**u-boot分析导读**

置顶 2016年03月30日 21:25:56 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：1540

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/51019523

**u-boot分析导读——itxiebo版**

Universal Boot Loader，普遍的引导程序。是非常常用的一个引导程序，可作为主流系统的引导程序，如OpenBSD, NetBSD, FreeBSD,4.4BSD, Linux, SVR4, Esix, Solaris等等，当然同时也支持多种常见的cpu处理器，如MIPS、 x86、ARM、NIOS、XScale等。

u-boot是一种普遍用于嵌入式系统中的[Bootloader](https://www.baidu.com/s?wd=Bootloader&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，Bootloader是在操作系统运行之前执行的一小段程序，通过它，我们可以初始化硬件设备、建立内存空间的映射表，从而建立适当的软硬件环境，为最终调用[操作系统内核](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)做好准备。Boot Loader的主要运行任务就是将内核映象从硬盘(e.g. NAND flash or eMMC)上读到RAM中，然后跳转到内核的入口点去运行，即开始启动操作系统。系统在上电或复位时通常都从地址0x00000000处开始执行，而在这个地址处安排的通常就是系统的Boot Loader程序。

前些日子，利用业余时间，整理了几篇u-boot相关的文章，通过这几天的u-boot学习，相关知识的整理以及博客的撰写，对u-boot的目录结构，u-boot启动内核流程等有了比较清晰的认识。

**u-boot分析 一**

2016年03月18日 01:23:09 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：2522

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50920094

学习u-boot难吗？ 不难！

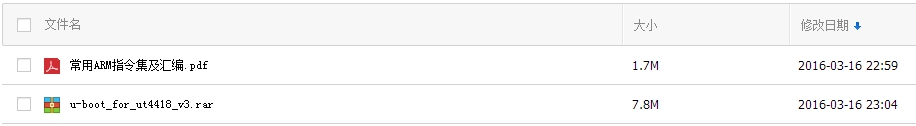
不知你是否和我一样

1. 不懂gcc编译链接原理
2. 不熟悉常用ARM指令集
3. 没有ARM汇编编程经验   
   ……

没有关系，这一切的困难都是纸老虎，即便如此，我们一样可以搞清u-boot启动原理。

欲善其事，必先利其器。

此博文为u-boot学习计划之启动篇，暂不涉及u-boot相关的任何知识，但有两点必须要达到：

1. 安抚民心。   
   结合博主自己的学习历程，个人认为， 对新手而言，学习u-boot是一个痛苦的过程。现在细想，让人痛苦的原因主要是咱们没有一个合理的学习顺序和可以借鉴的学习方法。就如无头苍蝇，东看看，西学学。学习过程中肯定会有很多的疑问和不明白，我的建议是不要死扣细节，把握大体思路，弄清u-boot大体原理为优先前提。给自己一点消化吸收的时间，不可急功近利，妄自菲薄。博主只有业余时间学习之，我给自己的时间是一星期。你给自己准备了多少时间呢？也就是自己给自己多长时间的痛苦呢？   
   
2. 资料分享。   
   安抚民心后，如果你的心态已经OK，那就请在开始之前收下鄙人的“恩情”，百度网盘中share了基于UT4418开发板的**u-boot source code**以及**ARM常用指令集**。学习u-boot就靠它了。   
   <http://pan.baidu.com/s/1kUhYmEj>   
   

