|  |
| --- |
| Elvonzion.WX |
| Porting&BootLoader |
| 基于ARM/S3C2440/MINI2440开发板 |
|  |
| **elvonzion@gmail.com** |
| **2012/8/12** |

|  |
| --- |
| 移植os以及bootloader的过程。 |

**目 录**

[1 系统移植 1](#_Toc388299253)

[1.1 移植前的准备 1](#_Toc388299254)

[1.1.1 开发板准备 1](#_Toc388299255)

[1.1.2 工作计划准备 1](#_Toc388299256)

[1.2 移植linux 3](#_Toc388299257)

[1.2.1 获取Linux内核源码 3](#_Toc388299258)

[1.2.2 编译测试 3](#_Toc388299259)

[1.2.3 添加平台相关文件 3](#_Toc388299260)

[1.2.4 测试MTD 5](#_Toc388299261)

[1.2.5 添加yaffs2 8](#_Toc388299262)

[1.3 制作文件系统 10](#_Toc388299263)

[1.3.1 创建基础根文件系统 10](#_Toc388299264)

[1.3.2 使用Busybox添加系统基本命令 11](#_Toc388299265)

[1.3.3 添加系统配置文件 13](#_Toc388299266)

[1.3.4 验证文件系统 14](#_Toc388299267)

[1.3.5 yaffs2镜像制作 15](#_Toc388299268)

[1.3.6 RAMFS镜像制作 15](#_Toc388299269)

[1.3.7 RAMDISK镜像制作 16](#_Toc388299270)

[1.3.8 烧录镜像到nand 16](#_Toc388299271)

[1.4 移植u-boot 17](#_Toc388299272)

[1.4.1 获取uboot源码 17](#_Toc388299273)

[1.4.2 编译环境配置 17](#_Toc388299274)

[1.4.3 增加板级配置表 17](#_Toc388299275)

[1.4.4 生成配置文件 17](#_Toc388299276)

[1.4.5 编译源码 18](#_Toc388299277)

[1.4.6 添加SPL功能 20](#_Toc388299278)

[1.4.7 添加nand驱动 20](#_Toc388299279)

[1.4.8 修正编译错误以及警告 21](#_Toc388299280)

[1.4.9 配置初始化clock 22](#_Toc388299281)

[1.4.10 从nand启动 23](#_Toc388299282)

[1.4.11 优化绝对寻址 24](#_Toc388299283)

[1.4.12 逻辑地址布局规划 25](#_Toc388299284)

[1.5 系统烧录 27](#_Toc388299285)

[1.5.1 下载文件到内存 27](#_Toc388299286)

[1.5.2 烧录文件到nand 27](#_Toc388299287)

[1.5.3 制作uImage 28](#_Toc388299288)

[1.5.4 串口下载命令支持 29](#_Toc388299289)

[1.5.5 网络命令支持 30](#_Toc388299290)

[1.5.6 nand命令支持 30](#_Toc388299291)

[1.5.7 YAFFS命令支持 33](#_Toc388299292)

[1.5.8 配置mtd分区 38](#_Toc388299293)

[1.5.9 配置设备树 38](#_Toc388299294)

[1.6 移植Cross编译工具 39](#_Toc388299295)

[1.7 常见移植问题 40](#_Toc388299296)

[1.7.1 Makefile语法兼容问题 40](#_Toc388299297)

[1.7.2 mtd编译错误 41](#_Toc388299298)

[1.7.3 yaffs2编译错误 41](#_Toc388299299)

[1.7.4 yaffs2的xattr支持问题 41](#_Toc388299300)

[1.7.5 ABI引起的系统调用问题 41](#_Toc388299301)

[1.7.6 串口终端没有显示 41](#_Toc388299302)

[1.7.7 不能引导kernel 42](#_Toc388299303)

[1.7.8 启动后提示NFL错误 42](#_Toc388299304)

[1.8 init进程 43](#_Toc388299305)

[1.8.1 系统基本配置 43](#_Toc388299306)

[1.8.2 构造信号处理表 44](#_Toc388299307)

[1.8.3 信号的响应方式 46](#_Toc388299308)

[1.8.4 系统信号监控 47](#_Toc388299309)

[1.8.5 inittab格式 49](#_Toc388299310)

[1.9 Busybox 51](#_Toc388299311)

[1.9.1 构造定位信息 51](#_Toc388299312)

[1.9.2 二进制组织方式 52](#_Toc388299313)

[1.9.3 安装方法 52](#_Toc388299314)

[1.9.4 运行流程 53](#_Toc388299315)

[2 U-Boot 55](#_Toc388299316)

[2.1 启动流程 55](#_Toc388299317)

[2.1.1 初始化阶段 55](#_Toc388299318)

[2.1.2 board\_init\_f阶段 56](#_Toc388299319)

[2.1.3 重定位阶段 58](#_Toc388299320)

[2.1.4 重定位结束阶段 59](#_Toc388299321)

[2.1.5 board\_init\_r阶段 60](#_Toc388299322)

[2.2 代码重定位 61](#_Toc388299323)

[2.2.1 重定位原因 61](#_Toc388299324)

[2.2.2 重定位方法 61](#_Toc388299325)

[2.2.3 关于绝对寻址 61](#_Toc388299326)

[2.2.4 需要重定位的情况 62](#_Toc388299327)

[2.3 类LINUX驱动 63](#_Toc388299328)

[2.3.1 NandFlash设备配置 63](#_Toc388299329)

[2.3.2 YAFFS系统的配置 64](#_Toc388299330)

[3 附录 66](#_Toc388299331)

[3.1 MINI2440开发板配置 66](#_Toc388299332)

[3.1.1 S3C2440AL-40 66](#_Toc388299333)

[3.1.2 K9F2G08U0B 67](#_Toc388299334)

[3.1.3 S29AL016J70TFI02 67](#_Toc388299335)

[3.1.4 K4S561632N-LC75 67](#_Toc388299336)

[3.1.5 DM9000EP 67](#_Toc388299337)

[参考文献 68](#_Toc388299338)

# 系统移植

## 移植前的准备

首先要准备的是：明确本次行为的目的，即“移植LINUX到ARM，使之正常运行”；

其次要准备的是：信心、决心、恒心，这是本次行为顺利达成目的的基本条件，尤其是恒心，少了此心，一切都是零，甚至负数，谨记：“没有达到目的的行为就是浪费生命”；

最后要准备的是：一套ARM开发板(本文以MINI2440为例)和一个有LINUX开发环境和WINDOWS环境的电脑。

### 开发板准备

淘宝上购买了友善之臂的MINI2440-256M开发板，实际配置如表1-1所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **名称** | **型号** | **制造商** | **主要特性** | **板载配置** |
| MCU | [S3C2440AL-40](#_S3C2440AL-40) | SAMSUNG |  | x1片 |
| NAND | [K9F2G08U0B](#_K9F2G08U0B) | SAMSUNG 116 | 256M Byte | x1片= 256M |
| NOR FLASH | [S29AL016J70TFI02](#_S29AL016J70TFI02) | SPANSION | 2M Byte | x1片= 2M |
| SDRAM | [K4S561632N-LC75](#_K4S561632N-LC75) | SAMSUNG 128 | 32M Byte | x2片= 64M |
| 网络芯片 | [DM9000EP](#_DM9000EP) | DAVICO |  | x1片 |

表1-1 开发板实际配置

### 工作计划准备

为了提高整体移植的效率，实现各工作环节的可执行性和可验证性，在充分利用友善提供的现有资源(supervivi和移植好的linux)基础上，暂计划移植流程如图1-1所示，以后工作按照此流程推进：



图 1-1 移植工作流程

## 移植linux

### 获取Linux内核源码

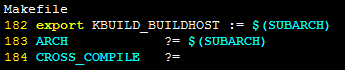
下载内核命令：wget [http:**//**www.kernel.org**/**pub**/**linux**/**kernel**/**v2.6**/**linux-2.6.32.2.tar.gz](http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.2.tar.gz)

解压内核命令：tar xzvf linux-2.6.32.2.tar.gz

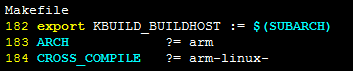
如果解压出现 “Error EOF” 错误，请检查压缩包的完整性，建议重新下载。

### 编译测试

1. 打开源码总目录下的Makefile，[跳转到所在行](#_跳转和移动)修改编译相关变量ARCH和CROSS\_COMPILE：



修改后：



1. 使用SMDK2440的缺省配置文件，s3c2410\_defconfig，执行：

创建config配置 文件

**make s3c2410\_defconfig**

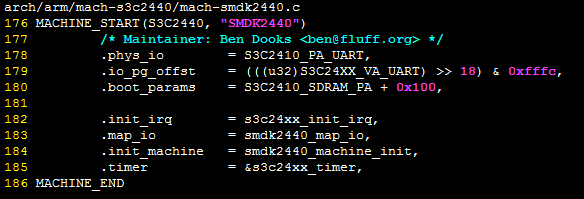
创建zImage等文件，编译时间较长

**make**

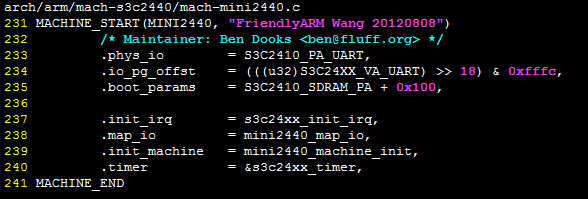
1. 如果编译成功，查看源码目录 arch**/**arm**/**boot**/**，是否生成了zImage文件，如果有，说明下载的源码和本地的编译环境设置是正确的，否则需要定位错误，检查是源码完整性问题还是编译环境设置问题。

### 添加平台相关文件

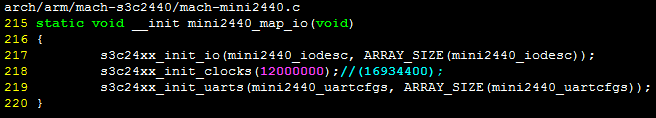
1. 删除arch**/**arm**/**mach-s3c2440**/**目录下的文件mach-mini2440.c(这个别人做的，我们要自己做)；
2. 复制arch**/**arm**/**mach-s3c2440**/**目录下的文件mach-smdk2440.c 到同样目录的mach-mini2440.c；
3. 修改复制的mach-mini2440.c 文件：



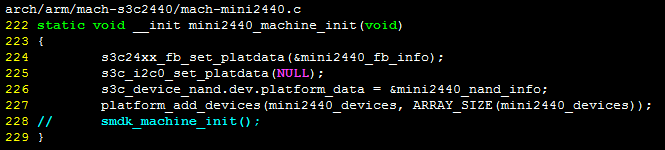
修改后：



1. 把mach-mini2440.c 文件中所有的字符串“smdk2440” [替换](#_查找替换)为“mini2440”；
2. 修改系统时钟为12000000，因为mini2440 开发板上的晶振12MHz (元器件标号为X2)，其中旧的16934400代表原SMDK2440 目标板上的晶振是16.9344MHz。



1. 在mini2440\_machine\_init(void)函数中，把函数调用smdk\_machine\_init()注释掉；



1. 在源码总目录下执行make menuconfig，在配置菜单里进入：

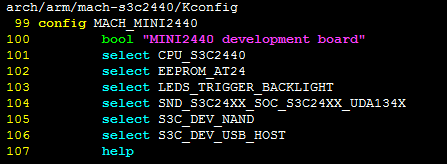
System Type-->S3C2440 Machines-->MINI2440 development board，按空格键选中，然后按ESC保存退出，即代表使用MINI2440开发板的默认配置。

Linux 内核对mini2440 开发板支持的默认选项，可以从文件arch/arm/mach-s3c2440/Kconfig 中看到；

要让选择的配置实际起效，还需要根据Kconfig在与其同一个目录下的Makefile中添加相应的源文件编译规则。

由于此处mach-mini2440.c文件以前就存在，且已经在Makefile中添加了相关文件的编译规则，所以就不用这个步骤。

Kconfig：



Makefile：



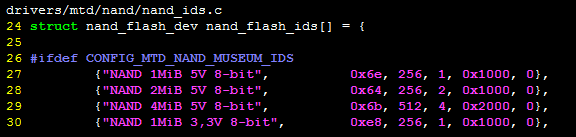
1. 在源码总目录下执行：

**make mini2440\_defconfig** 使用Linux 官方自带的mini2440 配置创建.config配置文件

**make zImage** 编译内核，时间较长，最后会生成zImage

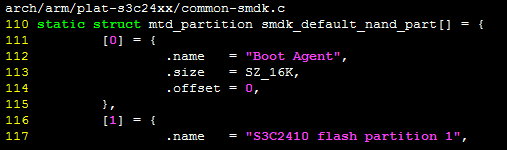
### 测试MTD

1. Linux2.6.32 已经自带了大部分NandFlash 驱动，可从支持列表中查询：

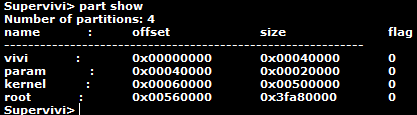


1. 系统默认的分区和bootloader中的分区信息不一致，需要修改：

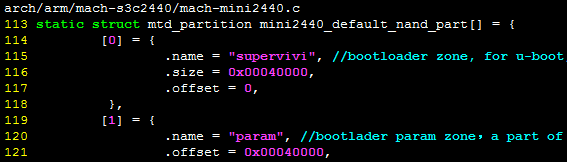
系统默认的分区：



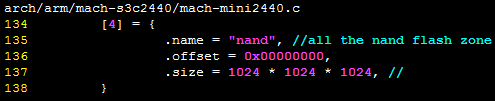
Bootloader中的分区信息:



修改后的分区：

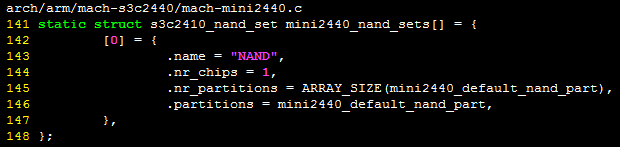


注意：



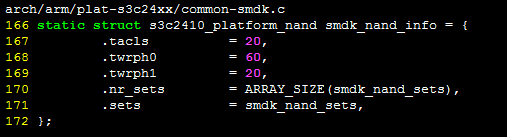
此区域代表了整片的nand flash，主要是预留使用，比如以后可以通过应用程序访问读取/dev/mtdblock4 就能实现备份整片nand flash 了。

1. 配置开发板的NandFlash 的mtd信息表，因为板子上只有一片，因此也就只有一个表：

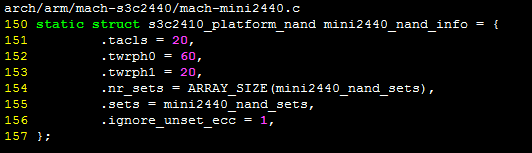


1. NandFlash的结构信息需要增加填写，以便能够适合系统自带的NandFlash 驱动接口，这可以参考SMDK2440 中关于NandFlash 设备注册的一些信息：

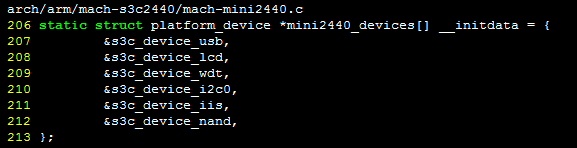
SMDK2440的Nand结构信息配置：



我们的Nand结构信息配置：



1. 最后把nand 驱动注册到系统的平台设备列表中：



注意：

函数 mini2440\_machine\_init(void)中：

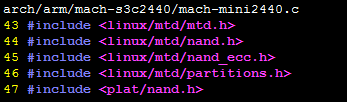


加入这个是因为注释掉smdk\_machine\_init后，实际Nand驱动没有真正注册到系统平台上，所以造成运行后出现异常：

*unable to handle kernel NULL pointer dereference at virtual address 00000018 resolve method*:

1. 编译后烧录运行。

注意：



加入这个是因为编译报错：*incomplete type...之类的*

### 添加yaffs2

1. 获取最新源码：git://www.aleph1.co.uk/yaffs2
2. 在yaffs2源码总目录下执行：

**./patch-ker.sh c m /opt/FriendlyARM/mini2440/linux-2.6.32.2/**

进入linux-2.6.32.2**/**fs 目录，可以看到已经多了一个yaffs2 文件夹，上面命令完成下面三件事情：

a 修改内核fs**/**Makefile



b 修改内核fs**/**Kconfig



c 在内核fs**/**目录下创建yaffs2目录

将yaffs2源码目录下面的Makefile.kernel文件复制为内核fs**/**yaffs2**/**Makefie;

将yaffs2 源码目录的Kconfig文件复制到内核fs**/**yaffs2目录下;

将yaffs2源码目录下的\*.c \*.h文件复制到内核fs**/**yaffs2目录下。

注意：

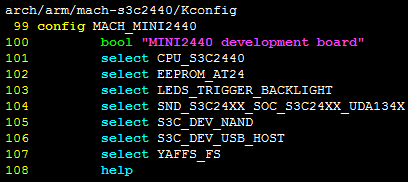
m代表ysffs2中的multive version，如果在新的版本中没有添加这个参数，则命令会执行失败；后面的路径是我们要修改的linux源码总目录的路径。

1. 在 Linux 内核源代码根目录运行：

**make menuconfig**，移动上下按键找到File Systems->Miscellaneous filesystems->yaffs2 file system support，并按空格选中它，这样我们就在内核中添加了yaffs2 文件系统的支持，按“ESC”退出内核配置。

注意：

配置了 menuconfig后，在编译时不要运行 **make distclean**，否则当调用**make s3c2410\_defconfig**后，配置就会丢失，需要重新配置。

可以修改mini2440的默认配置，使之默认打开YAFFS2支持：  


1. 烧录测试：

重新编译kernel，使用友善NOR上存储的suppervivi烧录到NAND，先烧录zImage，再烧录友善自带的文件系统映像文件 rootfs\_qtopia\_qt4.img，如果没有成功，则会打印“No filesystem could mount root arm”之类的提示信息；如果打印“Please press Enter to activate this console”则代表正常进入。

注意：

在虚拟机烧录BIN文件的时候，可能会对存放在主机上的文件读取失败，此时会出现错误信息“File Size Error”建议把要烧写的文件拷贝到虚拟机然后烧录。

## 制作文件系统

嵌入式 Linux 中都需要构建根文件系统，构建根文件系统的规则在FHS(Filesystem Hierarchy Standard)文档中，下面是根文件系统顶层目录：

|  |  |  |
| --- | --- | --- |
| 编号 | 目录 | 内容 |
| 1 | bin | 存放所有用户都可以使用的、基本的命令。 |
| 2 | sbin | 存放的是基本的系统命令，它们用于启动系统、修复系统等。 |
| 3 | usr | 里面存放的是共享、只读的程序和数据。 |
| 4 | proc | 这是个空目录，常作为proc 文件系统的挂载点。 |
| 5 | dev | 该目录存放设备文件和其它特殊文件。 |
| 6 | etc | 存放系统配置文件，包括启动文件。 |
| 7 | lib | 存放共享库和可加载块(即驱动程序)，共享库用于启动系统、运行根文件系统中的可执行程序。 |
| 8 | boot | 引导加载程序使用的静态文件 |
| 9 | home | 用户主目录，包括供服务账号锁使用的主目录，如FTP |
| 10 | mnt | 用于临时挂接某个文件系统的挂接点，通常是空目录。也可以在里面创建空的子目录。 |
| 11 | opt | 给主机额外安装软件所摆放的目录。 |
| 12 | root | 用户的主目录 |
| 13 | tmp | 存放临时文件，通常是空目录。 |
| 14 | var | 存放可变的数据。 |

表1 根文件系统目录说明

制作文件系统的基本步骤是：

1. 在本地创建一个文件夹作为目标系统的根目录，并在该文件夹下按一定规则(FHS)创建其他必需的文件(夹)；

2. 使用工具生成目标系统的基本命令，并将命令执行文件安装到对应目标根目录下的路径，busybox可创建linux的基本命令并安装；

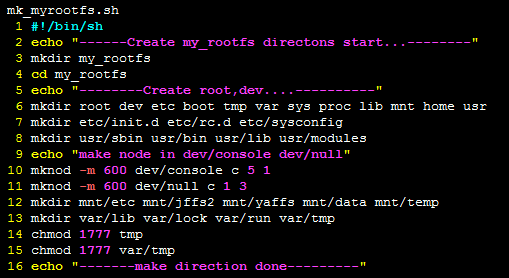
3. 添加配置文件到根文件系统，用于系统初始化和运行时必需的参数及命令等；

4. 根据文件系统要求(比如yaffs2)制作相应的转化工具，能够将文件以及文件组织信息合并转化为一个单独的数据映像；

5. 根据目标系统的文件系统规则和存储设备要求(这里指平台的mtd配置)制作烧录工具，将映像烧录到存储设备，这个工具一般集成在引导程序中，比如vivi和 uboot。

### 创建基础根文件系统

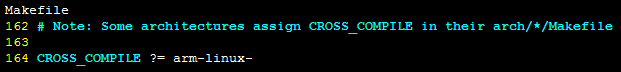
创建系统基本的文件作为系统的根文件系统，可执行如下脚本完成，注意脚本文件需要具有可执行属性，此处在my\_rootfs文件夹下创建了linux 根目录所需要的基本文件(夹)：



### 使用Busybox添加系统基本命令

1. 获取源码并解压：[http:**//**busybox.net**/**downloads**/**busybox-1.13.3.tar.bz2](http://busybox.net/downloads/busybox-1.13.3.tar.bz2)

2. 修改顶层Makefile，建立编译环境：





3. 添加linux的命令配置信息，用来决定使用那些命令。执行**cp fa.config .config**，使用默认配置，或者输入 **make menuconfig** 配置，需要注意的配置如下：

a. 指定编译方式

Build Options--->

[\*] Build BusyBox as a static binary(no shared libs)

注意：如果采用共享库的编译方式，选项如下：

Build Options  --->

[\*]Build shared libbusybox  
 选择共享库方式后要保证busybox依赖的所有共享库做到根文件系统中，具体参考附录 4.3章节中《共享库查看方法》。

共享库一般在交叉编译工具的lib目录下。本文使用的可在友善直臂qt\_rootfs的lib目录找到。

b. 指定mdev设备文件系统

Linux System Utilities --->

[\*]Support **/**etc**/**mdev.conf

[\*]Support command execution at device addition**/**removal

4. 编译busybox，将配置的命令文件安装到我们的根文件系统中：

说明：在busybox中创建dev有三种方法：

（1）手动创建：在制作根文件系统的时候，就在 dev目录下创建好要使用的设备文件，系统

挂接根文件系统后，就可以使用 dev目录下的设备文件了。

（2）使用 devfs 文件系统：这种方法已经过时，具有不确定的设备映射、没有足够的主/次设备号、devfs 消耗大量的内存。

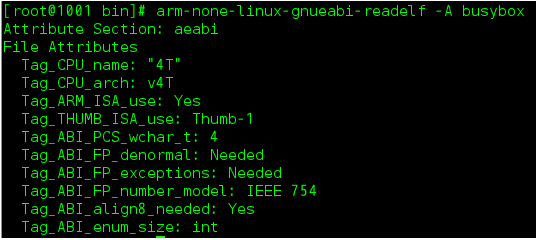
（3）udev：它是个用户程序，（u是指user space ,dev 是指device）能根据系统中硬件设备的状态动态的更新设备文件，包括设备文件的创建、删除等。使用udev机制也不需要/dev目录下创建设备节点，它需要一些用户程序的支持，并且内核要支持sysfs文件系统。它的操作相对复杂，但灵活性很高 。

busybox 自带的mdev一个简化版的 udev，适合于嵌入式的应用埸合。其具有使用简单的特点。它的作用，就是在系统启动和热插拔或动态加载驱动程序时，自动产生驱动程序所需的节点文件。在以busybox 为基础构建嵌入式linux 的根文件系统时，使用它是最优的选择。配置时需要增加对 mdev的支持。

当只执行**make**时，只是将busybox编译成功，当指定了install以及其安装的路径，执行**make CONFIG\_PREFIX=./my\_rootfs install**，即可将busybox生成的的命令文件放到根文件系统对应的目录中（如果之前安装过，先执行 **make uninstall**）。

注意：

生成的busybox是基于armv4指令集的，可查看是否正确的编译：



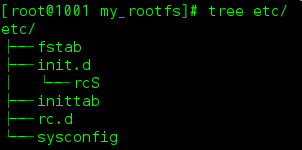
如果不是armv4的：

1. 可在makefile中指定CPU的架构ARCH ?=armv4t；

2. 或者同时指定具体的CPU型号，Makefile中添加 -mtune=arm920t

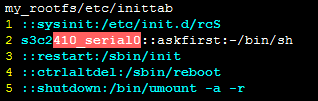
### 添加系统配置文件

配置文件存放的目录为etc，其内容主要取决于要运行的程序，这里只需要创建3个文件即可：etc**/**inittab、etc**/**init.d**/**rcS、etc**/**fstab。



1. 增加配置文件inittab

仿照Busybox的examples**/**inittab，创建etc**/**inittab：



注意：

1由于内核启动命令行参数中init=/linuxrc，因此，在文件系统挂载后，运行的第一个程序就是根目录下的linuxrc,其是一个指向/bin/busybox 的链接(busybox通过不同链接名称来决定实际运行哪种功能)，也就是说，系统起来后运行的第一个程序其实就是busybox本身。

2 如果在启动时报错：

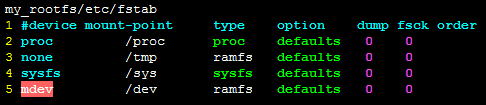
can't open /dev/s3c2410\_serial0: No such file or directory

首先要确定 rcS脚本中是否执行了 mdev –s 命令。如果还是报错，查看内核中用于console的串口设备名称，并修改initab，如下(其中console是串口设备名称)；



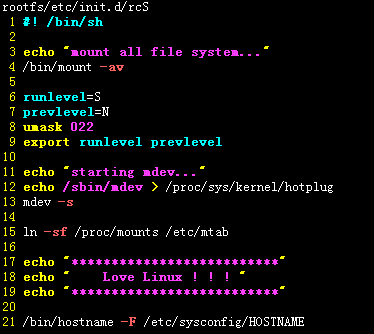
2. 增加配置文件fstab

创建etc**/**fstab，来定义文件系统的“静态信息”：



这些信息用来控制mount的行为，当使用mount -a命令，则会按照fstab中的信息逐项挂载。其中**/**proc、 **/**tmp为“虚拟”文件系统，只在系统运行时存在ram中。

3. 增加可执行文件rcS



这个脚本是inittab中定义的sysinit事件发生时需要执行的命令，需要该文件具有可执行属性。

代码解释：

**/**bin**/**mount -av: 读取etc**/**fstab配置，挂载默认文件系统；

mdev -s：以‘-s’为参数调用位于**/**sbin目录写的mdev，mdev扫描 **/**sys**/**class 和**/**sys**/**block中所有的类设备目录，如果在目录中含有名为“dev”的文件，且文件中包含的是设备号，则mdev就利用这些信息为这个设备在**/**dev下创建设备节点文件。

echo **/**sbin**/**mdev > **/**proc**/**sys**/**kernel**/**hotplug：将mdev命令的路径写入hotplug文件中，当有热插拔事件产生时，内核会读取hotplug脚本中的命令并执行。这时mdev就会被调用，通过环境变量中的 ACTION 和DEVPATH，（这两个变量是系统自带的，内核会更新）来确定此次热插拔事件的动作以及影响了**/**sys中的那个目录。接着会看看这个目录中是否有“dev”的属性文件，如果有就利用这些信息为这个设备在**/**dev 下创建设备节点文件。

### 验证文件系统

1. 解压友善提供的mkyaffs2image.tgz，将其中的mkyaffs2image-128M 可执行文件拷贝出来(另一个文件mkyaffs2image用于小于128M的NAND)；

2. 使用命令**mkyaffs2image-128M my\_rootfs rootfs.img** 生成根文件系统映像文件；

注意：如果使用友善提供的工具转换报错或失败，需要重新制作转换工具，友善提供的工具只保证在fedora平台上的正确性，可以使用本文1.2.5中方法获取YAFFS2源码重新制作工具。

3. 使用supervivi烧录 rootfs.img和前面移植的zImage，验证文件系统是否可以正常挂载到系统，如果失败，检查映像制作工具和根文件系统的配置文件。

注意：如果错误打印如下：

Failed to execute /linuxrc. Attempting defaults...

