Luffy Li (李壮)

嵌入式学习笔记（三）

Uboot和Linux内核移植

基于S5PV210

[一、uboot学习前传 1](#_Toc26949075)

[1.1为什么要有uboot 1](#_Toc26949076)

[1.1.1.计算机的主要部件 1](#_Toc26949077)

[1.1.2 .PC机的启动过程 1](#_Toc26949078)

[1.1.3.典型嵌入式linux系统启动过程 1](#_Toc26949079)

[1.1.4. android系统启动过程 1](#_Toc26949080)

[1.1.5.总结：uboot到底是干嘛的 1](#_Toc26949081)

[1.2 为什么是uboot 2](#_Toc26949082)

[1.2.1 .uboot从哪里来 2](#_Toc26949083)

[1.2.2 .uboot的发展历史 2](#_Toc26949084)

[1.2.3. uboot的版本号问题 2](#_Toc26949085)

[1.2.4. uboot的可移植性的正确理解 2](#_Toc26949086)

[1.2.5.总结：时势造英雄，任何牛逼的东西都是时代的产物 2](#_Toc26949087)

[1.3 uboot必须解决哪些问题 2](#_Toc26949088)

[1.3.1.能自身开机直接启动 2](#_Toc26949089)

[1.3.2.能引导操作系统内核启动并给内核传参 2](#_Toc26949090)

[1.3.3.能提供系统部署功能 3](#_Toc26949091)

[1.3.4.能进行SoC级和板级硬件管理 3](#_Toc26949092)

[1.3.5.uboot的生命周期 3](#_Toc26949093)

[1.4 uboot的工作方式 3](#_Toc26949094)

[1.4.1.从裸机程序镜像uboot.bin说起 3](#_Toc26949095)

[1.4.2.uboot的命令行式shell界面 4](#_Toc26949096)

[1.4.3.掌握uboot使用的2个关键点：命令和环境变量 4](#_Toc26949097)

[1.4.4.思考：结合ARM裸机部分进行理解和印证 4](#_Toc26949098)

[1.5 uboot的常用命令 4](#_Toc26949099)

[1.5.1.类似linux终端的行缓冲命令 4](#_Toc26949100)

[1.5.2.有些命令有简化的别名 4](#_Toc26949101)

[1.5.3.有些命令会带参数（注意格式是固定的） 5](#_Toc26949102)

[1.5.4.命令中的特殊符号（譬如单引号） 5](#_Toc26949103)

[1.5.5.有些命令是一个命令族（譬如movi） 5](#_Toc26949104)

[1.5.6.第一个命令：printenv/print 5](#_Toc26949105)

[1.5.7.设置（添加/更改）环境变量：setenv/set 5](#_Toc26949106)

[1.5.8.保存环境变量的更改：saveenv/save 5](#_Toc26949107)

[1.5.9.网络测试指令：ping 6](#_Toc26949108)

[1.5.10.tftp下载指令：tftp 6](#_Toc26949109)

[1.5.11.SD卡/iNand操作指令movi 7](#_Toc26949110)

[1.5.12.NandFlash操作指令nand 7](#_Toc26949111)

[1.5.13.内存操作指令：mm、mw、md 7](#_Toc26949112)

[1.6 开发板和主机的ping通 8](#_Toc26949113)

[1.6.1.开发板运行linux下和主机Windows的ping通 8](#_Toc26949114)

[1.6.2.开发板运行linux下和虚拟机ubuntu的ping通 8](#_Toc26949115)

[1.6.3.开发板运行uboot下和主机Windows的ping通 8](#_Toc26949116)

[1.6.4.开发板运行uboot下和虚拟机ubuntu的ping通 9](#_Toc26949117)

[1.7 uboot的常用环境变量 9](#_Toc26949118)

[1.7.1.环境变量如何参与程序运行 9](#_Toc26949119)

[1.7.2.自动运行倒计时：bootdelay 9](#_Toc26949120)

[1.7.3.网络设置：ipaddr serverip 9](#_Toc26949121)

[1.7.4.自动运行命令设置：bootcmd 9](#_Toc26949122)

[1.7.5.uboot给kernel传参：bootargs 9](#_Toc26949123)

[1.7.6.新建、更改、删除一个环境变量的方法 10](#_Toc26949124)

[1.7.7.注意：环境变量更改后的保存 10](#_Toc26949125)

[1.8 uboot中对Flash和DDR的管理 10](#_Toc26949126)

[1.8 .1.uboot阶段Flash的分区 10](#_Toc26949127)

[1.8 .2.uboot阶段DDR的分区 11](#_Toc26949128)

[二、Shell和Makefile 11](#_Toc26949129)

[2.1 Shell介绍 11](#_Toc26949130)

[2.1.1.Shell是什么？ 11](#_Toc26949131)

[2.1.2.shell概指一类编程语言（在shell中用于编写程序的语言） 11](#_Toc26949132)

[2.1.3.shell脚本的运行机制：解释执行（没有编译和链接的过程） 11](#_Toc26949133)

[2.2 写shell程序 12](#_Toc26949134)

[2.2.1.文本编辑器 12](#_Toc26949135)

[2.2.2.shell程序的运行方法 12](#_Toc26949136)

[2.2.3.shell程序注意事项： 12](#_Toc26949137)

[2.2.4.shell不神秘 12](#_Toc26949138)

[2.3 shell编程学习 13](#_Toc26949139)

[2.3.1.shell中的变量定义、初始化、赋值和引用 13](#_Toc26949140)

[2.3.2.shell中无引号、单引号、双引号的区别 13](#_Toc26949141)

[2.3.3.shell中调用linux命令 14](#_Toc26949142)

[2.4 shell中的选择分支结构 14](#_Toc26949143)

[2.5 shell中的循环结构 15](#_Toc26949144)

[2.6 Makefile基础回顾 17](#_Toc26949145)

[2.6.1.Makefile的作用和意义 17](#_Toc26949146)

[2.6.2.目标、依赖、命令 17](#_Toc26949147)

[2.6.3.通配符%和Makefile自动推导（规则） 18](#_Toc26949148)

[2.6.4.Makefile中定义和使用变量 18](#_Toc26949149)

[2.6.5.伪目标（.PHONY） 18](#_Toc26949150)

[2.6.6.Makefile的文件名 18](#_Toc26949151)

[2.6.7.Makefile中引用其他Makefile（include指令） 18](#_Toc26949152)

[2.7 Makefile补充学习 18](#_Toc26949153)

[2.7.1.Makefile中的注释 # 18](#_Toc26949154)

[2.7.2.命令前面的@用来静默执行 18](#_Toc26949155)

[2.7.3.Makefile中几种变量赋值运算符 19](#_Toc26949156)

[2.7.4.Makefile的环境变量 20](#_Toc26949157)

[2.7.5.Makefile中的通配符 20](#_Toc26949158)

[2.7.6.Makefile的自动变量 20](#_Toc26949159)

[三、零距离体验uboot 20](#_Toc26949160)

[3.1 X210官方uboot配置编译实践 20](#_Toc26949161)

[3.1.1.找到官方移植好的uboot（BSP概念） 21](#_Toc26949162)

[3.1.2.在linux源生目录下配置编译 21](#_Toc26949163)

[3.1.3.配置 21](#_Toc26949164)

[3.1.4.编译得到uboot.bin 21](#_Toc26949165)

[3.2 uboot的源码目录分析 22](#_Toc26949166)

[3.2.1.九鼎官方uboot和三星原版uboot对比 22](#_Toc26949167)

[3.2.2.各文件介绍 22](#_Toc26949168)

[3.2.3.各文件夹介绍 23](#_Toc26949169)

[3.3 SourceInsight的基本使用 25](#_Toc26949170)

[3.3.1.为什么要使用SourceInsight 25](#_Toc26949171)

[3.3.2.建立工程及添加文件 25](#_Toc26949172)

[3.3.3.解析工程文件 26](#_Toc26949173)

[3.3.4.常用技巧 26](#_Toc26949174)

[四、uboot配置和编译过程详解 26](#_Toc26949175)

[4.1 uboot主Makefile分析 26](#_Toc26949176)

[4.1.1.uboot version确定（Makefile的24-29行） 26](#_Toc26949177)

[4.1.2.HOSTARCH和HOSTOS 26](#_Toc26949178)

[4.1.3.静默编译（50-54行） 27](#_Toc26949179)

[4.1.4.两种编译方法 27](#_Toc26949180)

[4.1.5.OBJTREE、SRCTREE、TOPDIR 27](#_Toc26949181)

[4.1.6.MKCONFIG（Makefile的101行） 27](#_Toc26949182)

[4.1.7.include $(obj)include/config.mk（133行） 28](#_Toc26949183)

[4.1.8.ARCH CROSS\_COMPILE 28](#_Toc26949184)

[4.1.9.$(TOPDIR)/config.mk（主Makefile的185行） 29](#_Toc26949185)

[4.1.10.第一个目标all（主Makefile的第291行） 30](#_Toc26949186)

[4.2 uboot配置过程mkconfig详解 30](#_Toc26949187)

[4.3 uboot的u-boot.lds链接脚本 32](#_Toc26949188)

[五、uboot源码分析1-启动第一阶段 33](#_Toc26949189)

[5.1 start.S的引入 33](#_Toc26949190)

[5.1.1. u-boot.lds中找到start.S入口 33](#_Toc26949191)

[5.1.2. SI中如何找到文件 33](#_Toc26949192)

[5.1.3. SI中找文件技巧 33](#_Toc26949193)

[5.2 start.S解析 33](#_Toc26949194)

[5.2.1.不简单的头文件包含 34](#_Toc26949195)

[5.2.2.启动代码的16字节头部 34](#_Toc26949196)

[5.2.3.异常向量表的构建 35](#_Toc26949197)

[5.2.4.有点意思的deadbeef 35](#_Toc26949198)

[5.2.5.TEXT\_BASE等 36](#_Toc26949199)

[5.2.6.（107行）CFG\_PHY\_UBOOT\_BASE 36](#_Toc26949200)

[5.2.8.设置CPU为SVC模式（149行） 36](#_Toc26949201)

[5.2.9.设置L2、L1cache和MMU 36](#_Toc26949202)

[5.2.10.识别并暂存启动介质选择 36](#_Toc26949203)

[5.2.11.设置栈（SRAM中的栈）并调用lowlevel\_init 37](#_Toc26949204)

[5.3 lowlevel\_init.S解析 37](#_Toc26949205)

[5.3.1.检查复位状态 37](#_Toc26949206)

[5.3.2.IO状态恢复 37](#_Toc26949207)

[5.3.3.关看门狗 38](#_Toc26949208)

[5.3.4.一些SRAM SROM相关GPIO设置 38](#_Toc26949209)

[5.3.5.供电锁存 38](#_Toc26949210)

[5.3.6.判断当前代码执行位置 38](#_Toc26949211)

[5.3.7.system\_clock\_init 39](#_Toc26949212)

[5.3.8.mem\_ctrl\_asm\_init 39](#_Toc26949213)

[5.3.9.uart\_asm\_init 40](#_Toc26949214)

[5.3.10.tzpc\_init 40](#_Toc26949215)

[5.3.11.pop {pc}以返回 40](#_Toc26949216)

[5.4 回到start.S解析 40](#_Toc26949217)

[5.4.1.再次设置栈（DDR中的栈） 40](#_Toc26949218)

[5.4.2.再次判断当前地址以决定是否重定位 41](#_Toc26949219)

[5.5 uboot重定位详解 41](#_Toc26949220)

[5.6 start.S继续解析1 42](#_Toc26949221)

[5.6.1.关于虚拟地址和物理地址 42](#_Toc26949222)

[5.6.2.地址映射原理 43](#_Toc26949223)

[5.6.3.什么是页表(转换表)呢？ 43](#_Toc26949224)

[5.6.4.uboot中虚拟地址映射采用了段模式 43](#_Toc26949225)

[5.6.5．uboot中的映射页表 46](#_Toc26949226)

[5.6.6.MMU单元的作用 46](#_Toc26949227)

[5.6.7.地址映射的额外收益1：访问控制 46](#_Toc26949228)

[5.6.8.地址映射的额外收益2：cache 46](#_Toc26949229)

[5.7 start.S继续解析2 47](#_Toc26949230)

[5.7.1.使能域访问（cp15的c3寄存器） 47](#_Toc26949231)

[5.7.2.设置TTB（cp15的c2寄存器） 47](#_Toc26949232)

[5.7.3.使能MMU单元（cp15的c1寄存器） 47](#_Toc26949233)

[5.7.4.找到映射表待分析 48](#_Toc26949234)

[5.8 start.S继续解析3 48](#_Toc26949235)

[5.8.1.再三次设置栈 48](#_Toc26949236)

[5.8.2.清理bss 48](#_Toc26949237)

[5.8.3. ldr pc, \_start\_armboot 48](#_Toc26949238)

[5.8.4.总结：uboot的第一阶段做了哪些工作 49](#_Toc26949239)

[六、uboot源码分析2-启动第二阶段 50](#_Toc26949240)

[6.1.start\_armboot函数简介 50](#_Toc26949241)

[6.1.2.一个很长的函数组成uboot第二阶段 50](#_Toc26949242)

[6.1.3. 宏观分析：uboot第二阶段应该做什么 50](#_Toc26949243)

[6.1.4.思考：uboot第二阶段完结于何处？ 50](#_Toc26949244)

[6.2 start\_armboot解析1 51](#_Toc26949245)

[6.2.1. init\_fnc\_t 51](#_Toc26949246)

[6.2.2. DECLARE\_GLOBAL\_DATA\_PTR 51](#_Toc26949247)

[6.3内存使用排布 52](#_Toc26949248)

[6.3.1.为什么要分配内存 52](#_Toc26949249)

[6.3.2.内存排布 52](#_Toc26949250)

[6.3.3. uboot运行过程中的存储分布图解 52](#_Toc26949251)

[6.4 start\_armboot解析2 53](#_Toc26949252)

[6.4.1. for循环执行init\_sequence 53](#_Toc26949253)

[6.4.2. int cpu\_init(void) 55](#_Toc26949254)

[6.4.3. int board\_init(void) 55](#_Toc26949255)

[6.4.4. int interrupt\_init(void) 57](#_Toc26949256)

[6.4.5. int env\_init(void) 58](#_Toc26949257)

[6.4.6. int init\_baudrate(void) 59](#_Toc26949258)

[6.4.7. int serial\_init(void) 59](#_Toc26949259)

[6.4.8. int console\_init\_f(void) 59](#_Toc26949260)

[6.4.9. int display\_banner(void) 59](#_Toc26949261)

[6.4.10 int print\_cpuinfo(void) 60](#_Toc26949262)

[6.4.11. int checkboard(void) 60](#_Toc26949263)

[6.4.12. int init\_func\_i2c(void) 61](#_Toc26949264)

[6.4.13. int dram\_init(void) 61](#_Toc26949265)

[6.4.14. int display\_dram\_config(void) 61](#_Toc26949266)

[6.4.15. CFG\_NO\_FLASH 61](#_Toc26949267)

[6.4.16. 初始化堆管理器 mem\_malloc\_init 62](#_Toc26949268)

[6.4.17. 开发板独有初始化：mmc初始化 62](#_Toc26949269)

[6.4.18. env\_relocate 62](#_Toc26949270)

[6.4.19. IP地址、MAC地址的确定 63](#_Toc26949271)

[6.4.20. int devices\_init (void) 63](#_Toc26949272)

[6.4.21. void jumptable\_init (void) 63](#_Toc26949273)

[6.4.22. console\_init\_r () 63](#_Toc26949274)

[6.4.23. void enable\_interrupts (void) 64](#_Toc26949275)

[6.4.24. loadaddr、bootfile两个环境变量 64](#_Toc26949276)

[6.4.25. board\_late\_init (void) 64](#_Toc26949277)

[6.4.26. int eth\_initialize(bd\_t \*bis) 64](#_Toc26949278)

[6.4.27. x210\_preboot\_init(void)（LCD和logo显示） 64](#_Toc26949279)

[6.4.28. check menukey to update from sd 65](#_Toc26949280)

[6.4.29. main\_loop（uboot的归宿） 65](#_Toc26949281)

[6.4.30. 启动过程特征总结 66](#_Toc26949282)

[七、uboot源码分析3-uboot如何启动内核 66](#_Toc26949283)

[7.1 uboot和内核到底是什么 66](#_Toc26949284)

[7.1.1. uboot是一个裸机程序 66](#_Toc26949285)

[7.1.2.内核本身也是一个"裸机程序" 66](#_Toc26949286)

[7.1.3.部署在SD卡中特定分区内 66](#_Toc26949287)

[7.1.4.运行时必须先加载到DDR中链接地址处 67](#_Toc26949288)

[7.1.5.内核启动需要必要的启动参数 67](#_Toc26949289)

[7.2. 启动内核第一步：加载内核到DDR中 67](#_Toc26949290)

[7.2.1.静态内核镜像在哪里？ 67](#_Toc26949291)

[7.2.2.镜像要放在DDR的什么地址？ 67](#_Toc26949292)

[7. 3.zImage和uImage的区别联系 67](#_Toc26949293)

[7.3.1. bootm命令对应do\_bootm函数 68](#_Toc26949294)

[7.3.2. vmlinux和zImage和uImage 68](#_Toc26949295)

[7.4 zImage启动细节 69](#_Toc26949296)

[7.4.1. LINUX\_ZIMAGE\_MAGIC 69](#_Toc26949297)

[7.4.2. image\_header\_t 69](#_Toc26949298)

[7.5 uImage启动 70](#_Toc26949299)

[7.6 do\_bootm\_linux函数 70](#_Toc26949300)

[7.6.1.找到do\_bootm\_linux函数 70](#_Toc26949301)

[7.6.2.镜像的entrypoint 70](#_Toc26949302)

[7.6.3.机器码的再次确定 70](#_Toc26949303)

[7.6.4.传参并启动概述 70](#_Toc26949304)

[7.7传参详解 71](#_Toc26949305)

[7.7.1. tag方式传参 71](#_Toc26949306)

[7.7.2. x210\_sd.h中配置传参宏 71](#_Toc26949307)

[7.7.3.移植时注意事项 71](#_Toc26949308)

[7.8 uboot启动内核的总结 71](#_Toc26949309)

[八、uboot源码分析4-uboot的命令体系 72](#_Toc26949310)

[8.1 uboot命令体系基础 72](#_Toc26949311)

[8.1.1.使用uboot命令 72](#_Toc26949312)

[8.1.2. uboot命令体系实现代码在哪里 72](#_Toc26949313)

[8.1.3.每个命令对应一个函数 72](#_Toc26949314)

[8.1.4.命令参数以argc&argv传给函数 72](#_Toc26949315)

[8.2 uboot命令解析和执行过程分析 72](#_Toc26949316)

[8.2.1.从main\_loop说起 72](#_Toc26949317)

[8.2.2关键点分析 73](#_Toc26949318)

[8.3 uboot如何处理命令集1 73](#_Toc26949319)

[8.3.1.可能的管理方式 73](#_Toc26949320)

[8.3.2.命令结构体cmd\_tbl\_t 73](#_Toc26949321)

[8.3.2. uboot实现命令管理的思路 74](#_Toc26949322)

[8.4 uboot如何处理命令集2 74](#_Toc26949323)

[8.4.1. uboot命令定义具体实现分析 74](#_Toc26949324)

[8.4.2. find\_cmd函数详解 75](#_Toc26949325)

[8.5 uboot中添加自定义命令 76](#_Toc26949326)

[8.5.1.在已有的c文件中直接添加命令 76](#_Toc26949327)

[8.5.2.自建一个c文件并添加命令 76](#_Toc26949328)

[九、uboot源码分析5-uboot的环境变量 77](#_Toc26949329)

[9.1 uboot的环境变量基础 77](#_Toc26949330)

[9.1.1.环境变量的作用 77](#_Toc26949331)

[9.1.2.环境变量的优先级 77](#_Toc26949332)

[9.1.3.环境变量在uboot中工作方式 77](#_Toc26949333)

[9.2.环境变量相关命令源码解析 78](#_Toc26949334)

[9.2.1. printenv 78](#_Toc26949335)

[9.2.2. setenv 79](#_Toc26949336)

[9.2.3. saveenv 80](#_Toc26949337)

[9.2.4. getenv 80](#_Toc26949338)

[9.2.5. getenv\_r 81](#_Toc26949339)

[十、uboot源码分析6-uboot的硬件驱动部分 81](#_Toc26949340)

[10.1 uboot与linux驱动 81](#_Toc26949341)

[10.1.2. uboot的虚拟地址对硬件操作的影响 81](#_Toc26949342)

[10.1.3. uboot借用（移植）了linux驱动 81](#_Toc26949343)

[10.2 iNand/SD驱动解析 82](#_Toc26949344)

[10.2.1.从start\_armboot开始 82](#_Toc26949345)

[10.2.2. mmc\_initialize 82](#_Toc26949346)

[10.2.3. cpu\_mmc\_init 82](#_Toc26949347)

[10.2.4. smdk\_s3c\_hsmmc\_init 82](#_Toc26949348)

[10.2.5. s3c\_hsmmc\_initialize 82](#_Toc26949349)

[10.2.6. find\_mmc\_device 83](#_Toc26949350)

[10.2.7. mmc\_init 83](#_Toc26949351)

[10.2.8. 总结 83](#_Toc26949352)

[10.2.9. struct mmc 84](#_Toc26949353)

[10.2.10.分离思想 84](#_Toc26949354)

[10.2.11.分层思想 84](#_Toc26949355)

[十一、uboot的移植1-从三星官方uboot开始移植 84](#_Toc26949356)

[11.1移植前的准备工作 84](#_Toc26949357)

[11.1.1.三星移植过的uboot源代码准备 85](#_Toc26949358)

[11.1.2. SourceInsight准备 85](#_Toc26949359)

[11.1.3.便捷的文件传输工具sshsecureshell 85](#_Toc26949360)

[11.2 ubuntu14.04上网及安装openssh 85](#_Toc26949361)

[11.2.1. ubuntu14.04上网问题 85](#_Toc26949362)

[11.2.2.搭建openssh环境 86](#_Toc26949363)

[11.3 移植初体验 86](#_Toc26949364)

[11.3.1.直接编译三星移植版uboot尝试运行 86](#_Toc26949365)

[11.3.2.代码分析&问题查找 86](#_Toc26949366)

[11.4时钟和DDR的配置移植 87](#_Toc26949367)

[11.4.1.确认时钟部分的配置 87](#_Toc26949368)

[11.4.2. DDR配置信息的更改 87](#_Toc26949369)

[11.5 将DDR端口0地址配置为30000000开头 87](#_Toc26949370)

[11.4.1. DDR初始化参数更改 87](#_Toc26949371)

[11.4.2. smdkv210single.h中相关宏定义修改 88](#_Toc26949372)

[11.4.3.虚拟地址映射表中相应修改 88](#_Toc26949373)

[11.4.4.修改DMC0的配置参数 88](#_Toc26949374)

[11.4.5.修改修改虚拟地址到物理地址的映射函数 89](#_Toc26949375)

[11.5 iNand驱动问题的解决 89](#_Toc26949376)

[11.5.1.先从现象出发定位问题 89](#_Toc26949377)

[11.5.2.网络搜索解决方案 89](#_Toc26949378)

[11.5.3.尝试修改代码解决问题 89](#_Toc26949379)

[11.5.4.推测和实验验证（SD卡和iNand的区别） 90](#_Toc26949380)

[11.6 一些小问题的修补 90](#_Toc26949381)

[11.6.1控制台串口更换为串口0 90](#_Toc26949382)

[11.6.2修改默认网络地址设置 90](#_Toc26949383)

[11.6.3修改行提示符 91](#_Toc26949384)

[11.6.1总结 91](#_Toc26949385)

# 一、uboot学习前传

## 1.1为什么要有uboot

### 1.1.1.计算机的主要部件

(1)计算机系统就是由CPU来做核心进行运行的系统。典型的计算机系统有：PC机（台式机+笔记本）、嵌入式设备（手机、平板电脑、游戏机）、单片机（家用电器像电饭锅、空调）。

(2)计算机系统的组成部件非常多 ，不同计算机的组成部件也不同。但是所有的计算机系统运行时都需要的主要核心部件都是3个东西：CPU + 外部存储器（Flash/ 硬盘） + 内部存储器（DDR SDRAM/ SDRAM/ SRAM）。

### 1.1.2 .PC机的启动过程

(1)典型的PC机的部署：BIOS程序部署在PC机主板上（随主板出厂时就已经预制了），操作系统部署在硬盘上，内存在掉电时无作用，CPU在掉电时不工作。

(2)启动过程：PC上电后先执行BIOS程序（实际上PC的BIOS保存在NorFlash）,BIOS程序负责初始化DDR内存，负责初始化硬盘，然后从硬盘上将OS镜像读取到DDR中，然后跳转到DDR中去执行OS直到启动（OS启动后BIOS就无用了）。

### 1.1.3.典型嵌入式linux系统启动过程

(1)嵌入式系统的部署和启动都是参考的PC机的。只是设备上有一些差别。

(2)典型嵌入式系统的部署：uboot程序部署在Flash（能作为启动设备的Flash上），OS部署在Flash（嵌入式系统中使用了Flash代替了硬盘），内存在掉电时无作用，CPU在掉电时不工作。

(3)启动过程：嵌入式系统上电后先执行uboot，然后ubbot负责初始化DDR、初始化Flash，然后将OS从Flash中读取到DDR中，然后启动OS（OS启动后uboot无用了）。

总结：嵌入式系统和PC机的启动过程几乎没有两样，只是BIOS成了uboot，硬盘成了Flash。

### 1.1.4. android系统启动过程

(1)android系统的启动和linux系统（前面讲的典型的嵌入式系统启动）几乎一样。几乎一样意思就是前面完全一样，只是在内核启动后加载根文件系统不同了。

(2)可以认为启动分为2个阶段：第一个阶段是uboot到OS启动；第二个阶段是OS启动后到rootfs加载到命令行执行；现在我们主要研究第一个阶段，android的启动和linux的差别在第二阶段。

### 1.1.5.总结：uboot到底是干嘛的

(1)uboot主要作用是用来启动操作系统内核的。

(2)uboot还主要负责部署整个计算机系统。

(3)uboot中还有操作Flash等板子上硬盘的驱动。

(4)uboot还得提供一个命令行界面供人机交互。

## 1.2 为什么是uboot

### 1.2.1 .uboot从哪里来

(1)uboot是SourceForge上的开源项目

(2)uboot项目的作者：是由一个德国人最早发起的

(3)uboot就是由一个人发起，然后由整个网络上所有感兴趣的人共同维护发展而来的一个BootLoader

### 1.2.2 .uboot的发展历史

(1)自己使用的小开源项目。

(2)被更多人认可使用

(3)被SoC厂商默认支持。

总结：uboot经过多年发展，已经成为事实上的业内bootloader标准。现在大部分的嵌入式设备都会默认使用uboot来做为bootloader。

### 1.2.3. uboot的版本号问题

(1)早期的uboot的版本号类似于这样：uboot1.3.4。后来版本号便成了类似于uboot-2010.06。

(2)uboot的核心部分几乎没怎么变化，越新的版本支持的开发板越多而已，对于一个老版本的芯片来说，新旧版本的uboot并没有差异。

### 1.2.4. uboot的可移植性的正确理解

(1)uboot就是universal bootloader（通用的启动代码），通用的意思就是在各种地方都可以用。所以说uboot具有可移植性。

(2)uboot具有可移植性并不是说uboot在哪个开发板都可以随便用，而是说uboot具有在源代码级别的移植能力，可以针对多个开发板进行移植，移植后就可以在这个开发板上使用了。

### 1.2.5.总结：时势造英雄，任何牛逼的东西都是时代的产物

uboot的出现是一种必然，如果没有uboot也会有另一个bootloader来代替。

## 1.3 uboot必须解决哪些问题

### 1.3.1.能自身开机直接启动

(1)一般的SoC都支持多种方式启动，譬如SD卡启动、NorFlash启动、NandFlash启动等…… uboot要能够开机启动，必须根据具体的SoC的启动设计来设计uboot

(2)uboot必须进行和硬件相对应的代码级别的更改和移植，才能够保证可以从相应的启动介质启动。uboot中第一阶段的start.S文件就是具体处理了这一块。

### 1.3.2.能引导操作系统内核启动并给内核传参

(1)uboot的终极目标就是启动内核

(2)linux内核在设计的时候，设计为可以被传参。也就是说我们可以在uboot中事先给linux内核准备一些启动参数放在内存中特定的位置然后传给内核，内核启动后会到这个特定的位置去取uboot传给它的参数，然后在内核中解析这些函数，这些函数将来被用来指导linux内核的启动过程。

### 1.3.3.能提供系统部署功能

(1)uboot必须能够被人借助而完成整个系统（包括uboot、kernel、rootfs等的镜像）在Flash上的烧录下载工作。

(2)裸机中刷机就是利用uboot中的fastboot功能将各种镜像烧录到iNand中，然后从iNand启动。

### 1.3.4.能进行SoC级和板级硬件管理

(1)uboot中实现了一部分硬件的控制能力（uboot中初始化了一部分硬件），因为uboot为了完成一些任务必须让这些硬件工作。譬如uboot要实现刷机必须能驱动iNand，譬如uboot要在刷机时LCD上显示进度条就必须能驱动LCD，譬如uboot能够通过串口提供操作界面就必须驱动串口，譬如uboot要实现网络功能就必须驱动网卡芯片。

(2)SoC级（譬如串口）就是SoC内部外设，板级就是SoC外面开发板上面的硬件（譬如 网卡、iNand）

### 1.3.5.uboot的生命周期

(1)uboot的生命周期就是指：uboot什么时候开始运行，什么时候结束运行。

(2)uboot本质上是一个裸机程序（不是操作系统），一旦uboot开始SoC就会单纯运行uboot（意思就是uboot运行的时候别的程序是不可能同时运行的），一旦uboot结束运行则无法再回到uboot（所以uboot启动了内核后，uboot本身就死了，要想再次看到uboot界面只能重启系统。重启并不是复活了刚才的uboot，重启只是uboot的另一生）。

(3)uboot的入口和出口。uboot的入口就是开机自动启动，uboot的唯一出口就是启动内核。uboot还可以执行很多别的任务（譬如烧录系统），但是其他任务执行完后都可以回到uboot的命令行下继续执行uboot命令，而启动内核命令一旦执行就回不来了。

总结：一切都是为了启动内核。

## 1.4 uboot的工作方式

### 1.4.1.从裸机程序镜像uboot.bin说起

(1)uboot的本质就是一个裸机程序，而我们裸机全集中写的那些裸机程序xx.bin并没有本质区别。如果非要说有区别，那就是：我们写的大部分小于16KB，而uboot大于16KB（一般uboot在180K-400K）之间。

(2)uboot本身是一个开源项目，由若干个.c和.h文件组成，配置编译之后会生成一个uboot.bin，这就是uboot这个裸机程序的镜像文件。然后这个镜像文件被合理的烧录到启动介质中拿给SoC去启动。也就是说uboot在没有运行时表现为uboot.bin，一般躺在启动介质里。

(3)uboot运行时会被加载到内存中，然后一条指令一条指令的拿给CPU去运行。

### 1.4.2.uboot的命令行式shell界面

(1)普通的裸机程序运行起来就直接执行了，执行时效果和代码有关。

(2)有些程序需要和人进行交互，于是乎程序中就实现了一个shell（shell就是通过人机交互的一个界面），uboot就是实现了这样一个shell。

注意：shell并不是操作系统，和操作系统一点关系都没有。linux中打开一个终端后就得到一个shell，可以输入命令回车执行。uboot中的shell工作方式和linux中的终端shell非常像（几乎是一样的，只是命令集不一样。譬如linux中可以ls，而在uboot中ls是不被识别的）

### 1.4.3.掌握uboot使用的2个关键点：命令和环境变量

(1)uboot启动后大部分时间和工作都是在shell下完成的（譬如uboot要部署系统要在shell下输命令、要设置环境变量也得在命令行下，要启动内核也要在命令行下敲命令）

(2)命令就是uboot的shell中可以识别的各种命令。uboot中有几十个命令，其中有一些常用另一些不常用（我们还可以自己给uboot添加命令），后面会用几节课时间来依次学习uboot中常用命令。