最后，请自备**Source Insight**看code软件。

**u-boot分析 二 (u-boot目录结构)**

2016年03月20日 09:31:23 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：6572

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50935248

**u-boot分析 二**

继前一启动篇，此博文目的：

学习u-boot的目录结构，从code架构上对u-boot有一个整体的把握和了解。

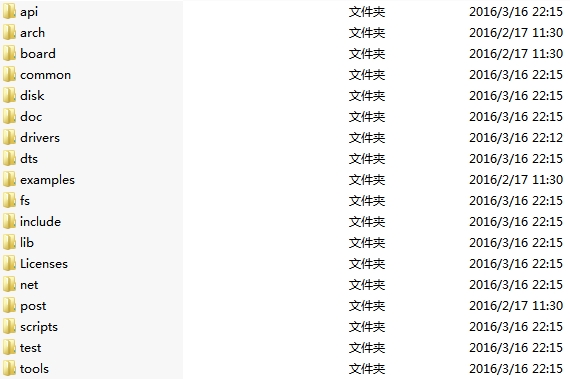
**一、学习u-boot，得问，什么是u-boot?**

u-boot是一段开源的程序。

Universal Boot Loader，普遍的引导程序。是非常常用的一个引导程序，可作为主流系统的引导程序，如OpenBSD, NetBSD, FreeBSD,4.4BSD, Linux, SVR4, Esix, Solaris等等，当然同时也支持多种常见的cpu处理器，如MIPS、 x86、ARM、NIOS、XScale等。

u-boot是一种普遍用于嵌入式系统中的[Bootloader](https://www.baidu.com/s?wd=Bootloader&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，Bootloader是在操作系统运行之前执行的一小段程序，通过它，我们可以初始化硬件设备、建立内存空间的映射表，从而建立适当的软硬件环境，为最终调用[操作系统内核](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)做好准备。Boot Loader的主要运行任务就是将内核映象从硬盘(e.g. NAND flash or eMMC)上读到RAM中，然后跳转到内核的入口点去运行，即开始启动操作系统。系统在上电或复位时通常都从地址0x00000000处开始执行，而在这个地址处安排的通常就是系统的Boot Loader程序。

**二、u-boot目录结构**

u-boot source code，无论是从网络下载(<ftp://ftp.denx.de/pub/u-boot/>)还是来自上一篇博客分享(<http://pan.baidu.com/s/1kUhYmEj>) ，第一次映入眼帘的风景基本是这样的：   


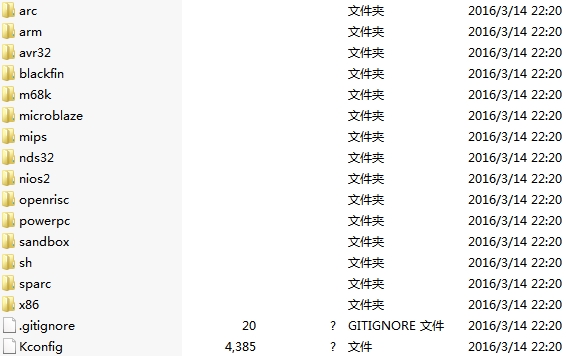
u-boot目录结构，从u-boot-2010.06版本开始，相比老版本

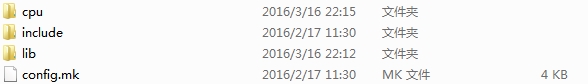
1. cpu与lib\_arch合二为一，命名arch
2. 增加include folder
3. 分离出通用库文件夹lib

u-boot-2010.06及以后版本，基本目录结构如下表

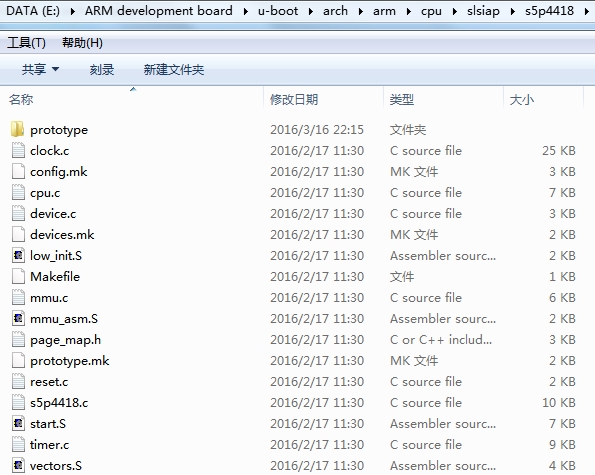
| **Name** | **Description** |
| --- | --- |
| api | 此目录下存放u-boot向外提供的接口函数 |
| arch | 与体系结构相关的代码，核心folder。s5p4418是arm体系结构。 |
| board | 此folder是根据不同的具体开发板而定制的代码，代码也不少 |
| common | 通用代码，涵盖各个方面，以命令行处理为主 |
| disk | 磁盘分区相关代码 |
| doc | 常见功能和问题的说明文档，一堆README开头的文件 |
| drivers | 常用的设备驱动程序，每个类型的设备驱动占用一个子目录 |
| examples | 示例程序 |
| fs | 文件系统，支持嵌入式开发常见的fs(cramfs,ext2,ext3,jffs2,etc) |
| include | 全局需要的头文件定义在这儿 |
| lib | 通用库文件 |
| net | 网络相关的代码，小型的协议栈 |
| post | Power On Self Test，开店自检 |
| Tools | 辅助程序，用于编译和检查uboot目标文件 |