Kernel panic - not syncing: No init found. Try passing init= option to kernel

可能原因是：

1. 编译busybox时没有采用静态库方法，可以使用readelf -a命令查看生成的busybox，确保busybox编译配置打开静态库选项；

2. 编译环境有问题，使用readelf –A 命令查询busybox的指令是否是armv4。

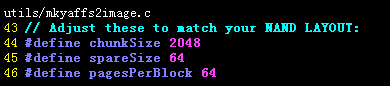
.

### yaffs2镜像制作

1. 进入yaffs2的源码的utils目录下，修改Makefile；



1. 配置nandflash的参数



1. 在utils目录下，执行 **make mkyaffs2image**，则会在utils目录下生成 mkyaffs2image可执行文件，这个文件即yaffs2 制作工具。

使用方式：mkyaffs2image 需要转换的文件夹 生成的image名称

### RAMFS镜像制作

1. 制作镜像

制作initramfs根文件系统镜像时，只需配置Linux内核的Initramfs source file(s)选项即可，该选项用于指定根文件系统目录，在编译内核期间会生成initramfs镜像。

自动生成的initramfs根文件系统镜像在Linux源码树的usr目录下。名字叫initramfs\_data.cpio.gz，它是gz格式的压缩文件。

1. 内核参数

root=ramfs

devfs=mount

### RAMDISK镜像制作

1. 制作工具
2. 源码路径

http://genext2fs.sourceforge.net/

1. 编译工具genext2fs。
2. 制作镜像
3. 跳转到要制作的rootfs的上一级目录

genext2fs -b 4096 -d <rootfs> <ramdisk>

-b是指制作的ramdisk大小为4096K字节

-d是指要制作成ramdisk的根文件系统目录

1. 将该ramdisk以最优方式压缩为ramdisk.gz

gzip -9 -f <ramdisk>

1. 内核参数

initrd=<内存位置>,<大小>来指定initramfs的定位信息

root=/dev/ram rw

### 烧录镜像到nand

因为ramdisk不需要额外烧录文件系统镜像，所以可使用ramfs先进入linux，然后将rootfs所在的分区挂载到一个空文件夹，将制作好的rootfs(注意不是镜像文件，而是busybox生成的文件)，直接拷贝到该空文件下，然后卸载该挂载点即可。

## 移植u-boot

在本文移植时，uboot最新发布版本为，u-boot2013.07，其已经集成了mini2440的配置，但只支持从nor启动，本次移植在其基础上增加nand启动。

### 获取uboot源码

下载命令：wget ftp:**//**ftp.denx.de**/**pub**/**u-boot**/**u-boot-2013.07.tar.bz2

解压命令：tar xjvf u-boot-2013.07.tar.bz2

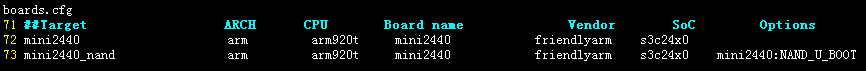
### 编译环境配置

默认已经配置了交叉编译环境，如果需要可修改, 如下:



### 增加板级配置表

增加一个从nand启动的配置,如下图第73行代码。

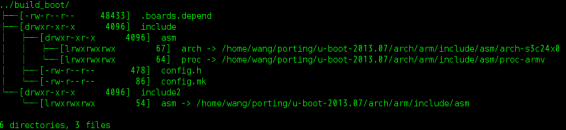


### 生成配置文件

在u-boot源码根目录下执行:



执行后将根据boards.cfg的配置生成相关文件，如下：



1. .boards.depend是解析boards.cfg生成，其包含了所有在boards.cfg中出现的配置，处理方法在主makefile，如下：



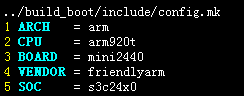
每项的格式如下，其中第30行即此次新添加的配置：



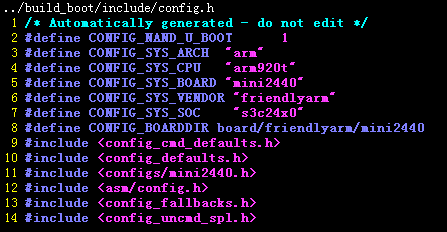
1. 其余文件由可执行程序mkconfig(shell脚本)根据boards.cfg的匹配项生成，在执行 make xxx\_config命令时mkconfig会被调用，如下：

  
mkconfig脚本的具体内容此处不做解释，执行后将生成相关配置文件，主要包括 config.mk，config.h, 以及链接arch,proc,asm,其中：

config.mk定义了在执行make命令（编译源码，不是生成配置）时主makfile中引用的几个变量：



config.h定义了源码中引用到的几个宏定义,如下：



其中boards.cfg的option中定义的NAND\_U\_BOOT对应此处的第２行。

### 编译源码

1. 指定代码的链接地址:



1. 在u-boot源码根目录下执行:

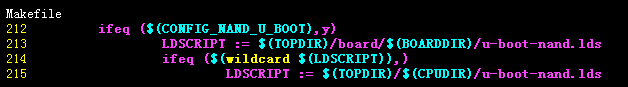
(测试编译，因为nand\_u\_boot相关代码没加，编译过程会报错)

执行后将生成u-boot.bin文件，是可以直接烧录的二进制image文件。

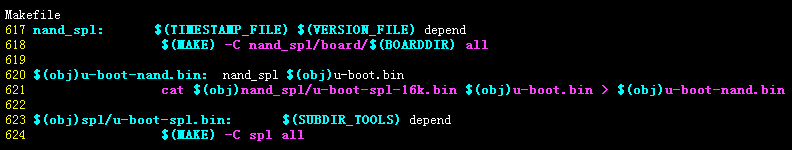
同时在tools目录下将生成[mkimage](#_烧录内核)可执行文件，用于在主机上制作uImage。



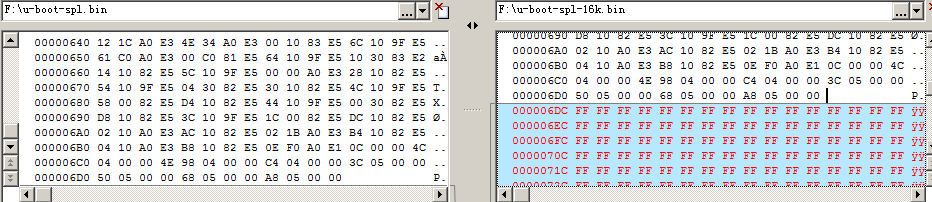
1. 如果指定了NAND\_U\_BOOT(本章节的配置)，在生成u-boot.bin后，make还会继续编译nand\_spl下的代码并将其生成的bin和u-boot.bin合并，其makefile相应规则为：



编译和合并规则如下：



其中的u-boot-spl-16k.bin 是由u-boot-spl.bin填充生成，如下所示:



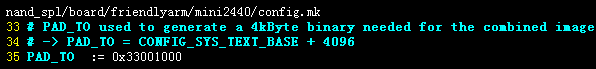
填充的实现如下：



OBJCFLAGS定义如下:



PAD-TO定义如下:

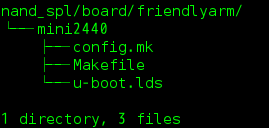




### 添加SPL功能

上个小节的编译在运行过程中，会报错，因为默认代码里没有实现mini2440对应的的SPL(second program loader)，而在打开了NAND\_U\_BOOT配置宏后，主makefile中会去nand\_spl目录中查找对应配置的makefile用于继续编译spl，所以此处需要增加SPL相关代码。

1. 增加链接脚本以及makefile,用于编译spl的实例u-boot- spl-16k.bin(不一定是16K，具体大小可配置)，增加的文件如下：



此处只是定义了编译规则，代码还是和u-boot是复用的，所以针对SPL运行的代码需要编程者用配置宏 CONFIG\_SPL\_BUILD做隔离保护:



针对NAND\_SPL运行的代码用 CONFIG\_NAND\_SPL做隔离保护:

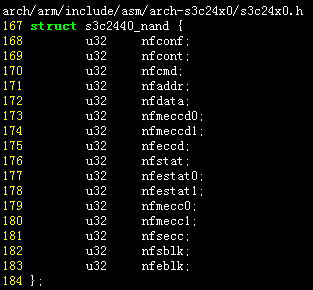


### 添加nand驱动

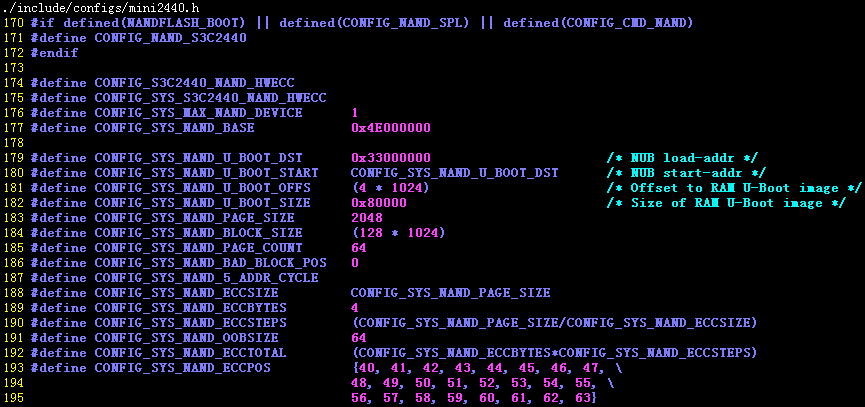
1. 增加**/**drivers**/**mtd**/**nand**/**s3c2440\_nand.c，具体可参考s3c2410\_nand.c，并将文件编译规则加入makfile,如下:



1. 因为默认的2440 nand寄存器配置是错的，所以需要修改s3c24x0.h，修改后如下：

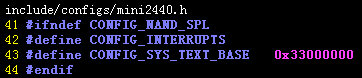


1. 增加NandFlash的配置参数，根据实际NandFlash spec配置如下：

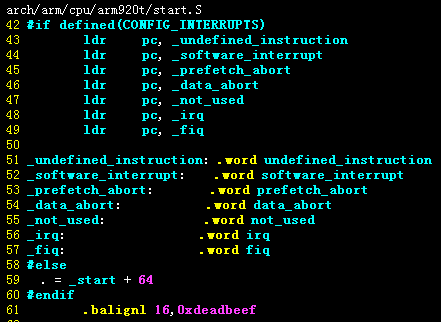


### 修正编译错误以及警告

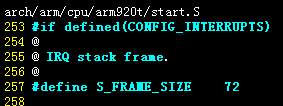
1. 打开中断代码配置宏 CONFIG\_INTERRUPPTS：



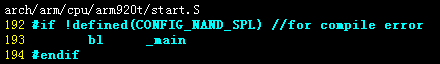
1. 用配置宏将报错的相关代码隔离，如下42行，253行代码:



修改:

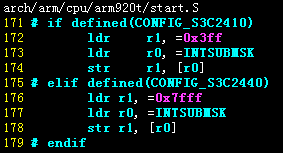
修改:



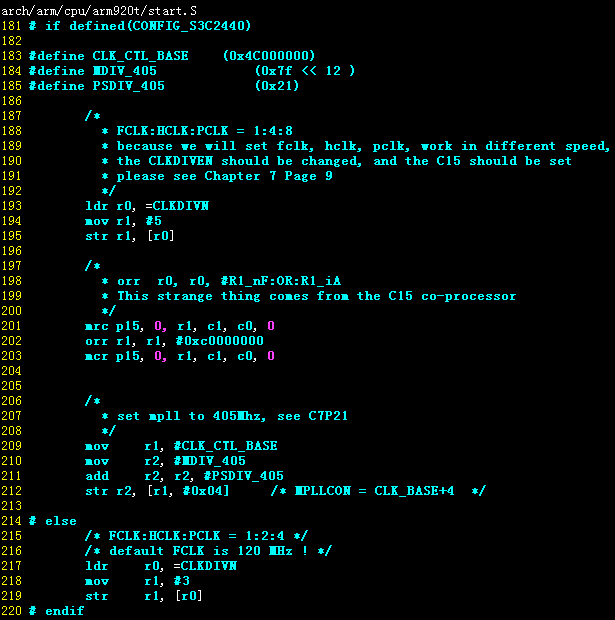
### 配置初始化clock

Start.S中的时钟配置只有2410的，需要对2440重新配置，加入代码：

1. 屏蔽中断

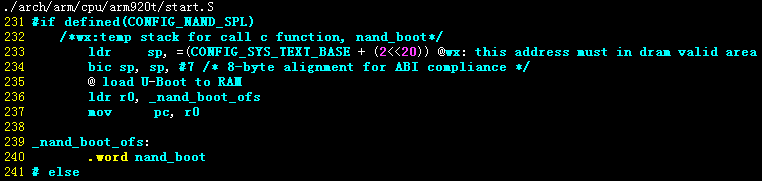


2 配置时钟寄存器

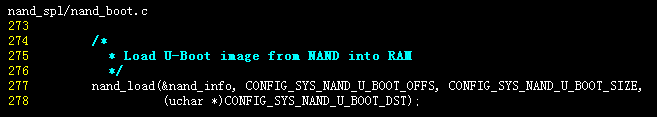


### 从nand启动

1. 当采用了nand\_spl方式时，则在启动时直接调用nand\_boot函数完成对剩余代码的加载，如下：



1. Nand\_boot函数将通过从u-boot.bin在nand的偏移位置读取代码，将其载入对应内存。加载u-boot.bin的代码如下：



其中nand\_load的参数表示意义如下：

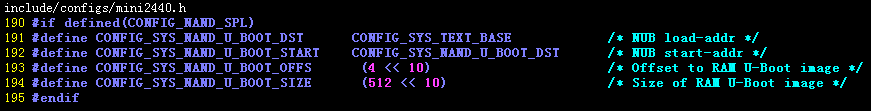
第一个: nand驱动的数据结构，用于控制nand读取数据；

第二个: 数据在NandFlash的偏移地址；

第三个: 数据从偏移地址开始，需要读取的总长度；

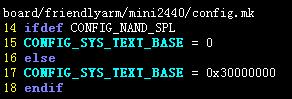
第四个: 数据从NandFlash读取时使用的起始内存地址。

1. u-boot.bin存储信息以及载入信息配置如下：



其中U\_BOOT\_OFFS的4K表示nand-spl.bin的大小为4K，之后的数据是u-boot.bin。

1. 在使用nand\_spl启动方式时，由于nand\_spl代码中有C函数指针调用(mtd代码中大量存在)，这些语句编译后会使用绝对地址寻址的方法定位，而CPU复位后的起始地址为0，所以需要将nand-spl的逻辑入口地址改为0，使前4K的代码逻辑地址和复位后的运行地址一致。修改如下：

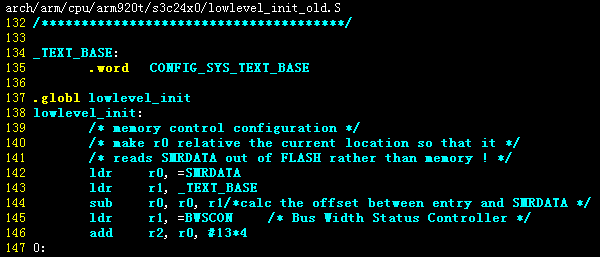


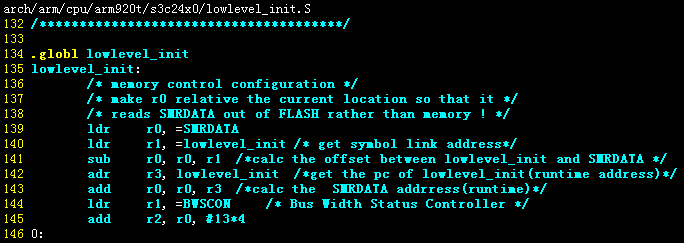
### 优化绝对寻址

s3c2440默认启动代码的一些地方使用绝对寻址，这样就限制了代码使用方式的灵活性。

1. lowlevel\_init对sdram参数的定位。

默认代码使用的是绝对地址，如果代码载入内存的位置和链接地址不一致则执行会异常，修改如下：

修改前：

修改后：

1. nand\_boot跳转。

默认代码是间接绝对寻址，且nand\_boot函数调用不会返回(函数反汇编代码最后没有lr->pc的指令)，故可以修改如下：



### 逻辑地址布局规划

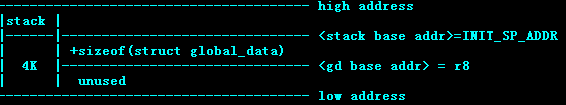
代码重定位需要考虑内存规划，由于u-boot还要负责下载代码并烧录，这些用于特殊作用的地址一般是固定的，所以规划时要一起考虑。

规划时既要考虑各地址区的安全，也要杜绝内存的浪费。

1. 堆栈规划

c语言编译成汇编代码时，在函数跳转时一般会将当前相关上下文入栈，返回调用时再出栈，以便每个函数对寄存器的修改不影响其外部调用者。

Uboot默认将栈底相邻区域作为u-boot的全局数据使用，而且栈会在函数调用过程中向高地址动态增长，其布局如下：



所以要将栈的基址规划到一个既能让栈增长尽量不受限制，又能防止全局数据区被破坏。最符合这样要求的区域就是内存起始位置附近。

如果内存起始位置附近区域已经占用(当前代码加载位置或其他特殊用途)，则可以将其放到内存末端附近。

1. 逻辑首地址(编译的链接入口地址)规划

对应入口地址往后的一块内存区是u-boot代码最终的载入位置，这片内存因为比较大，一般规划到靠中间的内存区，一般使用逻辑首地址作为内存布局的参考点。

1. 数据缓存地址规划

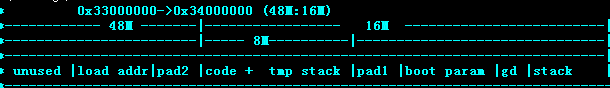
使用fttp,Kermit,usb等下载其他代码时需要一片内存，这片内存可能会很大，所以其要规划到一个相对比较独立的区域，避免和其他内存区重叠。

1. 临时堆栈规划

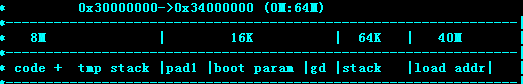
在启动代码的汇编部分，如果在系统堆栈没有初始化前要调用C函数，则需要在每次跳转到C函数前设置一个临时堆栈基址用，因为大多只是在启动阶段临时用，所以最好的方法就是在当前代码最大加载区的末端开辟小块内存即可。

综上所述，b,c两项是相对较大的区域，则在规化时可将其作为关键区域，先给这两个区域分配，其余相对较小的可以附属到相对较小的一方。比如：

当逻辑入口为0x33000000时：



当逻辑入口为0x3000000时：



## 系统烧录

### 下载文件到内存

如果有jlink之类的在线调试工具，则可以将其连接到目标板的jtag接口进行代码下载烧录；如果系统中已经烧录了一个bootloader，而且提供了通过网络或者串口，usb等方式烧录的功能，则可以将该bootloader作为烧录工具使用。

在烧录前代码必须先下载到内存，才能后续的烧录。这里以mini2440为例，介绍两种常用的下载方式。

1. 直接使用vivi下载。

在DNW 程序的option菜单中设置下载地址，然后按[d]，当提示usb连接成功时，点击DNW 程序的“USB Port”->“Transmit”，则开始下载文件到RAM,完成后会自动从下载地址执行。

该方法的缺点在于如果需要增加新特性到烧录工具(比如增加新的下载方式或烧录方式)，则需要先把带有新特性的bootloader烧录到固件后才可以使用。

1. 间接用u-boot下载。

编译一个可以直接运行在内存的u-boot，先用vivi将其下载到内存并运行，然后用这个u-boot下载其他的文件。

该方法与1相比多了一次下载，这样的好处是增加了烧录方式的灵活性，只需将修改的bootloader通过旧有bootloader载入内存，则可以直接用这些新特性，默认的bootloader只需要提供最简单的的功能（下载文件到内存），而带有扩展特性的新bootloader也不需要烧录到固件。

### 烧录文件到nand

由于用vivi烧录时，其mtd数据格式我们并不清楚，所以需要一个我们清楚mtd格式的烧录工具来烧录代码，步骤如下:

1. 使用上节中“[间接用u-boot下载](#_下载到内存)”的方法将文件下载到内存；
2. 将下载到内存的文件代码用u-boot相应的[nand命令](#_nand命令支持)写入对应位置；
3. 如果需要从nand启动，可将启动模式开关从Nor拨到Nand，重启板子。

注意:

一般情况下bootloader所在的nandflash位置不会有坏块，如果u-boot不做坏块检测则可以直接用vivi的 [a]或者[v]功能把u-boot烧录到nand flash偏移为0的位置。

### 制作uImage

u-boot引导linux一般使用uImage映像，该文件是在zImge上增加了一些头信息生成，可在编译linux时生成，也可以用u-boot的mkimage工具生成，使用方法如下：

mkimage使用说明

1. mkimage -l image

-l ==> list image header information

2 mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data\_file[:data\_file...] image

-A ==> set architecture to 'arch' //用于指定CPU类型，比如ARM

-O ==> set operating system to 'os' //用于指定操作系统，比如Linux

-T ==> set image type to 'type'//用于指定image类型，比如Kernelramdisk

-C ==> set compression type 'comp'//指定压缩类型

-a ==> set load address to 'addr' (hex)//指定image的载入地址

-e ==> set entry point to 'ep' (hex)//内核的入口地址，

一般是：image的载入地址+0x40（信息头的大小）

-n ==> set image name to 'name'//image在头结构中的命名

-d ==> use image data from 'datafile'//无头信息的image文件名

-x ==> set XIP (execute in place)//设置执行位置即不进行文件的拷贝，在当前位置执行

3 mkimage [-D dtc\_options] -f fit-image.its fit-image.

### 串口下载命令支持

1. 打开配置宏。

默认u-boot已经内置了loadb命令，该命令使用串口的kermit协议下载文件到内存，格式如下：

loadb　<内存位置>

1. 终端工具配置。

要正常使用该功能，终端必须提供了kermit协议，比如超级终端，putty等工具。

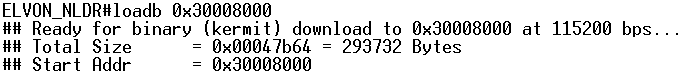
当输入命令后u-boot进入等待终端发送文件的状态，如下：



超级终端则进入发送文件状态，如下：



当u-boot接收完成后，终端显示如下：



表示本次下载的内存位置以及大小。

### 网络命令支持

### nand命令支持

1. 打开配置宏。



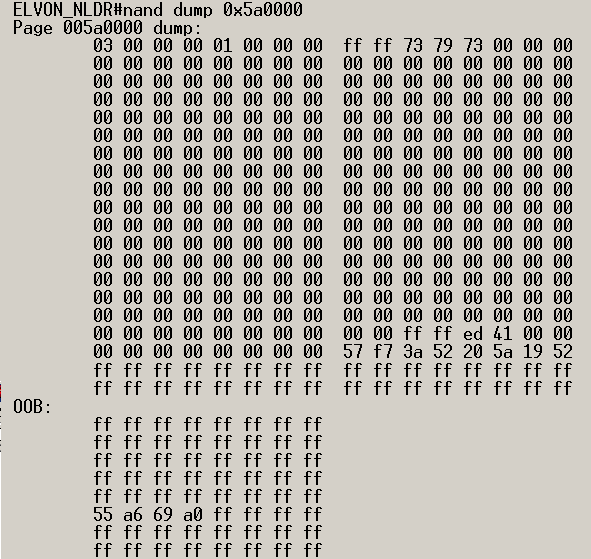
这样可以在u-boot的命令行使用 nand命令来实现相关操作，格式如下：

nand <子命令> [<内存位置>]　[<偏移>]　[<大小>]

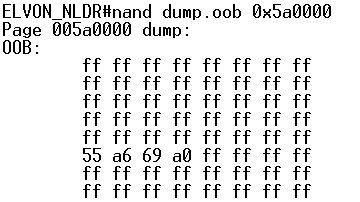
注意：nand　子命令中的位置，偏移和大小是以字节为单位的。

1. 子命令使用。
2. 查看page的数据区：nand dump <起始偏移>

该命令会打印出起始偏移的一个Page大小，并打印出OBB：

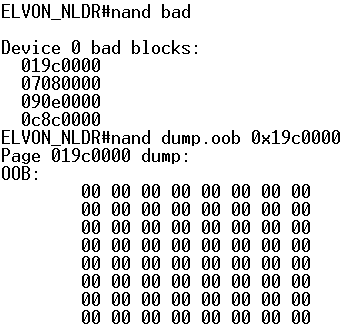


1. 查看page的oob区：nand dump.oob <起始偏移>



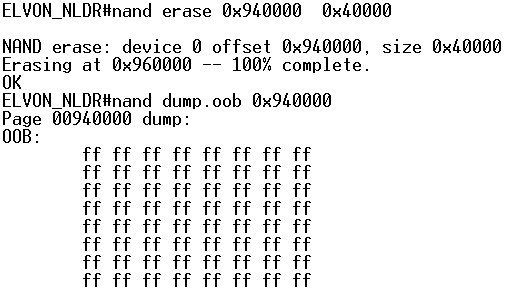
因为当前nandflash的oob大小为64 个字节，所以打印了8x8个字节，其中非ff的四个字节”55 a6 69 a0”即存储的ECC，其起始位置在OOB中的偏移为40(软件配置)，具体可参考[oob格式章节](#_OOB的数据格式)。