(3)uboot的环境变量和操作系统的环境变量工作原理和方式几乎完全相同。uboot在设计时借助了操作系统的设计理念（命令行工作方式借鉴了linux终端命令行，环境变量借鉴了操作系统的环境变量，uboot的驱动管理几乎完全照抄了linux的驱动框架）。

(4)环境变量可以被认为是系统的全局变量，环境变量名都是系统内置的（认识就认识，不认识就不认识，这部分是系统自带的默认的环境变量，譬如PATH；但是也有一部分环境变量是自己添加的，自己添加的系统就不认识，但是我们认识）。系统或者我们自己的程序在运行时可以通过读取环境变量来指导程序的运行。这样设计的好处就是灵活，譬如我们要让一个程序更改运行方法，就不用去重新修改程序代码再重新编译运行，而只要修改相应的环境变量就可以了。

(5)环境变量就是运行时的配置属性。

### 1.4.4.思考：结合ARM裸机部分进行理解和印证

(1)及时复习ARM裸机中和现在讲到的相关的知识点

(2)及时对照原来ARM裸机中讲到的相关部分，可以帮助理解当前讲到的知识点。

(3)结合ARM裸机中和现在讲的，对比分析思考，会得到更多。

## 1.5 uboot的常用命令

### 1.5.1.类似linux终端的行缓冲命令

(1)行缓冲的意思就是：当我们向终端命令行输入命令的时候，这些命令没有立即被系统识别，而是被缓冲到一个缓存区（也就是系统认为我们还没有输入完），当我们按下回车键（换行）后系统就认为我们输入完了，然后将缓冲区中所有刚才输入的作为命令拿去分析处理。

(2)linux终端设计有3种缓冲机制：无缓冲、行缓冲、全缓冲

### 1.5.2.有些命令有简化的别名

譬如printenv命令可以简化为print，譬如setenv可以简化为set。

### 1.5.3.有些命令会带参数（注意格式是固定的）

uboot中的每个命令都有事先规定好的各种格式。有命令就是不带参数的，譬如printenv/print命令；有些命令带可选的参数（可以带也可以不带，当然带不带参数的执行结果是不同的）；有些命令带必须的参数（譬如setenv/set命令）。

### 1.5.4.命令中的特殊符号（譬如单引号）

(1)uboot的有些命令带的参数非常长，为了告诉uboot这个非常长而且中间有好多个空格的东西是给它的一个参数，所以单引号将这个很长且中间由空格隔开的参数引起来。

(2)别的符号也许也有，而且有特定的意义。当碰到uboot的命令行有特殊符号时要注意不是弄错了，而是可能有特别的意义。

### 1.5.5.有些命令是一个命令族（譬如movi）

(1)命令族的意思就是好多个命令开头都是同一个命令关键字的，但是后面的参数不一样，这些命令的功能和作用也不同。

(2)同一个命令族中所有的命令都有极大的关联，譬如movi开头的命令族都和moviNand（EMMC、iNand）操作有关。

### 1.5.6.第一个命令：printenv/print

(1)print命令是不带参命令，作用是打印出系统中所有的环境变量。

(2)环境变量就好像程序的全局变量一样。程序中任何地方都可以根据需要去调用或者更改环境变量（一般都是调用），环境变量和全局变量不同之处在于：全局变量的生命周期是在程序的一次运行当中，开始运行时诞生，程序结束时死亡，下一次运行程序时从头开始；但是环境变量被存储在Flash的另一块专门区域（Flash上有一个环境变量分区），一旦我们能在程序中保存了该环境变量，那么下次开机时该环境变量的值将维持上一次更改保存后的值。

### 1.5.7.设置（添加/更改）环境变量：setenv/set

用法：set name value

### 1.5.8.保存环境变量的更改：saveenv/save

save命令不带参数，直接执行，作用是将内存中的环境变量的值同步保存到Flash中对应的环境变量的分区。注意：环境变量的保存是整体的覆盖保存，也就是说内存中所有的环境变量都会整体的将Flash中环境变量分区中原来的内容覆盖。

总结：彻底的更改一个环境变量的值，需要2步：第一步set命令来更改内存中的环境变量，第二步用save命令将其同步到Flash中环境变量的分区。

有时候我们只是想测试下这个环境变量，不希望影响到下一次开机，那就只set不save，这样set后当前本次运行的uboot已经起效果了，只不过没save下一次开机还是会恢复到原来的状况。

### 1.5.9.网络测试指令：ping

(1)命令用法： ping ip地址

注意：ping是测试开发板和主机之间的网络链接，注意以下步骤：

1)首先要插上网线。

2)先试图ping通主机windows。注意Windows中有线网卡的地址设置（设置本地连接）。设置主机windows的本地连接IPv4地址为192.168.1.10

3)第三步确认开发板中uboot里几个网络相关的环境变量的值对不对。最重要的是ipaddr（这个环境变量表示当前开发板的IP地址），这个地址必须和主机windows的IP地址在同一个网段。

网段的概念：一个IP地址分为2部分，一部分是网段的地址，另一部分是网段内的主机地址（由子网掩码来区分哪一部分是网段地址，哪一部分是IP地址）。在子网掩码是255.255.255.0的情况下，192.168.1.10这个IP地址的前三部分（192.168.1.）属于网段地址，第4部分（10）属于主机地址。

### 1.5.10.tftp下载指令：tftp

(1)uboot本身主要目标是启动内核，为了完成启动内核必须要能够部署内核，uboot为了部署内核就需要将内核镜像从主机中下载过来然后烧录到本地flash中。uboot如何从主机（windows或者虚拟机ubuntu）下载镜像到开发板上？有很多种方式，主流方式是：fastboot和tftp。

fastboot的方式是通过USB线进行数据传输。

tftp的方式是通过有线网络的。典型的方式就是通过网络，fastboot是近些年才新发展的。

(2)tftp方式下载时实际上uboot扮演的是tftp客户端程序角色，主机windows或虚拟机ubuntu中必须有一个tftp服务器，然后将要下载的镜像文件放在服务器的下载目录中，然后开发板中使用uboot的tftp命令去下载即可。

(3)有些人习惯在windows中搭建tftp服务器，一般是用一些软件来搭建（譬如tftpd32，使用起来比较简单）；有些人习惯在linux下搭建tftp服务器，可以参考网盘中的虚拟机下载目录下的一个教程《嵌入式开发环境搭建-基于14.04.pdf》，这里面有ubuntu中搭建tftp服务器的教程，也可以自己上网搜索教程尝试。

(4)检查开发板uboot的环境变量，注意serverip必须设置为虚拟机ubuntu的ip地址。（serverip这个环境变量的意义就是主机tftp服务器的ip地址）

(5)然后在开发板的uboot下先ping通虚拟机ubuntu，然后再尝试下载：tftp 0x30000000 zImage-qt（意思是将服务器上名为zImage-qt的文件下载到开发板内存的0x30000000地址处。）

(6)镜像下载到开发板的DDR中后，uboot就可以用movi指令进行镜像的烧写了。

注意：

1）如果你是用的windows下的tftp服务器，那uboot的serverip就要设置为和windwos下tftp服务器的ip地址一样（windows下的tftp服务器软件设置的时候就有个步骤是让你设置服务器的ip地址，这个ip地址和主机windows必须在一个网段）。

2）整个过程中间环节比较多，实际做的时候可能最后会下载不下来。这时候可能的问题非常多，不要问我，自己对照视频课程讲的思路来排查。（譬如：第一步应该先保证uboot和ubuntu可以ping通；第二步再保证ubuntu中tftp服务器搭建没错；第三步再实现tftp下载。如果第一步有问题参考网络设置部分，第二步有问题（tftp本地测试下载ok，但是开发板就是不行），有一个解决方案就是使用windows下的tftp服务器）

### 1.5.11.SD卡/iNand操作指令movi

(1)开发板如果用SD卡/EMMC/iNand等作为Flash，则在uboot中操作flash的指令为movi（或mmc）

(2)movi指令是一个命令集，有很多子命令，具体用法可以help movi查看。

(3)movi的指令都是movi read和movi write一组的，movi read用来读取iNand到DDR上，movi write用来将DDR中的内容写入iNand中。理解这些指令时一定要注意涉及到的2个硬件：iNand和DDR内存。

(4)movi read {u-boot | kernel} {addr} 这个命令使用了一种通用型的描述方法来描述：movi 和 read外面没有任何标记说明每一次使用这个指令都是必选的；一对大括号{}括起来的部分必选1个，大括号中的竖线表是多选一。中括号[]表示可选参数（可以有也可以没有）

(5)指令有多种用法，譬如 movi read u-boot 0x30000000，意思就是把iNand中的u-boot分区读出到DDR的0x30000000起始的位置处。（uboot代码中将iNand分成了很多个分区，每个分区有地址范围和分区名，uboot程序操作中可以使用直接地址来操作iNand分区，也可以使用分区名来操作分区。）；注意这里的0x30000000也可以直接写作30000000，意思是一样的（uboot的命令行中所有数字都被默认当作十六进制处理，不管你加不加0x都一样）。

### 1.5.12.NandFlash操作指令nand

理解方法和操作方法完全类似于movi指令

### 1.5.13.内存操作指令：mm、mw、md

(1)DDR中是没有分区的（只听说过对硬盘、Flash进行分区，没听说过对内存进行分区……），但是内存使用时要注意，千万不能越界踩到别人了。因为uboot是一个裸机程序，不像操作系统会由系统整体管理所有内存，系统负责分配和管理，系统会保证内存不会随便越界。然而裸机程序中uboot并不管理所有内存，内存是散的随便用的，所以如果程序员（使用uboot的人）自己不注意就可能出现自己把自己的数据给覆盖了。（所以你思考下我们为什么把uboot放在23E00000地址处）

(2)md就是memory display，用来显示内存中的内容。

(3)mw就是memory write，将内容写到内存中

(4)mm就是memory modify，修改内存中的某一块，说白了还是写内存（如果需要批量的逐个单元的修改内存，用mm最合适）

1.5.14.启动内核指令：bootm、go

(1)uboot的终极目标就是启动内核，启动内核在uboot中表现为一个指令，uboot命令行中调用这个指令就会启动内核（不管成功与否，所以这个指令是一条死路）。

(2)差别：bootm启动内核同时给内核传参，而go命令启动内核不传参。bootm其实才是正宗的启动内核的命令，一般情况下都用这个；go命令本来不是专为启动内核设计的，go命令内部其实就是一个函数指针指向一个内存地址然后直接调用那个函数，go命令的实质就是PC直接跳转到一个内存地址去运行而已。go命令可以用来在uboot中执行任何的裸机程序（有一种调试裸机程序的方法就是事先启动uboot，然后在uboot中去下载裸机程序，用go命令去执行裸机程序）

## 1.6 开发板和主机的ping通

### 1.6.1.开发板运行linux下和主机Windows的ping通

(1)先将开发板刷机成linux+QT镜像，然后启动进入linux命令行终端下。

(2)在linux下使用ifconfig命令将开发板中linux系统的IP地址设置为和主机windows同一网段（为了上课方便，以后就固定：主机windows地址192.168.1.10，开发板uboot或linux的地址为192.168.1.20，虚拟机ubuntu地址为192.168.1.141）

(3)此时开发板端ping windows通的。

(4)windows中ping开发板也是通的。

说明：首先开发板和主机的网络部分硬件都是好的，网络连接也是好的，主机windows中的网络软件设置是好的。

### 1.6.2.开发板运行linux下和虚拟机ubuntu的ping通

(1)在linux基础课中讲过：虚拟机的网卡设置可以选择好几种方式，常用的就是NAT和桥接（bridged）。

(2)虚拟机要和开发板进行网络通信，只能通过桥接方式连接。

(3)虚拟机要想被开发板ping通，设置步骤如下：

第一步：虚拟机设置成桥接方式。

第二步：虚拟机的菜单中有个“虚拟网络编辑器”，这里面要设置为桥接到有线网卡。（默认是自动的，自动的一般会影响ping通。因为电脑现在一般都有2个网卡：一个有线的一个无线的。如果选了自动，那么虚拟机会自动桥接到无线网卡上，但是我们却是通过有线网卡来连接开发板的，自然ping不通）

第三步：在虚拟机ubuntu中设置IP地址为192.168.1.141（可以通过/etc/network/interfaces文件来设置静态的然后重启；也可以直接命令行ifconfig去设置）

(4)此时开发板ping虚拟机ubuntu应该就通了。

(5)此时虚拟机ubuntu中ping开发板也是通的。

### 1.6.3.开发板运行uboot下和主机Windows的ping通

(1)刚才开发板运行linux时和主机windows、虚拟机ubuntu都ping通了，说明硬件和连接和主机设置没错。

(2)此时开发板重启进入uboot，设置好ipaddr、gatewayip，然后去ping windows发现还是不通。 怀疑uboot本身网络驱动有问题。

(3)然后同样情况下尝试去ping通虚拟机ubuntu，理论分析应该也不通，但是实际发现是通的。

### 1.6.4.开发板运行uboot下和虚拟机ubuntu的ping通

uboot和虚拟机ubuntu互相ping通（前提是虚拟机ubuntu设置为桥接，且桥接到有线网卡，且ip地址设置正确的情况下）

结论：开发板中运行的uboot有点小bug，ping windows就不通，ping虚拟机ubuntu就通。

## 1.7 uboot的常用环境变量

### 1.7.1.环境变量如何参与程序运行

(1)环境变量有2份，一份在Flash中，一份在DDR中。uboot开机时一次性从Flash中读取全部环境变量到DDR中作为环境变量的初始化值，然后使用过程中都是用DDR中这一份，用户可以用saveenv指令将DDR中的环境变量重新写入Flash中去更新Flash中环境变量。

(2)环境变量在uboot中是用字符串表示的，也就是说uboot是按照字符匹配的方式来区分各个环境变量的。因此用的时候一定要注意不要打错字了。

### 1.7.2.自动运行倒计时：bootdelay

### 1.7.3.网络设置：ipaddr serverip

(1)ipaddr是开发板的本地IP地址

(2)serverip是开发板通过tftp指令去tftp服务器下载东西时，tftp服务器的IP地址。

(3)gatewayip是开发板的本地网关地址

(4)netmask是子网掩码

(5)ethaddr是开发板的本地网卡的MAC地址

### 1.7.4.自动运行命令设置：bootcmd

(1)uboot启动后会开机自动倒数bootdelay秒，如果没人按下回车打断，则uboot会自动执行启动命令来启动内核。

(2)uboot开机自动启动时实际就是在内部执行了bootcmd这个环境变量的值所对应的命令集。

(3)bootcmd=movi read kernel 30008000; bootm 30008000 意思是：将iNand的kernel分区读取到DDR内存的0x30008000地址处，然后使用bootm启动命令从内存0x30008000处去启动内核。

4)set bootcmd printenv，然后saveenv；然后重启则会看到启动倒数后自动执行printenv命令打印出环境变量。这个小实验说明开机自动执行了bootcmd。

(5)set bootcmd 'movi read kernel 30008000; bootm 30008000'

### 1.7.5.uboot给kernel传参：bootargs

(1)linux内核启动时可以接收uboot给他传递的启动参数，这些启动参数是uboot和内核约定好的形式、内容，linux内核在这些启动参数的指导下完成启动过程。这样的设计是为了灵活，为了内核在不重新编译的情况下可以用不同的方式启动。

(2)我们要做的事情就是：在uboot的环境变量中设置bootargs，然后bootm命令启动内核时会自动将bootargs传给内核。

(3)bootargs=console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3

意义解释：

console=ttySAC2,115200 控制台使用串口2，波特率115200.

root=/dev/mmcblk0p2 rw 根文件系统在SD卡端口0设备（iNand）第2分区，根文件系统是可读可写的

init=/linuxrc linux的进程1（init进程）的路径

rootfstype=ext3 根文件系统的类型是ext3

(4)内核传参非常重要。在内核移植的时候，新手经常因为忘记给内核传参，或者给内核传递的参数不对，造成内核启动不起来。

### 1.7.6.新建、更改、删除一个环境变量的方法

(1)新建一个环境变量，使用set var value

(2)更改一个环境变量，使用set var value

(3)删除一个环境变量，使用set var

### 1.7.7.注意：环境变量更改后的保存

修改完成环境变量后一定要保存，否则下次开机更改就又没了。

## 1.8 uboot中对Flash和DDR的管理

### 1.8 .1.uboot阶段Flash的分区

(1)所谓分区，就是对Flash进行分块管理。

(2)PC机等产品中，因为大家都是在操作系统下使用硬盘的，整个硬盘由操作系统统一管理，操作系统会使用文件系统帮我们管理硬盘空间。（管理保证了文件之间不会互相堆叠），于是乎使用者不用自己太过在意分区问题。

(3)在uboot中是没有操作系统的，因此我们对Flash（相当于硬盘）的管理必须事先使用分区界定（实际上在uboot中和kernel中都有个分区表，分区表就是我们在做系统移植时对Flash的整体管理分配方法）。有了这个界定后，我们在部署系统时按照分区界定方法来部署，uboot和kernel的软件中也是按照这个分区界定来工作，就不会错。

(4)分区方法不是一定的，不是固定的，是可以变动的。但是在一个移植中必须事先设计好定死，一般在设计系统移植时就会定好，定的标准是：

uboot:uboot必须从Flash起始地址开始存放（也许是扇区0，也许是扇区1，也许是其他，取决于SoC的启动设计），uboot分区的大小必须保证uboot肯定能放下，一般设计为512KB或者1MB（因为一般uboot肯定不足512KB，给再大其实也可以工作，但是浪费）；

环境变量：环境变量分区一般紧贴着uboot来存放，大小为32KB或者更多一点。

kernel：kernel可以紧贴环境变量存放，大小一般为3MB或5MB或其他。

rootfs：……

剩下的就是自由分区，一般kernel启动后将自由分区挂载到rootfs下使用

总结：一般规律如下：

(1)各分区彼此相连，前面一个分区的结尾就是后一个分区的开头。

(2)整个flash充分利用，从开头到结尾。

(3)uboot必须在Flash开头，其他分区相对位置是可变的。

(4)各分区的大小由系统移植工程师自己来定，一般定为合适大小（不能太小，太小了容易溢出；不能太大，太大了浪费空间）

(5)分区在系统移植前确定好，在uboot中和kernel中使用同一个分区表。将来在系统部署时和系统代码中的分区方法也必须一样。

### 1.8 .2.uboot阶段DDR的分区

(1)DDR的分区和Flash的分区不同，主要是因为Flash是掉电存在的，而DDR是掉电消失，因此可以说DDR是每次系统运行时才开始部署使用的。

(2)内存的分区主要是在linux内核启动起来之前，linux内核启动后内核的内存管理模块会接管整个内存空间，那时候就不用我们来管了。

(3)注意内存分区关键就在于内存中哪一块用来干什么必须分配好，以避免各个不同功能使用了同一块内存造成的互相踩踏。譬如说我们tftp 0x23E00000 zImage去下载zImage到内存的0x23E00000处就会出错，因为这个内存处实际是uboot的镜像所在。这样下载会导致下载的zImage把内存中的uboot给冲掉。

# 二、Shell和Makefile

## 2.1 Shell介绍

### 2.1.1.Shell是什么？

shell是操作系统的终端命令行，他是连接用户和操作系统内核的中介。shell本身就是一个软件。

### 2.1.2.shell概指一类编程语言（在shell中用于编写程序的语言）

(1)在shell中编写程序时使用的语言就是shell语言，又叫做脚本语言。

(2)常见的shell语言：sh、bash、csh、ksh、perl、python等。

(3)linux下常用的脚本语言就是bash、sh；以bash为主来学习。

(4)perl、python这样的高级shell脚本语言，常用在网络管理配置等领域。

(5)shell语言在嵌入式中的应用主要是做配置。

### 2.1.3.shell脚本的运行机制：解释执行（没有编译和链接的过程）

(1)shell解析器会以顺序结构逐行的解释和运行shell程序代码。

(2)CPU只认识二进制代码，而shell程序代码不做编译和链接，那么是怎样被CPU识别的呢？

shell作为user和kernel的中间商，可直接调用kernel暴露出来的接口函数，通过解释运行shell程序代码即可直接调用被隐藏起来的接口函数来和底层硬件实现交互。

例子：比如，我们都知道在 Shell 中输入cat log.txt命令就可以查看 log.txt 文件中的内容，然而，log.txt 放在磁盘的哪个位置？分成了几个数据块？在哪里开始？在哪里终止？如何操作探头读取它？这些底层细节 Shell 统统不知道的，它只能去调用内核提供的 open() 和 read() 函数，告诉内核我要读取 log.txt 文件，请帮助我，然后内核就乖乖地按照 Shell 的吩咐去读取文件了，并将读取到的文件内容交给 Shell，最后再由 Shell 呈现给用户（其实呈现到显示器上还得依赖内核）。整个过程中 Shell 就是一个“中间商”，它在用户和内核之间“倒卖”数据，只是用户不知道罢了。

## 2.2 写shell程序

### 2.2.1.文本编辑器

linux下的换行符是'\n'，而windows下的换行符是"\r\n"。

因此在windows下写的shell程序不能在linux下执行。

### 2.2.2.shell程序的运行方法

(1)./xx.sh 这样运行要求shell程序必须具有可执行权限。chmod a+x xx.sh来添加可执行权限。

(2)source xx.sh 这样运行不需要shell程序具有可执行权限。source是linux的一个命令，这个命令就是用来执行脚本程序的。

(3)bash xx.sh 这样执行相当于我们执行了bash程序，然后把xx.sh作为argv[1]传给他运行。bash是一个脚本程序解释器，本质上是一个可执行程序。

### 2.2.3.shell程序注意事项：

(1)第一行一般是：#!/bin/sh

这行话以#!开始，后面加上一个pathname，这行话的意思就是指定shell程序执行时被哪个解释器解释执行。所以我们这里写上/bin/sh意思就是这个shell将来被当前机器中/bin目录下的sh可执行程序执行。

可以将第一行写为：#!/bin/bash来指定使用bash执行该脚本。

注意：在ubuntu上面默认使用的解释器sh其实不是bash，而是dash。dash是ubuntu中默认使用的脚本解释器。

(2)脚本中的注释使用#

(3)shell程序的正文，由很多行shell语句构成。

### 2.2.4.shell不神秘

(1)shell就是把以前命令行中键入执行的命令写成了程序。shell其实就是为了避免反复的在命令行下手工输入而发明的一种把手工输入步骤记录下来，然后通过执行shell脚本程序就能再次复述原来记录的手工输入过程的一种技术。

(2)shell编辑完可以直接运行（不需编译）

## 2.3 shell编程学习

### 2.3.1.shell中的变量定义、初始化、赋值和引用

(1)变量定义。和C语言不同，在shell编程中定义变量不需要指定类型，也没有类型这个概念。

(2)变量定义时可以使用等号(=)进行初始化赋值。在shell中赋值的等号(=)两边是不能有空格的。

(3)变量定义后，可以再赋值。shell中并不刻意区分变量的定义和赋值，反正每个变量就是一个符号，这个符号的值就是最后一个给他赋值时的值。

(4)变量引用。$符号

注意：$符号后面跟一个字符串，这个字符串就会被当作变量去解析。如果这个字符串本身没有定义，执行时并不会报错，而是把这个变量解析为空。也就是说在shell中没有被定义的变量其实就相当于是一个定义并赋值为空的变量。

注意：变量引用的时候可以$var，也可以${var}。这两种的区别是在某些情况下只能用${var}而不能简单的$var。

例：

string="hello "

# 以下会输出空，即判定stringworld为定义为负值的变量，其值为空。

echo "$stringworld"

# 以下会输出hello world

echo "${string}world"

$VAR

最好VAR在定义的时候，采用VAR="abcdefg" 这种带引号的定义。

${VAR}

这种写法比较规范，VAR在定义的时候，采用VAR=abcdefg 这种不带引号的定义也没有关系。

### 2.3.2.shell中无引号、单引号、双引号的区别

(1)shell中可不加引号来使用字符串。缺点是，不能输出"或其他转义字符。

(2)shell中可加单引号来使用字符串。缺点是，不能输出转义字符。

单引号中：完全字面替换（不可包含单引号 本身）

(3)双引号：

$加变量名可以取变量的值

"\$"、"\`"、"\"" 、"\\" ，除此4种输出转义字符外，其他字符前面加\无特殊含义。

### 2.3.3.shell中调用linux命令

(1)直接执行

(2)反引号括起来执行。格式：`linux命令`，可以得到这个命令的返回值（结果值）。

PWD=`pwd`

echo ${PWD}

## 2.4 shell中的选择分支结构

(1)典型if语言格式

if [表达式]; then

xxx

yyy

zzz

else

xxx

ddd

uuu

fi

(2)if的典型应用

判断文件是否存在。（-f），注意[]里面前后都有空格，不能省略。

if [ -f a.txt ]; then

echo "yes"

else

echo "no"

touch a.txt

fi

判断目录是否存在 （-d）

if [ -d test ]; then

echo "yes"

else

echo "no"

mkdir test

fi

判断字符串是否相等（"str1" = "str2"），注意用一个等号而不是两个

if [ "str1"="str2" ]; then

echo "yes"

else

echo "no"

fi

判断数字是否相等（-eq）、大于（-gt）、小于（-lt）、大于等于（-ge）、小于等于（-le）

if [ 3 -eq 5 ]; then

echo "yes"

else

echo "no"

fi

判断字符串是否为空（-z）注意-z判断时如果变量本身没定义也是不成立（也就是说-z认为没定义不等于为空）

str=""

if [ -z $str ]; then

echo "yes"

else

echo "no"

fi

(3)if判断式中使用“-o”表示逻辑或

相当于C语言中在if后面的条件式中用逻辑与、逻辑或来连接2个式子，最终的if中是否成立取决于2个式子的逻辑运算结果。

(4)逻辑与&&和逻辑或||与简写的if表达式相结合

## 2.5 shell中的循环结构

2.5.1.for 循环

要求：能看懂、能改即可。不要求能够完全不参考写出来。因为毕竟嵌入式并不需要完全重新手写shell，系统管理员（服务器运维人员，应用层系统级管理开发的才需要完全掌握shell）

2.5.2.while 循环

(1)和C语言的循环在逻辑上无差别

(2)要注意很多格式要求，譬如：while后面的[]两边都有空格，[]后面有分号分号（如果do放在一行的话），i++的写法中有两层括号。

2.5.3.echo的创建和追加输入文件内容

(1)在shell中可以直接使用echo指令新建一个文件，并且将一些内容传入这个文件中。创建文件并输入内容的关键就是>。

echo "hello " > a.txt

(2)还可以使用echo指令配合追加符号>> 向一个已经存在的文件末尾追加输入内容。

echo "world" >> a.txt

2.5.4.shell中的case语句

(1)shell中的case语句和C语言中的switch case语句作用一样，格式有差异

(2)shell中的case语句天生没有break，也不需要break，和C语言中的switch case不同。shell中的case默认就是匹配上哪个执行哪个，不会说执行完了还去执行后面的其他case（就好像shell中的case语言默认都带了break）。

read -p "please input a number: " n

case "$n" in

1)

echo "变量是1";;

2)

echo "变量是2";;

3)

echo "变量是3";;

\*)

echo "please input a number between 1 and 3";;

exit;

esac

2.5.5.调用shell程序的传参

(1)C语言中可以通过main函数的argc和argv给程序传参（详情参考《2.8.3.argc、argv与main函数的传参》）

(2)shell程序本身也可以在调用时传参给他。在shell程序内部使用传参也是使用的一些特定符号来表示的，包括：

$#表示调用该shell时传参的个数。（$#计数时只考虑真正的参数个数）

$0、$1、$2…… 则依次表示传参的各个参数。

C语言：./a.out aa bb cc argc = 4, argv[0] = ./a.out, argv[1]是第一个有效参数····

shell：source a.sh aa bb cc $# = 3, $0是执行这个shell程序的解析程序的名字，$1是第一个有效参数的值，$2是第2个有效参数的值……

2.5.6.while循环和case语言和传参结合

(1)shell中的break关键字和C语言中意义相同（都是跳出）但是用法不同。因为shell中case语句默认不用break的，因此在shell中break只用于循环跳出。所以当while中内嵌case语句时，case中的break是跳出外层的while循环的，不是用来跳出case语句的。

(2)shell中的$# $1等内置变量的值不是不可变的，而是可以被改变，被shift指令改变。shift指令有点像左移运算符，把我们给shell程序的传参左移了一个移出去了，原来的$2变成了新的$1，原来的$#少了1个。

#!/bin/bash

#脚本名称 dir

#作用：统计当前目录下的文件夹个数和文件个数

#定义一个函数fun\_directory

fun\_directory() {

let "filenum=0"

let "dirnum=0"

for i in $( ls )

do

if [ -d $i ]

then

let dirnum+=1

else

let filenum+=1

fi

done

echo "The number of directorys is $dirnum"

echo "The number of files is $filenum"

}

#调用函数

fun\_directory

#!/bin/bash

#The shell function used to reverse the input number

echo -n "Please input the number"

read n

sd=0

rev=""

on=$n

echo "Your input is $n"

while [ $n -gt 0 ]

do

sd=$(($n % 10))

n=$(($n / 10))

rev="$rev$sd"

done

echo "The reverse number is $rev"

## 2.6 Makefile基础回顾

### 2.6.1.Makefile的作用和意义

(1)工程项目中c文件太多管理不方便，因此用Makefile来做项目管理，方便编译链接过程。

(2)uboot和linux kernel本质上都是C语言的项目，都由很多个文件组成，因此都需要通过Makefile来管理。所以要分析uboot必须对Makefile有所了解。

### 2.6.2.目标、依赖、命令

(1)目标就是我们要去make xxx的那个xxx，就是我们最终要生成的东西。

(2)依赖是用来生成目录的原材料。

(3)命令就是加工方法，所以make xxx的过程其实就是使用命令将依赖加工成目标的过程。

### 2.6.3.通配符%和Makefile自动推导（规则）

(1)%是Makefile中的通配符，代表一个或几个字母。也就是说%.o就代表所有以.o为结尾的文件。

(2)所谓自动推导其实就是Makefile的规则。当Makefile需要某一个目标时，他会把这个目标去套规则说明，一旦套上了某个规则说明，则Makefile会试图寻找这个规则中的依赖，如果能找到则会执行这个规则用依赖生成目标。

### 2.6.4.Makefile中定义和使用变量

Makefile中定义和使用变量，和shell脚本中非常相似。相似是说：都没有变量类型，直接定义使用，引用变量时用$var

### 2.6.5.伪目标（.PHONY）

(1)伪目标意思是这个目标本身不代表一个文件，执行这个目标不是为了得到某个文件或者东西，而是单纯为了执行这个目标下面的命令。

(2)伪目标一般都没有依赖，因为执行伪目标就是为了执行目标下面的命令。既然一定要执行命令，那就不必加依赖，以为不加依赖就是无条件执行。

(3)伪目标可以直接写，不影响使用；但是有时候为了明确声明这个目标是伪目标会在伪目标的前面用.PHONY来明确它是伪目标。

### 2.6.6.Makefile的文件名

Makefile的文件名合法的一般有2个，Makefile和makefile

### 2.6.7.Makefile中引用其他Makefile（include指令）

有时候Makefile总体比较复杂，因此分成好几个Makefile来写。然后在主Makefile中引用其他的，用include指令来引用。引用的效果也是原地展开，和C语言中的头文件包含非常相似。