自问，学习u-boot又是为何？   
[无可厚非](https://www.baidu.com/s?wd=%E6%97%A0%E5%8F%AF%E5%8E%9A%E9%9D%9E&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，很大机会是为了移植u-boot到自己的开发板。   
如此，我们就不得不再分析**/arch，/board**两个folder

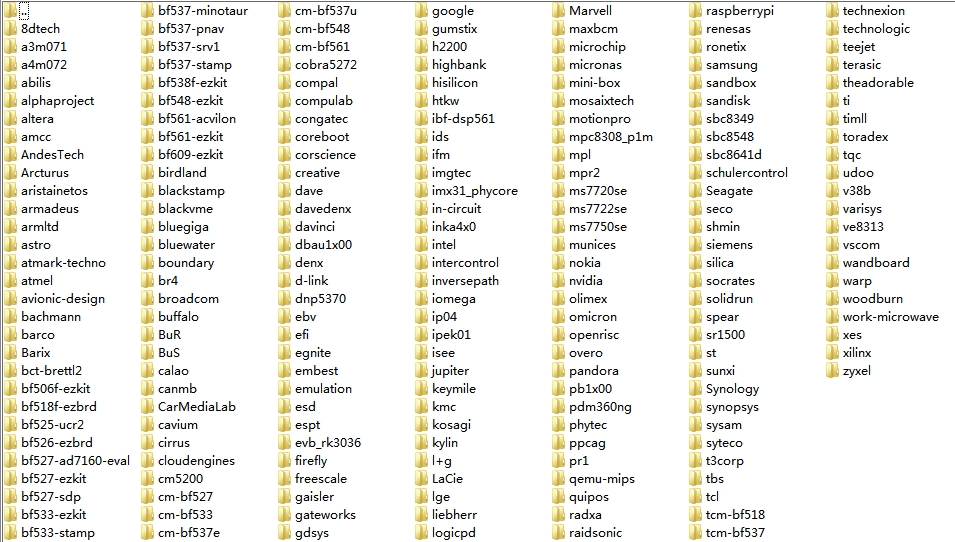
/arch   


每一个folder对应一个体系结构，如/arm便是本文涉及到的体系结构。进入/arch/arm   
   
**注，此图截自博主分享的u-boot source code，该code已经将不相干的arch folder移除，只留下了arm folder，以方便大家更好的学习。**

1. cpu子目录对应一种处理器的不同产品型号或者系列;
2. include子目录是处理器用到的头文件；
3. lib目录对应用到处理器公用的代码；

继续展开cpu文件夹，即/arch/arm/cpu/slsiap/s5p4418/   


其中，start.s文件是整个u-boot代码的入口点。

/board   


此图截自u-boot官网下载的source code，可见里面有非常多的现成的开发板配置，非常壮观。而我们的示例并不包括在其内，所以，再回看博主在上一篇文章中分享的source code(<http://pan.baidu.com/s/1kUhYmEj>) 。为方便学习，只留下了相关的文件和文件夹/u-boot/board/s5p4418/…

**我们总结一下，从目录结构上看，u-boot包括两个经常需要动到的/arch和/board目录，和另外一些暂不会动到的folders。作为初学，或者还没有涉及到u-boot裁剪的话，我们尽力搞清楚前两个folder的code原理即可搞明白u-boot的工作流程。**

**最后，u-boot.lds**

还有一个文件是必须要知道的，即u-boot/arch/arm/cpu/slsiap/u-boot.lds

u-boot.lds是ld程序也就是连接器的脚本文件，这个文件描述了如何连接目标文件，ld程序会根据这个文件的指示按照需求把不同的目标文件连接在一起生成供烧写到开发板的image。