1. 查看坏块： nand bad



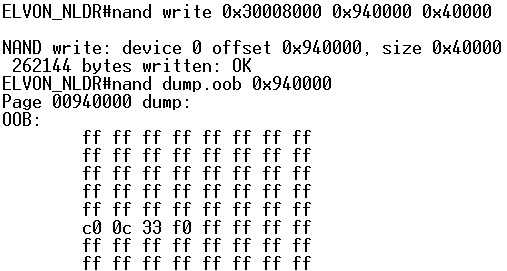
1. 擦除：nand erase <起始偏移> <擦除大小>

该命令除了擦除数据区外，还同时擦除了OOB的非坏块标记区，即除了坏块标记所在区域，OOB区的其他位置都被擦除为0xFF。

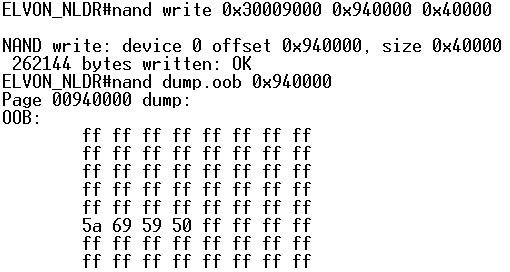


1. 写入：nand write　 <内存位置> <起始偏移> <映像文件大小>

该命令除了写入数据外，还同时更新了OOB区的ECC值，如果写失败还会更新坏块标记值，具体参考[OOB格式章节](#_OOB的数据格式)。

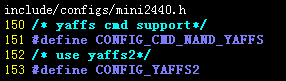


当写入其他数据时，ECC也会更新，如下：



### YAFFS命令支持

1. 打开配置宏。



1. YAFFS映像的写入。

该功能在nand的命令的子命令中，使用方法如下：

nand write.YAFFS <YAFFS镜像在内存的位置> <起始偏移> <文件大小>

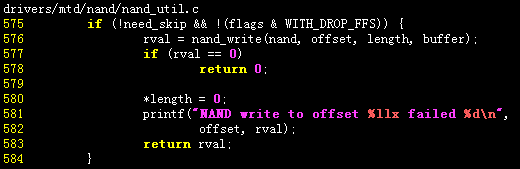
比如：



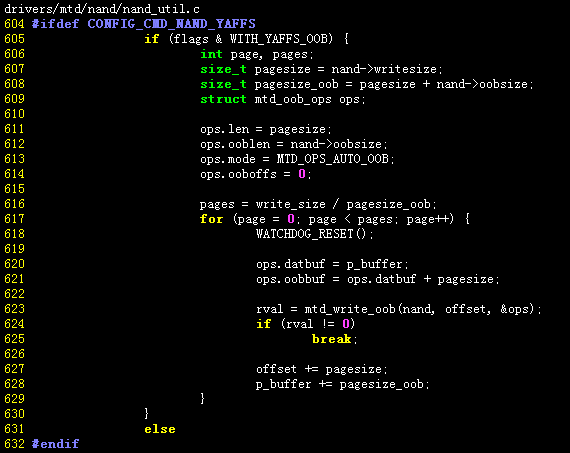
表示将下载到内存0x30008000的YAFFS镜像文件写到nandflash的0x5a000偏移处，其总共写入大小为0x39c000。

YAFFS镜像文件的格式必须正确，具体参考[YAFFS格式](#_yaffs格式)，该命令实现的基本流程如下：

1. 将把镜像文件的数据区写入nand，同时将ecc和坏块标记写入nand的oob区：



1. 然后将镜像文件的OOB区内容作为nandflash 的OOB的一部分，并单独写入nand。



1. YAFFS分区的文件读写：
2. 配置YAFFS分区

ydevconfig <挂载点名称> < mtd设备号> <起始块> <结束块>

比如 ：



表示将设备0的第45块到 80块之间的分区配置为按照YAFFS格式读写(因为本文中每个块是128K，所以0x5a0000的起始块为45)，其挂载点命名为rootfs。注意：起始和结束位置的单位为块号，不是字节。

1. 查看YAFFS分区的配置

ydevls

比如 ：



将会显示所有已配置的分区信息。

1. 挂载分区

ymount <挂载点名称>

比如：



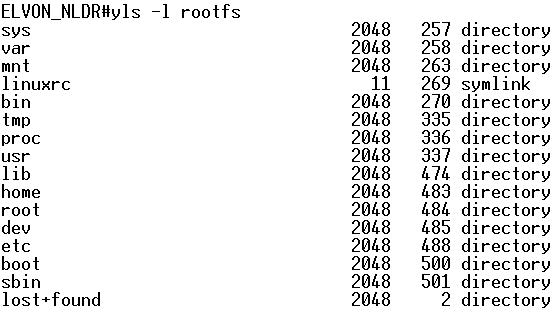
此时查看YAFFS分区显示如下：



其中Free 0x3c0000表示分区中的空闲空间字节，即YAFFS的空间。

1. 查看挂载点文件列表

yls <挂载点名称>

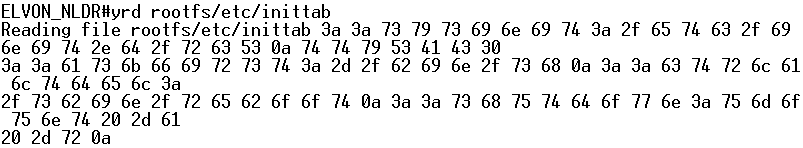


如果命令执行后如下所示，说明分区挂载失败：

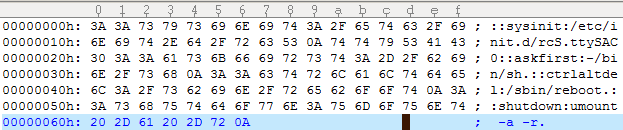


1. 读取文件

yrd <挂载点名称**/**文件路径>



该文件的实际的内容(制作rootfs时的源文件)如下：



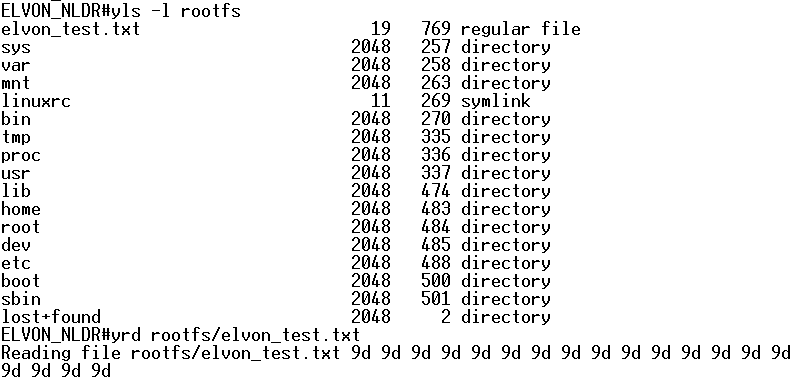
对比可以确认数据读取是正确的。

1. 写入文件

ywr <挂载点名称**/**文件路径> <数值> <次数>

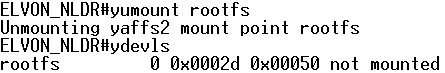


此时查看文件列表以及文件内容，如下：

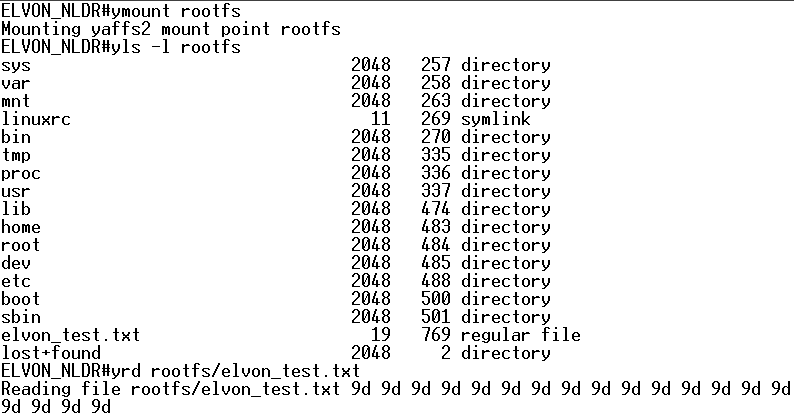


1. 卸载分区

yumount <挂载点名称>



当重启系统后重新挂载，可以看到之前创建的文件：



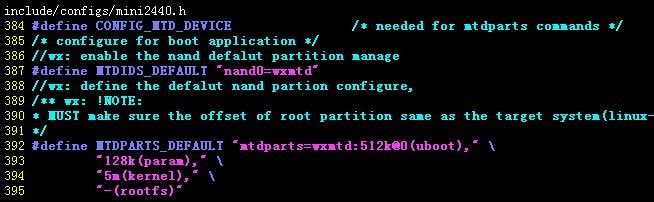
### 配置mtd分区

1. 配置的原因。

当两个分区重叠时，则数据会发生覆盖，从而导致系统异常，此时需要重新配置mtd分区信息，一般容易出现的情况如下：

1. U-boot由于新增了代码而导致和参数区覆盖，当参数保存后则u-boot后面的代码丢失，从而导致从nand启动失败；
2. kernel区被前面的覆盖，导致系统读取内核失败；
3. 由于u-boot的分区配置和内核不一致，导致内核启动后读取nand的文件错误。
4. 配置方法。
5. 计算出各分区实际占用大小，重新定义分区的偏移，当读写nand时使用新的偏移来定位各分区。
6. 修改内核的mtd分区配置，保证和u-boot的分区一致。
7. 可选：配置mtdparts命令，则在u-boot命令行下输入mtdparts 可以查看分区信息，如下：





第一次使用mtdparts命令时，需要先执行 mtdparts default，使修改后的默认配置生效。

### 配置设备树

U-Boot从 v1.1.3开始支持Device Tree，在引导内核时会将.dtb读入内存，然后使用bootz命令来启动内核，增加设备树支持需增加配置如下：

#define CONFIG\_OF\_LIBFDT

## 移植Cross编译工具

## 常见移植问题

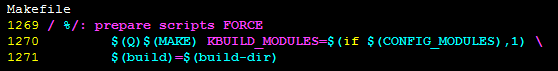
由于工具的升级或者源码的更新，造成各源码之间依赖的版本不一致，当在移植某个源码到另一个源码中时可能发生冲突或者错误。

在新版本的linux源码中，已经加入了对mini2440开发板的支持，由于内核一些接口的变更，可能造成其他依赖于内核的源码和旧有系统的不兼容，所以在移植时需要做相应处理，使之和内核保持一致。

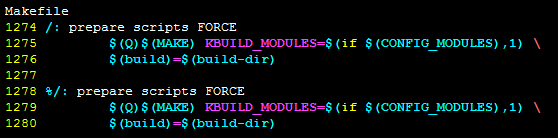
### Makefile语法兼容问题

在使用新版本的make命令时，解析makefile会报错：“混和的隐含和普通规则。 停止”，需要修改makefile的部分语句以适应新版本make对语法的约束；

修改前：



修改后：

  
类似的语句都要拆分：



### mtd编译错误

新版本内核中对mtd数据结构以及部分接口有调整，例如:

1. 在结构体struct mtd\_info中的成员函数命名改成以\_开头的，如mtd->read 改为mtd->\_read,mtd->write改为mtd->\_write等;
2. inode结构体i\_nlink成员变量改为const只读类型，旧有的可读写的i\_nlink等价于新成员\_\_i\_nlink;
3. d\_alloc\_root 接口重命名为d\_make\_root；
4. end\_writeback 接口重命名为clear\_inode。

这些问题发生在一些使用mtd接口的源码中，一般最新的版本中已经同步了内核mtd修改，如果没有修改，而且不打算更换掉旧的源码，那就将以上变更同步使之和新内核兼容。

### yaffs2编译错误

新版本yaffs2源码中对YAFFS\_dev数据结构有调整，例如:

1. 新增成员drv，tagger；
2. 将旧成员param中的部分函数移动到drv，tagger成员中，移动后个别函数名也做了修改。
3. 在u-boot中，其中个别配置宏也有轻微修改。

这些问题在新的yaffs2源码中已经同步，如果不打算更换旧的YAFFS代码，那就将以上变更同步使之和新内核兼容。

### yaffs2的xattr支持问题

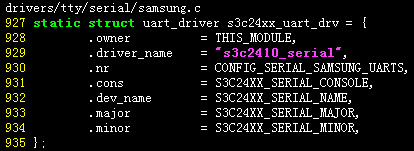
内核中使用xattr获取文件权能失败，具体参考yaffs2的xattr支持章节。

### ABI引起的系统调用问题

EABI和OABI系统调用方式不一样，如果C库和内核的ABI方式不一样，则调用一些命令会出问题。

### 串口终端没有显示

1. mini2440的串口驱动中driver\_name命名为s3c2410\_serial0，所以制作rootfs时 etc**/**inittab文件中需要将串口事件处理的串口设备名称改为s3c2410\_serial0；



1. 在移植中可以打开 DEBUG\_LL可以跟踪内核启动过程，对应选项为：

Kernel hacking->Kernel low-level debugging functions；

除此之外还要确保串口驱动是编译进内核，而不是以模块形式。

1. 如果使用友善之臂的bootloader，需要修改内核启动参数（或者不使用bootloader传入的参数，通过修改内核配置让linux使用固定的启动参数），在console配置中加入串口的波特率,修改后为：

noinitrd root=**/**dev**/**mtdblock3 init=**/**linuxrc **console=ttySAC0,115200**