## 2.7 Makefile补充学习

### 2.7.1.Makefile中的注释 #

### 2.7.2.命令前面的@用来静默执行

(1)在makefile的命令行中前面的@表示静默执行。

(2)Makefile中默认情况下在执行一行命令前会先把这行命令给打印出来，然后再执行这行命令。

(3)如果你不想看到命令本身，只想看到命令执行就静默执行即可。

### 2.7.3.Makefile中几种变量赋值运算符

(1)= 最简单的赋值

(2):= 一般也是赋值

以上这两个大部分情况下效果是一样的，但是有时候不一样

用”=”看前方最近的值来赋值，用”:=”看最新的值来赋值。

str1="hello"

str2="$(str1) world"

str1="fuck"

echo $str2

# 结果是hello world

str1="hello"

str2:="$(str1) world"

str1="fuck"

echo $str2

# 结果是fuck world

用:=来赋值的，则是就地直接解析，只用往前看即可。

(3)?= 如果变量前面并没有赋值过则执行这条赋值，如果前面已经赋值过了则本行被忽略。（实验可以看出：所谓的没有赋值过其实就是这个变量没有被定义过）

str="hello"

str?="fuck"

echo $str

# 结果是hello

str?="fuck"

echo $str

# 结果是fuck

(4)+= 用来给一个已经赋值的变量接续赋值，意思就是把这次的值加到原来的值的后面，有点类似于strcat。（在shell makefile等文件中，可以认为所有变量都是字符串，+=就相当于给字符串stcat接续内容）（注意一个细节，+=续接的内容和原来的内容之间会自动加一个空格隔开）

str="hello"

str+="world"

echo $str

# 结果是hello world

# hello 和 world中间的空格是自动添加的

注意：Makefile中并不要求赋值运算符两边一定要有空格或者无空格，这一点比shell的格式要求要松一些。

### 2.7.4.Makefile的环境变量

(1)makefile中用export导出的就是环境变量。一般情况下要求环境变量名用大写，普通变量名用小写。

(2)环境变量和普通变量不同，可以这样理解：环境变量类似于整个工程中所有Makefile之间可以共享的全局变量，而普通变量只是当前本Makefile中使用的局部变量。所以要注意：定义了一个环境变量会影响到工程中别的Makefile文件，因此要小心。

(3)Makefile中可能有一些环境变量可能是makefile本身自己定义的内部的环境变量或者是当前的执行环境提供的环境变量（譬如我们在make执行时给makefile传参。make CC=arm-linux-gcc，其实就是给当前Makefile传了一个环境变量CC，值是arm-linux-gcc。我们在make时给makefile传的环境变量值优先级最高的，可以覆盖makefile中的赋值）。这就好像C语言中编译器预定义的宏\_\_LINE\_\_ \_\_FUNCTION\_\_等

一样。

### 2.7.5.Makefile中的通配符

(1)\* 若干个任意字符

(2)? 1个任意字符

(3)[] 将[]中的字符依次去和外面的结合匹配

(4)还有个%，也是通配符，表示任意多个字符，和\*很相似，但是%一般只用于规则描述中，又叫做规则通配符。

关于通配符，Makefile还有一些wildcard等比较复杂的通配符用法，具体参考《跟我一起学Makefile》即可。

### 2.7.6.Makefile的自动变量

(1)为什么使用自动变量。在有些情况下文件集合中文件非常多，描述的时候很麻烦，所以我们Makefile就用一些特殊的符号来替代符合某种条件的文件集，这就形成了自动变量。

(2)自动变量的含义：预定义的特殊意义的符号。就类似于C语言编译器中预制的那些宏\_\_FILE\_\_一样。

(3)常见自动变量：

$@ 规则的目标

$< 规则的一个依赖条件

$^ 规则中的所有依赖条件



# 三、零距离体验uboot

## 3.1 X210官方uboot配置编译实践

### 3.1.1.找到官方移植好的uboot（BSP概念）

(1)原始的源代码是uboot官网下载的。这个下载的源代码可能没有你当前使用的开发板的移植，甚至找不到当前开发板使用的SoC对应的移植版本。

(2)SoC厂商在推出一款SoC后，厂商的工程师会去uboot官网下载一个uboot，根据自己的SoC进行第一步的移植，移植的目标是厂商推出的开发板。（譬如三星的S5PV210芯片厂商出的开发板就叫SMDKV210）.所以三星的工程师移植的uboot是根据他们自己的SMDKV210开发板移植的。

(3)具体的开发板供应商（譬如X210的生产商深圳市九鼎科技）首先购买三星的SMDKV210开发板，然后进行裁剪（把一些无用的接口功能裁剪去，配置降低一下，某些配置会被替换）。硬件替换和裁剪之后生成的新的开发板（譬如X210）和三星官方的SMDKV210有所不同，因此uboot也不同。但是因为SoC是相同的，所以相似度至少有60%以上。所以具体开发板供应商会以三星SMDKV210中移植的uboot为蓝本来移植得到自己的开发板的一个uboot移植。我们买X210开发板时厂商光盘中带的BSP中的uboot源码就是他移植过的。

总结：uboot可以有3种获取途径：uboot官方、SoC Mfr.官方、具体开发板Supplier的官方

### 3.1.2.在linux源生目录下配置编译

(1)X210移植过的uboot在开发板光盘的BSP中。

(2)BSP就是board support package(板级支持包，一般由开发板供应商提供)，里面的内容就是这个开发板的所有相关的源代码、文档、教程等。

(3)将整个BSP打包文件弄到linux的源生目录中去解压分析，不要在windows中的共享文件夹中解压开。（除非你的代码只在windows下去分析而不去编译，如果你想编译工程就一定不要在windows共享文件夹下，否则会出错）

(4)tar -jxvf qt\_x210v3\_130807.tar.bz2

(5)我们在linux下维持一份uboot，在windows下也维持一份uboot，在我们没有开始任何工作之前，这两份uboot内容一样的，都是九鼎官方的uboot 内容。我们这样做目的是：在linux中进行编译、在windwos下进行代码分析和观看。（windwos下有SourceInsight等很好的工具辅助我们看代码、编辑代码，在linux下编译和看代码都很麻烦……）

### 3.1.3.配置

(1)uboot和linux kernel等复杂项目，都不能直接编译，都要先配置才能编译。

(2)uboot也要先配置，配置方法是：首先cd进入uboot源码的根目录，然后在根目录下执行：make x210\_sd\_config。执行配置命令后，如果出现：Configuring for x210\_sd board...说明配置好了，如果不是这个是别的说明配置出错了。

### 3.1.4.编译得到uboot.bin

(1)编译之前一定要注意检查arm-linux-gcc对不对，检查份2步：

第一步：检查当前编译环境中有没有安装合适的arm-linux-gcc。我们装的是arm-2009q3，因为这个是三星官方、九鼎官方开发uboot时使用的。

第二步：检查当前目录下（uboot根目录）的Makefile中编译器的设置是否正确。在工程的总Makefile中会设置交叉编译工具链的路径和名字，必须确保这个路径和名字和我们自己装的一致，否则编译会出错。

(2)确保了以上2点，即可进行编译。编译很简单，直接make即可。或者可以make -j4 (多线程编译，主机如果是多核心电脑，可以尝试多线程编译，会快一些)

## 3.2 uboot的源码目录分析

### 3.2.1.九鼎官方uboot和三星原版uboot对比

(1)以九鼎官方的uboot为蓝本来学习的，以三星官方的这份为对照。

(2)不同版本的uboot或者同一版本不同人移植的uboot，可能目录结构和文件内容都有所不同。将来大家懂了后也可以自己根据需要去添加/删除/更改目录结构。

(3)九鼎在以三星的uboot为原材料进行移植时，把三星版本的uboot中很多不必要的文件夹、文件给删除掉了。这个删除把很多完全用不到的文件清除出去，减少了整体的文件数量，便于工作。

### 3.2.2.各文件介绍

(1).gitignore。git工具的文件，git是一个版本管理工具（类似的还有个svn），这个文件和git有关，和uboot本身无关的，不用去管。

(2)arm\_config.mk。后缀是.mk，是一个Makefile文件，将来在某个Makefile中会去调用它。

(3)三个Changelog文件，修改记录文件，该文件记录了这个uboot项目的版本变迁以及每个版本较上个版本修改的记录。正式的项目都有这些记录的。可以直接忽略，主要是给维护uboot的人用的。

(4)config.mk。和arm\_config.mk差不多性质。

(5)COPYING。版权声明，uboot本身是GPL许可证的。

(6)CREDITS。鸣谢，里面记录了对uboot有贡献的人，感谢目录。

(7)image\_split。一个脚本，看说明是用来分割uboot.bin到BL1的，暂时用不到，先不管。

(8)MAINTAINERS。维护者，就是当前在参与维护uboot源码的社区工作者。

(9)MAKEALL。一个脚本，应该是帮助编译uboot的。

(10)Makefile。这个很重要，是uboot源代码的主Makefile，将来整个uboot被编译时就是用这个Makefile管理编译的，所以我们在下个课程中研究uboot配置编译过程时就要分析这个Makefile。

(11)mk。快速编译的脚本，其实就是先清理然后配置然后编译而已

#!/bin/sh

make distclean

make x210\_sd\_config

make -j4

(12)mkconfig。这个很重要，是uboot配置阶段的主要配置脚本。uboot的可移植性很大程度就是靠这个配置脚本在维护的。我们在下个课程中研究uboot配置编译过程时就要分析这个配置脚本。

(13)mkmovi。暂时不去管他，一个脚本，和iNand/SD卡启动有关

(14)README。所有的软件都有README，一般拿到一个东西要先读README，这个东西其实就是个简单的使用说明书。

(15)rules.mk。这个文件是我们uboot的Makefile使用的规则，本身非常重要，但是我们不去分析他，不去看他。

总结：以上这些文件中，对我们比较重要，需要认真看的有2个：mkconfig和Makefile。一个负责uboot的配置，一个负责编译。我们在第四部分的课程中会详细讲解分析这两个东西。

### 3.2.3.各文件夹介绍

(1)api. 硬件无关的功能函数的API。uboot移植时基本不用管，这些函数是uboot本身使用的。

(2)api\_examples. API相关的测试事例代码。

(3)board。board是板的意思，板就是开发板。board文件夹下每一个文件都代表一个开发板，这个文件夹下面放的文件就是用来描述这一个开发板的信息的。board目录下有多少个文件夹，就表示当前这个uboot已经被移植到多少个开发板上了（当前的uboot支持多少个开发板）。

问题一：思考uboot如何支持多套开发板，如何具有可移植性

不同开发板的uboot不同，要使uboot具有可移植性，需要建立层级目录，明确划分厂商、型号等。

问题二：board下有这么多文件夹，究竟如何确定具体使用的是哪一个？uboot在配置阶段会有一些手段帮助我们来确定具体使用的是board目录下的哪一个文件夹。（想想为什么不能直接编译而要先配置）

因为不同board对应不同uboot，对应不同文件或文件夹，当然需要先配置

问题三：开发板越来越多，board目录下文件夹越来越多不方便管控。于是乎uboot就新增了一种机制，可以在board目录下不直接放开发板目录，而是在board下放厂家目录（vendor目录，以具体芯片厂商名字命名），然后将这个IC厂商的所有芯片开发板都丢到这个vendor目录下面去。所以大家会发现我们X210对应的开发板目录在board/samsung/x210。多了这层目录会影响配置阶段，在uboot的配置阶段要注意配置时的路径深度和实际存放要对应，不然配置后编译时找不到文件编译就会失败。注意一个细节就是历史原因造成的兼容性麻烦。最开始时board目录下就是开发板名字，后来才改成厂商名字的。但是因为要向前兼容，同一个厂商原来还是外面的开发板并没有挪移到厂商目录下面去。这样就造成后来的人不知道原委的感到很奇怪，感觉很混乱。

注意：强调一下，uboot的配置阶段（其实就是根目录下面的mkconfig脚本和Makefile中配置有关的部分）主要解决的问题就是在可移植性领域能够帮助我们确定具体的文件夹的路径，然后编译时可以找到应该找到的文件，才能编译成功。因此board目录下的不同会造成配置时的不同。做配置时要看清楚这里的目录。如果移植时没注意这里肯定要失败。

总结一句话：uboot的配置阶段主要目的是确保编译可以找到相关文件的路径！

(4)common。common是普遍的普通的，这个文件夹下放的是一些与具体硬件无关的普遍适用的一些代码。譬如控制台实现、crc校验的。但是更多的主要是两类：一类是cmd开头的，是用来实现uboot的命令系统的；另一类是env开头的，是用来实现环境变量的。

(5)cpu。这个目录是SoC相关的，里面存放的代码都是SoC相关初始化和控制代码（譬如CPU的、中断的、串口等SoC内部外设的，包括起始代码start.S也在这里）。里面很多子文件夹，每一个子文件夹就是一个SoC系列。（其实这个文件夹叫SOC最为合适，因为uboot开源所以这帮人懒得理，将错就错）这个目录是SOC片上级别（例如时钟系统、中断系统、串口控制器等）的一些初始化代码，而board目录是板级（板级的意思就是一些硬件无法集成到SOC上，因此只能体现在开发板上，例如Nand、DDR、网卡等）的一些初始化代码。

注意：这个文件是严格和硬件相关的，因此移植时也是要注意的。但是因为这个文件夹内都是SoC有关的，我们自己的开发板和三星的开发板虽然板子设计不同但是SoC都是同一个，因此实际移植时这个目录几乎不用动。

(6)disk。磁盘有关的，没研究过，没用过。

(7)doc。文档目录，里面存放了很多uboot相关文档，这些文档可以帮助我们理解uboot代码。但是因为是纯英文的，而且很杂乱，所以几乎没用。

(8)drivers。顾名思义，驱动。这里面放的就是从linux源代码中扣出来的原封不动的linux设备驱动，主要是开发板上必须用到的一些驱动，如网卡驱动、Inand/SD卡、NandFlash等的驱动。要知道：uboot中的驱动其实就是linux中的驱动，uboot在一定程度上移植了linux的驱动给自己用。但是linux是操作系统而uboot只是个裸机程序，因此这种移植会有不同，让我说：uboot中的驱动其实是linux中的驱动的一部分。

(9)examples。示例代码，没用过。

(10)fs。filesystem，文件系统。这个也是从linux源代码中移植过来的，用来管理Flash等资源。

(11)include。头文件目录。uboot和linux kernel在管理头文件时都采用了同一个思路，就是把所有的头文件全部集中存放在include目录下，而不是头文件跟着自己对应的c文件。所以在uboot中头文件包含时路径结构要在这里去找。

(12)lib\_开头的一坨。（典型的lib\_arm和lib\_generic）架构相关的库文件。譬如lib\_arm里面就是arm架构使用的一些库文件。lib\_generic里是所有架构通用的库文件。这类文件夹中的内容移植时基本不用管。

(13)libfdt。设备树有关的。linux内核在3.4左右的版本的时候更改了启动传参的机制，改用设备树来进行启动传参，进行硬件信息的描述了。

(14)nand\_spl。nand相关的，不讲。

(15)net。网络相关的代码，譬如uboot中的tftp nfs ping命令 都是在这里实现的。

(16)onenand开头的，是onenand相关的代码，是三星加的，标准uboot中应该是没有的。

(17)post。没关注过，不知道干嘛的。

(18)sd\_fusing。这里面代码实现了烧录uboot镜像到SD卡的代码。后面要仔细研究的。

(19)tools。里面是一些工具类的代码。譬如mkimage。

总结：文件夹里面比较重要的，后面会分析涉及到的有：board、common、cpu、drivers、include、lib\_arm、lib\_generic、sd\_fusing



## 3.3 SourceInsight的基本使用

### 3.3.1.为什么要使用SourceInsight

对于一个真正的一个项目，往往有几十几百甚至上万个c文件、h文件，代码量非常大，代码之间关联非常复杂，就带来一个问题，代码的分析、阅读越来越难。譬如这里引用了一个函数但是这个函数可能在另外一个目录下的一个文件夹中存放，这样情况下代码查找和阅读很困难，我们希望有一个好工具能够帮助。于是乎有了SourceInsight。

### 3.3.2.建立工程及添加文件

(1)要使用SI看代码，首先要创建一个工程。菜单栏：Project->new project。在弹出的对话框中上面输入工程名字（自己起名字，随便写，但是一般要和工程相对应免得时间长了忘记了），下面输入工程文件存放的位置。

(2)工程项目文件和工程中管理的源代码文件目录可以不同，但是我一般习惯放在一起。放在：C:\winshare\s5pv210\uboot\uboot-jiuding\SI\_Proj

点确定，进入new project setting，直接点ok进入下一步。

(3)到了向项目中添加文件的步骤。

在左侧选择uboot-jiuding这个目录，然后点右侧边栏的add tree即可添加，发现添加了1054个文件进去。

(4)本来应该已经结束了，但是有遗留问题。因为SI软件有个特点，它只能发现自己识别了的文件类型，对于它未识别的文件类型它就看不到。譬如start.S文件就未包含在内，因为SI默认不认识.S后缀的文件。

解决方案：第一种是自己配置；第二种是加载我提供的AstonSICFG。

(5)解决了4中的问题然后再次添加文件。菜单栏　Project->Add and Remove Project Files 再次浏览到uboot-jiuding目录下，再次add tree，发现添加了额外的19个文件。

### 3.3.3.解析工程文件

(1)SI工作原理就是预先把所有源代码中的所有符号全部解析存储到数据库中，然后等我们进行符号查找时，SI不是查文件而是查数据库帮我们索引符号，因此SI查找速度非常快。

(2)因此我们使用SI查阅源码前应该预先进行源码解析。在菜单栏Project->Syneronize Files，选中上面2个，然后确定。

### 3.3.4.常用技巧

# 四、uboot配置和编译过程详解

## 4.1 uboot主Makefile分析

### 4.1.1.uboot version确定（Makefile的24-29行）

(1)uboot的版本号分3个级别：

VERSION：主板本号

PATCHLEVEL：次版本号

SUBLEVEL：再次版本号

EXTRAVERSION:另外附加的版本信息

这4个用.分隔开共同构成了最终的版本号。

(2)Makefile中版本号最终生成了一个变量U\_BOOT\_VERSION，这个变量记录了Makefile中配置的版本号。

(3)include/version\_autogenerated.h文件是编译过程中自动生成的一个文件，

所以源目录中没有，但是编译过后的uboot中就有了。它里面的内容是一个宏定义，宏定义的值内容就是我们在Makefile中配置的uboot的版本号。

(4)验证方法：自己修改主Makefile中几个Version有关的变量，然后重新编译uboot，然后烧录到SD卡中，从SD卡启动，然后去看启动时uboot打印出来的版本信息，看看变化是不是和自己的分析一致。

### 4.1.2.HOSTARCH和HOSTOS

(1)HOSTARCH这个名字：HOST是主机；ARCH是architecture(架构)的缩写，表示CPU的架构。所以HOSTARCH就表示主机的CPU的架构。

(2)直接在shell中执行uname -m得到i686，得到的值其实你当前执行这个命令的电脑的CPU的版本号。

(3)shell中的 | 叫做管道，管道的作用就是把管道前面一个运算式的输出作为后面一个的输入再去做处理，最终的输出才是我们整个式子的输出。

(4)这两个环境变量是主机的操作系统和主机的CPU架构，得出后保存备用，后面自然会用到。

### 4.1.3.静默编译（50-54行）

(1)平时默认编译时命令行会打印出来很多编译信息。但是有时候我们不希望看到这些编译信息，就后台编译即可。这就叫静默编译。

(2)使用方法就是编译时make -s，-s会作为MAKEFLAGS传给Makefile，在50-54行这段代码作用下XECHO变量就会被变成空（默认等于echo），于是实现了静默编译。

### 4.1.4.两种编译方法

(1)编译复杂项目，Makefile提供2种编译管理方法。默认情况下是当前文件夹中的.c文件，编译出来的.o文件会放在同一文件夹下。这种方式叫原地编译。原地编译的好处就是处理起来简单。

(2)原地编译有一些坏处：第一，污染了源文件目录。第二的缺陷就是一套源代码只能按照一种配置和编译方法进行处理，无法同时维护2个或2个以上的配置编译方式。

（假如这套源代码有2种配置，那么编译其中一种配置后，要想编译另一种配置需要先make distclean清除第一种的编译生成文件才可以进行。）

(3)为了解决以上2种缺陷，uboot支持单独输出文件夹方式的编译（linux kernel也支持，而且uboot的这种技术就是从linux kernel偷师而来）。基本思路就是在编译时另外指定一个输出目录，将来所有的编译生成的.o文件或生成的其他文件全部丢到那个输出目录下去。源代码目录不做任何污染，这样输出目录就只是承载了本次配置编译的所有结果。

(4)具体用法：默认的就是原地编译。如果需要指定具体的输出目录编译则有2种方式来指定输出目录。（具体参考Makefile 56-76行注释内容）

第一种：make O=输出目录

第二种：export BUILD\_DIR=输出目录 然后再make

如果两个都指定了（既有BUILD\_DIR环境变量存在，又有O=xx），则O=xx具有更高优先级，听他的。

(5)两种编译的实现代码在Makefile的78-123行。

### 4.1.5.OBJTREE、SRCTREE、TOPDIR

(1)OBJTREE：编译出的.o文件存放的目录的根目录。在默认编译下，OBJTREE等于当前目录；在O=xx编译下，OBJTREE就等于我们设置的那个输出目录。

(2)SRCTREE: 源码目录，其实就是源代码的根目录，也就是当前目录。

总结：在默认编译下，OBJTREE和SRCTREE相等；在O=xx这种编译下OBJTREE和SRCTREE不相等。Makefile中定义这两个变量，其实就是为了记录编译后的.o文件往哪里放，就是为了实现O=xx的这种编译方式的。

### 4.1.6.MKCONFIG（Makefile的101行）

Makefile中定义的一个变量（在这里定义，在后面使用），它的值就是我们源码根目录下面的mkconfig。这个mkconfig是一个脚本，这个脚本就是uboot配置阶段的配置脚本。后面要用至少3节课详细讲这个配置脚本的工作。

### 4.1.7.include $(obj)include/config.mk（133行）

(1)include/config.mk不是源码自带的（你在没有编译过的源码目录下是找不到这个文件的），要在配置过程（make x210\_sd\_config）中才会生成这个文件。因此这个文件的值和我们配置过程有关，是由配置过程根据我们的配置自动生成的。

(2)我们X210在iNand情况下配置生成的config.mk内容为：

ARCH = arm

CPU = s5pc11x

BOARD = x210

VENDOR = samsung

SOC = s5pc110

(3)我们在下一行（134行）export导出了这5个变量作为环境变量。所以这两行加起来其实就是为当前Makefile定义了5个环境变量而已。之所以不直接给出这5个环境变量的值，是因为我们希望这5个值是可以被人很容易的、集中的配置的。

(4)这里的配置值来自于2589行那里的配置项。如果我们要更改这里的某个配置值要到2589行那里调用MKCONFIG脚本传参时的参数。



### 4.1.8.ARCH CROSS\_COMPILE

(1)这是两个很重要的环境变量，一个是ARCH，是上面导出的，值来自于我们的配置过程，它的值直接影响后边的CROSS\_-COMPILE的值。ARCH是用来定义当前编译的目标CPU的架构。

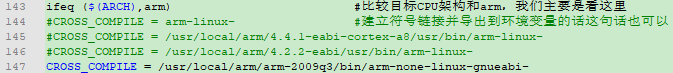
(2)CROSS\_COMPILE是定义交叉编译工具链的前缀的。

为什么要区分前后缀？

因为不同目标机架构下在linux中使用的交叉编译工具链不同，但是像gcc、ld、objcopy等后缀都是一样的，为了方便起见，将前后缀分开，只管控前缀，方便移植。

[定义这些前缀是为了在后面用（用前缀加上后缀来定义编译过程中用到的各种工具链中的工具）。我们把前缀和后缀分开还有一个原因就是：在不同CPU架构上的交叉编译工具链，只是前缀不一样，后缀都是一样的。因此定义时把前缀和后缀分开，只需要在定义前缀时区分各种架构即可实现可移植性。]

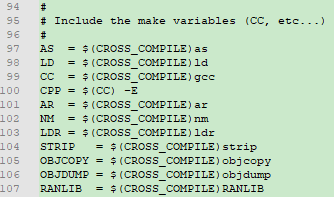
(3)CROSS\_COMPILE在136-182行来确定。CROSS\_COMPILE是被ARCH所确定的，只要配置了ARCH=arm，那么我们就只能在ARM的那个分支去设置CROSS\_COMPILE的值。这个设置值只要能保证找到那个交叉编译工具链即可，不一定非得是全路径的，相对路径也可以。（如果已经将工具链导出到环境变量，并且设置了符号链接，这样CROSS\_COMPILE = arm-linux-就可以）



(4)实际运用时，我们可以在Makefile中去更改设置CROSS\_COMPILE的值，也可以在编译时用make CROSS\_COMPILE=xxxx来设置，而且编译时传参的方法可以覆盖Makefile里面的设置。

### 4.1.9.$(TOPDIR)/config.mk（主Makefile的185行）

(1)config.mk中编译工具的定义（config.mk 94-107行）



(2)包含开发板配置项目（config.mk, 112行）

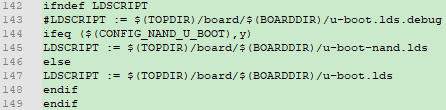
①autoconfig.mk文件不是源码提供的，是配置过程自动生成的。



②这个文件的作用就是用来指导整个uboot的编译过程。这个文件的内容其实就是很多CONFIG\_开头的宏（可以理解为变量），这些宏/变量会影响我们uboot编译过程的走向（原理就是条件编译）。在uboot代码中有很多地方使用条件编译进行编写，这个条件编译是用来实现可移植性的。（可以说uboot的源代码在很大程度来说是拼凑起来的，同一个代码包含了各种不同开发板的适用代码，用条件编译进行区别。）

③这个文件不是凭空产生的，配置过程也是需要原材料来产生这个文件的。原材料在源码目录的inlcude/configs/xxx.h头文件。（X210开发板中为include/configs/x210\_sd.h）。这个h头文件里面全都是宏定义，这些宏定义就是我们对当前开发板的移植。每一个开发板的移植都对应这个目录下的一个头文件，这个头文件里每一个宏定义都很重要，这些配置的宏定义就是我们移植uboot的关键所在。

(3)链接脚本（config.mk 142-149行）



①如果定义了CONFIG\_NAND\_U\_BOOT宏，则链接脚本叫u-boot-nand.lds，如果未定义这个宏则链接脚本叫u-boot.lds。

②从字面意思分析，即可知：CONFIG\_NAND\_U\_BOOT是在Nand版本情况下才使用的，我们使用的X210都是iNand版本的，因此这个宏没有的。

③实际在board\samsung\x210目录下有u-boot.lds，这个就是链接脚本。我们在分析uboot的编译链接过程时就要考虑这个链接脚本。

(4)TEXT\_BASE（config.mk 156-158行）

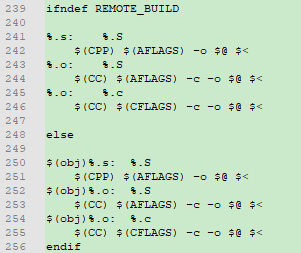


①Makefile中在配置X210开发板时，在board/samsung/x210目录下生成了一个文件config.mk，其中的内容就是：TEXT\_BASE = 0xc3e00000相当于定义了一个变量。

②TEXT\_BASE是将来我们整个uboot链接时指定的链接地址。因为uboot中启用了虚拟地址映射，因此这个0xC3E00000地址就等于0x33E00000。

(5)自动推导规则（config.mk 239-256行）

我们在讲Makefile时提到过自动推导规则，具体理解可以参考《跟我一起学Makefile》



### 4.1.10.第一个目标all（主Makefile的第291行）

(1)291行出现了整个主Makefile中第一个目标all（也就是默认目标，我们直接在uboot根目录下make其实就等于make all，就等于make这个目标）。

(2)目标中有一些比较重要的。譬如：u-boot是最终编译链接生成的elf格式的可执行文件。

(3)unconfig字面意思来理解就是未配置。这个符号用来做为我们各个开发板配置目标的依赖。目的是当我们已经配置过一个开发板后再次去配置时还可以配置。



(4)我们配置开发板时使用：make x210\_sd\_config，因此分析x210\_sd\_config肯定是主Makefile中的一个目标。

## 4.2 uboot配置过程mkconfig详解

(0)mkconfig脚本的6个参数

x210\_sd\_config : unconfig

@$(MKCONFIG) $(@:\_config=) arm s5pc11x x210 samsung s5pc110

@echo "TEXT\_BASE = 0xc3e00000" > $(obj)board/samsung/x210/config.mk

$(@:\_config=) arm s5pc11x x210 samsung s5pc110

x210\_sd\_config里的\_config部分用空替换，得到：x210\_sd，这就是第一个参数，所以：

$1: x210\_sd

$2: arm

$3: s5pc11x

$4: x210

$5: samsumg

$6: s5pc110

所以，$# = 6

1. 第23行：其实就是看BOARD\_NAME变量是否有值，如果有值就维持不变；如果无值就给他赋值为$1，实际分析结果：BOARD\_NAME=x210\_sd



(2)第25行：如果$#小于4，则exit 1（mkconfig脚本返回1）

(3)第26行：如果$#大于6，则也返回1.

所以：mkconfig脚本传参只能是4、5、6，如果大于6或者小于4都不行。



(4)[从第33行到第118行，都是在创建符号链接。为什么要创建符号链接？这些符号链接文件的存在就是整个配置过程的核心，这些符号链接文件（文件夹）的主要作用是给头文件包含等过程提供指向性连接。根本目的是让uboot具有可移植性。

uboot可移植性的实现原理：在uboot中有很多彼此平行的代码，各自属于各自不同的架构/CPU/开发板，我们在具体到一个开发板的编译时用符号连接的方式提供一个具体的名字的文件夹供编译时使用。这样就可以在配置的过程中通过不同的配置使用不同的文件，就可以正确的包含正确的文件。]

(5)创建的符号链接：

第一个：在include目录下创建asm文件，指向asm-arm。（46-48行）

第二个：在inlcude/asm-arm下创建一个arch文件，指向include/asm-arm/arch-s5pc110

第三个：在include目录下创建regs.h文件，指向include/s5pc110.h

删除第二个。

第四个：在inlcude/asm-arm下创建一个arch文件，指向include/asm-arm/arch-s5pc11x

第五个：在include/asm-arm下创建一个proc文件，指向include/asm-arm/proc-armv

总结：一共创建了4个符号链接。asm--->asm-arm、regs.h---> s5pc110.h、arch---> arch-s5pc11x、proc---> proc-armv这4个符号链接将来在写代码过程中，头文件包含时非常有用。譬如一个头文件包含可能是：#include <asm/xx.h>

如何理解创建符号链接呢？

为了在写代码时头文件包含不用动。例如#include <asm/xx.h>不用动，当更换不同架构下的目标机时，只需要让asm再创建符号链接指向其架构下的目录即可。

(6)创建include/config.mk文件（mkconfig文件123-129行）。

(7)创建include/config.mk文件是为了让主Makefile在第133行去包含的

(8)思考：uboot的配置和编译过程的配合。编译的时候需要ARCH=arm、CPU=xx等这些变量来指导编译，配置的时候就是为编译阶段提供这些变量。那为什么不在Makefile中直接定义这些变量去使用，而要在mkconfig脚本中创建config.mk文件然后又在Makefile中include这些文件呢？

这样做的目的是便于不同开发板在移植前的更改，只需要在相应配置文件配置开发板信息即可，而主Makefile不用动。试想一下，如果每次都更改Makefile，那么Makefile就硬件相关了呀！我们主要的目的就是让Makefile与硬件无直接关系，而是在配置时索取硬件信息。

这里的做法和上边创建符号链接的方法的目的是一致的，就是为了方便移植。相似之处在于，被引用的都是在配置中修改，因为这部分都是与硬件直接相关的，而引用方一动不动是王八，因为这部分是与硬件没有直接关系！

(9)理解这些脚本时，时刻要注意自己当前所处的路径。

(10)创建（默认情况）/追加（make -a时追加）include/config.h文件（mkconfig文件的134-141行）。

(11)这个文件里面的内容就一行#include <configs/x210\_sd.h>，这个头文件是我们移植x210开发板时，对开发板的宏定义配置文件。这个文件是我们移植x210时最主要的文件。

(12)x210\_sd.h文件会被用来生成一个autoconfig.mk文件，这个文件会被主Makefile引入，指导整个编译过程。这里面的这些宏定义会影响我们对uboot中大部分.c文件中一些条件编译的选择。从而实现最终的可移植性。

注意：uboot的整个配置过程，很多文件之间是有关联的（有时候这个文件是在那个文件中创建出来的；有时候这个文件被那个文件包含进去；有时候这个文件是由那个文件的内容生成的决定的）

注意：uboot中配置和编译过程，所有的文件或者全局变量都是字符串形式的（不是指的C语言字符串的概念，指的是都是字符组成的序列）。这意味着我们整个uboot的配置过程都是字符串匹配的，所以一定要细节，注意大小写，要注意不要输错字符，因为一旦错一个最后会出现一些莫名其妙的错误，很难排查，这个是uboot移植过程中新手来说最难的地方。

## 4.3 uboot的u-boot.lds链接脚本

(1)uboot的链接脚本和我们之前裸机中的链接脚本并没有本质区别，只是复杂度高一些，文件多一些，使用到的技巧多一些。

(2)ENTRY(\_start)用来指定整个程序的入口地址。所谓入口地址就是整个程序的开头地址，可以认为就是整个程序的第一句指令。有点像C语言中的main。

(3)之前在裸机中告诉大家，指定程序的链接地址有2种方法：一种是在Makefile中ld的flags用-Ttext 0x20000000来指定；第二种是在链接脚本的SECTIONS开头用.=0x20000000来指定。两种都可以实现相同效果。其实，这两种技巧是可以共同配合使用的，也就是说既在链接脚本中指定也在ld flags中用-Ttext来指定。两个都指定以后以-Ttext指定的为准。

(4)uboot的最终链接起始地址就是在Makefile中用-Ttext 来指定的，具体参见3.4.5.2节，注意TEXT\_BASE变量。最终来源是Makefile中配置对应的命令中，在make xxx\_config时得到的。

(5)在代码段中注意文件排列的顺序。指定必须放在前面部分的那些文件就是那些必须安排在前16KB内的文件，这些文件中的函数在前16KB会被调用。在后面第二部分（16KB之后）中调用的程序，前后顺序就无所谓了。

(6)链接脚本中除了.text .data .rodata .bss段等编译工具自带的段之外，编译工具还允许我们自定义段。譬如uboot总的.u\_boot\_cmd段就是自定义段。自定义段很重要。

# 五、uboot源码分析1-启动第一阶段

## 5.1 start.S的引入

### 5.1.1. u-boot.lds中找到start.S入口

(1)在C语言中整个项目的入口就是main函数（这是C语言规定的），所以譬如说一个有10000个.c文件的项目，第一个要分析的文件就是包含了main函数的那个文件。

(2)在uboot中因为有汇编阶段参与，因此不能直接找main.c。整个程序的入口取决于链接脚本中ENTRY声明的地方。ENTRY(\_start)因此\_start符号所在的文件就是整个程序的起始文件，\_start所在处的代码就是整个程序的起始代码。

### 5.1.2. SI中如何找到文件

(1)当前状况：我们知道在uboot中的1000多个文件中有一个符号叫\_start，但是我们不知道这个符号在哪个文件中。这种情况下要查找一个符号在所有项目中文件中的引用，要使用SourceInsight的搜索功能。

(2)利用SI工具搜索到一共7个\_start，然后分析搜索出来的7处，发现有2个是api\_example，2个是onenand相关的，都不是我们要找的。剩下3个都在uboot/cpu/s5pc11x/start.S文件中。

(3)然后进入start.S文件中，发现57行中就是\_start标号的定义处，于是乎我们就找到了整个uboot的入口代码，就是第57行。

### 5.1.3. SI中找文件技巧

(1)以上，找到了start.S文件，下面我们就从start.S文件开始分析uboot第一阶段。

(2)在SI中，如果我们知道我们要找的文件的名字，但是我们又不知道他在哪个目录下，我们要怎样找到并打开这个文件？方法是在SI中先打开右边的工程项目管理栏目，然后点击最左边那个（这个是以文件为单位来浏览的），然后在上面输入栏中输入要找的文件的名字。我们在输入的时候，SI在不断帮我们进行匹配，即使你不记得文件的全名只是大概记得名字，也能帮助你找到你要找的文件。

## 5.2 start.S解析

### 5.2.1.不简单的头文件包含

(1)#include <config.h>。config.h是在include目录下的，这个文件不是源码中本身存在的文件，而是配置过程中自动生成的文件。（详见mkconfig脚本）。这个文件的内容其实是包含了一个头文件：#include <configs/x210\_sd.h>".