回溯本文，

1. 什么是u-boot?
2. u-boot目录结构？
3. u-boot.lds是干什么的？

完

**u-boot分析 三 (u-boot.lds脚本)**

2016年03月20日 21:12:26 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：6639

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50938753

**u-boot分析 三 (u-boot.lds脚本)**

目的，   
了解链接器用到的脚本文件u-boot.lds。

在开始这篇博文之前，需要先了解一些GNU linker script的基本知识，可以参考博主的另外一篇分享[《GNU linker script，ld script，GNU链接脚本》](http://blog.csdn.net/itxiebo/article/details/50937412)

在《u-boot分析 二》中，我们分析u-boot的目录结构，提及到了程序入口start.S，但在开始了解start.S之前，我们先聊聊链接器ld程序的脚本文件u-boot.lds。

下面我们就来品味一下UT4418开发板的u-boot.lds脚本，即u-boot/arch/arm/cpu/slsiap/u-boot.lds。如果读者正好需要看source code，可以参看之前的文章[《u-boot分析 一》](http://blog.csdn.net/itxiebo/article/details/50920094)中的源码分享。

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")

*/\*指定输出可执行文件是elf格式，32位ARM指令，小端\*/*

OUTPUT\_ARCH(arm)

*/\*指定输出可执行文件的平台为ARM\*/*

ENTRY(\_stext)

*/\*指定输出可执行文件的起始代码段为\_stext\*/*

SECTIONS

{

*/\*指定可执行文件的全局入口点，通常这个地址都放在ROM(flash)0x0位置。必须使编译器知道这个地址，通常都是修改此处来完成\*/*

. = 0x00000000*;*

*/\*从0x0位置开始\*/*

. = ALIGN(4)*;*

*/\*代码以4字节对齐\*/*

.text :

*/\*代码段\*/*

{

\*(.\_\_image\_copy\_start)

*/\*u-boot将自己copy到RAM，此为需要copy的程序的start\*/*

SOCDIR/start.o (.text\*)

*/\*./arch/arm/cpu/slsiap/s5p4418/start.S\*/*

SOCDIR/vectors.o (.text\*)

*/\*./arch/arm/cpu/slsiap/s5p4418/vectors.S，异常向量表\*/*

\*(.text\*)

*/\*其他的代码段放在这里，即start.S/vector.S之后\*/*

}

. = ALIGN(4)*;*

*/\*代码段结束后，有可能4bytes不对齐了，此时做好4bytes对齐，以开始后面的.rodata段\*/*

.rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) }

*/\*在代码段之后，存放read only数据段\*/*

. = ALIGN(4)*;*

*/\*和前面一样，4bytes对齐，以开始接下来的.data段\*/*

.data : {

\*(.data\*)

*/\*可读写数据段\*/*

}

. = ALIGN(4)*;*

*/\*和前面一样，4bytes对齐\*/*

. = .*;*

. = ALIGN(4)*;*

.u\_boot\_list : {

KEEP(\*(SORT(.u\_boot\_list\*)))*;*

*/\*.data段结束后，紧接着存放u-boot自有的一些function，例如u-boot command等\*/*

}

. = ALIGN(4)*;*

.image\_copy\_end :

{

\*(.\_\_image\_copy\_end)

*/\*至此，u-boot需要自拷贝的内容结束，总结一下，包括代码段，数据段，以及u\_boot\_list\*/*

}

.rel\_dyn\_start :

*/\*在老的uboot中，如果我们想要uboot启动后把自己拷贝到内存中的某个地方，只要把要拷贝的地址写给TEXT\_BASE即可，然后boot启动后就会把自己拷贝到TEXT\_BASE内的地址处运行，在拷贝之前的代码都是相对的，不能出现绝对的跳转，否则会跑飞。在新版的uboot里（2013.07），TEXT\_BASE的含义改变了。它表示用户要把这段代码加载到哪里，通常是通过串口等工具。然后搬移的时候由uboot自己计算一个地址来进行搬移。新版的uboot采用了动态链接技术，在lds文件中有\_\_rel\_dyn\_start和\_\_rel\_dyn\_end，这两个符号之间的区域存放着动态链接符号，只要给这里面的符号加上一定的偏移，拷贝到内存中代码的后面相应的位置处，就可以在绝对跳转中找到正确的函数。\*/*

