现无限地询问你是否打补丁的时候不要理会一路回车就OK了

到此，u-boot的移植工作就全部完成了！