(2)经过分析后，发现start.S中包含的第一个头文件就是：include/configs/x210\_sd.h，这个文件是整个uboot移植时的配置文件。这里面是好多宏。因此这个头文件包含将include/configs/x210\_sd.h文件和start.S文件关联了起来。因此之后在分析start.S文件时，主要要考虑的就是x210\_sd.h文件。（在start.S中后续会有很多条件编译，其判断标准很多就是ifdefined or ifndef宏定义，而这些宏定义很多恰巧都是在x210\_sd.h中配置的，x210\_sd.h是通过mkconfig创建的）

(3)#include <version.h>。include/version.h中包含了include/version\_autogenerated.h，这个头文件就是配置过程中自动生成的。里面就一行内容：#define U\_BOOT\_VERSION "U-Boot 1.3.4"。这里面定义的宏U\_BOOT\_VERSION的值是一个字符串，字符串中的版本号信息来自于Makefile中的配置值。这个宏在程序中会被调用，在uboot启动过程中会串口打印出uboot的版本号，那个版本号信息就是从这来的。（版本号是在Makefile中配置的）

(4)#include <asm/proc/domain.h>。asm目录不是uboot中的原生目录，uboot中本来是没有这个目录的。asm目录是配置时创建的一个符号链接，实际指向的是就是asm-arm（详解上一章节分析mkconfig脚本时）。

为什么要创建符号链接？ 为什么start.S不直接包含asm-arm/proc-armv/domain.h，而要用asm/proc/domain.h。这样的设计主要是为了可移植性。因为如果直接包含，则start.S文件和CPU架构（和硬件）有关了，可移植性就差了。譬如我要把uboot移植到mips架构下，则start.S源代码中所有的头文件包含全部要修改。我们用了符号链接之后，则start.S中源代码不用改，只需要在具体的硬件移植时配置不同，创建的符号链接指向的不同，则可以具有可移植性。

### 5.2.2.启动代码的16字节头部

#if defined(CONFIG\_EVT1) && !defined(CONFIG\_FUSED)

.word 0x2000

.word 0x0

.word 0x0

.word 0x0

#endif

(1)裸机中讲过，在SD卡启动/Nand启动等整个镜像开头需要16字节的校验头。（mkv210image.c中就是为了计算这个校验头）。我们以前做裸机程序时根本没考虑这16字节校验头，因为：1、如果我们是usb启动直接下载的方式启动的则不需要16字节校验头（irom application note）；2、如果是SD卡启动mkv210image.c中会给原镜像前加16字节的校验头。

(2)uboot这里start.S中在开头位置放了16字节的填充占位，这个占位的16字节只是保证正式的image的头部确实有16字节，但是这16字节的内容是不对的，还是需要后面去计算校验和然后重新填充的。

### 5.2.3.异常向量表的构建

(1)异常向量表是硬件决定的，软件只是参照硬件的设计来实现它。

(2)异常向量表中每种异常都应该被处理，否则真遇到了这种异常就跑飞了。但是我们在uboot中并未非常细致的处理各种异常。

(3)复位异常处的代码是：b reset，因此在CPU复位后真正去执行的有效代码是reset处的代码，因此reset符号处才是真正的有意义的代码开始的地方。

.globl \_start

\_start: b reset

ldr pc, \_undefined\_instruction

ldr pc, \_software\_interrupt

ldr pc, \_prefetch\_abort

ldr pc, \_data\_abort

ldr pc, \_not\_used

ldr pc, \_irq

ldr pc, \_fiq

\_undefined\_instruction:

.word undefined\_instruction

\_software\_interrupt:

.word software\_interrupt

\_prefetch\_abort:

.word prefetch\_abort

\_data\_abort:

.word data\_abort

\_not\_used:

.word not\_used

\_irq:

.word irq

\_fiq:

.word fiq

\_pad:

.word 0x12345678 /\* now 16\*4=64 \*/

.global \_end\_vect

\_end\_vect:

.balignl 16,0xdeadbeef

### 5.2.4.有点意思的deadbeef

(1).balignl 16,0xdeadbeef. 这一句指令是让当前地址对齐排布，如果当前地址不对齐则自动向后走地址直到对齐，并且向后走的那些内存用0xdeadbeef来填充。

(2)0xdeadbeef这是一个十六进制的数字，这个数字很有意思，组成这个数字的十六进制数全是abcdef之中的字母，而且这8个字母刚好组成了英文的dead beef这两个单词，字面意思是坏牛肉。

(3)为什么要对齐访问？有时候是效率的要求，有时候是硬件的特殊要求。

### 5.2.5.TEXT\_BASE等

(1)第100行这个TEXT\_BASE就是上个课程中分析Makefile时讲到的那个配置阶段的TEXT\_BASE，其实就是我们链接时指定的uboot的链接地址。（值就是0xc3e00000）



1. 这里就是在Makefile中配置的变量运送到了start.S中了！

源代码中和配置Makefile中很多变量是可以互相运送的。简单来说有些符号的值可以从Makefile中传递到源代码中。

5.2.6.（107行）CFG\_PHY\_UBOOT\_BASE

0x33e00000 uboot在DDR中的物理地址

这个物理地址其实和上方TEXT\_BASE指的是同一个地址，因为uboot中使用了虚拟地址映射，将0xc3e00000映射到物理地址就是0x33e00000。

为什么要使用虚拟地址映射呢？

原因是为了两个收益：1.实现对内存的权限控制 2.cache相关

### 5.2.8.设置CPU为SVC模式（149行）

(1)msr cpsr\_c, #0xd3 将CPU设置为禁止FIQ IRQ，ARM状态，SVC模式。

(2)其实ARM CPU在复位时默认就会进入SVC模式，但是这里还是使用软件将其置为SVC模式。整个uboot工作时CPU一直处于SVC模式。

### 5.2.9.设置L2、L1cache和MMU

(1)bl disable\_l2cache // 禁止L2 cache

(2)bl set\_l2cache\_auxctrl\_cycle // l2 cache相关初始化

(3)bl enable\_l2cache // 使能l2 cache

(4)刷新L1 cache的icache和dcache。

(5)关闭MMU

总结：上面这5步都是和CPU的cache和mmu有关的，不用去细看，大概知道即可。

### 5.2.10.识别并暂存启动介质选择

(1)从哪里启动是由SoC的OM5:OM0这6个引脚的高低电平决定的。

(2)实际上在210内部有一个寄存器（地址是0xE0000004），这个寄存器中的值是硬件根据OM引脚的设置而自动设置值的。这个值反映的就是OM引脚的接法（电平高低），也就是真正的启动介质是谁。

(3)我们代码中可以通过读取这个寄存器的值然后判断其值来确定当前选中的启动介质是Nand还是SD还是其他的。

(4)start.S的225-227行执行完后，在r2寄存器中存储了一个数字，这个数字等于某个特定值时就表示SD启动，等于另一个特定值时表示从Nand启动……

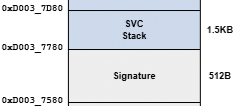


(5)260行中给r3中赋值#BOOT\_MMCSD(0x03)，这个在SD启动时实际会被执行，因此执行完这一段代码后r3中存储了0x03，以后备用。



### 5.2.11.设置栈（SRAM中的栈）并调用lowlevel\_init

(1)284-286行第一次设置栈。这次设置栈是在SRAM中设置的，因为当前整个代码还在SRAM中运行，此时DDR还未被初始化还不能用。栈地址0xd0036000是自己指定的，指定的原则就是这块空间只给栈用，不会被别人占用。



iRAM中SVC栈的分配图，讲道理应该设置在这个SVC栈里边才合适吧！

1. 在调用函数前初始化栈，主要原因是在被调用的函数内还有再次调用函数，而BL只会将返回地址存储到LR中，但是我们只有一个LR，所以在第二层调用函数前要先将LR入栈，否则函数返回时第一层的返回地址就丢了。

函数1调用函数2，将函数1的返回地址保存在LR中，压入SP栈；

函数2再调用函数3，将函数2的返回地址保存在LR中，压入SP栈；

因为栈是后入先出原则，只有一个口子，所以当执行完函数3返回函数2时，SP弹栈弹出保存在LR被压入SP的函数2的返回地址；当执行完函数2返回函数1时，SP再弹栈弹出保存在原来LR中被压入SP的函数1的返回地址。就是这样！所以初始化栈很重要，没有栈的话程序将一条死路走到黑。

## 5.3 lowlevel\_init.S解析

使用SourceInsight的Reference功能，找到lowlevel\_init函数真正的地方，是在uboot/board/samsumg/x210/lowlevel\_init.S中。

### 5.3.1.检查复位状态

(1)复杂CPU允许多种复位情况。譬如直接冷上电、热启动、睡眠(低功耗)状态下的唤醒等，这些情况都属于复位。所以我们在复位代码中要去检测复位状态，来判断到底是哪种情况。

(2)判断哪种复位的意义在于：冷上电时DDR是需要初始化才能用的；而热启动或者低功耗状态下的复位则不需要再次初始化DDR。

总结：检查复位状态的主要目的是确定在后边要不要进行DDR初始化！

### 5.3.2.IO状态恢复

这个和检查复位状态和主线启动代码都无关，因此不用去管他。

### 5.3.3.关看门狗

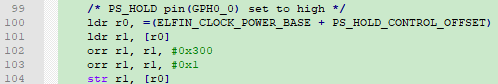
### 5.3.4.一些SRAM SROM相关GPIO设置

SROM指的是外接SRAM和SROM的部分。

与主线启动代码无关，不用管

### 5.3.5.供电锁存

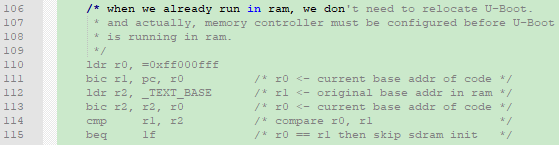
lowlevel\_init.S的第100-104行，开发板供电锁存。



总结：在前100行，lowlevel\_init.S中并没有做太多有意义的事情（除了关看门狗、供电锁存外），然后下面从110行才开始进行有意义的操作。

### 5.3.6.判断当前代码执行位置

(1)lowlevel\_init.S的110-115行。



(2)这几行代码的作用就是判定当前代码执行的位置在SRAM中还是在DDR中。为什么要做这个判定？

原因1：BL1（uboot的前一部分）在SRAM中有一份，在DDR中也有一份，因此如果是冷启动那么当前代码应该是在SRAM中运行的BL1，如果是低功耗状态的复位这时候应该就是在DDR中运行的。

原因2：我们判定当前运行代码的地址是有用的，可以指导后面代码的运行。譬如在lowlevel\_init.S中判定当前代码的运行地址，就是为了确定要不要执行时钟初始化和初始化DDR的代码。如果当前代码是在SRAM中，说明冷启动，那么时钟和DDR都需要初始化；如果当前代码是在DDR中，那么说明是热启动则时钟和DDR都不用再次初始化。

(2)bic r1, pc, r0 这句代码的意义是：将pc的值中的某些bit位清0，剩下一些特殊的bit位赋值给r1（r0中为1的那些位清零）相等于：r1 = pc & ~(ff000fff)

ldr r2, \_TEXT\_BASE 加载链接地址到r2，然后将r2的相应位清0剩下特定位。

(3)最后比较r1和r2.

总结：这一段代码是通过读取当前运行地址和链接地址，然后处理两个地址后对比是否相等，来判定当前运行是在SRAM中（不相等）还是DDR中（相等）。从而决定是否跳过下面的时钟和DDR初始化。而我们在裸机重定位章节中使用的是adr短加载地址（运行基地址）和ldr长加载地址（链接基地址）进行比较来判断代码是在哪里运行。

### 5.3.7.system\_clock\_init

(1)使用SI搜索功能，确定这个函数就在当前文件的205行，一直到第385行。这个初始化时钟的过程和裸机中初始化的过程一样的，只是更加完整而且是用汇编代码写的。

(2)在x210\_sd.h中300行到428行，都是和时钟相关的配置值。这些宏定义就决定了210的时钟配置是多少。也就是说代码在lowlevel\_init.S中都写好了，但是代码的设置值都被宏定义在x210\_sd.h中了。因此，如果移植时需要更改CPU的时钟设置，根本不需要动代码，只需要在x210\_sd.h中更改配置值即可。

### 5.3.8.mem\_ctrl\_asm\_init

(1)该函数用来初始化DDR

(2)函数位置在uboot/cpu/s5pc11x/s5pc110/cpu\_init.S文件中。

(3)该函数和裸机中初始化DDR代码是一样的。实际上裸机中初始化DDR的代码就是从这里抄的。配置值也可以从这里抄，但是当时我自己根据理解+抄袭整出来的一份。

(4)配置值中其他配置值参考裸机中的解释即可明白，有一个和裸机中讲的不一样。DMC0\_MEMCONFIG\_0，在裸机中配置值为0x20E01323；在uboot中配置为0x30F01313。这个配置不同就导致结果不同。

在裸机中DMC0的256MB内存地址范围是0x20000000-0x2FFFFFFF;

在uboot中DMC0的256MB内存地址范围为0x30000000-0x3FFFFFFF

(5)之前在裸机中时配置为2开头的地址，当时并没有说可以配置为3开头。从分析九鼎移植的uboot可以看出：DMC0上允许的地址范围是20000000-3FFFFFFF（一共是512MB），而我们实际只接了256MB物理内存，SoC允许我们给这256MB挑选地址范围。

(6)总结一下：在这份uboot中，可用的物理地址范围为：0x30000000-0x4FFFFFFF。一共512MB，其中30000000-3FFFFFFF为DMC0，40000000-4FFFFFFF为DMC1。

【疑问】在uboot中DMC0的256MB内存地址范围为0x30000000-0x3FFFFFFF。而通过USB启动，使用用DNW刷机的时候我们将uboot刷到0x23E00000处，为什么可以呢？

猜测：

因为可能在通过USB下载到iRAM的0xD0020010处的x210\_usb.bin（作为BL1）被BL0加载，然后这部分初始化了DDR的0x20000000-0x2FFFFFFF范围。再通过USB下载整个uboot到DDR的0x23E00000处，然后x210\_usb.bin（作为BL1）再重定位到DDR的0x23E00000处去运行，所以实际上uboot是在DDR的0x23E00000处运行着的。

SD卡启动的话，BL0加载SD卡上的uboot的前16KB，这时候初始化DDR的0x30000000-0x3FFFFFFF，然后BL1加载整个uboot到0x33E00000处去运行，实际上uboot是在0x33E00000处运行着。

(7)我们需要的内存配置值在x210\_sd.h的438行到468行之间。分析的时候要注意条件编译的条件，配置头文件中考虑了不同时钟配置下的内存配置值，这个的主要目的是让不同时钟需求的客户都能找到合适自己的内存配置值。

(8)在uboot中DMC0和DMC1都工作了，所以在裸机中只要把uboot中的配置值和配置代码全部移植过去，应该是能够让DMC0和DMC1都工作的。

### 5.3.9.uart\_asm\_init

(1)这个函数用来初始化串口

(2)初始化完了后通过串口发送了一个'O'

### 5.3.10.tzpc\_init

trust zone初始化，没搞过，不管

### 5.3.11.pop {pc}以返回

返回前通过串口打印'K'

分析：lowlevel\_init.S执行完如果没错那么就会串口打印出"OK"字样。这应该是我们uboot中看到的最早的输出信息。

总结回顾：lowlevel\_init.S中总共做了哪些事情：

检查复位状态、IO恢复、关看门狗、开发板供电锁存、时钟初始化、DDR初始化、串口初始化并打印'O'、tzpc初始化、打印'K'。

其中值得关注的：关看门狗、开发板供电锁存、时钟初始化、DDR初始化、打印"OK"

## 5.4 回到start.S解析

### 5.4.1.再次设置栈（DDR中的栈）

(1)再次开发板供电锁存。第一，做2次是不会错的；第二，做2次则第2次无意义；做代码移植时有一个古怪谨慎保守策略就是尽量添加代码而不要删除代码。

(2)之前在调用lowlevel\_init程序前设置过1次栈（start.S 284-287行），那时候因为DDR尚未初始化，因此程序执行都是在SRAM中，所以在SRAM中分配了一部分内存作为栈。本次因为DDR已经被初始化了，因此要把栈挪移到DDR中，所以要重新设置栈，这是第二次（start.S 297-299行）；这里实际设置的栈的地址是0x33E00000，刚好在uboot的代码段的下面紧挨着。



1. 为什么要再次设置栈？

因为DDR已经初始化了，已经有大片内存可以用了，没必要再把栈放在SRAM中可怜兮兮的了；原来SRAM中内存大小空间有限，栈放在那里要注意不能使用过多的栈否则栈会溢出，我们及时将栈迁移到DDR中也是为了尽可能避免栈使用时候的小心翼翼。

感慨：uboot的启动阶段主要技巧就在于小范围内有限条件下的辗转腾挪。

### 5.4.2.再次判断当前地址以决定是否重定位

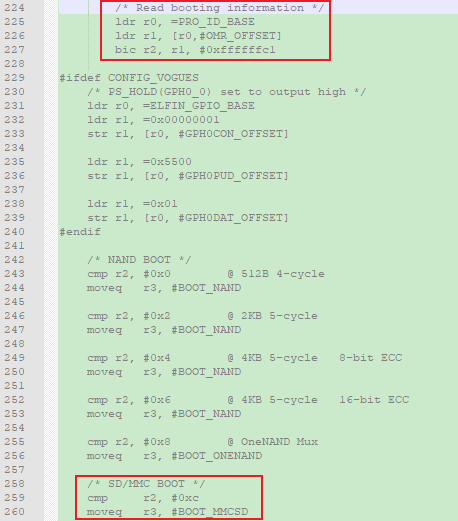
(1)再次用相同的代码判断运行地址是在SRAM中还是DDR中，不过本次判断的目的不同（上次判断是为了决定是否要执行初始化时钟和初始化DDR的代码）这次判断是为了决定是否进行uboot的relocate。

(2)冷启动时当前情况是uboot的前一部分（16kb或者8kb）开机自动从SD卡加载到SRAM中正在运行，uboot的第二部分（其实第二部分是整个uboot）还躺在SD卡的某个扇区开头的N个扇区中。此时uboot的第一阶段已经即将结束了（第一阶段该做的事基本做完了），结束之前要把第二部分加载到DDR中链接地址处（0x33e00000），这个加载过程就叫重定位。

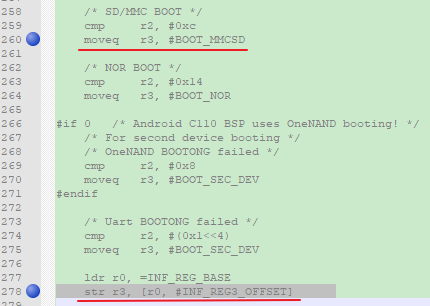
## 5.5 uboot重定位详解

(1)0xD0037488这个内存地址在SRAM中，这个地址中的值是被硬件自动设置的。硬件根据我们实际电路中SD卡在哪个通道中，会将这个地址中的值设置为相应的数字。譬如我们从SD0通道启动时，这个值为0xEB000000；从SD2通道启动时，这个值为0xEB200000

(2)我们在start.S的260行确定了从MMCSD启动，

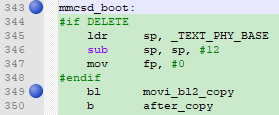


然后又在278行将#BOOT\_MMCSD写入了INF\_REG3寄存器中存储着。



然后又在322行读出来，再和#BOOT\_MMCSD去比较，确定是从MMCSD启动。最终跳转到mmcsd\_boot函数中去执行重定位动作。

(3)真正的重定位是通过调用movi\_bl2\_copy函数完成的，在uboot/cpu/s5pc11x/movi.c中。是一个C语言的函数。



(4)copy\_bl2(2, MOVI\_BL2\_POS, MOVI\_BL2\_BLKCNT,

CFG\_PHY\_UBOOT\_BASE, 0);

分析参数：2表示通道2；MOVI\_BL2\_POS是uboot的第二部分在SD卡中的开始扇区，这个扇区数字必须和烧录uboot时烧录的位置相同；MOVI\_BL2\_BLKCNT是uboot的长度占用的扇区数；CFG\_PHY\_UBOOT\_BASE是重定位时将uboot的第二部分复制到DDR中的起始地址（0x33E00000）。

## 5.6 start.S继续解析1

### 5.6.1.关于虚拟地址和物理地址

(1)物理地址就是物理设备设计生产时赋予的地址。像裸机中使用的寄存器的地址就是CPU设计时指定的，这个就是物理地址。物理地址是硬件编码的，是设计生产时确定好的，一旦确定了就不能改了。

(2)一个事实就是：寄存器的物理地址是无法通过编程修改的，是多少就是多少，只能通过查询数据手册获得并操作。一个解决方案就是使用虚拟地址。

(3)虚拟地址意思就是在我们软件操作和硬件被操作之间增加一个层次，叫做虚拟地址映射层。有了虚拟地址映射后，软件操作只需要给虚拟地址，硬件操作还是用原来的物理地址，映射层建立一个虚拟地址到物理地址的映射表。当我们软件运行的时候，软件中使用的虚拟地址在映射表中查询得到对应的物理地址再发给硬件去执行（虚拟地址到物理地址的映射是不可能通过软件来实现的）

### 5.6.2.地址映射原理

把虚拟地址划分为一定大小空间的存储块，同样，物理空间也划分为同样大小的块。然后，依照存储块的大小，可分为：1、段（1MB） 2、大页（64KB） 3、小页（4KB）4、极小页（1KB）。第一种称为段模式，后面三种称为页模式。这些映射，都是通过页表实现的，页表又可以分为：一级页表（用于段模式）和 二级页表（用于页模式）

### 5.6.3.什么是页表(转换表)呢？

页表就是存储在内存中（会被拷贝到 SDRAM 中存放，以供 MMU 查询），用于表示 VA 与 PA的映射关系的一个表格。表格中每项称为条目，条目里的内容称为描述符（段描述符和页描述符）

宏观上理解转换表：整个转换表可以看作是一个int类型的数组，数组中的一个元素就是一个表索引和表项的单元。这个元素的数组下标就是表索引，数组中的元素值就是表项。

### 5.6.4.uboot中虚拟地址映射采用了段模式

以段模式映射时，因为 VA 大小位 4GB，段模式每个条目表为 1MB 大小的空间，因此一个映射单元只能管1MB内存，所以全映射时，条目总数（全映射时页表所占内存空间） =4GB / 1MB =4096 单元（每条 32 位，即 4 字节，共 4 KB）。也就是说这个数组的元素个数是4096.实际上我们做的时候并没有依次单个处理这4096个单元，而是把4096个分成几部分，然后每部分用for循环做相同的处理。下面便是uboot的lowlevel\_init.S文件的593行中有这么一段代码，将4G空间分成了4096/256=16个部分，分别进行虚拟地址映射处理。

//uboot的lowlevel\_init.S文件的593行中有这么一段汇编代码

.set \_\_base,0 //设置初值为0

// Access for iRAM

.rept 0x100 //汇编语言循环体开始，循环0x100次，也就是256次

FL\_SECTION\_ENTRY \_\_base,3,0,0,0 //这里的\_\_base数值为0，ap值为3，d、c、b值均为0

.set \_\_base,\_\_base+1 //这里相当于C语言的i=i+1

.endr //循环体结束

对于循环体中。FL\_SECTION\_ENTRY \_\_base这个宏定义的定义方式如下：

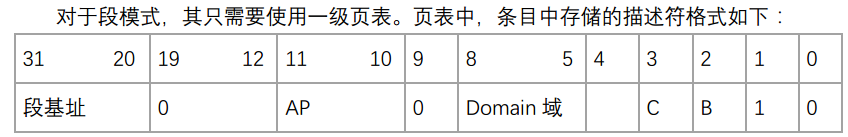
.macro FL\_SECTION\_ENTRY base,ap,d,c,b

.word (\base << 20) | (\ap << 10) | \

(\d << 5) | (1<<4) | (\c << 3) | (\b << 2) | (1<<1)

.endm

对于上面代码的位操作，通过下面这个一级页表来理解



(1)段基址： 在设计地址映射时，从20bit开始的目的是要映射的物理地址要 1MB 对齐，段基址就是这段1MB 物理地址起始地址的高[31:20]位，每个条目中的描述符的段基址都不一样（以段来说，相差 1MB）。

(2) AP： AP 是用来设置权限的，与 C1 的 R/S 位结合使用。

(3) Domain 域： 不管是段模式还是页模式，系统都把 4GB 空间分为 16 个域，每个域有相同的权限检查（在 C3 设置 ），这里的 Domain 是用来标识本段所在的域

(4) C/B： C/B 位是控制位，与本条目（描述符）所在域的 Cache 和 Buffer 有关（是否允许本域开启 Cache 和 Buffer）

(5) [1:0]=0b10： 表示本描述符是表示段模式（段描述符标识）

【补充】uboot中虚拟地址映射表对应的汇编代码，以及每一段代码完成的地址映射如下：

mmu\_table:

.set \_\_base,0

// Access for iRAM

.rept 0x100

FL\_SECTION\_ENTRY \_\_base,3,0,0,0

.set \_\_base,\_\_base+1

.endr

// Not Allowed

.rept 0x200 - 0x100

.word 0x00000000

.endr

.set \_\_base,0x200

// should be accessed

.rept 0x600 - 0x200

FL\_SECTION\_ENTRY \_\_base,3,0,1,1

.set \_\_base,\_\_base+1

.endr

.rept 0x800 - 0x600

.word 0x00000000

.endr

// should be accessed

.rept 0xb00 - 0x800

FL\_SECTION\_ENTRY \_\_base,3,0,0,0

.set \_\_base,\_\_base+1

.endr

/\*.rept 0xc00 - 0xb00

.word 0x00000000

.endr \*/

.set \_\_base,0xB00

.rept 0xc00 - 0xb00

FL\_SECTION\_ENTRY \_\_base,3,0,0,0

.set \_\_base,\_\_base+1

.endr

// 0xC000\_0000 0x2000\_0000

.set \_\_base,0x300

//.set \_\_base,0x200

// 256MB for SDRAM with cacheable

.rept 0xD00 - 0xC00

FL\_SECTION\_ENTRY \_\_base,3,0,1,1

.set \_\_base,\_\_base+1

.endr

// access is not allowed.