注意：

在使用bootloader的setparam/setenv命令时,需要在配置字符串前后加双引号。

### 不能引导kernel

新版本linux默认配置是不支持旧版本命令行参数的，如果bootloader仍旧使用旧的方式传递内核启动参数，则需要打开内核配置选项：

Boot options-> Provide old way to pass kernel parameters

### 启动后提示NFL错误

内核默认支持NFL，而uboot烧录时没有支持NFL，所以会造成内核启动后NFL报错，需要关闭该功能，在内核配置时取消下面这三个选项即可：

Device Drivers ->

Memory Technology Devices (MTD) ->

<>FTL (Flash Translation Layer) support

<> NFTL (NandFlash Translation Layer) support

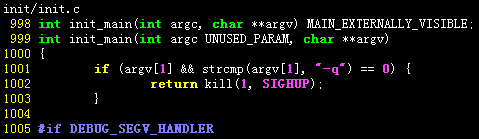
<>INFTL (Inverse NandFlash Translation Layer) support

## init进程

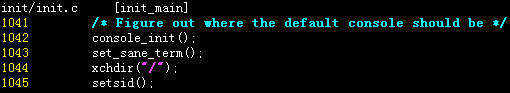
init进程负责系统启动后的用户初始化工作，在内核启动阶段的末期由内核调用，是系统中的第一个用户进程， 其PID为1，本文使用busybox项目中的init程序。

### 系统基本配置

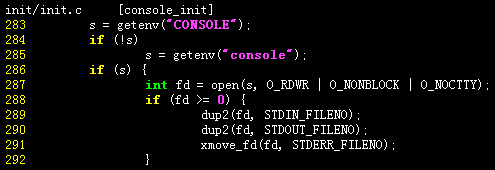
1. init程序的入口



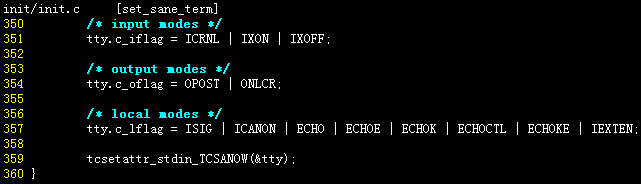
1. 初始化终端



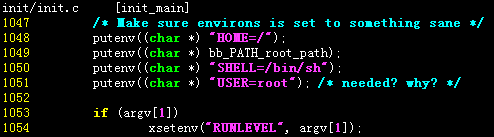
1. 打开终端设备



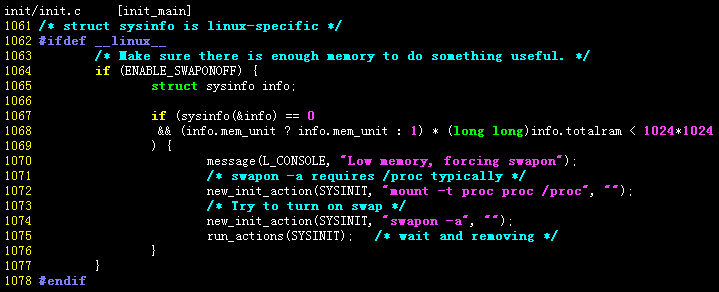
1. 设置终端属性



1. 设置默认环境变量

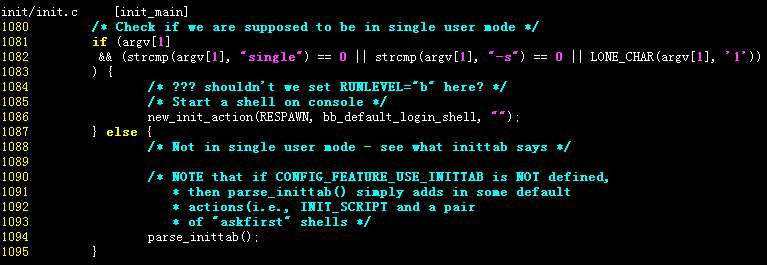


1. 设置交换区内存



### 构造信号处理表

如果是单用户模式则直接创建一个只含有respawn信号响应的处理表，否则通过解析inittab文件来生成系统信号处理表。如下：

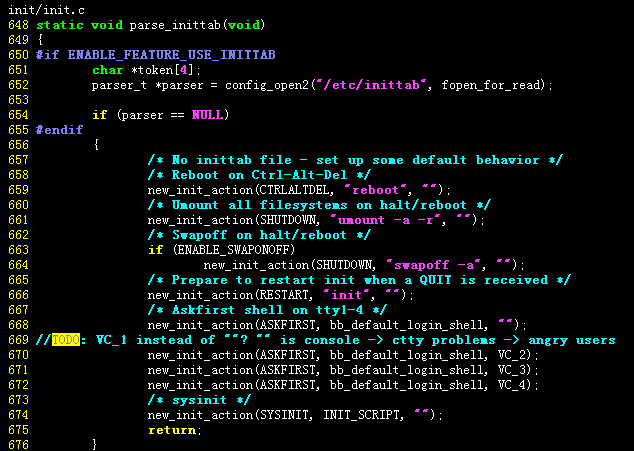


1. respawn的默认处理为打开shell

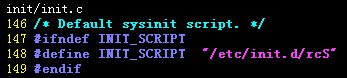




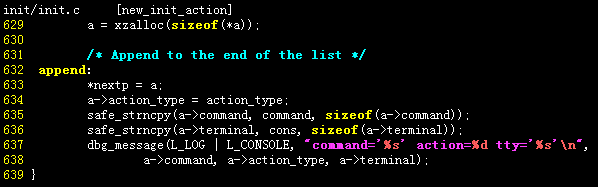
1. 非单用户模式下，如果没有**/**etc**/**inittab文件，则会生成默认的处理表。



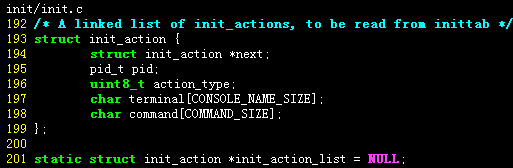
该处理表中SYSINIT信号的默认处理程序为/etc/init.d/rcS脚本，如下配置：



1. 添加信号处理项的方式

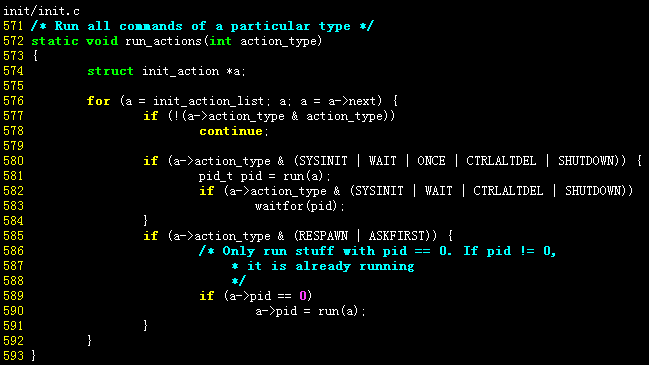


其信号处理表为一个全局链表，如下：

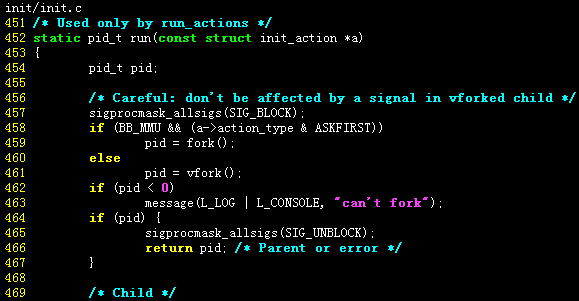


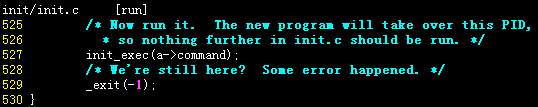
### 信号的响应方式

在需要响应的位置，调用响应处理函数。该函数通过信号类型从信号处理表查找对应命令执行，如下：

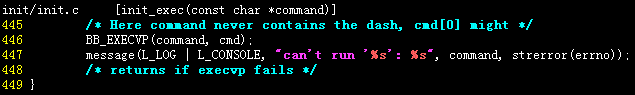


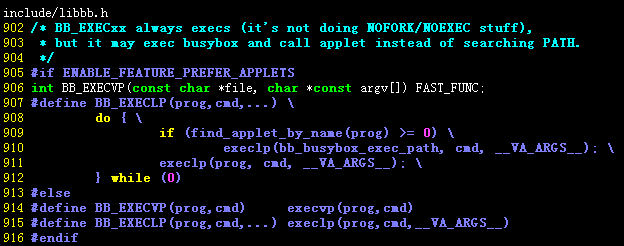
1. 主要通过run函数实现





1. init\_exec实现如下：

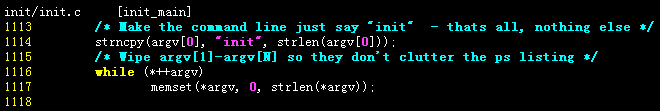




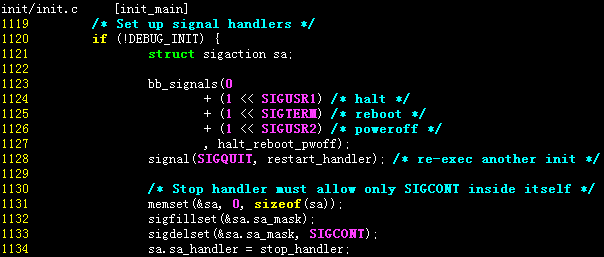
### 系统信号监控

通过监听的系统信号类型，从信号处理表中选择不同的处理方式。

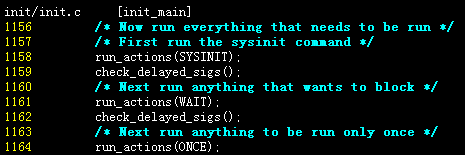
1. 将命令名统一改为init，并清除参数区。



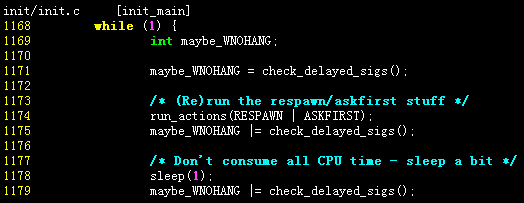
1. 注册信号处理函数，用于捕获系统信号。



1. 运行初始化信号对应的命令

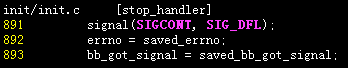


1. 检查是否有系统信号

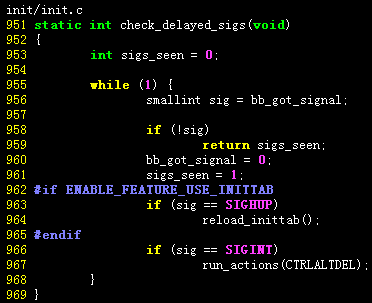


检查系统信号由两个部分组成，信号处理函数负责捕获，然后init\_main负责检查。

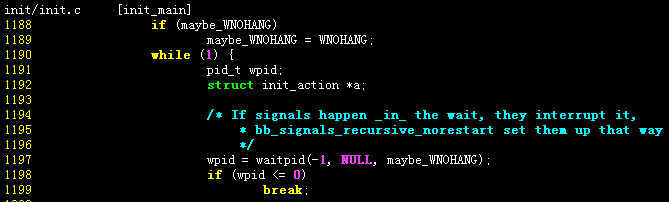
1. 捕获函数



1. 检查函数



1. 等待所有子进程结束



### inittab格式

inittab是一个文本文件，用于描述系统事件的响应方式。init进程通过读取配置文件/etc/inittab来响应系统定义的事件（参考busybox源代码init/init.c中的parse\_inittab()函数）。

每行的命令格式为：id:runlevel:action:process

1. id表示入口标识符

id是一个字符串，对于getty或mingetty等其他login程序项，要求id与tty的编号相同，否则getty程序将不能正常工作。

1. runlevel表示init进程所处于的运行级别

一般使用0－6以及s或S，runlevel可以是并列的多个值，以匹配多个运行级别，此时大多action仅当runlevel与当前运行级别匹配成功才会执行。

在一般的系统实现中，都使用了2、3、4、5几个级别，0、1、6为系统保留，7- 9在传统的Unix系统中没有定义。各运行级别意义如下：

|  |  |
| --- | --- |
| 运行级别 | 含义 |
| 0 | 关机动作 |
| 1 | 重启至单用户模式 |
| 2 | 无NFS支持的多用户模式 |
| 3 | 完全多用户模式（也是最常用的级别） |
| 4 | 保留给用户自定义 |
| 5 | XDM图形登录方式 |
| 6 | 重启动作 |
| S或s | 单用户模式，且无需inittab文件，因此也不在inittab中出现，实际上，进入单用户模式时，init直接在控制台（/dev/console）上运行/sbin/sulogin |

1. action表示运行方式

描述其后的process如何运行。action可取的值包括：initdefault、sysinit、boot、bootwait等：

1. initdefault用于标识缺省的启动级别

当init启动后，将读取inittab中的 initdefault项，取得其中的runlevel，并作为当前的运行级别。如果没有inittab文件，或者其中没有initdefault项， init将在控制台上请求输入runlevel。