{

\*(.\_\_rel\_dyn\_start)

}

.rel.dyn : {

\*(.rel\*)

*/\*动态链接符存放在的段\*/*

}

.rel\_dyn\_end :

{

\*(.\_\_rel\_dyn\_end)

*/\*动态链接符段结束\*/*

}

.end :

{

\*(.\_\_end)

}

\_image\_binary\_end = .*;*

*/\*bin文件结束\*/*

*/\**

*\* Deprecated: this MMU section is used by pxa at present but*

*\* should not be used by new boards/CPUs.*

*\*/*

. = ALIGN(4096)*;*

.mmutable : { */\*for MMU\*/*

\*(.mmutable)

}

*/\**

*\* Compiler-generated \_\_bss\_start and \_\_bss\_end, see arch/arm/lib/bss.c*

*\* \_\_bss\_base and \_\_bss\_limit are for linker only (overlay ordering)*

*\*/*

*/\*bss段的描述\*/*

.bss\_start (OVERLAY) : {

KEEP(\*(.\_\_bss\_start))*;*

\_\_bss\_base = .*;*

}

.bss \_\_bss\_base (OVERLAY) : {

\*(.bss\*)

. = ALIGN(4)*;*

\_\_bss\_limit = .*;*

}

.bss\_end \_\_bss\_limit (OVERLAY) : {

KEEP(\*(.\_\_bss\_end))*;*

}

*/\*bss段的描述结束\*/*

.dynsym \_image\_binary\_end : { \*(.dynsym) }

.dynbss : { \*(.dynbss) }

.dynstr : { \*(.dynstr\*) }

.dynamic : { \*(.dynamic\*) }

.plt : { \*(.plt\*) }

.interp : { \*(.interp\*) }

.gnu.hash : { \*(.gnu.hash) }

.gnu : { \*(.gnu\*) }

.ARM.exidx : { \*(.ARM.exidx\*) }

.gnu.linkonce.armexidx : { \*(.gnu.linkonce.armexidx.\*) }

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120

总结一下，u-boot.lds脚本文件告诉链接器linker如何布局代码段、数据段、bss段等，已经配置了u-boot自拷贝(从flash到RAM的copy)的内容。另外，还简要的涉及了动态链接技术等。

完

**u-boot分析 四 (程序入口start.S)**

2016年03月21日 22:10:14 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：4650

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50950772

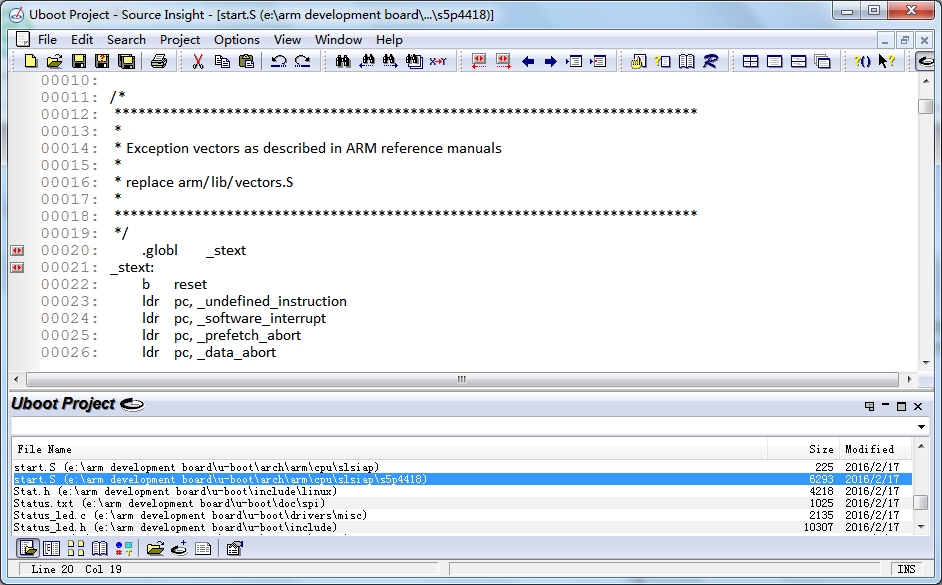
**u-boot分析 四 (程序入口start.S)**

注：部分内容摘抄自网络，如有问题，请联络博主。

本文内容：了解以stars.S为开始的ARM汇编程序部分。

回顾前几篇博文，咱们见识过了u-boot的目录结构，另外简要分析了u-boot.lds脚本文件的link原理。而今天我们要来听听嵌入式程序君告诉咱们的第一句“话”。