@.rept 0xD00 - 0xC80

@.word 0x00000000

@.endr

.set \_\_base,0xD00

// 1:1 mapping for debugging with non-cacheable

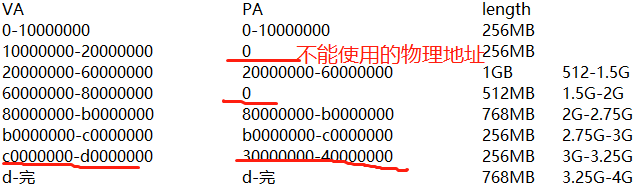
.rept 0x1000 - 0xD00

FL\_SECTION\_ENTRY \_\_base,3,0,0,0

.set \_\_base,\_\_base+1

.endr

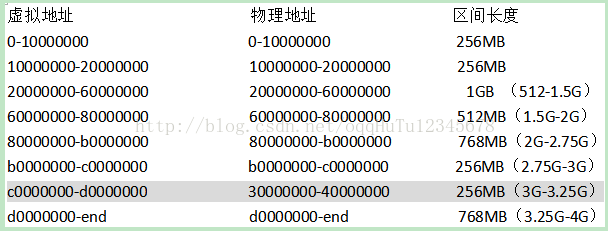
【注意】映射后的PA，如果PA数值为0，说明这段物理地址是不能使用的。



结论：虚拟地址映射只是把虚拟地址的c0000000开头的256MB映射到了DMC0的30000000开头的256MB物理内存上去了。其他的虚拟地址空间根本没动，还是原样映射的。

思考：为什么配置时将链接地址设置为c3e00000，因为这个地址将来会被映射到33e00000这个物理地址。

### 5.6.5．uboot中的映射页表



(1)把虚拟地址的以c0000000开头的256MB，映射到DMC0的以30000000开头的256MB物理内存，其他的虚拟地址空间原样映射。

(2) DRAM有效范围:

DMC0：0x30000000-0x3FFFFFFF

DMC1：0x40000000-0x4FFFFFFF

(3)链接地址c3e00000，会被映射到33e00000这个物理地址

### 5.6.6.MMU单元的作用

(1)MMU就是memory management unit，内存管理单元。MMU实际上是SOC中一个硬件单元，它的主要功能就是实现虚拟地址到物理地址的映射。

(2)MMU单片在CP15协处理器中进行控制，也就是说要操控MMU进行虚拟地址映射，方法就是对cp15协处理器的寄存器进行编程。

### 5.6.7.地址映射的额外收益1：访问控制

(1)访问控制就是：在管理上对内存进行分块，然后每块进行独立的虚拟地址映射，然后在每一块的映射关系中同时还实现了访问控制（对该块可读、可写、只读、只写、不可访问等控制）

(2)回想在C语言中编程中经常会出现一个错误：Segmentation fault。实际上这个段错误就和MMU实现的访问控制有关。当前程序只能操作自己有权操作的地址范围（若干个内存块），如果当前程序指针出错访问了不该访问的内存块则就会触发段错误。

### 5.6.8.地址映射的额外收益2：cache

(1)cache的工作和虚拟地址映射有关系。

(2)cache是快速缓存，意思就是比CPU慢但是比DDR块。CPU嫌DDR太慢了，于是乎把一些DDR中常用的内容事先读取缓存在cache中，然后CPU每次需要找东西时先在cache中找。如果cache中有就直接用cache中的；如果cache中没有才会去DDR中寻找。

## 5.7 start.S继续解析2

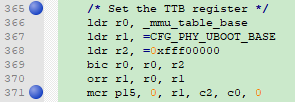
### 5.7.1.使能域访问（cp15的c3寄存器）



(1)cp15协处理器内部有c0到c15共16个寄存器，这些寄存器每一个都有自己的作用。我们通过mrc和mcr指令来访问这些寄存器。所谓的操作cp协处理器其实就是操作cp15的这些寄存器。

(2)c3寄存器在mmu中的作用是控制域访问。域访问是和MMU的访问控制有关的。

### 5.7.2.设置TTB（cp15的c2寄存器）



(1)TTB就是translation table base，转换表基地址。首先要明白什么是TT（translation table转换表），TTB其实就是转换表的基地址。

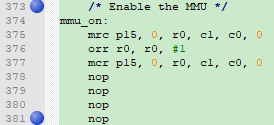
(2)转换表是建立一套虚拟地址映射的关键。转换表分2部分，表索引和表项。表索引对应虚拟地址，表项对应物理地址。一对表索引和表项构成一个转换表单元，能够对一个内存块进行虚拟地址转换。（映射中基本规定中规定了内存映射和管理是以块为单位的，至于块有多大，要看你的MMU的支持和你自己的选择。在ARM中支持3种块大小，细表1KB、粗表4KB、段1MB）。真正的转换表就是由若干个转换表单元构成的，每个单元负责1个内存块，总体的转换表负责整个内存空间（0-4G）的映射。

(3)整个建立虚拟地址映射的主要工作就是建立这张转换表

(4)转换表放置在内存中的，放置时要求起始地址在内存中要xx位对齐。转换表不需要软件去干涉使用，而是将基地址TTB设置到cp15的c2寄存器中，然后MMU工作时会自动去查转换表。

### 5.7.3.使能MMU单元（cp15的c1寄存器）

cp15的c1寄存器的bit0控制MMU的开关。只要将这一个bit置1即可开启MMU。开启MMU之后上层软件层的地址就必须经过TT的转换才能发给下层物理层去执行。



### 5.7.4.找到映射表待分析

通过SI中符号查找，确定转换表在lowlevel\_init.S文件的593行



## 5.8 start.S继续解析3

### 5.8.1.再三次设置栈

(1)第三次设置栈。这次设置栈还是在DDR中，之前虽然已经在DDR中设置过一次栈了，但是本次设置栈的目的是将栈放在比较合适（安全，紧凑而不浪费内存）的地方。

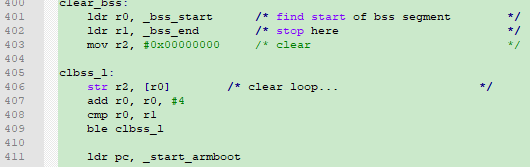
疑问，为什么不在第二次设置DDR中栈的时候就将其放在比较合适的位置啊？？

(2)我们实际将栈设置在uboot起始地址上方2MB处，这样安全的栈空间是：2MB-uboot大小-0x1000=1.8MB左右。这个空间既没有太浪费内存，又足够安全。



### 5.8.2.清理bss

清理bss段代码和裸机中讲的一样。注意表示bss段的开头和结尾地址的符号是从链接脚本u-boot.lds得来的。



### 5.8.3. ldr pc, \_start\_armboot

(1)start\_armboot是uboot/lib\_arm/board.c中，这是一个C语言实现的函数。这个函数就是uboot的第二阶段。这句代码的作用就是将uboot第二阶段执行的函数的地址传给pc，实际上就是使用一个远跳转直接跳转到DDR中的第二阶段开始地址处。

(2)远跳转的含义就是这句话加载的地址和当前运行地址无关，而和链接地址有关。因此这个远跳转可以实现从SRAM中的第一阶段跳转到DDR中的第二阶段。

(3)这里这个远跳转就是uboot第一阶段和第二阶段的分界线。

### 5.8.4.总结：uboot的第一阶段做了哪些工作

★进入uboot/cpu/s5pc11x/start.S:

1. 构建异常向量表

(2)设置CPU工作在SVC模式，ARM状态，禁止FIQ IRQ。

★(3)在SRAM设置栈并进入uboot/board/samsuang/具体开发板名字/lowlevel\_init.S:

（注，为什么要设置栈，因为在lowlevel\_init.S中还要调用别的函数，而LR只有一个）

(4)检查复位状态

直接冷上电、热启动、睡眠(低功耗)状态下的唤醒等

(5)IO状态恢复

(6)关看门狗

(7)开发板供电置锁

判断当前代码执行位置SRAM(初始化CLK,DDR)，DRAM（跳过初始化CLK,DDR）

(8)时钟初始化 system\_clock\_init

(9)内存初始化 mem\_ctrl\_asm\_init

(10)串口初始化 uart\_asm\_init 并打印“OK”

★回到uboot/cpu/s5pc11x/start.S:

(11)判断是从哪种介质启动的

(12)代码重定位 movi\_bl2\_copy

从SD卡把整个uboot重定位到DDR的指定位置（TEXT\_BASE）开始

(13)建立映射表（TTB）并打开MMU

在ARM中支持3种块大小，细表1KB、粗表4KB、段1MB，这里使用段式。

使能域访问（cp15的c3寄存器）

设置TTB（cp15的c2寄存器）

使能MMU单元（cp15的c1寄存器）

MMU相关： <https://blog.csdn.net/KayChanGEEK/article/details/50049103>

(14)清DDR的BSS段

(15)长跳转到第二阶段ldr pc, \_start\_armboot 跳转到DDR的\_start\_armboot开始运行

总结：第一阶段是在SRAM中运行的，主要是初始化CPU内部相关的，比如cache，时钟，看门狗,DDR控制器。接着跳转到第二阶段，大概就是初始化一些必要的外设，比如网卡，串口，flash等等，接着倒计时bootdelay，或者进入命令行，接收，解析，执行命令。直到执行驱动内核的命令，uboot的使命就终结了！

# 六、uboot源码分析2-启动第二阶段

## 6.1.start\_armboot函数简介

### 6.1.2.一个很长的函数组成uboot第二阶段

(1)这个函数在uboot/lib\_arm/board.c的第444行开始到908行结束。

(2)450行还不是全部，因为里面还调用了别的函数。

(3)为什么这么长的函数，怎么不分成两三个函数？主要因为这个函数整个构成了uboot启动的第二阶段。

### 6.1.3. 宏观分析：uboot第二阶段应该做什么

(1)概括来讲uboot第一阶段主要就是初始化了SoC内部的一些部件（譬如看门狗、时钟），然后初始化DDR并且完成重定位。

(2)由宏观分析来讲，uboot的第二阶段就是要初始化剩下的还没被初始化的硬件。主要是SoC外部硬件（譬如iNand、网卡芯片等……）、uboot本身的一些东西（uboot的命令、环境变量等……）。然后最终初始化完必要的东西后进入uboot的命令行准备接受命令。

### 6.1.4.思考：uboot第二阶段完结于何处？

(1)uboot启动后自动运行打印出很多信息（这些信息就是uboot在第一和第二阶段不断进行初始化时，打印出来的信息）。然后uboot进入了倒数bootdelay秒然后执行bootcmd对应的启动命令。

(2)如果用户没有干涉则会执行bootcmd进入自动启动内核流程（uboot就死掉了）；此时用户可以按下回车键打断uboot的自动启动进入uboot的命令行下。然后uboot就一直工作在命令行下。

(3)uboot的命令行就是一个死循环，循环体内不断重复：接收命令、解析命令、执行命令。这就是uboot最终的归宿。

## 6.2 start\_armboot解析1

### 6.2.1. init\_fnc\_t

(1)typedef int (init\_fnc\_t) (void); 这是一个函数类型

(2)init\_fnc\_ptr是一个二重函数指针，回顾高级C语言中讲过：二重指针的作用有2个（其中一个是用来指向一重指针），一个是用来指向指针数组。因此这里的init\_fuc\_ptr可以用来指向一个函数指针数组。

### 6.2.2. DECLARE\_GLOBAL\_DATA\_PTR



(1)#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm ("r8")

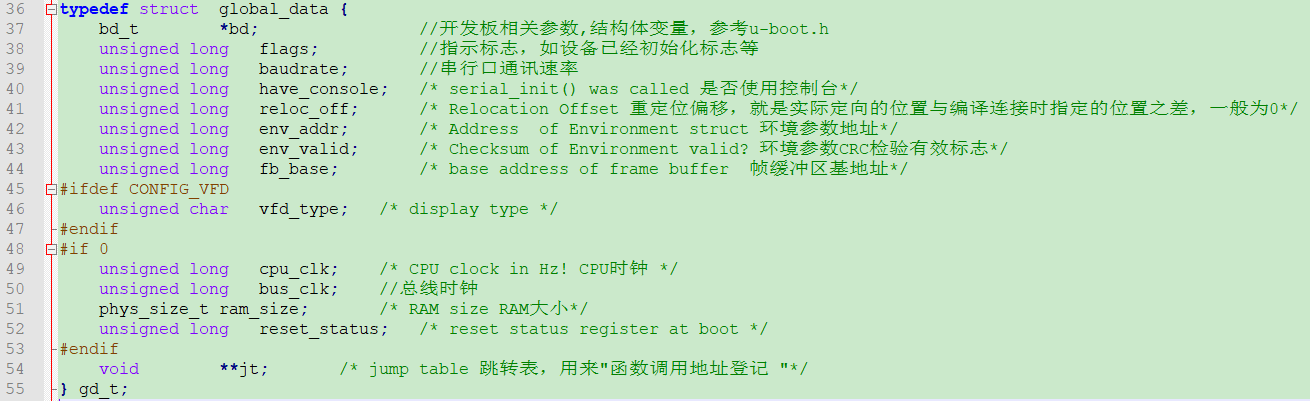
定义了一个全局变量名字叫gd,这个全局变量是一个指针类型，占4字节。用volatile修饰表示可变的，用register修饰表示这个变量要尽量放到寄存器中，后面的asm("r8")是gcc支持的一种语法，意思就是要把gd放到寄存器r8中。

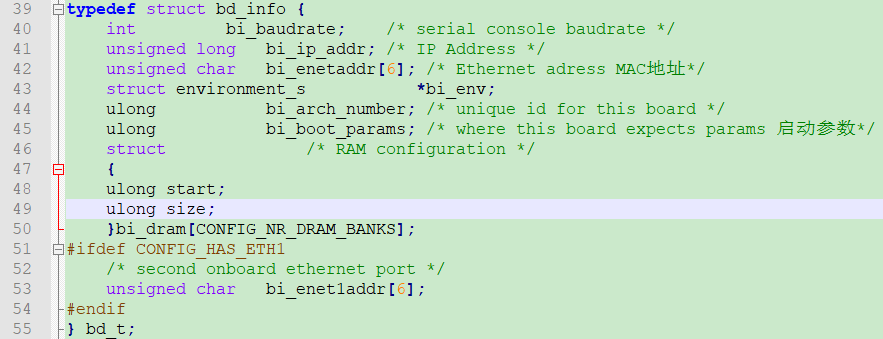
(2)综合分析，DECLARE\_GLOBAL\_DATA\_PTR就是定义了一个要放在寄存器r8中的全局变量，名字叫gd，类型是一个指向gd\_t类型变量的指针。

(3)为什么要定义为register？因为这个全局变量gd（global data的简称）是uboot中很重要的一个全局变量（准确的说这个全局变量是一个结构体，里面有很多内容，这些内容加起来构成的结构体就是uboot中常用的所有的全局变量），这个gd在程序中经常被访问，因此放在register中提升效率。因此纯粹是运行效率方面考虑，和功能要求无关。并不是必须的。

(4)gd\_t定义在include/asm-arm/global\_data.h中。

gd\_t中定义了很多全局变量，都是整个uboot使用的；其中有一个bd\_t类型的指针，指向一个bd\_t类型的变量，bd\_t的定义在include/asm-arm/u-boot.h中。这个bd是开发板的板级信息的结构体，里面有不少硬件相关的参数，譬如波特率、IP地址、机器码、DDR内存分布。





## 6.3内存使用排布

### 6.3.1.为什么要分配内存

(1)DECLARE\_GLOBAL\_DATA\_PTR只能定义了一个指针，也就是说gd里的这些全局变量并没有被分配内存，我们在使用gd之前要给他分配内存，否则gd也只是一个野指针而已。

(2)gd和bd需要内存，内存当前没有被人管理（因为没有操作系统统一管理内存），大片的DDR内存散放着可以随意使用（只要使用内存地址直接去访问内存即可）。但是因为uboot中后续很多操作还需要大片的连着内存块，因此这里使用内存要本着够用就好，紧凑排布的原则。所以我们在uboot中需要有一个整体规划。



### 6.3.2.内存排布

(1)uboot区 CFG\_UBOOT\_BASE-xx（长度为uboot的实际长度，但这里实给了2MB）

(2)堆区 长度为CFG\_MALLOC\_LEN，实际为912KB

(3)栈区 长度为CFG\_STACK\_SIZE，实际为512KB

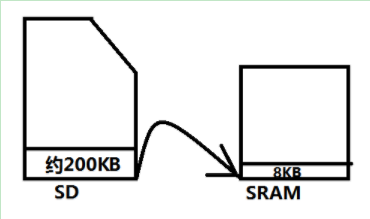
(4)gd 长度为sizeof(gd\_t)，实际36字节

(5)bd 长度为sizeof(bd\_t)，实际为44字节左右

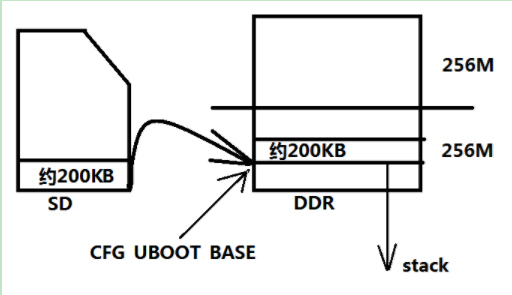
(6)内存间隔 为了防止高版本的gcc的优化造成错误。

### 6.3.3. uboot运行过程中的存储分布图解

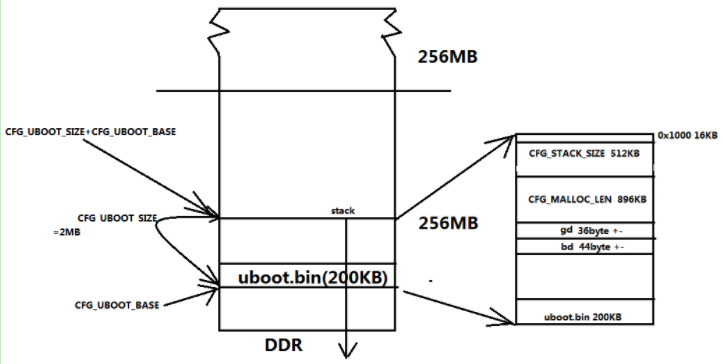
(1)上电后，从SD自拷贝8KB uboot到SRAM中运行（uboot的第一阶段）



(2) uboot被重定位到DDR，并指定栈，并跳到DDR中开始运行第二阶段

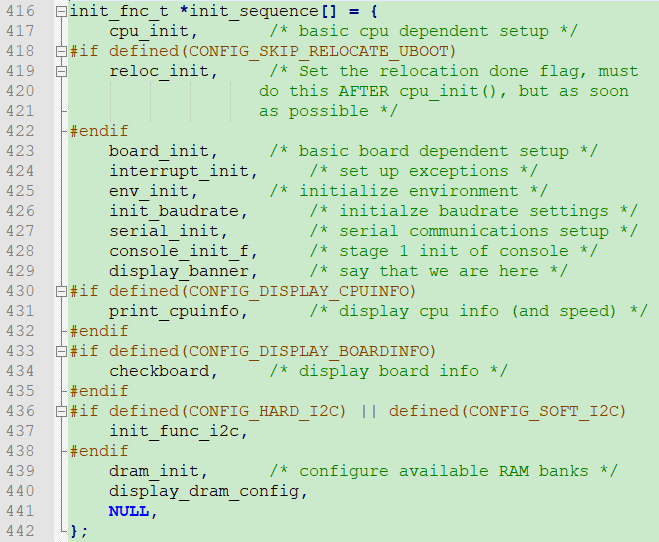


(3) 填充gd、bd，设置堆、栈



## 6.4 start\_armboot解析2

### 6.4.1. for循环执行init\_sequence



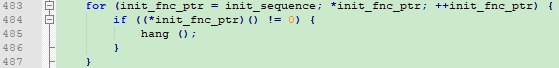
(1)init\_sequence是一个函数指针数组，数组中存储了很多个函数指针，这些指向指向的函数都是init\_fnc\_t类型（特征是接收参数是void类型，返回值是int）。

(2)init\_sequence在定义时就同时给了初始化，初始化的函数指针都是一些函数名。（C语言高级专题中讲过：函数名的实质）

(3)init\_fnc\_ptr是一个二重函数指针，可以指向init\_sequence这个函数指针数组。

(4)用for循环肯定是想要去遍历这个函数指针数组（遍历的目的也是去依次执行这个函数指针数组中的所有函数）。思考：如何遍历一个函数指针数组？有2种方法：第一种也是最常用的一种，用下标去遍历，用数组元素个数来截至。第二种不常用，但是也可以。就是在数组的有效元素末尾放一个标志，依次遍历到标准处即可截至（有点类似字符串的思路）。

我们这里使用了第二种思路。因为数组中存的全是函数指针，因此我们选用了NULL来作为标志。我们遍历时从开头依次进行，直到看到NULL标志截至。这种方法的优势是不用事先统计数组有多少个元素。



(5)init\_fnc\_t的这些函数的返回值定义方式一样的，都是：函数执行正确时返回0，不正确时返回-1.所以我们在遍历时去检查函数返回值，如果遍历中有一个函数返回值不等于0则hang()挂起。从分析hang函数可知：uboot启动过程中初始化板级硬件时不能出任何错误，只要有一个错误整个启动就终止，除了重启开发板没有任何办法。

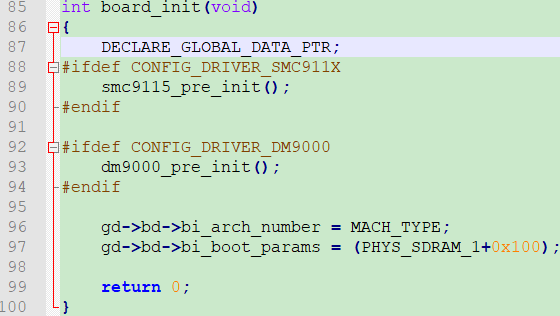


(6)init\_sequence中的这些函数，都是board级别的各种硬件初始化。

### 6.4.2. int cpu\_init(void)

看名字这个函数应该是cpu内部的初始化，如果使用了IRQ的话，里面做的是设置IRQ,FIQ的栈。否则什么也没做。

### 6.4.3. int board\_init(void)



(1)board\_init在uboot/board/samsung/x210/x210.c中，这个看名字就知道是x210开发板相关的初始化。

(2)DECLARE\_GLOBAL\_DATA\_PTR在这里声明是为了后面使用gd方便。可以看出把gd的声明定义成一个宏的原因就是我们要到处去使用gd，因此就要到处声明，定义成宏比较方便。

(3)网卡初始化。CONFIG\_DRIVER\_DM9000这个宏是x210\_sd.h中定义的，这个宏用来配置开发板的网卡的。dm9000\_pre\_init函数就是对应的DM9000网卡的初始化函数。开发板移植uboot时，如果要移植网卡，主要的工作就在这里。

(4)这个函数中主要是网卡的GPIO和端口的配置，而不是驱动。因为网卡的驱动都是现成的正确的，移植的时候驱动是不需要改动的，关键是这里的基本初始化。因为这些基本初始化是硬件相关的。

简要总结：dm9000\_pre\_init();

\*初始化自己开发板对应的网卡（主要是GPIO等，驱动是通用的。）

※ gd->bd->bi\_arch\_number

(1)bi\_arch\_number是board\_info中的一个元素，含义是：开发板的机器码。所谓机器码就是uboot给这个开发板定义的一个唯一编号。

(2)机器码的主要作用就是在uboot和linux内核之间进行比对和适配。

(3)嵌入式设备中每一个设备的硬件都是定制化的，不能通用。嵌入式设备的高度定制化导致硬件和软件不能随便适配使用。这就告诉我们：这个开发板移植的内核镜像绝对不能下载到另一个开发板去，否则也不能启动，就算启动也不能正常工作，有很多隐患。因此linux做了个设置：给每个开发板做个唯一编号（机器码），然后在uboot、linux内核中都有一个软件维护的机器码编号。然后开发板、uboot、linux三者去比对机器码，如果机器码对上了就启动，否则就不启动（因为软件认为我和这个硬件不适配）。

(4)MACH\_TYPE在x210\_sd.h中定义，值是2456，并没有特殊含义，只是当前开发板对应的编号。这个编号就代表了x210这个开发板的机器码，将来这个开发板上面移植的linux内核中的机器码也必须是2456，否则就启动不起来。

(5)uboot中配置的这个机器码，会作为uboot给linux内核的传参的一部分传给linux内核，内核启动过程中会比对这个接收到的机器码，和自己本身的机器码相对比，如果相等就启动，如果不想等就不启动。

(6)理论上来说，一个开发板的机器码不能自己随便定。理论来说有权利去发放这个机器码的只有uboot官方，所以我们做好一个开发板并且移植了uboot之后，理论上应该提交给uboot官方审核并发放机器码（好像是免费的）。但是国内的开发板基本都没有申请（主要是因为国内开发者英文都不行，和国外开源社区接触比较少），都是自己随便编号的。随便编号的问题就是有可能和别人的编号冲突，但是只要保证uboot和kernel中的编号是一致的，就不影响自己的开发板启动。

※ gd->bd->bi\_boot\_params

(1)bd\_info中另一个主要元素，bi\_boot\_params表示uboot给linux kernel启动时的传参的内存地址。也就是说uboot给linux内核传参的时候是这么传的：uboot事先将准备好的传参（字符串，就是bootargs）放在内存的一个地址处（就是bi\_boot\_params），然后uboot就启动了内核（uboot在启动内核时真正是通过寄存器r0 r1 r2来直接传递参数的，其中有一个寄存器中就是bi\_boot\_params）。内核启动后从寄存器中读取bi\_boot\_params就知道了uboot给我传递的参数到底在内存的哪里。然后自己去内存的那个地方去找bootargs。

(2)经过计算得知：X210中bi\_boot\_params的值为0x30000100，这个内存地址就被分配用来做内核传参了。所以在uboot的其他地方使用内存时要注意，千万不敢把这里给淹没了。

此处初始化来的莫名其妙！背景：关于DDR的配置：

注意：这里的初始化DDR和汇编阶段lowlevel\_init中初始化DDR是不同的。当时是硬件的初始化，目的是让DDR可以开始工作。现在是软件结构中一些DDR相关的属性配置、地址设置的初始化，是纯软件层面的。

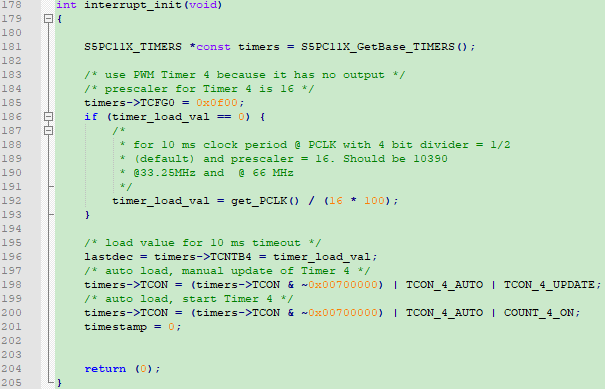
(1)软件层次初始化DDR的原因：对于uboot来说，他怎么知道开发板上到底有几片DDR内存，每一片的起始地址、长度这些信息呢？在uboot的设计中采用了一种简单直接有效的方式：程序员在移植uboot到一个开发板时，程序员自己在x210\_sd.h中使用宏定义去配置出来板子上DDR内存的信息，然后uboot只要读取这些信息即可。（实际上还有另外一条思路：就是uboot通过代码读取硬件信息来知道DDR配置，但是uboot没有这样。实际上PC的BIOS采用的是这种）

(2)x210\_sd.h的496行到501行中使用了标准的宏定义来配置DDR相关的参数。主要配置了这么几个信息：有几片DDR内存、每一片DDR的起始地址、长度。这里的配置信息我们在uboot代码中使用到内存时就可以从这里提取使用（想象uboot中使用到内存的地方都不是直接用地址数字的，都是用宏定义的）

### 6.4.4. int interrupt\_init(void)

初始化定时器。为bootdelay的倒计时打下基础。

(1)这个函数的位置在uboot/cpu/s5pc11x/interrupts.c。看名字函数是和中断初始化有关的，但是实际上不是，实际上这个函数是用来初始化定时器的（实际使用的是Timer4）。



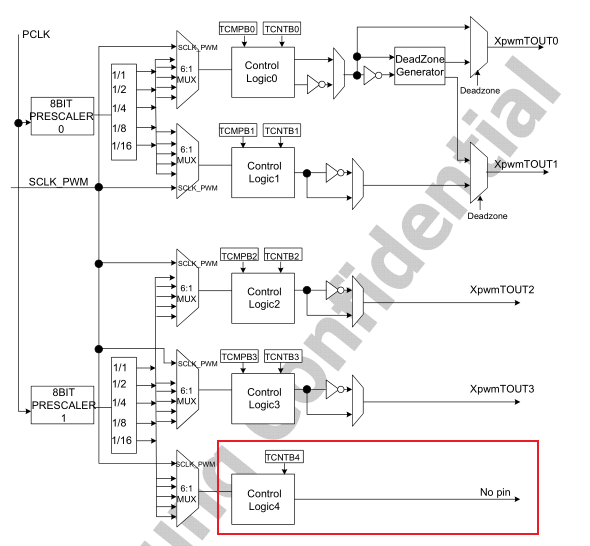
(2)裸机中讲过：210共有5个PWM定时器。其中Timer0-timer3都有一个对应的PWM信号输出的引脚。而Timer4没有引脚，无法输出PWM波形。Timer4在设计的时候就不是用来输出PWM波形的（没有引脚，没有TCMPB寄存器），这个定时器被设计用来做计时。

(3)Timer4用来做计时时要使用到2个寄存器：TCNTB4、TCNTO4。TCNTB4中存了一个数，这个数就是定时次数（每一次时间是由时钟决定的，其实就是由2级时钟分频器决定的）。我们定时时只需要把定时时间/基准时间=数，将这个数放入TCNTB中即可；我们通过TCNTO寄存器即可读取时间有没有减到0，读取到0后就知道定的时间已经到了。

(4)使用Timer4来定时，因为没有中断支持，所以CPU不能做其他事情同时定时，CPU只能使用轮询方式来不断查看TCNTO寄存器才能知道定时时间到了没。因此Timer4的定时是不能实现宏观上的并行。uboot中定时就是通过Timer4来实现定时的。所以uboot中定时时不能做其他事（考虑下，典型的就是bootdelay，bootdelay中实现定时并且检查用户输入是用轮询方式实现的，原理参考裸机中按键章节中的轮询方式处理按键）

(5)interrupt\_init函数将timer4设置为定时10ms。关键部位就是get\_PCLK函数获取系统设置的PCLK\_PSYS时钟频率，然后设置TCFG0和TCFG1进行分频，然后计算出设置为10ms时需要向TCNTB中写入的值，将其写入TCNTB，然后设置为auto reload模式，然后开定时器开始计时。

总结：在学习这个函数时，注意标准代码和之前裸机代码中的区别，重点学会：通过定义结构体的方式来访问寄存器，通过函数来自动计算设置值以设置定时器。



### 6.4.5. int env\_init(void)

环境变量的初始化，因为SOC的启动方式有多种，所以环境变量的存储方式也就有多种，所以这么有多个 env\_init 函数，比如我们是SD卡启动的，属于iNand所以选择env\_movi.c里面的int env\_init(void)函数。当前只是在内存中初始化一些环境变量必要的环境，实际上还没有从存储介质重定位到DDR中。

(1)env\_init，看名字就知道是和环境变量有关的初始化。uboot/common/env\_movi.c

(2)为什么有很多env\_init函数，主要原因是uboot支持各种不同的启动介质（譬如norflash、nandflash、inand、sd卡……），我们一般从哪里启动就会把环境变量env放到哪里。而各种介质存取操作env的方法都是不一样的。因此uboot支持了各种不同介质中env的操作方法。所以有好多个env\_xx开头的c文件。实际使用的是哪一个要根据自己开发板使用的存储介质来定（这些env\_xx.c同时只有1个会起作用，其他是不能进去的，通过x210\_sd.h中配置的宏来决定谁被包含的），对于x210来说，我们应该看env\_movi.c中的函数。

(3)经过基本分析，这个函数只是对内存里维护的那一份uboot的env做了基本的初始化或者说是判定（判定里面有没有能用的环境变量）。当前因为我们还没进行环境变量从SD卡到DDR中的relocate，因此当前环境变量是不能用的。

(4)在start\_armboot函数中（776行）调用env\_relocate才进行环境变量从SD卡中到DDR中的重定位。重定位之后需要环境变量时才可以从DDR中去取，重定位之前如果要使用环境变量只能从SD卡中去读取。

### 6.4.6. int init\_baudrate(void)

初始化串口的波特率填入gd,bd里面对应的波特率成员

gd->bd->bi\_baudrate = gd->baudrate = ···

(1)init\_baudrate看名字就是初始化串口通信的波特率的。

(2)getenv\_r函数用来读取环境变量的值。用getenv函数读取环境变量中“baudrate”的值（注意读取到的不是int型而是字符串类型），然后用simple\_strtoul函数将字符串转成数字格式的波特率。

(3)baudrate初始化时的规则是：先去环境变量中读取"baudrate"这个环境变量的值。如果读取成功则使用这个值作为环境变量，记录在gd->baudrate和gd->bd->bi\_baudrate中；如果读取不成功则使用x210\_sd.h中的的CONFIG\_BAUDRATE的值作为波特率。从这可以看出：环境变量的优先级是很高的。

### 6.4.7. int serial\_init(void)

其实没做什么实际的东西，我们在第一阶段已经初始化串口相关的硬件

(1)serial\_init看名字是初始化串口的。（疑问：start.S中调用的lowlevel\_init.S中已经使用汇编初始化过串口了，这里怎么又初始化？这两个初始化是重复的还是各自有不同？）

(2)SI中可以看出uboot中有很多个serial\_init函数，我们使用的是uboot/cpu/s5pc11x/serial.c中的serial\_init函数。

(3)进来后发现serial\_init函数其实什么都没做。因为在汇编阶段串口已经被初始化过了，因此这里就不再进行硬件寄存器的初始化了。

### 6.4.8. int console\_init\_f(void)

控制台初始化，填充gd里面的have\_console ：gd->have\_console = 1;

(1)console\_init\_f是console（控制台）的第一阶段初始化。\_f表示是第一阶段初始化，\_r表示第二阶段初始化。有时候初始化函数不能一次一起完成，中间必须要夹杂一些代码，因此将完整的一个模块的初始化分成了2个阶段。（我们的uboot中start\_armboot的826行进行了console\_init\_r的初始化）