1. sysinit、boot、bootwait等action将在系统启动时无条件运行，而忽略其中的runlevel。
2. 其余的action都与某个runlevel相关。各个action的定义在inittab的man手册中有详细的描述。
3. process

process为具体的执行程序。程序后面可以带参数。

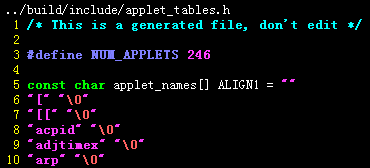
## Busybox

busybox用于构造Linux常用命令，可以将多个命令编译到一起生成一个可执行文件，通过命令名称来运行不同的功能；也可以分别编译

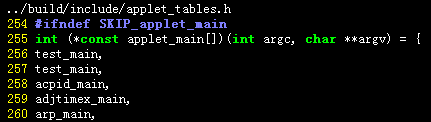
### 构造定位信息

在编译期间生成定位信息，用于查找指定程序的位置，如下：

1. 所有可支持的程序名称列表



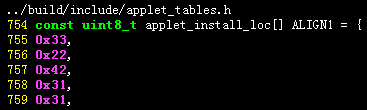
1. 程序入口表



1. 程序名在可支持列表中的偏移位置



1. 程序安装时候的位置(用来表示安装到rootfs的哪个目录)

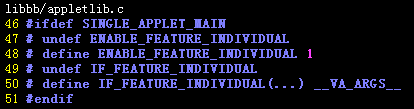


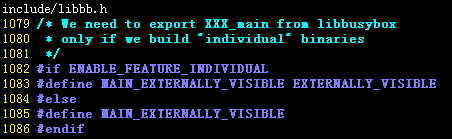
1. 程序名称最大长度



### 二进制组织方式

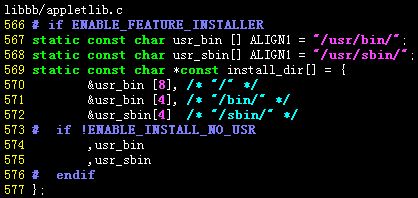
根据编译控配置来确定程序组织方式



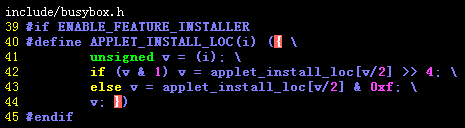


### 安装方法

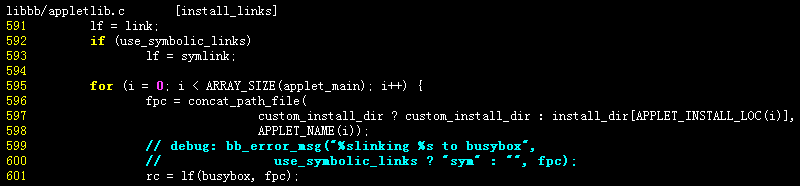
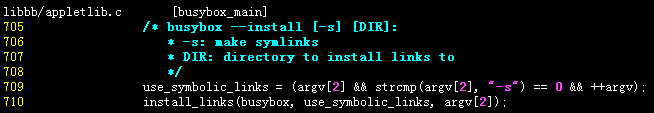
1. 安装路径表



1. 路径索引转换方式

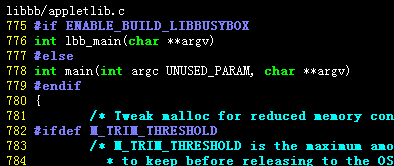


1. 创建链接

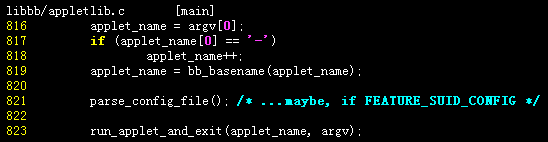


### 运行流程

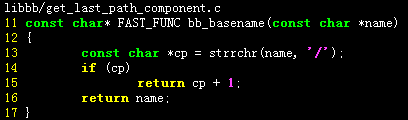
1. 入口函数



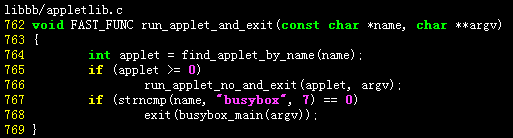
1. 查找对应程序的信息



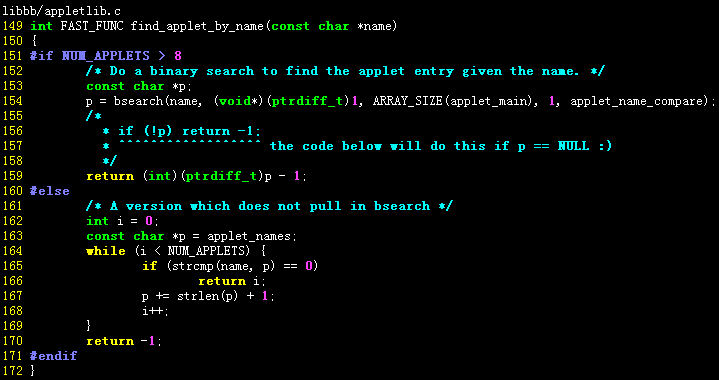
获取命令名称实现如下，即获取路径中的文件名



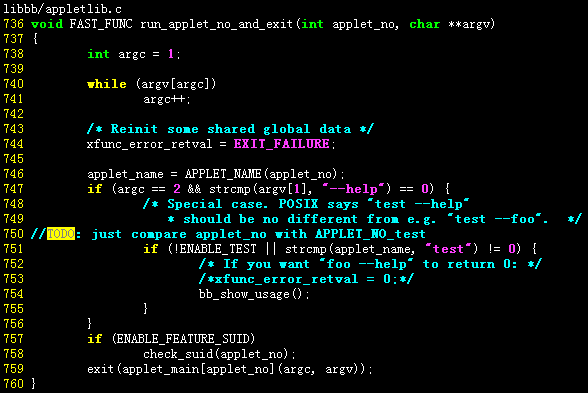
1. 运行对应程序



1. 查找程序入口代码



1. 运行程序，并以程序的返回码作为退出码



# U-Boot

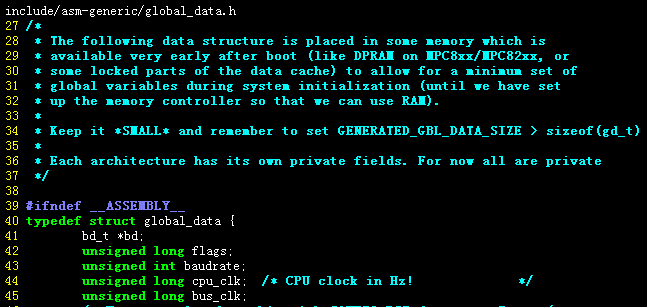
## 启动流程

### 初始化阶段

Uboot代码启动时候的会将系统运行信息放在一个全局变量中，其中包括重定位信息。

因为重定位信息是各模块相关的，需要统计各模块的内存配置，所以配置方式是在系统启动阶段通过收集各模块的配置信息，最终综合处理后确定的。

1. 全局运行信息的定义
2. 类型定义

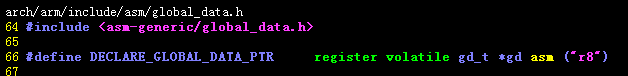


1. 偏移定义

因为汇编中访问结构体的成员变量时只能通过相应的偏移获取，而系统全局数据中一些成员需要汇编中使用，为了方便编程，kbuild工具在编译前会生成对应偏移值的宏定义，如下：



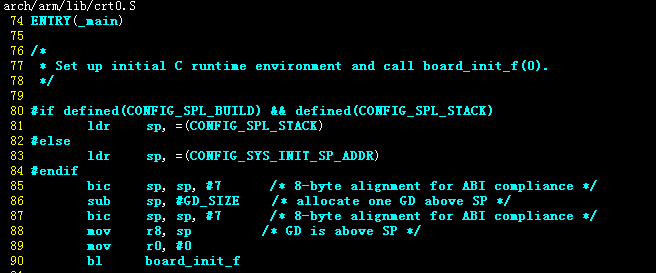
1. 全局数据的变量定义



其定义为一个寄存器指针变量gd，该指针的值在代码中等价于寄存器r8的值，表示全局数据在内存的起始地址。

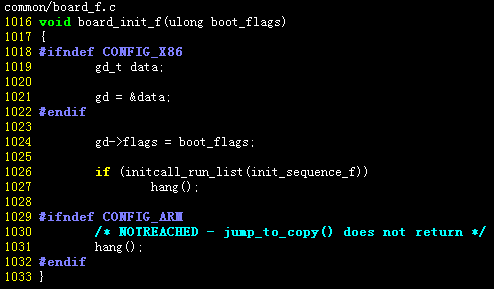
1. 初始化运行信息的存储空间

当进入\_\_main时会将一个内存地址传入r8寄存器，该内存位置用于存储全局数据，即此时gd指针变量指向SP\_ADDR – GD\_SIZE的位置。

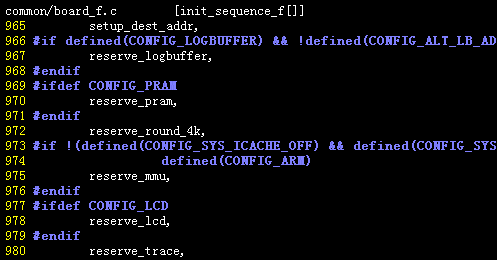


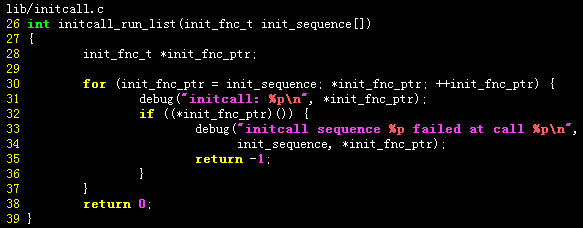
### board\_init\_f阶段

该阶段各模块通过修改gd指针中相关成员完成各自的配置。



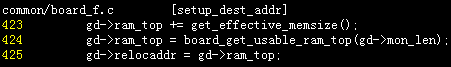
各模块的初始化列表如下；





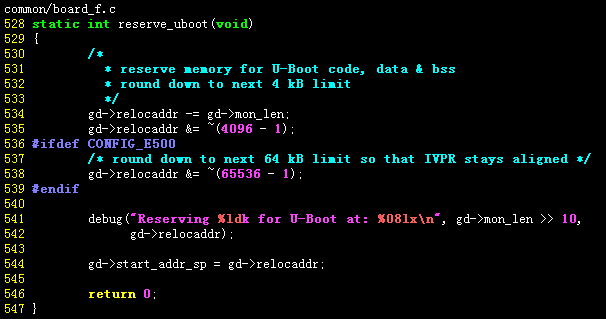
其中重定位主要的模块为：

1. 重定位信息初始化



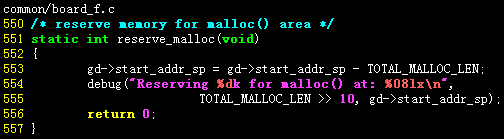
其将gd->relocaddr的重定位地址初始化为内存最顶端。

1. 重定位堆栈位置初始化



其将初始重定位信息(内存顶端)往低端内存偏移代码需要的长度并按页对齐。对其后的地址设置为系统的堆栈指针。

1. 各模块修改堆栈配置，将各自需求的内存加入放入代码和堆栈之间，同时堆栈指针继续往低端地址靠拢，比如：

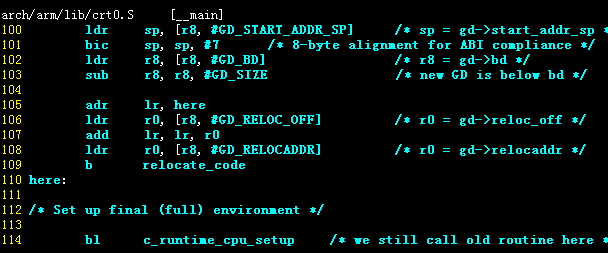


更新系统堆栈指针，实现预留内存给malloc使用。

### 重定位阶段

1. 进入重定位

当各模块的内存需求收集完成后，堆栈的位置也更新到一个新的位置，则开始进入重定位。

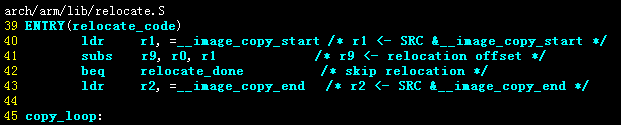


1. 在进入定位前堆栈指针继续往低端内存减小，减小的大小为全局数据的大小，同时将r8初始化为该地址，即全局数据指针的值。
2. 将之前计算的重定位地址作为重定位函数的参数
3. 将lr寄存器的指令地址附加上目标重定位的偏移值，即lr存储了代码重定位后here标签所在的内存位置。
4. 当r0和lr设置后，直接跳转到重定位函数而不更新lr寄存器。
5. 重定位

重定位函数实现将当前代码从内存搬移到目标地址，其中关键代码如下：

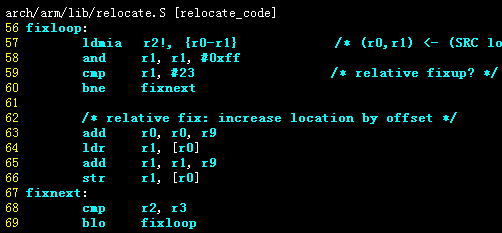
1. 搬移代码的text段

在调用前需要传入重定位的目标地址，即r0的值。



1. 更新全局符号的重定位信息

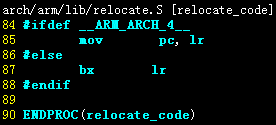
用于对更新全局符号的定位信息，全局数据访问时的汇编代码需要通过这个表的偏移来确定位置。