正式开始之前，我们需要准备三样东西：

1. [u-boot source code](http://pan.baidu.com/s/1kUhYmEj)
2. [常用ARM指令集](http://pan.baidu.com/s/1kUhYmEj)
3. Source Insight(用于trace code，使用方法略过，不会问百度)   
   

bootloader通常stage1和stage2两步骤，u-boot也不例外。

1. Stage1：依赖于CPU体系结构的代码（如设备初始化代码等）通常都放在这个程序段，且可以用汇编语言来实现；
2. stage2：通常用C语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

具体的说，

1. **Stage1 start.S代码结构**   
   u-boot的stage1代码通常放在start.S文件中，用汇编语言写成，其主要代码部分如下：   
   (1) 定义入口。由于一个可执行文件必须有一个入口点，并且只能有一个全局入口，通常这个入口放在ROM（Flash）的0x0地址。因此，必须通知编译器以使其知道这个入口，该工作可通过修改链接器脚本(.lds文件)来完成。   
   (2) 设置异常向量（Exception Vector）。   
   (3) 设置CPU的速度、时钟频率及终端控制寄存器。   
   (4) 初始化内存控制器。   
   (5) 将ROM中的代码复制到RAM中。   
   (6) 初始化堆栈。   
   (7) 转到RAM中执行，该工作可使用指令ldr pc来完成。
2. **Stage2 C语言代码部分**   
   ./arch/arm/lib/board.c中的board\_init\_r()是C语言开始的函数，也是整个启动代码中C语言的主函数，同时还是整个u-boot(armboot)的主函数，该函数主要完成如下操作：   
   (1) 调用一系列的初始化函数。   
   (2) 初始化Flash设备。   
   (3) 初始化系统内存分配函数。   
   (4) 如果目标系统拥有NAND设备，则初始化NAND设备。   
   (5) 如果目标系统有显示设备，则初始化该类设备。   
   (6) 初始化相关网络设备，填写IP、MAC地址等。   
   (7) 进去命令循环（即整个boot的工作循环），接收用户从串口输入的命令，然后进行相应的工作。

[事不宜迟](https://www.baidu.com/s?wd=%E4%BA%8B%E4%B8%8D%E5%AE%9C%E8%BF%9F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，我们赶紧打开start.S：   
./arch/arm/cpu/slsiap/s5p4418/start.S

*/\**

*\* armboot - Startup Code for NXPxxxx/ARM Cortex CPU-core*

*\*/*

#include <asm-offsets.h>

#include <config.h>

#include <version.h>

#include <asm/system.h>

#include <linux/linkage.h>

*/\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* Exception vectors as described in ARM reference manuals*

*\**

*\* replace arm/lib/vectors.S*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\*/*

.globl \_stext

*/\*程序的全局入口，《u-boot分析 三》中u-boot.lds设置此入口地址为0x00000000\*/*

\_stext:

b reset

*/\*参阅*[*《常用ARM指令集及汇编》*](https://www.baidu.com/s?wd=%E3%80%8A%E5%B8%B8%E7%94%A8ARM%E6%8C%87%E4%BB%A4%E9%9B%86%E5%8F%8A%E6%B1%87%E7%BC%96%E3%80%8B&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)*可知，b为跳转指令，跳转到reset函数处，reset在后面\*/*