(2)console\_init\_f在uboot/common/console.c中，仅仅是对gd->have\_console设置为1而已，其他事情都没做。

### 6.4.9. int display\_banner(void)

显示uboot的信息

(1)display\_banner用来串口输出显示uboot的logo

(2)display\_banner中使用printf函数向串口输出了version\_string这个字符串。那么上面的分析表示console\_init\_f并没有初始化好console怎么就可以printf了呢？

(3)通过追踪printf的实现，发现printf->puts，而puts函数中会判断当前uboot中console有没有被初始化好。如果console初始化好了则调用fputs完成串口发送（这条线才是控制台）；如果console尚未初始化好则会调用serial\_puts(再调用serial\_putc直接操作串口寄存器进行内容发送)。

(4)控制台也是通过串口输出，非控制台也是通过串口输出。究竟什么是控制台？和不用控制台的区别？实际上分析代码会发现，控制台就是一个用软件虚拟出来的设备，这个设备有一套专用的通信函数（发送、接收……），控制台的通信函数最终会映射到硬件的通信函数中来实现。uboot中实际上控制台的通信函数是直接映射到硬件串口的通信函数中的，也就是说uboot中用没用控制器其实并没有本质差别。

(5)但是在别的体系中，控制台的通信函数映射到硬件通信函数时可以用软件来做一些中间优化，譬如说缓冲机制。（操作系统中的控制台都使用了缓冲机制，所以有时候我们printf了内容但是屏幕上并没有看到输出信息，就是因为被缓冲了。我们输出的信息只是到了console的buffer中，buffer还没有被刷新到硬件输出设备上，尤其是在输出设备是LCD屏幕时）

(6)U\_BOOT\_VERSION在uboot源代码中找不到定义，这个变量实际上是在makefile中定义的，然后在编译时生成的include/version\_autogenerated.h中用一个宏定义来实现的。

printf ("\n\n%s\n\n", version\_string);

const char version\_string[] =

U\_BOOT\_VERSION" (" \_\_DATE\_\_ " - " \_\_TIME\_\_ ")"CONFIG\_IDENT\_STRING;

### 6.4.10 int print\_cpuinfo(void)

主要是自动获取开发板配置的时钟系统的情况并打印到串口

(1)uboot启动过程中：

CPU: S5PV210@1000MHz(OK)

APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz

MPLL = 667MHz, EPLL = 96MHz

HclkDsys = 166MHz, PclkDsys = 83MHz

HclkPsys = 133MHz, PclkPsys = 66MHz

SCLKA2M = 200MHz

Serial = CLKUART

这些信息都是print\_cpuinfo打印出来的。

(2)回顾ARM裸机中时钟配置一章的内容，比对这里调用的函数中计算各种时钟的方法，自己去慢慢分析体会这些代码的原理和实现方法。这就是学习。

### 6.4.11. int checkboard(void)

主要是打印开发板的名字信息（可以自己再次添加你想打印的）

printf("\nBoard: Luffy210\n");

(1)checkboard看名字是检查、确认开发板的意思。这个函数的作用就是检查当前开发板是哪个开发板并且打印出开发板的名字。

### 6.4.12. int init\_func\_i2c(void)

初始化IIC并打印

(1)这个函数实际没有被执行，X210的uboot中并没有使用I2C。如果将来我们的开发板要扩展I2C来接外接硬件，则在x210\_sd.h中配置相应的宏即可开启。

### 6.4.13. int dram\_init(void)

DDR的软件初始化，主要是填充bd里面和DDR相关的结构体成员

（几片DDR，起始地址，容量大小）

gd->bd->bi\_dram[0].start = PHYS\_SDRAM\_1;

gd->bd->bi\_dram[0].size = PHYS\_SDRAM\_1\_SIZE;

(1)dram\_init看名字是关于DDR的初始化。疑问：在汇编阶段已经初始化过DDR了否则也无法relocate到第二部分运行，怎么在这里又初始化DDR？

(2)dram\_init都是在给gd->bd里面关于DDR配置部分的全局变量赋值，让gd->bd数据记录下当前开发板的DDR的配置信息，以便uboot中使用内存。

(3)从代码来看，其实就是初始化gd->bd->bi\_dram这个结构体数组。

### 6.4.14. int display\_dram\_config(void)

主要是打印DDR的总大小：

puts("DRAM: ");

print\_size(size, "\n");

也可以在uboot命令行里面使用命令bdinfo来获取DRAM的详细配置信息。

(1)看名字意思就是打印显示dram的配置信息。

(2)启动信息中的：（DRAM: 512 MB）就是在这个函数中打印出来的。

(3)思考：如何在uboot运行中得知uboot的DDR配置信息？uboot中有一个命令叫bdinfo，这个命令可以打印出gd->bd中记录的所有硬件相关的全局变量的值，因此可以得知DDR的配置信息。

DRAM bank = 0x00000000

-> start = 0x30000000

-> size = 0x10000000

DRAM bank = 0x00000001

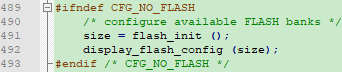
-> start = 0x40000000

-> size = 0x10000000

init\_sequence总结

(1)都是板级硬件的初始化以及gd、gd->bd中的数据结构的初始化。譬如：网卡初始化、机器码（gd->bd->bi\_arch\_number）、内核传参DDR地址（gd->bd->bi\_boot\_params）、Timer4初始化为10ms一次、波特率设置（gd->bd->bi\_baudrate和gd->baudrate）、console第一阶段初始化（gd->have\_console设置为1）、打印uboot的启动信息、打印cpu相关设置信息、检查并打印当前开发板名字、DDR配置信息初始化（gd->bd->bi\_dram）、打印DDR总容量。

### 6.4.15. CFG\_NO\_FLASH



(1)虽然NandFlash和NorFlash都是Flash，但是一般NandFlash会简称为Nand而不是Flash，一般讲Flash都是指的Norflash。这里2行代码是Norflash相关的。

(2)flash\_init执行的是开发板中对应的NorFlash的初始化、display\_flash\_config打印的也是NorFlash的配置信息（Flash: 8 MB就是这里打印出来的）。但是实际上X210中是没有Norflash的。所以着两行代码是可以去掉的（我也不知道为什么没去掉？猜测原因有可能是去掉着两行代码会导致别的地方工作不正常，需要花时间去移植调试，然后移植的人就懒得弄。实际上不去掉除了显示有8MB Flash实际没用之外也没有别的影响）

CONFIG\_VFD和CONFIG\_LCD是显示相关的，这个是uboot中自带的LCD显示的软件架构。但是实际上我们用LCD而没有使用uboot中设置的这套软件架构，我们自己在后面自己添加了一个LCD显示的部分。

## 6.4.16. 初始化堆管理器 mem\_malloc\_init

(1)mem\_malloc\_init函数用来初始化uboot的堆管理器。

(2)uboot中自己维护了一段堆内存，肯定自己就有一套代码来管理这个堆内存。有了这些东西uboot中你也可以malloc、free这套机制来申请内存和释放内存。我们在DDR内存中给堆预留了896KB的内存。

### 6.4.17. 开发板独有初始化：mmc初始化

(1)从536到768行为开发板独有的初始化。意思是三星用一套uboot同时满足了好多个系列型号的开发板，然后在这里把不同开发板自己独有的一些初始化写到了这里。用#if条件编译配合CONFIG\_xxx宏来选定特定的开发板。

(2)X210相关的配置在599行到632行。

(3)mmc\_initialize看名字就应该是MMC相关的一些基础的初始化，其实就是用来初始化SoC内部的SD/MMC控制器的。函数在uboot/drivers/mmc/mmc.c里。

(4)uboot中对硬件的操作（譬如网卡、SD卡……）都是借用的linux内核中的驱动来实现的，uboot根目录底下有个drivers文件夹，这里面放的全都是从linux内核中移植过来的各种驱动源文件。

(5)mmc\_initialize是具体硬件架构无关的一个MMC初始化函数，所有的使用了这套架构的代码都掉用这个函数来完成MMC的初始化。mmc\_initialize中再调用board\_mmc\_init和cpu\_mmc\_init来完成具体的硬件的MMC控制器初始化工作。

(6)cpu\_mmc\_init在uboot/cpu/s5pc11x/cpu.c中，这里面又间接的调用了drivers/mmc/s3c\_mmcxxx.c中的驱动代码来初始化硬件MMC控制器。这里面分层很多，分层的思想一定要有，否则完全就糊涂了。

### 6.4.18. env\_relocate

gd->env\_addr：环境变量在SD卡处的起始位置

gd->env\_addr：环境变量在DDR处的起始位置

(1)env\_relocate是环境变量的重定位，完成从SD卡中将环境变量读取到DDR中的任务。

(2)环境变量到底从哪里来？SD卡中有一些（8个）独立的扇区作为环境变量存储区域的。但是我们烧录/部署系统时，我们只是烧录了uboot分区、kernel分区和rootfs分区，根本不曾烧录env分区。所以当我们烧录完系统第一次启动时ENV分区是空的，本次启动uboot尝试去SD卡的ENV分区读取环境变量时失败（读取回来后进行CRC校验时失败），我们uboot选择从uboot内部代码中设置的一套默认的环境变量出发来使用（这就是默认环境变量）；这套默认的环境变量在本次运行时会被读取到DDR中的环境变量中，然后被写入（也可能是你saveenv时写入，也可能是uboot设计了第一次读取默认环境变量后就写入）SD卡的ENV分区。然后下次再次开机时uboot就会从SD卡的ENV分区读取环境变量到DDR中，这次读取就不会失败了。

(3)真正的从SD卡到DDR中重定位ENV的代码是在env\_relocate\_spec内部的movi\_read\_env完成的。

### 6.4.19. IP地址、MAC地址的确定

/\* IP Address \*/

gd->bd->bi\_ip\_addr = getenv\_IPaddr ("ipaddr");

/\* MAC Address \*/

gd->bd->bi\_enetaddr；

(1)开发板的IP地址是在gd->bd中维护的，来源于环境变量ipaddr。getenv函数用来获取字符串格式的IP地址，然后用string\_to\_ip将字符串格式的IP地址转成字符串格式的点分十进制格式。

(2)IP地址由4个0-255之间的数字组成，因此一个IP地址在程序中最简单的存储方法就是一个unsigend int。但是人类容易看懂的并不是这种类型，而是点分十进制类型（192.168.1.2）。这两种类型可以互相转换。

### 6.4.20. int devices\_init (void)

(1)devices\_init看名字就是设备的初始化。这里的设备指的就是开发板上的硬件设备。放在这里初始化的设备都是驱动设备，这个函数本来就是从驱动框架中衍生出来的。uboot中很多设备的驱动是直接移植linux内核的（譬如网卡、SD卡），linux内核中的驱动都有相应的设备初始化函数。linux内核在启动过程中就有一个devices\_init(名字不一定完全对，但是差不多)，作用就是集中执行各种硬件驱动的init函数。

(2)uboot的这个函数其实就是从linux内核中移植过来的，它的作用也是去执行所有的从linux内核中继承来的那些硬件驱动的初始化函数。

### 6.4.21. void jumptable\_init (void)

实就是在用C语言实现面向对象编程，建立一个函数跳转表。

(1)jumptable跳转表，本身是一个函数指针数组，里面记录了很多函数的函数名。看这阵势是要实现一个函数指针到具体函数的映射关系，将来通过跳转表中的函数指针就可以执行具体的函数。这个其实就是在用C语言实现面向对象编程。在linux内核中有很多这种技巧。

(2)通过分析发现跳转表只是被赋值从未被引用，因此跳转表在uboot中根本就没使用。

### 6.4.22. console\_init\_r ()

/\* fully init console as a device 控制台的第二次初始化 \*/

(1)console\_init\_f是控制台的第一阶段初始化，console\_init\_r是第二阶段初始化。实际上第一阶段初始化并没有实质性工作，第二阶段初始化才进行了实质性工作。

(2)uboot中有很多同名函数，使用SI工具去索引时经常索引到不对的函数处（回忆下当时start.S中找lowlevel\_init.S时，自动索引找到的是错误的，真正的反而根本没找到。）

(3)console\_init\_r就是console的纯软件架构方面的初始化（说白了就是去给console相关的数据结构中填充相应的值），所以属于纯软件配置类型的初始化。

(4)uboot的console实际上并没有干有意义的转化，它就是直接调用的串口通信的函数。所以用不用console实际并没有什么分别。（在linux内console就可以提供缓冲机制等不用console不能实现的东西）。

### 6.4.23. void enable\_interrupts (void)

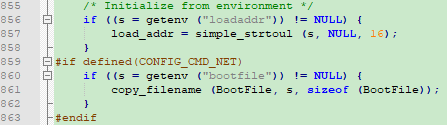
中断初始化，配置CPSR罢了。如果没使用IRQ，FIQ，则该函数为空

(1)看名字应该是中断初始化代码。这里指的是CPSR中总中断标志位的使能。

(2)因为我们uboot中没有使用中断，因此没有定义CONFIG\_USE\_IRQ宏，因此我们这里这个函数是个空壳子。

(3)uboot中经常出现一种情况就是根据一个宏是否定义了来条件编译决定是否调用一个函数内部的代码。uboot中有2种解决方案来处理这种情况：方案一：在调用函数处使用条件编译，然后函数体实际完全提供代码。方案二：在调用函数处直接调用，然后在函数体处提供2个函数体，一个是有实体的一个是空壳子，用宏定义条件编译来决定实际编译时编译哪个函数进去。

### 6.4.24. loadaddr、bootfile两个环境变量



这两个环境变量都是内核启动有关的，在启动linux内核时会参考这两个环境变量的值。

### 6.4.25. board\_late\_init (void)

(1)看名字这个函数就是开发板级别的一些初始化里比较晚的了，就是晚期初始化。所以晚期就是前面该初始化的都初始化过了，剩下的一些必须放在后面初始化的就在这里了。侧面说明了开发板级别的硬件软件初始化告一段落了。

(2)对于X210来说，这个函数是空的。

### 6.4.26. int eth\_initialize(bd\_t \*bis)

(1)看名字应该是网卡相关的初始化。这里不是SoC与网卡芯片连接时SoC这边的初始化，而是网卡芯片本身的一些初始化。

(2)对于X210（DM9000）来说，这个函数是空的。X210的网卡初始化在board\_init函数中，网卡芯片的初始化在驱动中。

### 6.4.27. x210\_preboot\_init(void)（LCD和logo显示）

(1)x210开发板在启动起来之前的一些初始化，以及LCD屏幕上的logo显示。（自己添加的）

### 6.4.28. check menukey to update from sd

(1)uboot启动的最后阶段设计了一个自动更新的功能。就是：我们可以将要升级的镜像放到SD卡的固定目录中，然后开机时在uboot启动的最后阶段检查升级标志（是一个按键。按键中标志为"LEFT"的那个按键，这个按键如果按下则表示update mode，如果启动时未按下则表示boot mode）。如果进入update mode则uboot会自动从SD卡中读取镜像文件然后烧录到iNand中；如果进入boot mode则uboot不执行update，直接启动正常运行。

(2)这种机制能够帮助我们快速烧录系统，常用于量产时用SD卡进行系统烧录部署。

### 6.4.29. main\_loop（uboot的归宿）



(1)解析器

(2)开机倒数自动执行

(3)命令补全

##下面是稍稍的总结##

==========================================================

init\_sequence

cpu\_init 空的

board\_init 网卡、机器码、内存传参地址

dm9000\_pre\_init 网卡

gd->bd->bi\_arch\_number 机器码

gd->bd->bi\_boot\_params 内核传参的内存地址

interrupt\_init 定时器

env\_init

init\_baudrate gd 数据结构中波特率

serial\_init 空的

console\_init\_f 空的

display\_banner 打印启动信息

print\_cpuinfo 打印CPU时钟设置信息

checkboard 检验开发板名字

dram\_init gd数据结构中DDR信息

display\_dram\_config 打印DDR配置信息表

mem\_malloc\_init 初始化uboot自己维护的堆管理器的内存

mmc\_initialize nand/SD卡的SoC控制器和卡的初始化

env\_relocate 环境变量重定位

gd->bd->bi\_ip\_addrgd 数据结构赋值

gd->bd->bi\_enetaddrgd 数据结构赋值

devices\_init 空的

jumptable\_init 不用关注的

console\_init\_r 真正的控制台初始化

enable\_interrupts 空的

loadaddr、bootfile 环境变量读出初始化全局变量

board\_late\_init 空的

eth\_initialize 空的

x210\_preboot\_init LCD初始化和显示logo

check\_menu\_update\_from\_sd 检查自动更新

main\_loop 主循环

### 6.4.30. 启动过程特征总结

(1)第一阶段为汇编阶段、第二阶段为C阶段

(2)第一阶段在SRAM中、第二阶段在DRAM中

(3)第一阶段注重SoC内部、第二阶段注重SoC外部Board内部

移植时的注意点

(1)x210\_sd.h头文件中的宏定义

(2)特定硬件的初始化函数位置（譬如网卡）

# 七、uboot源码分析3-uboot如何启动内核

## 7.1 uboot和内核到底是什么

### 7.1.1. uboot是一个裸机程序

(1)uboot的本质就是一个复杂点的裸机程序。和我们在ARM裸机全集中学习的每一个裸机程序并没有本质区别。

(2)ARM裸机第十六部分写了个简单的shell，这东西其实就是个mini型的uboot

### 7.1.2.内核本身也是一个"裸机程序"

(1)操作系统内核本身就是一个裸机程序，和uboot、和其他裸机程序并没有本质区别。

(2)区别就是操作系统运行起来后在软件上分为内核层和应用层，分层后两层的权限不同，内存访问和设备操作的管理上更加精细（内核可以随便访问各种硬件，而应用程序只能被限制的访问硬件和内存地址）。

直观来看：uboot的镜像是u-boot.bin，linux系统的镜像是zImage，这两个东西其实都是两个裸机程序镜像。从系统的启动角度来讲，内核其实就是一个大的复杂点裸机程序。

### 7.1.3.部署在SD卡中特定分区内

(1)一个完整的软件+硬件的嵌入式系统，静止时（未上电时）bootloader、kernel、rootfs等必须的软件都以镜像的形式存储在启动介质中（X210中是iNand/SD卡）；运行时都是在DDR内存中运行的，与存储介质无关。上面2个状态都是稳定状态，第3个状态是动态过程，即从静止态到运行态的过程，也就是启动过程。

(2)动态启动过程就是一个从SD卡逐步搬移到DDR内存，并且运行启动代码进行相关的硬件初始化和软件架构的建立，最终达到运行时稳定状态。

(3)静止时u-boot.bin zImage rootfs都在SD卡中，他们不可能随意存在SD卡的任意位置，因此需要对SD卡进行一个分区，然后将各种镜像各自存在各自的分区中，这样在启动过程中uboot、内核等就知道到哪里去找谁。（uboot和kernel中的分区表必须一致，同时和SD卡的实际使用的分区要一致）

### 7.1.4.运行时必须先加载到DDR中链接地址处

(1)uboot在第一阶段中进行重定位时将第二阶段（整个uboot镜像）加载到DDR的0xc3e00000地址处，这个地址就是uboot的链接地址。

(2)内核也有类似要求，uboot启动内核时将内存从SD卡读取放到DDR中（其实就是个重定位的过程），不能随意放置，必须放在内核的链接地址处，否则启动不起来。譬如我们使用的内核链接地址是0x30008000。

### 7.1.5.内核启动需要必要的启动参数

(1)uboot是无条件启动的，从零开始启动的。

(2)内核是不能开机自动完全从零开始启动的，内核启动要别人帮忙。uboot要帮助内核实现重定位（从SD卡到DDR），uboot还要给内核提供启动参数。

## 7.2. 启动内核第一步：加载内核到DDR中

uboot要启动内核，分为2个步骤：第一步是将内核镜像从启动介质中加载到DDR中，第二步是去DDR中启动内核镜像。（内核代码根本就没考虑重定位，因为内核知道会有uboot之类的把自己加载到DDR中链接地址处的，所以内核直接就是从链接地址处开始运行的）

### 7.2.1.静态内核镜像在哪里？

(1)SD卡/iNand/Nand/NorFlash等：raw分区（即原始格式分区）

常规启动时各种镜像都在SD卡中，因此uboot只需要从SD卡的kernel分区去读取内核镜像到DDR中即可。读取要使用uboot的命令来读取（譬如X210的iNand版本是movi命令，X210的Nand版本就是Nand命令）

(2)这种启动方式来加载DDR，使用命令：movi read kernel 30008000。其中kernel指的是uboot中的kernel分区（就是uboot中规定的SD卡中的一个区域范围，这个区域范围被设计来存放kernel镜像，就是所谓的kernel分区）

(2)tftp、nfs等网络下载方式从远端服务器获取镜像

uboot还支持远程启动，也就是内核镜像不烧录到开发板的SD卡中，而是放在主机的服务器中，然后需要启动时uboot通过网络从服务器中下载镜像到开发板的DDR中。

分析总结：最终结果要的是内核镜像到DDR中特定地址即可，不管内核镜像是怎么到DDR中的。以上2种方式各有优劣。产品出厂时会设置为从SD卡中启动（客户不会还要搭建tftp服务器才能用……）；tftp下载远程启动这种方式一般用来开发。

### 7.2.2.镜像要放在DDR的什么地址？

内核一定要放在链接地址处，链接地址去内核源代码的链接脚本或者Makefile中去查找。X210中是0x30008000。

## 7. 3.zImage和uImage的区别联系

### 7.3.1. bootm命令对应do\_bootm函数

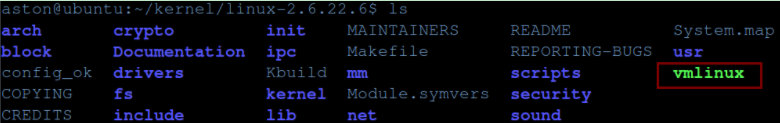
(1)命令名前加do\_即可构成这个命令对应的函数，因此当我们bootm命令执行时，uboot实际执行的函数叫do\_bootm函数，在cmd\_bootm.c。

(2)do\_bootm刚开始定义了一些变量，然后用宏来条件编译执行了secureboot的一些代码（主要进行签名认证），先不管他；然后进行了一些一些细节部分操作，也不管他。然后到了CONFIG\_ZIMAGE\_BOOT，用这个宏来控制进行条件编译一段代码，这段代码是用来支持zImage格式的内核启动的。

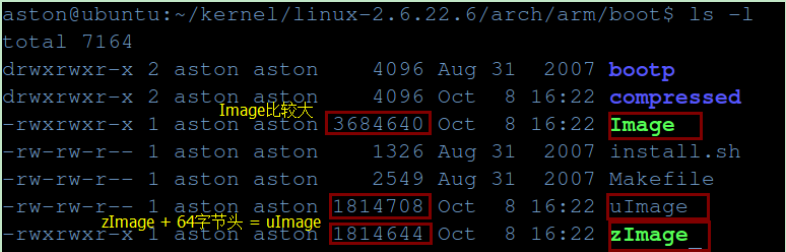
### 7.3.2. vmlinux和zImage和uImage

我们编译内核完（编译成功）会生成vmlinux，Image，zImage，再通过uboot提供的工具mkimage，执行make uImage会生成uImage，那么他们谁是内核镜像。如下图为这3个东东：

vmlinux在kernel根目录下：



Image和zImage和uImage在根目录下的arch/arm/boot目录下：



SO，哪个是镜像文件？哈哈，三个都是。

vmlinux：这个就是原始的未经任何处理加工的原版内核elf文件，他一般加载在PC机这种的硬盘中，因为未加工（比较大），所以PC机有大硬盘不怕放不下，所以PC机的上电后直接加载vmlinux内核镜像就可以运行了。比如（Ubuntu）。

Image：vmlinux经过objcopy去掉一些不需要的东西之后得到的。

zImage：Image经过压缩得到的。原则上，Image就可以直接运行了，但是，嵌入式系统要求精简所以把Image压缩，再加上压缩算法放到真正的压缩镜像前面就得到zImage。如图可知zImage变小了。

uImage：uImage是经过加64字节的头部信息放到zImage的前面合成得到的。

如果直接在kernel底下去make uImage会提供mkimage command not found。解决方案是去uboot/tools下cp mkimage /usr/local/bin/，复制mkimage工具到系统目录下。再去make uImage即可。

## 7.4 zImage启动细节

do\_bootm函数中一直到397行的after\_header\_check这个符号处，都是在进行镜像的头部信息校验。校验时就要根据不同种类的image类型进行不同的校验。所以do\_bootm函数的核心就是去分辨传进来的image到底是什么类型，然后按照这种类型的头信息格式去校验。校验通过则进入下一步准备启动内核；如果校验失败则认为镜像有问题，所以不能启动。

### 7.4.1. LINUX\_ZIMAGE\_MAGIC

(1)这个是一个定义的魔数，这个数等于0x016f2818，表示这个镜像是一个zImage。也就是说zImage格式的镜像中在头部的一个固定位置存放了这个数作为格式标记。如果我们拿到了一个image，去他的那个位置去取4字节判断它是否等于LINUX\_ZIMAGE\_MAGIC，则可以知道这个镜像是不是一个zImage。

(2)命令 bootm 0x30008000，所以do\_boom的argc=2，argv[0]=bootm argv[1]=0x30008000。但是实际bootm命令还可以不带参数执行。如果不带参数直接bootm，则会从CFG\_LOAD\_ADDR地址去执行（定义在x210\_sd.h中）。

(3)zImage头部开始的第37-40字节处存放着zImage标志魔数，从这个位置取出然后对比LINUX\_ZIMAGE\_MAGIC。可以用二进制阅读软件来打开zImage查看，就可以证明。很多软件都可以打开二进制文件，如winhex、UltraEditor。



### 7.4.2. image\_header\_t

(1)这个数据结构是我们uboot启动内核使用的一个标准启动数据结构，zImage头信息也是一个image\_header\_t，但是在实际启动之前需要进行一些改造。下边两句就是在进行改造。

hdr->ih\_os = IH\_OS\_LINUX;

hdr->ih\_ep = ntohl(addr);

(2)images全局变量是do\_bootm函数中使用，用来完成启动过程的。zImage的校验过程其实就是先确认是不是zImage，确认后再修改zImage的头信息到合适，修改后用头信息去初始化images这个全局变量，然后就完成了校验。

如果那个zImage的宏CONFIG\_ZIMAGE\_BOOT没有定义将会执行uImage或者设备树的方式。取决于genimg\_get\_format (os\_hdr)的结果。设备数启动方式暂且搁置。

## 7.5 uImage启动

(1)LEGACY(遗留的)，在do\_bootm函数中，这种方式指的就是uImage的方式。

(2)uImage方式是uboot本身发明的支持linux启动的镜像格式，但是后来这种方式被一种新的方式替代，这个新的方式就是设备树方式（在do\_bootm方式中叫FIT）

(3)uImage的启动校验主要在boot\_get\_kernel函数中，主要任务就是校验uImage的头信息，并且得到真正的kernel的起始位置去启动。

总结1：uboot本身设计时只支持uImage启动，原来uboot的代码也是这样写的。后来有了fdt方式之后，就把uImage方式命令为LEGACY方式，fdt方式命令为FIT方式，于是乎多了写#if #endif添加的代码。后来移植的人又为了省事添加了zImage启动的方式，又为了省事把zImage启动方式直接写在了uImage和fdt启动方式之前，于是乎又有了一对#if #endif。于是乎整天的代码看起来很糟心。

总结2：第二阶段校验头信息结束，下面进入第三阶段，第三阶段主要任务是启动linux内核，调用do\_bootm\_linux函数来完成

## 7.6 do\_bootm\_linux函数

### 7.6.1.找到do\_bootm\_linux函数

(1)函数在uboot/lib\_arm/bootm.c中。

(2)SI找不到（是黑色的）不代表就没有，要搜索一下才能确定；搜索不到也不能代表就没有，因为我们在向SI工程中添加文件时，SI只会添加它能识别的文件格式的文件，有一些像Makefile、xx.conf等Makefile不识别的文件是没有被添加的。所以如果要搜索的关键字在makefile中或者脚本中，可能就是搜索不到的。（譬如TEXT\_BASE）

### 7.6.2.镜像的entrypoint

(1)ep就是entrypoint的缩写，就是程序入口。一个镜像文件的起始执行部分不是在镜像的开头（镜像开头有n个字节的头信息），真正的镜像文件执行时第一句代码在镜像的中部某个字节处，相当于头是有一定的偏移量的。这个偏移量记录在头信息中。

(2)一般执行一个镜像都是：第一步先读取头信息，然后在头信息的特定地址找MAGIC\_NUM，由此来确定镜像种类；第二步对镜像进行校验；第三步再次读取头信息，由特定地址知道这个镜像的各种信息（镜像长度、镜像种类、入口地址）；第四步就去entrypoint处开始执行镜像。

(3)theKernel = (void (\*)(int, int, uint))ep;将ep赋值给theKernel，则这个函数指向就指向了内存中加载的OS镜像的真正入口地址（就是操作系统的第一句执行的代码）。

### 7.6.3.机器码的再次确定

(1)uboot在启动内核时，机器码要传给内核。uboot传给内核的机器码是怎么确定的？第一顺序备选是环境变量machid，第二顺序备选是gd->bd->bi\_arch\_num（x210\_sd.h中硬编码配置的）

### 7.6.4.传参并启动概述

(1)从110行到144行就是uboot在给linux内核准备传递的参数处理。

(2)Starting kernel ... 这个是uboot中最后一句打印出来的东西。这句如果能出现，说明uboot整个是成功的，也成功的加载了内核镜像，也校验通过了，也找到入口地址了，也试图去执行了。如果这句后串口就没输出了，说明内核并没有被成功执行。原因一般是：传参（80%）、内核在DDR中的加载地址……

## 7.7传参详解

### 7.7.1. tag方式传参

(1)struct tag，tag是一个数据结构，在uboot和linux kernel中都有定义tag数据结构，而且定义是一样的。

(2)tag\_header和tag\_xxx。tag\_header中有这个tag的size和类型编码，kernel拿到一个tag后先分析tag\_header得到tag的类型和大小，然后将tag中剩余部分当作一个tag\_xxx来处理。

(3)tag\_start与tag\_end。kernel接收到的传参是若干个tag构成的，这些tag由tag\_start起始，到tag\_end结束。

(4)tag传参的方式是由linux kernel发明的，kernel定义了这种向我传参的方式，uboot只是实现了这种传参方式从而可以支持给kernel传参。

### 7.7.2. x210\_sd.h中配置传参宏

(1)CONFIG\_SETUP\_MEMORY\_TAGS，tag\_mem，传参内容是内存配置信息。

(2)CONFIG\_CMDLINE\_TAG，tag\_cmdline，传参内容是启动命令行参数，也就是uboot环境变量的bootargs.