### 重定位结束阶段

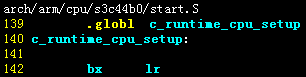
1. 当定位执行完后，代码返回lr记录的内存位置，即执行

bl 　c\_runtime\_cpu\_setup，代码如下：

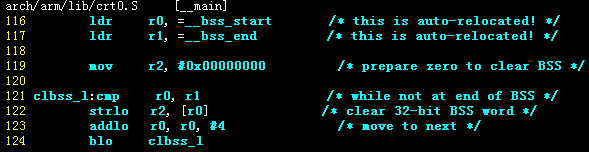




1. 当 c\_runtime\_cpu\_setup返回后，则继续下个指令，即进入清除BSS的代码。



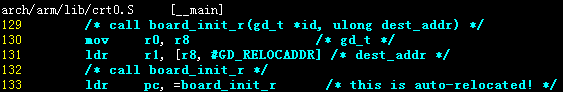
1. 清除重定位后的BSS区



### board\_init\_r阶段

在重定位完成后，系统BSS段清除，此时所有定位工作完成，系统进入init\_r阶段，该阶段是系统一些后续的初始化工作，最终将进入引导内核。

在进入init\_t前将全局数据变量的指针，和重定位后的代码入口地址传递给init\_r函数：



board\_init\_r中一些比较关键的代码如下：

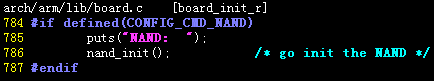
1. 串口初始化



1. Malloc初始化

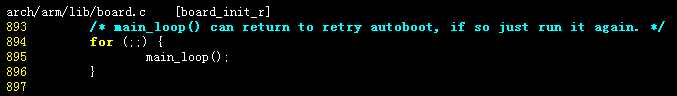


1. NandFlash初始化



1. 用户交互阶段

当所有初始化完成后，进入用户交互的阶段，等待用户的命令输入



## 代码重定位

### 重定位原因

逻辑地址(编译期间确定的链接地址)和运行地址(实际加载的内存位置)使用不同布局时，将导致代码的绝对寻址指令运行混乱或异常。常见原因如下：

1. 内存布局限制。

在运行每个系统实例时,其占有全部内存空间，当需要引导另一个系统实例时，为了不覆盖当前系统的运行区域，所以需要限制另一份代码在内存中的加载布局。

一般cpu复位时，将从固件中读取**固定大小**的代码放入内存的**固定位置**，然后从**固定地址**运行(一般是地址0)，所以逻辑地址和实际运行地址可能不一致。

1. 内存布局需要。

一般的系统会将部分内存保留以作为其他用途(比如堆栈)，如果现有布局与目标布局不一致时，则需要将代码重定位;

1. 内存大小限制。

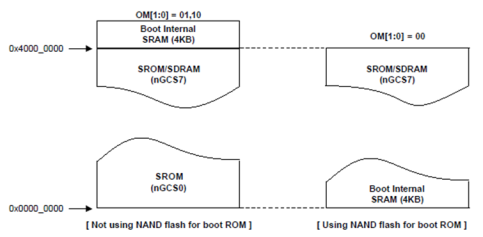
代码不能全部同时载入内存，此时需要保证已在内存的代码能够实现从固件中读取代码并加载到对应内存。

### 重定位方法

1. 如果是内存布局导致，则可以在使用绝对寻址之前对内存内容移动或者使用MMU设备对地址重映射，从而保证代码逻辑地址与其对应内存的实际内容一致，
2. 如果是内存大小导致，则可以增加将剩余代码载入内存的功能，并保证运行该功能的代码常驻在正确的内存区域。
3. 将对应内存内容搬移到新位置后，还需要将代码[GOT段](#_GOT表)的内容进行更新，该段用于存储全局符号的相对偏移(具体参考GCC的链接镜像文件的格式)。

### 关于绝对寻址

1. 这里的绝对寻址是指使用一个确定的值直接修改pc让cpu去执行一个确定位置的指令，或者引用一个确定位置的数据。
2. S3c2440复位时pc值为0，而不是ram的起始物理地址(mini2440的物理地址是0x30000000: nGCS6)，这是因为cpu启动时做了内存映射，将其保留的一块4K内存映射到0-4K的地址，如下：



1. C语言代码中的全局指针引用属于绝对寻址。比如使用函数指针或者引用全局变量(包括static临时变量)的指针。函数指针调用是通过直接更改pc实现的，全局变量的指针值表示的是绝对逻辑地址，引用时属于绝对寻址。
2. 尽量使用相对寻址，比如汇编代码可以用bl或b指令替换直接更改pc的代码跳转，C代码可以将函数指针的调用改为直接用实际函数名调用的方式，以及将全局变量的指针方式引用改为值引用。如果目标符号在相对寻址范围内，则cpu使用相对偏移和当前实际pc定位目标，此时的代码具有位置无关性；如果存在直接更改pc的代码，则代码是位置相关的，此时必须保证运行地址和逻辑地址的一致性，否则在这之前需要进行重定位。

### 需要重定位的情况

1. 从nand启动时，s3c2440cpu在复位时只是把NandFlash前4K的代码读取到内存的0地址，而其他代码还没载入内存，此时由启动代码的前4K负责把剩余代码加载到对应内存，此时相当于完成了重定位。
2. u-boot在正常引导后，需要对其内存布局重新规划，即启动后会按照其规划的内存布局重新对自身定位，然后转入定位后的区域运行。
3. 从ram启动时，一般代码会全部载入内存，若代码载入地址和逻辑地址不一致，则需要在其使用绝对寻址前做重定位。

## 类LINUX驱动

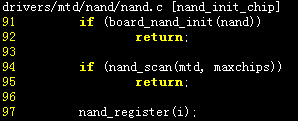
U-Boot的驱动是仿照Linux架构的，其基本方法是通过实现一个系统适配层，达到和linux一些内核模块兼容调用，从而可以方便的复用linux代码。

### NandFlash设备配置

1. 配置NandFlash驱动

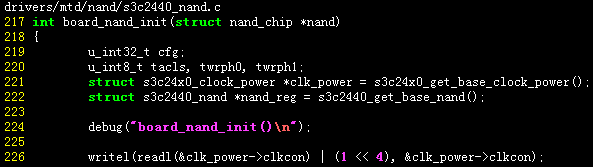
系统启动后会进入board\_init\_r阶段，nand\_init会被调用，用来初始化NandFlash。



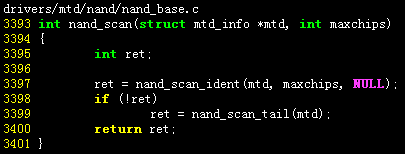


其中：

1. board\_nand\_init用于初始化驱动的NandFlash接口



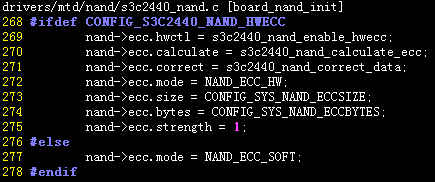
1. nand\_scan通过调用MTD的接口实现对NandFlash的配置



1. 硬件ECC实现

硬件ECC需要由驱动提供相关接口和参数，如下代码：

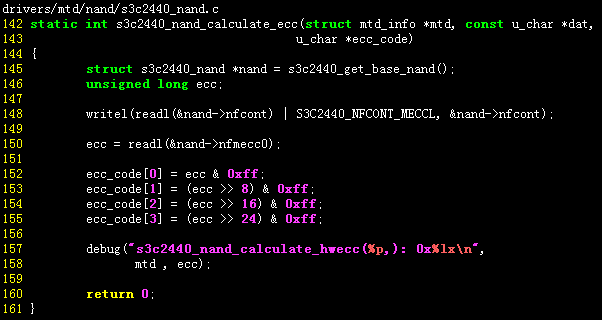




其中各成员作用如下：

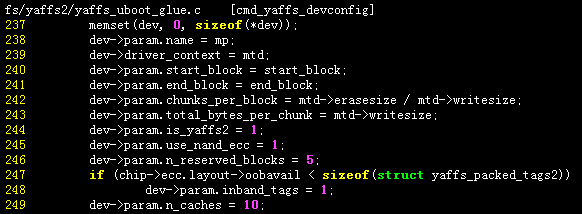
1. ecc.size指每次进行ecc运算时所使用的数据大小，这里配置为一个page的大小(2048)，其大小由ecc模块纠错能力决定。
2. ecc.bytes指每次进行ecc运算得到的ecc校验码大小，这里为4字节，即实际只占用了layout所定义范围的前4个对应位置,。
3. ecc算法实现由驱动提供。

具体算法实现已经封装入硬件内部，驱动只需访问相应IO即可：



### YAFFS系统的配置

U-Boot中是通过解析ydevconfig的命令参数来实现YAFFS系统的配置，其配置代码如下：



其中：

1. start\_block和end\_block表示YAFFS分区在NandFlash的起始和结束块。
2. is\_yaffs2=1表示使用yaffs2 文件系统格式。
3. use\_nand\_ecc表示使用MTD层进行ecc，只用于yaffs1，此处没有意义。
4. chunks\_per\_block表示一个块有多少个YAFFS读写单位。
5. total\_bytes\_per\_chunk表示一个读写单位的大小，一般配置为一个page的大小。
6. 因为param被初始化为0，所以no\_tags\_ecc = 0，即需要对tags进行ecc。

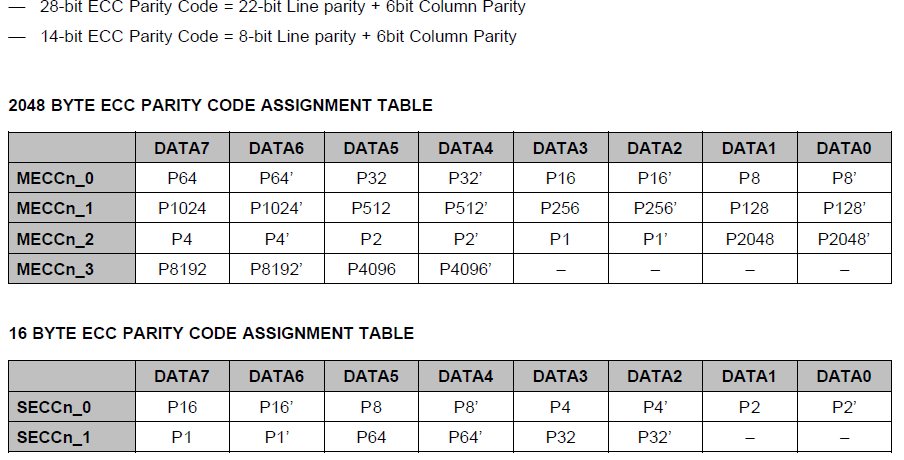
# 附录

## MINI2440开发板配置

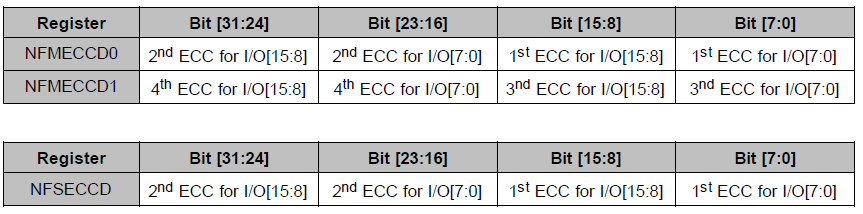
### S3C2440AL-40

* 类型：CPU
* ECC模块特性

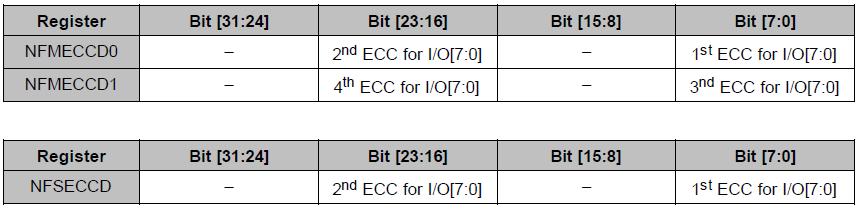
1. ECC纠错能力



1. 16bit NandFlash的ECC接口



1. 8bit NandFlash的ECC接口



### K9F2G08U0B

* 类型：NandFlash Memory
* 容量规格：

|  |  |  |
| --- | --- | --- |
| 名称 | 大小 | 单位 |
| Page | 2048(data) + 64(oob) | Byte |
| Block | 64 | Page |
| Device | 2048 | Block |
| 总计 | (2048+64) \*64 \* 2048  = (128K + 4K) \* 2048  = 2112Mbits = > 2M | Byte |

### S29AL016J70TFI02

* 类型：NOR Flash Memory
* 3.0 VOLT ONLY BOOT SECTOR FLASH MEMORY
* 容量规格：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | | 大小 | 单位 |
| Word | | 16 | Bit |
| Special Sector | Reserved | 8K + 4K + 4K + 16K | Word |
| Secured | 128 |
| Normal Sector | | 32K | Word |
| Device | | 31(Normal) + 1(Special) | Sector |
| 总计 | | 31\*32K + 32K + 128  = 1M + 128 **/** 2M + 256 (Word**/**Byte Mode) | Word**/**Byte |

### K4S561632N-LC75

* 类型：SDRAM

### DM9000EP

* 类型：Ethernet MAC Controller

参考文献

1. 友善之臂.Mini2440 Linux移植开发实战指南[M].广州:友善之臂公司,2010.