(3)CONFIG\_INITRD\_TAG

(4)CONFIG\_MTDPARTITION，传参内容是iNand/SD卡的分区表。

(5)起始tag是ATAG\_CORE、结束tag是ATAG\_NONE，其他的ATAG\_XXX都是有效信息tag。

思考：内核如何拿到这些tag？

uboot最终是调用theKernel函数来执行linux内核的，uboot调用这个函数（其实就是linux内核）时传递了3个参数。这3个参数就是uboot直接传递给linux内核的3个参数，通过寄存器来实现传参的。（第1个参数就放在r0中，第二个参数放在r1中，第3个参数放在r2中）第1个参数固定为0，第2个参数是机器码，第3个参数传递的就是大片传参tag的首地址。

### 7.7.3.移植时注意事项

(1)uboot移植时一般只需要配置相应的宏即可

(2)kernel启动不成功，注意传参是否成功。传参不成功首先看uboot中bootargs设置是否正确，其次看uboot是否开启了相应宏以支持传参。

## 7.8 uboot启动内核的总结

<https://blog.csdn.net/KayChanGEEK/article/details/50101555>

启动4步骤

第一步：将内核搬移到DDR中

第二步：校验内核格式、CRC等

第三步：准备传参

第四步：跳转执行内核

涉及到的主要函数是：do\_boom和do\_bootm\_linux

uboot能启动的内核格式：zImage uImage fdt方式

跳转与函数指针的方式运行内核：theKernel (0, machid, bd->bi\_boot\_params);

# 八、uboot源码分析4-uboot的命令体系

## 8.1 uboot命令体系基础

### 8.1.1.使用uboot命令

uboot启动后进入命令行环境下，在此输入命令按回车结束，uboot会收取这个命令然后解析，然后执行。

### 8.1.2. uboot命令体系实现代码在哪里

uboot命令体系的实现代码在uboot/common/cmd\_xxx.c中。有若干个.c文件和命令体系有关。（还有command.c main.c也是和命令有关的）

### 8.1.3.每个命令对应一个函数

(1)每一个uboot的命令背后都对应一个函数。这就是uboot实现命令体系的一种思路和方法。这个东西和我们在裸机第十六部分shell中实现shell命令的方法是一样的。

(2)我们要找到每一个命令背后所对应的那个函数，而且要分析这个函数和这个命令是怎样对应起来的。

### 8.1.4.命令参数以argc&argv传给函数

有些uboot的命令还支持传递参数。也就是说命令背后对应的函数接收的参数列表中有argc和argv，然后命令体系会把我们执行命令时的命令+参数（例如：md 30000000 10）以argc（3）和argv（argv[0]=md, argv[1]=30000000 argv[2]=10）的方式传递给执行命令的函数。

举例分析，以help命令为例：

help命令背后对应的函数名叫：do\_help。在uboot/common/command.c的236行。int do\_help (cmd\_tbl\_t \* cmdtp, int flag, int argc, char \*argv[])

## 8.2 uboot命令解析和执行过程分析

### 8.2.1.从main\_loop说起

uboot/common/main.c

(1)uboot启动的第二阶段，在初始化了所有该初始化的东西后，进入了一个死循环，死循环的循环体就是main\_loop。



(2)main\_loop函数执行一遍，就是一个获取命令、解析命令、执行命令的过程。

(3)main\_loop中调用的run\_command函数就是用来执行命令的函数。

### 8.2.2关键点分析

(1)控制台命令获取

(2)命令解析。int parse\_line (char \*line, char \*argv[])函数把"md 30000000 10"解析成argv[0]=md, argv[1]=30000000 argv[2]=10;

(3)命令集中查找命令。find\_cmd(argv[0])函数去uboot的命令集合当中搜索有没有argv[0]这个命令，

(4)执行命令。最后用函数指针的方式调用执行了对应函数。

思考：关键点就在于find\_cmd函数如何查找到这个命令是不是uboot的合法支持的命令？这取决于uboot的命令体系机制（uboot是如何完成命令的这一套设计的，命令如何去注册、存储、管理、索引。）。

## 8.3 uboot如何处理命令集1

### 8.3.1.可能的管理方式

(1)数组。结构体数组，数组中每一个结构体成员就是一个命令的所有信息。

(2)链表。链表的每个节点data段就是一个命令结构体，所有的命令都放在一条链表上。这样就解决了数组方式的不灵活。坏处是需要额外的内存开销，然后各种算法（遍历、插入、删除等）需要一定复杂度的代码执行。

(3)有第三种吗？uboot没有使用数组或者链表，而是使用了一种新的方式来实现这个功能。

### 8.3.2.命令结构体cmd\_tbl\_t

struct cmd\_tbl\_s {

char \*name; /\* Command Name \*/

int maxargs; /\* maximum number of arguments \*/

int repeatable; /\* autorepeat allowed? \*/

/\* Implementation function \*/

int (\*cmd)(struct cmd\_tbl\_s \*, int, int, char \*[]);

char \*usage; /\* Usage message (short) \*/

#ifdef CFG\_LONGHELP

char \*help; /\* Help message (long) \*/

#endif

#ifdef CONFIG\_AUTO\_COMPLETE

/\* do auto completion on the arguments \*/

int (\*complete)(int argc, char \*argv[], char last\_char, int maxv, char \*cmdv[]);

#endif

};

typedef struct cmd\_tbl\_s cmd\_tbl\_t;

(1)name：命令名称，字符串格式。

(2)maxargs：命令最多可以接收多少个参数

(3)repeatable：指示这个命令是否可重复执行。重复执行是uboot命令行的一种工作机制，就是直接按回车则执行上一条执行的命令。

(4)cmd：函数指针，命令对应的函数的函数指针，将来执行这个命令的函数时使用这个函数指针来调用。

(5)usage：命令的短帮助信息。对命令的简单描述。

(6)help：命令的长帮助信息。细节的帮助信息。

(7)complete：函数指针，指向这个命令的自动补全的函数。

总结：uboot的命令体系在工作时，一个命令对应一个cmd\_tbl\_t结构体的一个实例，然后uboot支持多少个命令，就需要多少个结构体实例。uboot的命令体系把这些结构体实例管理起来，当用户输入了一个命令时，uboot会去这些结构体实例中查找（查找方法和存储管理的方法有关）。如果找到则执行命令，如果未找到则提示命令未知。

### 8.3.2. uboot实现命令管理的思路

给所有命令结构体实例定义段属性。链接时链接在一起。所有命令结构体实例就紧挨着被重定位加载到DDR。通过类似于数组遍历的方式来查询命令，执行命令。这些命令结构体实例的开始和结束地址由段起始地址和结束地址确定。



(1)填充1个结构体实例构成一个命令

(2)给命令结构体实例附加特定段属性（用户自定义段），链接时将带有该段属性的内容链接在一起排列（挨着的，不会夹杂其他东西，也不会丢掉一个带有这种段属性的，但是顺序是乱序的）。

(3)uboot重定位时将该段整体加载到DDR中。加载到DDR中的uboot镜像中带有特定段属性的这一段其实就是命令结构体的集合，有点像一个命令结构体数组。

(4)段起始地址和结束地址（链接地址、定义在u-boot.lds中）决定了这些命令集的开始和结束地址。

## 8.4 uboot如何处理命令集2

### 8.4.1. uboot命令定义具体实现分析

U\_BOOOT\_CMD宏基本分析。

格式：U\_BOOT\_CMD(name,maxargs,repeatable,command,"usage","help")

name: 命令名，非字符串，但在U\_BOOT\_CMD中用“#”符号转化为字符串

maxargs: 命令的最大参数个数

repeatable: 是否自动重复（按Enter键是否会重复执行）

command: 该命令对应的响应函数指针

“usage”: 段帮助

“help”: 长帮助

这个宏定义在uboot/include/command.h中。



“##”与”#”都是预编译操作符，“##”有字符串连接的功能，”#”表示后面紧接着的是一个字符串。其中Struct\_Section在include/command.h中定义如下：



凡是带有\_\_attribute\_\_ ((unused,section (“.u\_boot\_cmd”))属性声明的变量都将被存放在”.u\_boot\_cmd”段中，并且即使该变量没有在代码中显式的使用编译器也不产生警告信息。

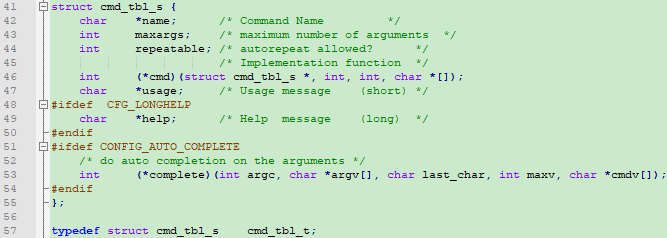
在u-Boot连接脚本 u-boot.lds中定义了”.u\_boot\_cmd”段：



这表明带有”.u\_boot\_cmd”声明的函数或变量将存储在”u\_boot\_cmd”段。

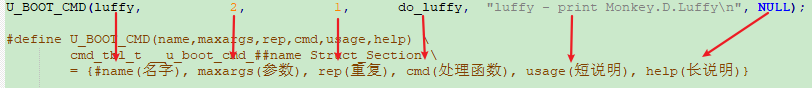
这样只要将u-boot所有命令对应的cmd\_tbl\_t变量加上”.u\_boot\_cmd”声明，编译器就会自动将其放在”u\_boot\_cmd”段，查找cmd\_tbl\_t变量时只要在 \_\_u\_boot\_cmd\_start 与 \_\_u\_boot\_cmd\_end 之间查找就可以了。

cmd\_tbl\_t在include/command.h中定义如下：



一个cmd\_tbl\_t结构体变量（命令结构体实例）包含了调用一条命令的所需要的信息。

以”luffy”命令的定义经过宏展开后如下：





### 8.4.2. find\_cmd函数详解

(1)find\_cmd函数的任务是从当前uboot的命令集中查找是否有某个命令。如果找到则返回这个命令结构体的指针，如果未找到返回NULL。

(2)函数的实现思路很简单，如果不考虑命令带点的情况（md.b md.w这种）就更简单了。查找命令的思路其实就是for循环遍历数组的思路，数组中的元素是命令结构体类型变量，不同的是数组的起始地址和结束地址是用地址值来给定的，而这里是由链接脚本中自定义段的起始和结束地址来确定的。

总结命令执行过程：

① 在u-boot控制台中输入”luffy”命令执行时，u-boot控制台接收输入的字符串”luffy”，传递给run\_command函数。

② run\_command函数调用common/command.c中实现的find\_cmd函数在\_\_u\_boot\_cmd\_start与\_\_u\_boot\_cmd\_end间查找命令，并返回luffy命令的cmd\_tbl\_t结构。

③ 然后run\_command函数使用返回的cmd\_tbl\_t结构中的函数指针调用luffy命令的响应函数do\_luffy，从而完成了命令的执行。

## 8.5 uboot中添加自定义命令

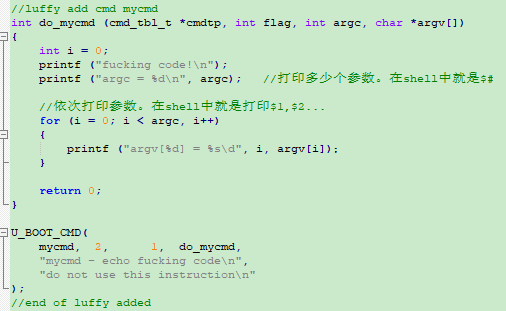
### 8.5.1.在已有的c文件中直接添加命令

(1)在uboot/common/command.c中添加一个命令，叫：mycmd

(2)在已有的.c文件中添加命令比较简单，直接使用U\_BOOT\_CMD宏即可添加命令，给命令提供一个do\_xxx的对应的函数这个命令就GG了。

(3)添加完成后要重新编译工程（make distclean; make x210\_sd\_config; make），然后烧录新的uboot去运行即可体验新命令。

(4)还可以在函数中使用argc和argv来验证传参。

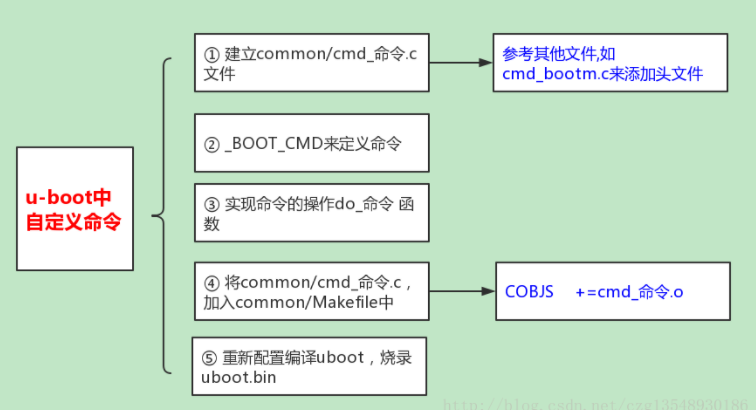


### 8.5.2.自建一个c文件并添加命令

(1)在uboot/common目录下新建一个命令文件，叫cmd\_luffy.c（对应的命令名就叫luffy，对应的函数就叫do\_luffy函数），然后在c文件中添加命令对应的U\_BOOT\_CMD宏和函数。注意头文件包含不要漏掉。

(2)在uboot/common/Makefile中添加上cmd\_luffy.o，目的是让Make在编译时能否把cmd\_luffy.c编译链接进去。

(3)重新编译烧录。重新编译步骤是：make distclean; make x210\_sd\_config; make



体会：uboot命令体系的优点

(1)uboot的命令体系本身稍微复杂，但是他写好之后就不用动了。我们后面在移植uboot时也不会去动uboot的命令体系。我们最多就是向uboot中去添加命令，就像本节课所做的这样。

(2)向uboot中添加命令非常简单。

# 九、uboot源码分析5-uboot的环境变量

## 9.1 uboot的环境变量基础

### 9.1.1.环境变量的作用

让我们可以不用修改uboot的源代码，而是通过修改环境变量来影响uboot运行时的一些数据和特性。譬如说通过修改bootdelay环境变量就可以更改系统开机自动启动时倒数的秒数。

### 9.1.2.环境变量的优先级

(1)uboot代码当中有一个值，环境变量中也有一个值。uboot程序实际运行时规则是：如果环境变量为空则使用代码中的值；如果环境变量不为空则优先使用环境变量对应的值。

(2)譬如machid（机器码）。uboot中在x210\_sd.h中定义了一个机器码2456，写死在程序中的不能更改。如果要修改uboot中配置的机器码，可以修改x210\_sd.h中的机器码，但是修改源代码后需要重新编译烧录，很麻烦；比较简单的方法就是使用环境变量machid。set machid 0x998类似这样，有了machid环境变量后，系统启动时会优先使用machid对应的环境变量，这就是优先级问题。

### 9.1.3.环境变量在uboot中工作方式

(1)默认环境变量，在uboot/common/env\_common.c中default\_environment，这东西本质是一个字符数组，大小为CFG\_ENV\_SIZE（16KB），里面内容就是很多个环境变量连续分布组成的，每个环境变量最末端以'\0'结束。

(2)SD卡中环境变量分区，在uboot的raw分区中。SD卡中其实就是给了个分区，专门用来存储而已。存储时其实是把DDR中的环境变量整体的写入SD卡中分区里。所以当我们saveenv时其实整个所有的环境变量都被保存了一遍，而不是只保存更改了的。

(3)DDR中环境变量，在default\_environment中，实质是字符数组。在uboot中其实是一个全局变量，链接时在数据段，重定位时default\_environment就被重定位到DDR中一个内存地址处了。这个地址处这个全局字符数组就是我们uboot运行时的DDR中的环境变量了。

总结：刚烧录的系统中环境变量分区是空白的，uboot第一次运行时加载的是uboot代码中自带的一份环境变量，叫默认环境变量（正所谓default\_environment）。我们在saveenv时DDR中的环境变量会被更新到SD卡中的环境变量中，就可以被保存下来，下次开机会在环境变量relocate时会SD卡中的环境变量会被加载到DDR中去。

default\_environment中的内容虽然被uboot源代码初始化为一定的值（这个值就是我们的默认环境变量），但是在uboot启动的第二阶段，env\_relocate时代码会去判断SD卡中的env分区的crc是否通过。如果crc校验通过说明SD卡中有正确的环境变量存储，则relocate函数会从SD卡中读取环境变量来覆盖default\_environment字符数组，从而每次开机可以保持上一次更改过的环境变量。

## 9.2.环境变量相关命令源码解析

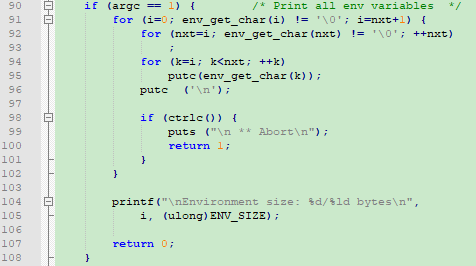
### 9.2.1. printenv

(1)命令定义和对应的函数在uboot/common/cmd\_nvedit.c中，对应的函数为do\_printenv。

(2)通过printenv的help可以看出，这个命令有2种使用方法。第一种直接使用不加参数则打印所有的环境变量；第二种是printenv name则只打印出name这个环境变量的值。(3)分析do\_printenv函数：do\_printenv函数首先区分argc=1还是不等于1的情况，若argc=1那么就循环打印所有的环境变量出来；如果argc不等于1，则后面的参数就是要打印的环境变量，给哪个就打印哪个。



(5)argc=1时用双重for循环来依次处理所有的环境变量的打印。第一重for循环就是处理各个环境变量。所以有多少个环境变量则第一重就执行循环多少圈。



1. 这个函数要看懂，首先要明白整个环境变量在内存中如何存储的问题。

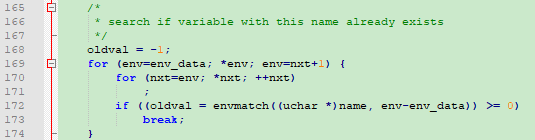
环境变量在内存中存储是以字符串形式的，每个环境变量的结尾都有个”\0”作为结束标志。

(6)关键点：第一要明白环境变量在内存中存储的方式；第二要C语言处理字符串的功底要好。

### 9.2.2. setenv

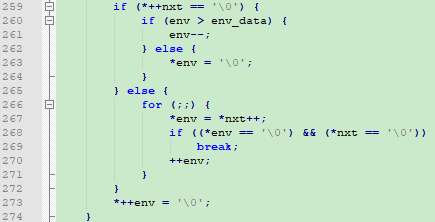
(1)命令定义和对应的函数在uboot/common/cmd\_nvedit.c中，对应的函数为do\_setenv。

(2)setenv的思路就是：先去DDR中的环境变量处寻找原来有没有这个环境变量，如果原来就有则需要覆盖原来的环境变量，如果原来没有则在最后新增一个环境变量即可。

第1步：遍历DDR中环境变量的数组，找到原来就有的那个环境变量对应的地址。168-174行。

第2步：擦除原来的环境变量，259-265行

第3步：写入新的环境变量，266-273行。



(3)本来setenv做完上面的就完了，但是还要考虑一些附加的问题。

问题一：环境变量太多超出DDR中的字符数组，溢出的解决方法。

问题二：有些环境变量如baudrate、ipaddr等，在gd中有对应的全局变量。这种环境变量在set更新的时候要同时去更新对应的全局变量，否则就会出现在本次运行中环境变量和全局变量不一致的情况。

### 9.2.3. saveenv

(1)命令定义和对应的函数在uboot/common/cmd\_nvedit.c中，对应函数为do\_saveenv

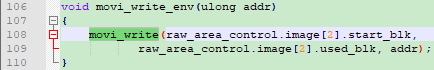
(2)从uboot实际执行saveenv命令的输出，和x210\_sd.h中的配置（#define CFG\_ENV\_IS\_IN\_AUTO）可以分析出：我们实际使用的是env\_auto.c中相关的内容。没有一种芯片叫auto的，env\_auto.c中是使用宏定义的方式去条件编译了各种常见的flash芯片（如movinand、norflash、nand等）。然后在程序中读取INF\_REG（OMpin内部对应的寄存器）从而知道我们的启动介质，然后调用这种启动介质对应的操作函数来操作。

(3)do\_saveenv内部调用env\_auto.c中的saveenv函数来执行实际的环境变量保存操作。

(4)寄存器地址：E010F000+0C=E010\_F00C，含义是用户自定义数据。我们在start.S中判断启动介质后将#BOOT\_MMCSD（就是3，定义在x210\_sd.h）写入了这个寄存器，所以这里读出的肯定是3，经过判断就是movinand。所以实际执行的函数是：saveenv\_movinand

(5)真正执行保存环境变量操作的是：cpu/s5pc11x/movi.c中的movi\_write\_env函数，这个函数肯定是写sd卡，将DDR中的环境变量数组（其实就是default\_environment这个数组，大小16kb，刚好32个扇区）写入iNand中的ENV分区中。

(6)raw\_area\_control是uboot中规划iNand/SD卡的原始分区表，这个里面记录了我们对iNand的分区，env分区也在这里，下标是2.追到这一层就够了，再里面就是调用驱动部分的写SD卡/iNand的底层函数了。



### 9.2.4. getenv

(1)是不可重入的。

(2)实现方式就是去遍历default\_environment数组，挨个拿出所有的环境变量比对name，找到相等的直接返回这个环境变量的首地址即可。

### 9.2.5. getenv\_r

(1)可重入版本。（可自行搜索补充可重入函数的概念）

(2)getenv函数是直接返回这个找到的环境变量在DDR中环境变量处的地址，而getenv\_r函数的做法是找到了DDR中环境变量地址后，将这个环境变量复制一份到提供的buf中，而不动原来DDR中环境变量。

所以差别就是：getenv中返回的地址只能读不能随便乱写，而getenv\_r中返回的环境变量是在自己提供的buf中，是可以随便改写加工的。

总结

(1)功能是一样的，但是可重入版本会比较安全一些，建议使用。

(2)有关于环境变量的所有操作，主要理解了环境变量在DDR中的存储方法，理解了环境变量和gd全局变量的关联和优先级，理解了环境变量在存储介质中的存储方式（专用raw分区），整个环境变量相关的都清楚了。

# 十、uboot源码分析6-uboot的硬件驱动部分

## 10.1 uboot与linux驱动

10.1.1 uboot本身是裸机程序

(1)裸机本来是没有驱动的概念的（狭义的驱动的概念就是操作系统中用来具体操控硬件的那部分代码叫驱动）

(2)裸机程序中是直接操控硬件的，操作系统中必须通过驱动来操控硬件。这两个有什么区别？本质区别就是分层。

### 10.1.2. uboot的虚拟地址对硬件操作的影响

(1)操作系统（指的是linux）下MMU肯定是开启的，也就是说linux驱动中肯定都使用的是虚拟地址。而纯裸机程序中根本不会开MMU，全部使用的是物理地址。这是裸机下和驱动中操控硬件的一个重要区别。

(2)uboot早期也是纯物理地址工作的，但是现在的uboot也开启了MMU做了虚拟地址映射，这个东西驱动也必须考虑。查uboot中的虚拟地址映射表，发现除了0x30000000-0x3FFFFFFF映射到了0xC0000000-0xCFFFFFFF之外，其余的虚拟地址空间全是原样映射的。而我们驱动中主要是操控硬件寄存器，而S5PV210的SFR都在0xExxxxxx地址空间，因此驱动中不必考虑虚拟地址。

### 10.1.3. uboot借用（移植）了linux驱动

(1)linux驱动本身做了模块化设计。linux驱动本身和linux内核不是强耦合的，这是linux驱动可以被uboot借用（移植）的关键。

(2)uboot移植了linux驱动源代码。uboot是从源代码级别去移植linux驱动的，这就是linux系统的开源性。

(3)uboot中的硬件驱动比linux简单。linux驱动本身有更复杂的框架，需要实现更多的附带功能，而uboot本质上只是个裸机程序，uboot移植linux驱动时只是借用了linux驱动的一部分而已。

## 10.2 iNand/SD驱动解析

### 10.2.1.从start\_armboot开始

驱动整体比较庞大，涉及很多个文件夹下的很多文件，函数更多，贸然插入根本不知道看哪里。学习时必须有顺序。

### 10.2.2. mmc\_initialize

(1)函数位于：uboot/drivers/mmc/mmc.c。

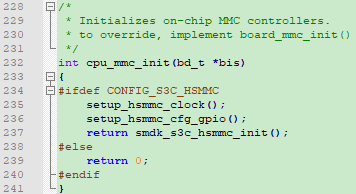
(2)从名字可以看出，这个函数的作用就是初始化开发板上MMC系统。MMC系统的初始化应该包含这么几部分：SoC里的MMC控制器初始化（MMC系统时钟的初始化、SFR初始化）、SoC里MMC相关的GPIO的初始化、SD卡/iNand芯片的初始化。

(3)mmc\_devices链表全局变量，用来记录系统中所有已经注册的SD/iNand设备。所以向系统中插入一个SD卡/iNand设备，则系统驱动就会向mmc\_devices链表中插入一个数据结构表示这个设备。



### 10.2.3. cpu\_mmc\_init

(1)函数位于：uboot/cpu/s5pc11x/cpu.c中。实质是通过调用3个函数来完成的。



(2)setup\_hsmmc\_clock，在uboot/cpu/s5pc11x/setup\_hsmmc.c中。看名字函数是用来初始化SoC中MMC控制器中的时钟部分的。

(3)setup\_hsmmc\_cfg\_gpio，在uboot/cpu/s5pc11x/setup\_hsmmc.c中。看名字函数是用来配置SoC中MMC控制器相关的GPIO的。

### 10.2.4. smdk\_s3c\_hsmmc\_init

(1)函数位于：uboot/drivers/mmc/s3c\_hsmmc.c中。

(2)函数内部通过宏定义USE\_MMCx来决定是否调用s3c\_hsmmc\_initialize来进行具体的初始化操作。

### 10.2.5. s3c\_hsmmc\_initialize

(1)函数位于：uboot/drivers/mmc/s3c\_hsmmc.c中。

(2)定义并且实例化一个struct mmc类型的对象（定义了一个指针，并且给指针指向有意义的内存，或者说给指针分配内存），然后填充它的各种成员，最后调用mmc\_register函数来向驱动框架注册这个mmc设备驱动。

(3)mmc\_register功能是进行mmc设备的注册，注册方法其实就是将当前这个struct mmc使用链表连接到mmc\_devices这个全局变量中去。

(4)我们在X210中定义了USE\_MMC0和USE\_MMC2，因此在我们的uboot初始化时会调用2次s3c\_hsmmc\_initialize函数，传递参数分别是0和2，因此完成之后系统中会注册上2个mmc设备，表示当前系统中有2个mmc通道在工作。

(5)至此cpu\_mmc\_init函数分析完成。

### 10.2.6. find\_mmc\_device

(1)这个函数位于：uboot/drivers/mmc/mmc.c中。

(2)这个函数其实就是通过mmc设备编号来在系统中查找对应的mmc设备（struct mmc的对象，根据上面分析系统中有2个，编号分别是0和2）。

(3)函数工作原理就是通过遍历mmc\_devices链表，去依次寻找系统中注册的mmc设备，然后对比其设备编号和我们当前要查找的设备编号，如果相同则就找到了要找的设备。找到了后调用mmc\_init函数来初始化它。

### 10.2.7. mmc\_init

(1)函数位于：drivers/mmc/mmc.c中。

(2)分析猜测这个函数应该要进行mmc卡的初始化了（前面已经进行了SoC端控制器的初始化）

(3)函数的调用关系为：

mmc\_init

mmc\_go\_idle

mmc\_send\_cmd

mmc\_send\_if\_cond

mmc\_send\_cmd

······

具体分析可以看出，mmc\_init函数内部就是依次通过向mmc卡发送命令码（CMD0、CMD2那些）来初始化SD卡/iNand内部的控制器，以达到初始化SD卡的目的。

### 10.2.8. 总结

(1)至此整个MMC系统初始化结束。

(2)整个MMC系统初始化分为2大部分：SoC这一端的MMC控制器的初始化，SD卡这一端卡本身的初始化。前一步主要是在cpu\_mmc\_init函数中完成，后一部分主要是在mmc\_init函数中完成。

(3)整个初始化完成后去使用sd卡/iNand时，操作方法和mmc\_init函数中初始化SD卡的操作一样的方式。读写sd卡时也是通过总线向SD卡发送命令、读取/写入数据来完成的。

(4)顺着操作追下去，到了mmc\_send\_cmd函数处就断了，真正的向SD卡发送命令的硬件操作的函数找不到。这就是学习驱动的麻烦之处。

(5)struct mmc结构体是关键。两部分初始化之间用mmc结构体来链接的，初始化完了后对mmc卡的常规读写操作也是通过mmc结构体来链接的。

### 10.2.9. struct mmc

(1)驱动的设计中有一个关键数据结构。譬如MMC驱动的结构体就是struct mmc这些结构体中包含一些变量和一些函数指针，变量用来记录驱动相关的一些属性，函数指针用来记录驱动相关的操作方法。这些变量和函数指针加起来就构成了驱动。驱动就被抽象为这个结构体。

(2)一个驱动工作时主要就分几部分：驱动构建（构建一个struct mmc然后填充它）、驱动运行时（调用这些函数指针指针的函数和变量）

### 10.2.10.分离思想

(1)分离思想就是说在驱动中将操作方法和数据分开。

(2)操作方法就是函数，数据就是变量。所谓操作方法和数据分离的意思就是：在不同的地方来存储和管理驱动的操作方法和变量，这样的优势就是驱动便于移植。

### 10.2.11.分层思想

(1)分层思想是指一个整个的驱动分为好多个层次。简单理解就是驱动分为很多个源文件，放在很多个文件夹中。譬如本课程讲的mmc的驱动涉及到drivers/mmc下面的2个文件和cpu/s5pc11x下的好几个文件。

(2)以mmc驱动为例来分析各个文件的作用：

uboot/drivers/mmc/mmc.c：本文件的主要内容是和MMC卡操作有关的方法，譬如MMC卡设置空闲状态的、卡读写数据等。但是本文件中并没有具体的硬件操作函数，操作最终指向的是struct mmc结构体中的函数指针，这些函数指针是在驱动构建的时候和真正硬件操作的函数挂接的（真正的硬件操作的函数在别的文件中）。

uboot/drivers/mmc/s3c\_hsmmc.c:本文件中是SoC内部MMC控制器的硬件操作的方法，譬如向SD卡发送命令的函数（s3c\_hsmmc\_send\_command），譬如和SD卡读写数据的函数（s3c\_hsmmc\_set\_ios），这些函数就是具体操作硬件的函数，也就是mmc.c中需要的那些硬件操作函数。这些函数在mmc驱动初始化构建时（s3c\_hsmmc\_initialize函数中）和struct mmc挂接起来备用。

分析：mmc.c和s3c\_hsmmc.c构成了一个分层，mmc.c中调用了s3c\_hsmmc.中的函数，所以mmc.c在上层，s3c\_hsmmc.c在下层。这两个分层后我们发现mmc.c中不涉及具体硬件的操作，s3c\_hsmmc.c中不涉及驱动工程时的时序操作。因此移植的时候就有好处：譬如我们要把这一套mmc驱动移植到别的SoC上mmc.c就不用动，s3c\_hsmmc.c动就可以了；譬如SoC没变但是SD卡升级了，这时候只需要更换mmc.c，不需要更换s3c\_hsmmc.即可。

(3)cpu/s5pc11x/下面还有一个setup\_hsmmc.c，也和MMC驱动有关。但是这些代码为什么不能放到drivers目录下去，而要放到cpu目录下去？因为这里面的2个函数（setup\_hsmmc\_clock和setup\_hsmmc\_cfg\_gpio）都是和SoC有关的初始化函数，这两个函数不能放到drivers目录下去。实际上如果非把这两个函数放在uboot/drivers/mmc/s3c\_hsmmc.c文件中也凑活能说过去。

# 十一、uboot的移植1-从三星官方uboot开始移植

## 11.1移植前的准备工作

### 11.1.1.三星移植过的uboot源代码准备

(1)三星对于S5PV210的官方开发板为SMDKV210，对应的移植过的uboot是：网盘/2.uboot和linux内核移植（努力更新中）\资源下载\三星官方为210移植过的uboot和kernel/android\_uboot\_smdkv210.tar.bz2

(2)这个源代码网上是下载不到的，三星官方是不会把这些东西放在官网上下载的。这些东西都是随着官方的开发板一起流出的。

### 11.1.2. SourceInsight准备

(1)移植的时候最重要的工作就是看代码、改代码然后编译运行测试。

(2)编译代码必须在linux中（windows共享文件夹中配置uboot不行的），那么看代码和改代码可以在linux中（vim、gedit）也可以在windows中（Sourceinsight）。

(3)我一般习惯的方式是：在windows中解压一份uboot源代码，在linux中也解压一份，注意这两份代码原始情况是一模一样的。然后移植的时候是在windows中这一份中去看代码、改代码；在linux中这一份去编译烧写。这种做法需要在windows中和linux中2份代码之间保持同步（直白点说就是windows中改过了后要把改过的源代码复制到linux中那一份去覆盖linux中那一份里面的同目录同文件）。

(4)问题来了，怎么在windows中和linux中同步代码？通过共享文件夹在linux中进行复制（cp /mnt/hgfs/winshare/xxx.c ./）；通过一些专用工具，譬如sshsecureshell。

### 11.1.3.便捷的文件传输工具sshsecureshell

(1)windows中安装sshsecureshell客户端。

(2)虚拟机上网。不同的ubuntu版本中网卡重启的命令不同，在ubuntu10.04中网卡重启使用：/etc/init.d/networking restart 或者 service network restart就可以；但是在ubuntu14.04中这两个都不行了，要重启网卡需要使用：ifdown eth0 然后ifup eth0即可。

## 11.2 ubuntu14.04上网及安装openssh

### 11.2.1. ubuntu14.04上网问题

(1)虚拟机上网有2种模式，NAT和桥接。

(2)虚拟机上网配置要注意这几个地方：第一个要注意选择NAT还是桥接模式；第二个要注意ubuntu中网络配置文件/etc/network/interfaces，这个文件中是配置网卡信息的（譬如静态ip还是dhcp，静态ip地址是多少等）；第三个如果是桥接模式要注意桥接到哪个网卡上（一般笔记本都有2个网卡，一个是有线网卡一个是无线网卡），如果是NAT模式没关系。

(3)ubuntu14.04中重启网卡的命令变了。

(4)演示：NAT方式下上网配置过程。

前提是主机windows通过无线wifi上网了（或者windows通过有线上网也可以）。然后在虚拟机中配置2点：第一选择NAT方式，第二配置/etc/network/inerfaces文件中使用dhcp方式；然后重启网卡，确认ip地址得到后即可上网。

(5)演示：桥接方式上网配置过程。

前提是主机windows通过无线wifi上网了(主机通过有线网卡上网的配置方式有不同)。然后虚拟机中配置注意三点：第一选择桥接方式，第二配置桥接到无线网卡（如果主机windows是通过有线上网的，则桥接到有线网卡）；第三配置/etc/network/inerfaces文件中使用dhcp方式；然后重启网卡，确认ip地址得到后即可上网。

(6)演示：虚拟机要ping通开发板的配置过程。

前提是开发板和电脑之间通过网线连接好。然后虚拟机中配置注意三点：第一选择桥接方式，第二配置桥接到有线网卡（如果配置为自动或者配置桥接到无线网卡则肯定无法ping通开发板）；第三配置/etc/network/inerfaces文件中使用static方式，ip地址配置保证和主机windows、开发板三者处于同一网段；然后重启网卡，确认ip地址得到后即可ping通开发板。

(7)windows系统中有一个bug，如果windows没有检测到有线网卡连接了外部网络则windows中本地连接是不工作的，网卡不工作。解决方案是用网线随便连接一个有联网能力的东西即可，譬如网线连接你的电脑到旁边兄弟的电脑上，譬如插上你的开发板（开发板中运行了linux系统），譬如插上路由器端口。

### 11.2.2.搭建openssh环境

(1)安装ssh-server。(sudo apt-get install openssh-server)如果报错提示依赖错误，可以参考：http://www.cnblogs.com/mliudong/p/4094519.html

(2)securecrt登录

(3)sshsecureshell登录

ssh登录不上，要修改/etc/ssh/sshd\_config，参考：http://blog.sina.com.cn/s/blog\_5f435c130102v6pv.html。 修改完重启时如果/etc/init.d/ssh restart不起作用，可以使用：ps -e | grep ssh，看sshd的进程号，然后kill -9 进程号杀死ssh进程以达到重启的目的，或者直接重启ubuntu系统

## 11.3 移植初体验

### 11.3.1.直接编译三星移植版uboot尝试运行

(1)复制到linux的源生目录下，然后解压开。

(2)检查Makefile中的交叉编译工具链

(3)配置时使用：make smdkv210single\_config，对应include/configs/smdkv210single.h头文件。

(4)配置完成后直接make编译，编译完成后就进入烧录步骤。

(5)uboot/sd\_fusing目录下有sd\_fusing.sh脚本，用来烧录。

### 11.3.2.代码分析&问题查找

运行结果是：第一，串口无输出；第二，开发板供电锁存成功。

分析运行结果：uboot中串口最早的输出在"OK"，在lowlevel\_init.S中初始化串口时打印出来的；串口无输出"O"说明在打印"O"之前代码已经死掉了；开发板供电锁存在lowlevel\_init.S中，开发板供电锁存成功说明这个代码之前的部分是没问题的。两个结合起来得到结论：错误在开发板供电锁存代码和串口初始化打印"O"代码之间。

(1)Windows下建立SourceInsight工程

(2)顺藤摸瓜去找可能出问题的地方

整个程序运行是从start.S开始的，看代码也从这里开始。

因为我们板子上没有Power Management IC，实际上只要屏蔽掉bl PMIC\_InitIp这一行代码，然后重新编译，整个uboot就启动起来了。但是很多配置信息是有问题的，很多功能应该也是不能用的，都要去一一查验。

## 11.4时钟和DDR的配置移植

更改CONFIG\_IDENT\_STRING为" for Luffy210"，然后同步到ubuntu中的一份代码，然后 make distclean; make smdkv210single\_config，然后make，然后烧录运行，检查打印出来的banner信息是否如我们改动的那样。

### 11.4.1.确认时钟部分的配置

(1)时钟部分的运行结果本来就是对的，时钟部分的代码在lowlevel\_init.S中的bl system\_clock\_init调用的这个函数中。函数的代码部分是没任何问题的，根本不需要改动，要改动的是寄存器写入的值，这些值都在配置头文件（smdkv210single.h）中用宏定义定义出来了。如果时钟部分要更改，关键是去更改头文件中的宏定义。

(2)三星移植时已经把210常用的各种时钟配置全都计算好用宏开关来控制了。只要打开相应的宏开关就能将系统配置为各种不同的频率。可以尝试切换宏开关到其他时钟频率下试一下，但是uboot可能会有问题，因为和时钟相关的一些配置（例如波特率）是写死的，这样的话也需要去修改相应的波特率才行。而我们这一份uboot中，波特率就是写死的。

### 11.4.2. DDR配置信息的更改

(1)从运行信息以及bdinfo命令看到的结果，显示DRAM bank0和1的size值都设置错了。

(2)使用md（memery display）和mw（memery write）命令测试内存，发现20000000和40000000开头的内存都是可以用的，说明代码中DDR初始化部分是正确的，只是size错了。同时还发现20000000和30000000使用的内存是一模一样的，具体为什么会是这样不得而知。

(3)内存部分配置成：

#define CONFIG\_NR\_DRAM\_BANKS 2 /\* we have 2 bank of DRAM \*/

//#define SDRAM\_BANK\_SIZE 0x20000000 /\* 512 MB \*/

#define SDRAM\_BANK\_SIZE 0x10000000 /\* 256 MB \*/

#define PHYS\_SDRAM\_1 MEMORY\_BASE\_ADDRESS /\* SDRAM Bank #1 \*/

#define PHYS\_SDRAM\_1\_SIZE SDRAM\_BANK\_SIZE

//#define PHYS\_SDRAM\_2 (MEMORY\_BASE\_ADDRESS + SDRAM\_BANK\_SIZE) /\* SDRAM Bank #2 \*/

#define PHYS\_SDRAM\_2 0x40000000

#define PHYS\_SDRAM\_2\_SIZE SDRAM\_BANK\_SIZE

## 11.5 将DDR端口0地址配置为30000000开头

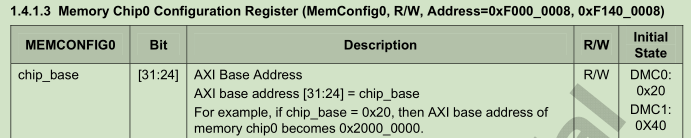
更改有2个目的：第一是让大家体验内存配置的更改过程；第二是3开头的地址和DRAM bank1上40000000开头的地址就连起来了。这样我们就得到了地址连续的512MB内存，而原来我们得到的512MB内存地址是断续的。

### 11.4.1. DDR初始化参数更改

(1)根据裸机中讲DDR初始化部分的课程，和uboot前面分析uboot中DDR初始化部分的代码的课程，得出结论就是：DDR的初始化代码部分是在lowlevel\_init.S中写的，是不动的。代码部分就是对相应寄存器做相应值的初始化；要动的是值，而uboot为了具有可移植性把值都宏定义在include/configs/xxx.h中了。因此我们只需要去这个配置头文件中更改配置值即可。

(2)更改内容是：#define DMC0\_MEMCONFIG\_0 0x20E01323改为：

#define DMC0\_MEMCONFIG\_0 0x30E01323 注意20改为30了。



这8位是用来配置AXI基地址。AXI是一种总线协议，我们只要知道，在ARM CPU内部，它用来传输数据，例如它用来传输内核和DMC0之间的数据。如果chip\_base=0x30，则DMC0的chip0的基地址就是0x30000000。可见它决定了AXI基址的高8位。

### 11.4.2. smdkv210single.h中相关宏定义修改

(1)寄存器的值改了后相当于是硬件配置部分做了更改。但是uboot中DDR相关的一些软件配置值还没更改，还在原来位置，所以要去更改。

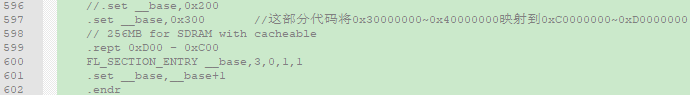
(2)#define MEMORY\_BASE\_ADDRESS 0x20000000改为：

#define MEMORY\_BASE\_ADDRESS 0x30000000

### 11.4.3.虚拟地址映射表中相应修改

(1)uboot中开启了MMU对内存进行了段式映射，有一张内存映射表。之前课程中分析过，分析方法是一样的。

(2)经过实际分析，发现这个内存映射只是把20000000开始的256MB映射到C0000000开头的256MB。我们更改方法是将2改成3.



(3)为了安全起见，再去配置头文件smdkv210single.h中查一遍，看看有没有其他的宏定义值和内存配置有关联的。

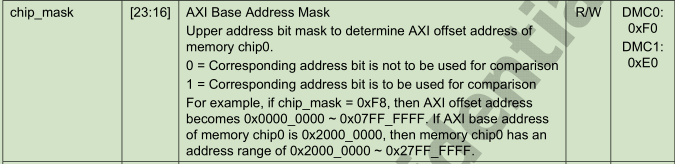
重新配置编译，烧录运行查看结果。

### 11.4.4.修改DMC0的配置参数

(1)修改DDR中DMC0的memconfig\_0寄存器的配置值，将

#define DMC0\_MEMCONFIG\_0 0x30E01323 改为：

#define DMC0\_MEMCONFIG\_0 0x30F01323



这8位是用来决定chip0在内存地址中的偏移地址的，也就是设置chip0所连接的DRAM容量。譬如，如果设定chip\_mask=0Xf0。把它换成二进制为：0b1111\_0000，那么它将屏蔽最高的4位[31:28]，那么得到的结果为：0b0000\_1111，即为0x0F。其余的低24位用1补齐，则连接DMC0的chip0上的DRAM容量为0x0FFF\_FFFF。得到chip0的地址范围为：0x3000\_0000~0x3FFF\_FFFF。即容量为256MBytes。

(2)然后重新同步、编译烧写运行，发现uboot第二阶段运行了，但是整个uboot还是不成功。

(3)分析问题，寻找解决方案。分析方法有2种：第一种靠经验、靠发现能力、靠直觉去找；第二种就是在整个代码中先基本定位错误地方，然后通过在源代码中添加打印信息来精确定位出错的代码，然后找到精确的出错位置后再去分析错误原因，从而找到解决方案。

### 11.4.5.修改修改虚拟地址到物理地址的映射函数

(1)修改uboot/board/samsung/smdkc110/smdkc110.c中的virt\_to\_phy\_smdkc110，将其中的20000000改为30000000即可。

(2)同步代码，然后重新编译烧录运行。

## 11.5 iNand驱动问题的解决

### 11.5.1.先从现象出发定位问题

(1)解决问题的第一步，是定位问题。所谓定位问题就是找到源代码当中导致这个问题的那一句或者那几句代码。有时候解决这个问题需要修改的代码和直接导致这个问题的代码是不同的。我们这里说的定位问题指的是定位到出问题的代码处，也就是运行了这一句代码时发生了这个错误。

(2)定位了问题之后，实际修改程序解决问题不一定改的是这一句代码。但是肯定和这一句代码有关联，我们要通过自己分析来找到这种关联，从而从定位的错误点找到真正需要修改的点，然后去修改他。

(3)实战方法：从打印出来的错误休息中挑选一个关键词，然后去源代码中搜索这个关键字，通过这种搜索的方法定位问题。通过搜索将问题定位在drivers/mmc/mmc.c的818行。

(4)然后就是解决问题了。

### 11.5.2.网络搜索解决方案

(1)初步的解决方案是自己先浏览一遍这个问题点周边代码上下文。通过浏览代码上下文，发现这个函数是在读取SD/iNand的ext\_csd寄存器的值。通过浏览代码结合出错地方，可以判断出：从卡端读取ext\_csd寄存器是成功的，并且从读取结果中拿到了卡的版本号信息。然后代码对版本号进行了判断，并且如果版本号大于5就会报错并且函数错误退出。这就是问题所正。

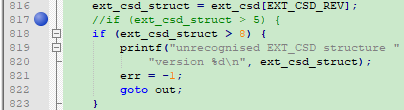
(2)问题就是：我们使用的iNand卡的版本号大于5，而uboot代码本身不处理版本号大于5的卡，因此出错了。

(3)怎么解决？第一可能，换卡；第二可能，软件修复。

(4)网络搜索错误关键字，然后逐个去查阅，看看哪个可以给我们提供解决问题的思路和方法。http://blog.csdn.net/wang\_shuai\_ww/article/details/22308853

### 11.5.3.尝试修改代码解决问题

(1)解决方法就是修改uboot中的代码，把判断的5改成更大的数字。譬如8，然后跳过这个错误。



### 11.5.4.推测和实验验证（SD卡和iNand的区别）

(1)当前板子上有一个iNand接在SD0上，有一个外置SD卡接在SD2上。那uboot中初始化的这个是iNand而不是SD卡。也就是说uboot中实际用的是SD0而不是SD2。

(2)大家可以尝试，使用外置SD卡时，这个版本号的问题不会出现。从这里可以推测出SD卡和iNand的区别，至少从一个角度可以看出：SD卡版本低，iNand的版本比较高。



## 11.6 一些小问题的修补

### 11.6.1控制台串口更换为串口0

(1)uboot中默认使用串口2来做控制台输入输出的。

(2)SOC中一共有4个串口（串口0、1、2、3），开发板X210上用DB9接口引出了2个串口，分别是串口2和串口0.（靠边的是串口2，靠里那个是串口0）。

(3)三星公司推荐使用串口2来作为调试串口，所以在三星移植的uboot和内核版本中都是以串口2默认为控制台串口的。

(4)有时候项目需要将调试串口修改为另外的串口（譬如串口0），这时候需要修改uboot的代码，做移植让uboot工作在串口0的控制台下。

(5)uboot中真正去硬件初始化串口控制器的代码在lowlevel\_init.S中的uart\_asm\_init中，其中初始化串口的寄存器用ELFIN\_UART\_CONSOLE\_BASE宏作为串口n的寄存器的基地址，结合偏移量对寄存器进行寻址初始化。所以uart\_asm\_init中到底初始化的是串口几（从0到3）？取决于ELFIN\_UART\_CONSOLE\_BASE宏。这个宏的值又由CONFIG\_SERIALn（n是从1到4）来决定

(6)同步代码、编译烧录运行，发现串口线插在串口2上，crt上只打印：SD checksum error.（这个是内部iROM打印出来的，内部iNand校验失败的信息）；然后将串口线改插到串口0上，启动，所有的信息出现。实验成功。

### 11.6.2修改默认网络地址设置

(1)修改配置头文件smdkv210single.h中的CONFIG\_IPADDR等宏，则可以修改uboot的默认环境变量。

(2)更改完成后如果环境变量还是原来的，正常。因为原来uboot执行过saveenv，因此环境变量已经被保存到iNand中的ENV分区中去了。uboot启动后校验时iNand的ENV分区中的环境变量是正确的，因此会优先加载。我们在uboot源代码中修改的只是默认的环境变量。解决方案是擦除掉iNand中的那一份环境变量，然后迫使uboot启动时使用uboot代码中自带的默认的这一份环境变量，就可以看到了。

(3)可以使用mmc write 0 30000000 11# 32（表示将DDR的0x30000000开头的一段内存中的内容写入iNand中的第17个扇区开始的32个扇区内，写入长度是32个扇区长度（16KB））

### 11.6.3修改行提示符

(1)#define CFG\_PROMPT "LUFFY210 # "

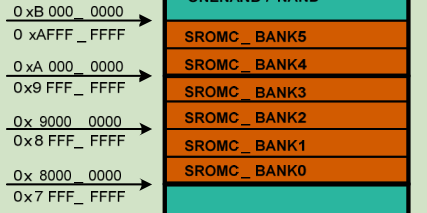
### 11.6.1总结

(1)结论就是对uboot的整体结构和过程了解之后，移植非常简单。

## 11.7 网卡驱动移植1

### 11.7.1.网卡芯片与开发板的连接方式

(1)SoC的SROM bank（SROM就是SRAM/ROM）和网卡芯片的CS引脚。SoC的SROM Controller其实就是SoC提供的对外总线式连接SRAM/ROM的接口。如果SoC要外部外接一些SRAM/ROM类的存储芯片（或者伪装成SROM接口的芯片，譬如网卡芯片）就要通过SROM Controller来连接。网卡接在SROM中好处就是网卡芯片好像一个存储芯片一样被扩展在SoC的一个地址空间中，主机SoC可以直接用一个地址来访问网卡芯片内部寄存器。



(2)网卡芯片内部寄存器使用相对地址访问。网卡芯片内部很多寄存器有一个地址，这个地址是从00开始的，但是实际上我们SoC不能用0地址去访问这个网卡的芯片内部寄存器。SoC访问网卡芯片00寄存器时的地址应该是：起始地址+00这里的起始地址就是网卡芯片对应接在SROM bankn中的bankn对应的基地址。

(3)主机SoC上网，其实就是通过操控网卡芯片内部的寄存器、缓冲区等资源来上网的。也就是说其实SoC是通过网卡芯片来间接上网的。

(4)总结：实际上也是一种总线式连接方式。优势是SoC内部不需要内置网卡控制器，所有的SFR全都在外部网卡芯片中，而且还可以通过地址直接访问（IO与内存统一编址），不用像Nand/SD接口一样使用时序来访问。

(5)从逻辑上来看，网卡更像是串口，而不像是SD/Nand。

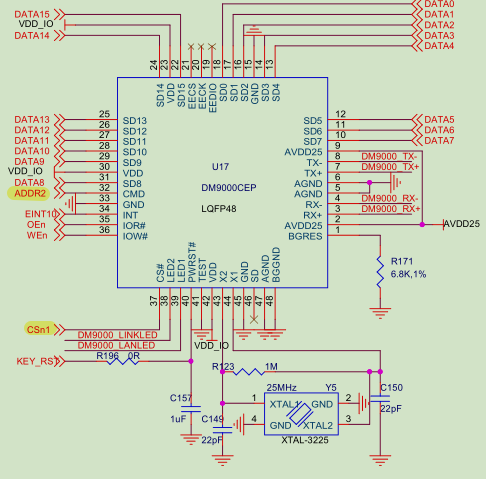
### 11.7.2.原理图浏览

(1)210的SROM控制器允许8/16bit的接口，我们实际使用的是16位接口。

(2)网线有8根线，但是实际只有4根有效通信线，另外4根都是GND，用来抗干扰的。4根通信线中管发送的有2根（Tx-和Tx+），管接收的有2根（Rx+和Rx-）。因为网线上传输的是差分信号。

(3)网卡芯片有个CS引脚，（CS就是chip select，片选信号，主机向CS发送有效信号则从机芯片工作，主机向CS发送无效信号则从机芯片不工作。），这个引脚要接主机SoC的片选信号引脚，主机S5PV210的每一个SROM bank中有一个片选信号CSn(n=0-5)，从原理图可以看出，我们X210上将DM9000的CS引脚接到了CSn1上，对应SROM bank1（推断出DM9000的总线地址基地址是0x88000000）。

(4)DM9000的CMD引脚接到了S5PV210的ADDR2引脚上。DM9000为了减少芯片引脚数，数据线和地址线是复用的（DATA0到DATA15这16根线是有时候做数据线传输数据，有时候做地址线传输地址的。什么时候做什么用就由CMD引脚决定。）通过查询数据手册知道：当CMD为高电平时对应传输是DATA，当CMD为低电平时对应传输为INDEX（offset，寄存器地址）。



注明：这些引脚上的电平变化都是控制器自动的，不需要程序员手工干预。程序员所需要做的就是在配置寄存器值时充分考虑到硬件电路的接法，然后给相应寄存器配置正确的数值即可。

## 11.8网卡驱动移植2

### 11.8.1网卡驱动文件介绍

(1)uboot中本来就提供了很多网卡芯片的驱动程序，在uboot/drivers/net/dm9000x.c和dm9000x.h。这个驱动来自于linux kernel源代码。所以我们uboot中是移植而不是编写。

(2)要想彻底看懂这个驱动，必须对linux的驱动模型中网络设备驱动有一定的理解才可以。因为我们还没学驱动，因此这个源代码就不用看了。

(3)这个驱动是linux内核中做好的，根本不用动可以在uboot中直接使用的。而且因为linux驱动设计的很合理（数据和代码是分开的，这里驱动主要是代码，数据是由硬件开发板中的接法决定的，数据由一定的数据结构来提供。），所以驱动本身具有可移植性。这个就决定了我们移植DM9000驱动时这个驱动文件dm9000x.c和h不用动，要动的是数据。

### 11.8.2.网卡移植的关键：初始化

(1)uboot在第二阶段init\_sequences中进行了一系列的初始化，其中就有网卡芯片的初始化。这个初始化就是关键，在这里的初始化中只要将网卡芯片正确的初始化了，则网卡芯片就能工作（意思是网卡驱动dm9000x.c和dm9000x.h依赖于这里的初始化而工作）。

(2)网卡初始化代码地方在：

start\_armboot

init\_sequence

board\_init

dm9000\_pre\_init 这个函数就是移植的关键

(3)dm9000\_pre\_init函数主要功能就是初始化DM9000网卡。这个初始化过程和我们开发板上DM9000网卡芯片的硬件连接方式有关。必须要结合开发板原理图来分析，然后决定这个函数怎么编程。

(4)原来的代码是三星的工程师根据三星的开发板SMDKV210的硬件接法来写的程序，我们要根据自己的开发板的硬件接法去修改这个程序，让网卡在我们的开发板上能工作。

(5)#define DM9000\_16BIT\_DATA这个宏用来表示DM9000工作在16位总线模式下。根据上节课的硬件原理图的分析，可以看到我们开发板上DM9000确实工作在16位模式下。

(6)从三星版本的代码中可以看出，它操作的是bit20-bit23，对照数据手册中寄存器定义，可以看出三星的开发板DM9000是接在Bank5上的。而我们接在bank1上的，因此我们需要操作的bit位是bit4-bit7

(7)总结：三个寄存器的修改。主要是三星的开发板DM9000接在bank5，我们接在了bank1上，因此要做一些修改。

### 11.8.3.基地址的配置等

(1)之前说过，驱动分为2部分：代码和数据。代码不用动，数据要修改。

(2)CONFIG\_DM9000\_BASE是DM9000网卡通过SROM bank映射到SoC中地址空间中的地址。这个地址的值取决于硬件接到了哪个bank，这个bank的基地址是SoC自己定义好的。譬如我们这里接到了bank1上，bank1的基地址是0x88000000.

(3)DM9000\_IO表示访问芯片IO的基地址，直接就是CONFIG\_DM9000\_BASE；DM9000\_DATA表示我们访问数据时的基地址，因为DM9000芯片的CMD引脚接到了ADDR2，因此这里要+4（0b100，对应ADDR2）

(4)本来这样配置就完了，重新编译运行网卡就应该工作了。但是实际测试发现不工作，要怎么样修改呢？修改方式是将CONFIG\_DM9000\_BASE改成0x88000300就工作了。

问题？这个0x300从哪里来的？我得出的感觉最靠谱的解释是：跟DM9000网卡芯片型号版本有关，我认为这个0x300是DM9000网卡本身的问题，他本身的内部寄存器就有一个0x300的一个偏移量。

## 11.9网卡驱动如何工作

### 11.9.1.网卡移植代码实践

(1)经过实践，网卡驱动移植成功。

(2)其实还可以做一些实验。譬如说对网卡驱动初始化部分寄存器的设置，还有网卡CONFIG\_DM9000\_BASE也可以配成0x88000000再去试一试。

### 11.9.2.linux系统中网卡驱动的典型工作方式简介

(1)在linux系统中，网卡算是一个设备，这个设备驱动工作后会生成一个设备名叫ethn（n是0、1、2、…）（无线网卡名字一般叫wlan0、wlan1…）。然后linux系统用一些专用命令来操作网卡，譬如ifconfig命令。

(2)linux下的应用程序如何使用网卡驱动来进行网络通信？最通用的方法就是socket接口。linux系统中有一系列的API和库函数，提供了一个socket编程接口，linux下的应用程序都是通过socket来实现上网的，socket内部就是间接调用的网卡驱动实现网络通信的。

(3)linux设计是非常完备的，应用层和驱动层是严格分离的。也就是说写网络编程应用层的人根本不用管驱动，只要会用socket接口即可；写底层驱动的人根本不用管应用层，只要面向linux的网络驱动框架模型即可。

### 11.9.3.uboot中网卡驱动的工作方式简介

(1)一定要记住：uboot本身是一个裸机程序，是一个整体，没有分层。所以uboot中根本没有驱动和应用的概念。

(2)按照逻辑来说，ping这样的命令实现的代码就是网络应用的应用程序，像dm9000x.c和dm9000x.h这样的代码属于驱动程序。所以在uboot中这些东西是揉在一起的，应用是直接调用驱动实现的。也就是说ping命令内部是直接调用了dm9000的网卡驱动中的函数来实现自己的。

### 11.9.4.以ping命令为例查找代码验证分析

(1)ping命令是uboot的众多命令之一，ping命令实现的函数叫do\_ping

(2)函数的调用关系：

do\_ping

NetLoop

PingStart

PingSend

ArpRequest

eth\_send（dm9000x.c中）

(3)验证了11.9.3中说的uboot中应用程序（ping）调用驱动程序（dm9000x.c）的方式。这就是一种直接调用的方式。

## 11.10 使用自己移植的uboot启动内核

问题：当前uboot不能启动内核

(1)用同样的方法（使用tftp 0x30008000 zImage-qt; 然后bootm 0x30008000），分别使用我们自己移植的uboot和使用九鼎移植版本的uboot去启动内核，发现九鼎移植版本的可以启动，但是我们自己移植的不可以启动。到此我们就断定我们的uboot有问题，不能启动内核。

(2)做基本检查：首先怀疑是机器码不对。经过和九鼎移植版本的uboot对比发现machid都是2456，说明机器码没错。

(3)想到一个问题，我们之前做实验时将串口改为了串口0，而内核zImage-qt的串口输出在串口2.怀疑可能的问题是uboot使用了串口0而内核使用了串口2所以在uboot后看不到内核的启动信息。

解决第一步：将串口改回串口2

(1)在smdkv210single.h中修改串口编号即可。

根据现象分析，定位问题并试图解决

(1)如果已经启动了内核，那没什么好说的了。应该是可以直接启动了。

(2)如果内核没有启动，是smdkv210single.h中没有定义bootm传参需要的那几个宏造成的。

# 十二、从官方原版uboot开始移植

## 12.1官方原版uboot的选择

### 12.1.1.官方原版uboot的版本

(1)版本号。刚开始是1.3.4，后来变成2009.08

(2)新版和旧版的差别。uboot的架构很早就定下来了，然后里面普遍公用的东西（common目录下、drivers目录下、fs目录下等……）在各个版本之间几乎是完全一样的。差别最大的是board和cpu目录，这两个目录正是单板（开发板）相关的。越新的uboot版本支持越多的开发板（CPU），所以越新的uboot越庞大。

(3)并不是越新的版本就越好。越新的uboot中会多出更多的开发板的支持代码，如果我们的开发板并不是很新，就没必要去用很新版本的uboot。因为多出来的代码自己也用不到而且还会成为累赘。

### 12.1.2.新版uboot配置体系的改变

(1)在最新的uboot版本（准确的说是2013.10到2014.10中的某个版本）中，uboot的文件体系发生了一个很大的变化。这个变化就是uboot引入了linux kernel的配置体系（Kbuild、Kconfig、menuconfig），从而让我们可以在图形界面下，像配置内核一样配置uboot。

(2)所以新版本的uboot配置时和我们之前的课程讲的就不同了。我们移植时不能选择这种配置方式更改之后的uboot版本。我们要选择更改之前的。

(3)新版本的配置方式本质上和linux kernel一样的，所以在学完linux kernel移植后自己就能看懂，因此不用担心。在工作中一般是不需要从uboot官方版本出发去做移植的，而是从SoC厂商提供的开发板配套的uboot去做移植的。

## 12.2 初步浏览官方原版uboot

### 12.2.1.文件夹结构浏览

(1)文件夹结构分析、主要文件检视

总的来说，文件夹结构和以前基本一样。不同的主要是lib，以前是lib\_arm和lib\_generic，现在是arch和lib。arch目录下放的是和cpu架构有关的东西。

总的来说，2013.10版本的uboot在结构上和1.3.4版本的uboot还是有所不同的。

(2)参照物开发板的选择

我们开发板使用的CPU是S5PV210，所以要找uboot中针对S5PV210或者S5PC110进行移植的作为参考。

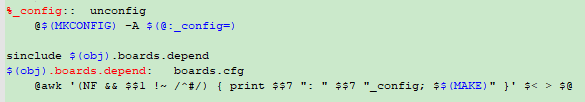
根据规律，我们应该参考include/configs/s5p\_goni.h，对应的board在uboot/board/samsung/goni这个目录。

(3)删除无关文件和文件夹

### 12.2.2.建立SI工程并预解析

### 12.2.3.主Makefile浏览及boards.cfg文件

(1)2013.10版本的uboot的Makefile中使用了boards.cfg文件，因此在配置uboot时make xxx\_config，这个xxx要到boards.cfg文件中查找。



(2)其实就相当于把以前的版本的uboot中各种开发板的配置部分规则抽离出来写到了Makefile中，然后把配置信息部分写到了一个独立文件boards.cfg。

### 12.2.4.结论

(1)参照物开发板为：55p\_goni

(2)配置对应的cpu、board文件夹分别为：

cpu: u-boot-2013.10\arch\arm\cpu\armv7

board: u-boot-2013.10\board\samsung\goni

## 12.3 mkconfig脚本分析

### 12.3.1.脚本功能浏览

(1)首先我们在命令行配置uboot时，是：make s5p\_goni\_config，对应Makefile中的一个目标。

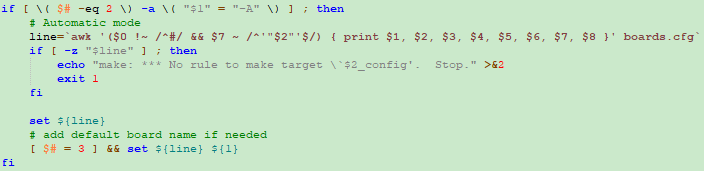
(2)新版本的Makefile中：

%\_config:: unconfig

@$(MKCONFIG) -A $(@:\_config=)

从这里分析得出结论，实际配置时是调用mkconfig脚本，然后传参2个：-A和s5p\_goni

1. 到了mkconfig脚本中了。在24到35行中使用awk正则表达式将boards.cfg中与刚才$1（s5p\_goni）能够匹配上的那一行截取出来赋值给变量line，然后将line的内容以空格为间隔依次分开，分别赋值给$1、$2……$8

。

(4)注意在解析完boards.cfg之后，$1到$8就有了新的值。

$1 = Active

$2 = arm

$3 = armv7

$4 = s5pc1xx

$5 = samsung

$6 = goni

$7 = s5p\_goni

$8 = -

### 12.3.2.几个传参和其含义

(1)几个很重要的变量

arch=arm

cpu=armv7

vendor=samsung

soc=s5pc1xx

### 12.3.3.符号链接

(1)include/asm -> arch/arm/include/asm

(2)include/asm/arch -> include/asm/arch-s5pc1xx

(3)include/asm/proc -> include/asm/proc-armv

最后创建了include/config.h文件。

### 12.3.4.Makefile中添加交叉编译工具链

(1)官方原版的uboot中CROSS\_COMPLIE是没有定义的，需要自己去定义。如果没定义就直接去编译，就会用gcc编译。

(2)添加一行：

CROSS\_COMPILE = /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-

### 12.3.5.配置编译测试

(1)编译过程：

make distclean

make s5p\_goni\_config

make

(2)结果：得到u-boot.bin即可