*/\*ARM体系结构规定在上电复位后的起始位置，必须有8条连续的跳转指令，通过硬件实现。他们就是异常向量表\*/*

*/\*ldr，用于加载32bit的立即数或一个地址值到指定寄存器\*/*

ldr pc, \_undefined\_instruction */\*未定义指令异常,0x04\*/*

ldr pc, \_software\_interrupt */\*软中断异常,0x08\*/*

ldr pc, \_prefetch\_abort */\*内存操作异常,0x0c\*/*

ldr pc, \_data\_abort */\*数据异常,0x10\*/*

ldr pc, \_not\_used */\*未使用,0x14\*/*

ldr pc, \_irq */\*慢速中断异常,0x18\*/*

ldr pc, \_fiq */\*快速中断异常,0x1c\*/*

*/\*.word的意思是将后边的符号所对应的32bit值赋予前面的符号\*/*

*/\*而如下的七条语句，后面的符号正好是对应的中断异常服务程序的入口地址\*/*

*/\*这七个中断服务程序位于./arch/arm/cpu/slsiap/s5p4418/vector.S\*/*

\_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

.balignl 16,0xdeadbeef

*/\*16bytes对齐，并且使用0xdeadbeef填充\*/*

*/\*.balignl是.balign的变体*

*.align伪操作用于表示对齐方式：通过添加填充字节使当前位置满足一定的对齐方式。*

*.balign的作用同.align。*

*.align {alignment} {,fill} {,max}*

*其中：*

*alignment用于指定对齐方式，可能的取值为2的次幂，缺省为4。*

*fill是填充内容，缺省用0填充。*

*max是填充字节数最大值，如果填充字节数超过max，就不进行对齐\*/*

*/\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* Text and section base*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\*/*

.globl TEXT\_BASE

*/\*全局定义TEXT\_BASE\*/*

TEXT\_BASE:

.word CONFIG\_SYS\_TEXT\_BASE

*/\*TEXT\_BASE使用.word赋予的值为CONFIG\_SYS\_TEXT\_BASE，而CONFIG\_SYS\_TEXT\_BASE是在*

*u-boot/include/configs/s5p4418\_urbetter.h中定义，值为0x42c00000\*/*

*/\**

*\* These are defined in the board-specific linker script(u-boot.lds中定义)，可参看前面的博客*

*\*/*

.globl \_bss\_start\_ofs

\_bss\_start\_ofs:

.word \_\_bss\_start - \_stext

.globl \_bss\_end\_ofs

\_bss\_end\_ofs:

.word \_\_bss\_end - \_stext

.globl \_end\_ofs

\_end\_ofs:

.word \_end - \_stext

*/\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* Reset handling*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\*/*

.globl reset

*/\*这里便是reset函数的入口\*/*

*/\*其实在CPU一上电以后就是跳到这里执行的\*/*

reset:

bl save\_boot\_params

*/\*call save\_boot\_params，此函数在下面有定义，但是实际do nothing\*/*

*/\**

*\* set the CPU to SVC32 mode，即管理模式*

*\*/*

*/\*对状态寄存器CPSR的修改要按照：读出-修改-写回的顺序来执行\*/*

*/\**

*31 30 29 28 ----- 7 6 - 4 3 2 1 0*

*N Z C V I F M4 M3 M2 M1 M0*

*0 0 0 0 0 User26 模式*

*0 0 0 0 1 FIQ26 模式*

*0 0 0 1 0 IRQ26 模式*

*0 0 0 1 1 SVC26 模式*

*1 0 0 0 0 User 模式*

*1 0 0 0 1 FIQ 模式*

*1 0 0 1 0 IRQ 模式*

*1 0 0 1 1 SVC 模式*

*1 0 1 1 1 ABT 模式*

*1 1 0 1 1 UND 模式*

*1 1 1 1 1 SYS 模式*

*\*/*

mrs r0, cpsr

*/\*读出cpsr的值\*/*

bic r0, r0, #0x1f

*/\*清零低5位\*/*

orr r0, r0, #0xd3

*/\*0xd3，即1 1 0 1 0 0 1 1，即为SVC模式，禁止IRQ,FIQ中断(bit6,7)\*/*

msr cpsr,r0

*/\*写回cpsr使其生效\*/*

*/\* the mask ROM code should have PLL and others stable \*/*

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

*/\*UT4418 u-boot source code中没有define CONFIG\_SKIP\_LOWLEVEL\_INIT，所以如下两条命令会执行\*/*

bl cpu\_init\_cp15

*/\*bl为跳转指令，即call cpu\_init\_cp15函数，该函数在下面定义，主要功能是*

*Invalidate L1 I/D，disable MMU stuff and I-cache\*/*

bl cpu\_init\_crit

*/\*继续调用cpu\_init\_crit函数，函数定义在下面，该函数中只有一句函数，即b lowlevel\_init，*

*lowlevel\_init函数的具体功能，此处先略过\*/*

#endif

#ifdef CONFIG\_RELOC\_TO\_TEXT\_BASE

*/\*u-boot/include/configs/s5p4418\_urbetter.h中define CONFIG\_RELOC\_TO\_TEXT\_BASE\*/*

relocate\_to\_text:

*/\*程序继续执行，此程序段便是要将ROM中的code copy到TEXT\_BASE处，即从0x0开始的数据copy到0x42c00000处\*/*

*/\**

*\* relocate u-boot code on memory to text base*

*\* for nexell arm core (add by jhkim)*

*\*/*

adr r0, \_stext */\* r0 <- current position of code \*/*

ldr r1, TEXT\_BASE */\* test if we run from flash or RAM \*/*

cmp r0, r1 */\* don't reloc during debug \*/*

beq clear\_bss

ldr r2, \_bss\_start\_ofs

add r2, r0, r2 */\* r2 <- source end address \*/*

copy\_loop\_text:

ldmia r0!, {r3-r10} */\* copy from source address [r0] \*/*

stmia r1!, {r3-r10} */\* copy to target address [r1] \*/*

cmp r0, r2 */\* until source end addreee [r2] \*/*

ble copy\_loop\_text

ldr r1, TEXT\_BASE */\* restart at text base \*/*

mov pc, r1

*/\*程序从TEXT\_BASE处开始执行\*/*

clear\_bss:

ldr r0, \_bss\_start\_ofs

ldr r1, \_bss\_end\_ofs

ldr r4, TEXT\_BASE */\* text addr \*/*

add r0, r0, r4

add r1, r1, r4

mov r2, #0x00000000 /\* clear \*/

clbss\_l:str r2, [r0] */\* clear loop... \*/*

add r0, r0, #4

cmp r0, r1

bne clbss\_l

#ifdef CONFIG\_MMU\_ENABLE

*/\*同样在u-boot/include/configs/s5p4418\_urbetter.h中已经define\*/*

bl mmu\_turn\_on

*/\*ROM到RAM的copy完成后，继续调用 mmu\_turn\_on函数，此函数位于/u-boot/arch/arm/cpu/slsiap/s5p4418/mmu\_asm.S*

*主要功能是打开MMU，详细过程需要进一步trace code\*/*

#endif

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

*/\*u-boot/include/configs/s5p4418\_urbetter.h中*

*define CONFIG\_SYS\_INIT\_SP\_ADDR = CONFIG\_SYS\_TEXT\_BASE，即0x42c00000\*/*

bic sp, sp, #7

*/\* 8-byte alignment for ABI compliance \*/*

sub sp, #GD\_SIZE

*/\* allocate one GD above SP \*/*

bic sp, sp, #7

*/\* 8-byte alignment for ABI compliance \*/*

mov r9, sp

*/\* GD is above SP \*/*

mov r0, #0

bl board\_init\_f

*/\*call board\_init\_f, /u-boot/arch/arm/lib/board.c\*/*

mov sp, r9

*/\* SP is GD's base address \*/*

bic sp, sp, #7

*/\* 8-byte alignment for ABI compliance \*/*

sub sp, #GENERATED\_BD\_INFO\_SIZE

*/\* allocate one BD above SP \*/*

bic sp, sp, #7

*/\* 8-byte alignment for ABI compliance \*/*

mov r0, r9

*/\* gd\_t \*gd \*/*

ldr r1, TEXT\_BASE

*/\* ulong text \*/*

mov r2, sp

*/\* ulong sp \*/*

bl gdt\_reset

*/\*调用gdt\_reset函数初始化gdt，u-boot/arch/arm/cpu/slsisp/s5p4418/Cpu.c\*/*

*/\* call board\_init\_r(gd\_t \*id, ulong dest\_addr) \*/*

mov r0, r9 */\* gd\_t \*/*

ldr r1, =(CONFIG\_SYS\_MALLOC\_END) */\* dest\_addr for malloc heap end \*/*

*/\* call board\_init\_r \*/*

ldr pc, =board\_init\_r

*/\* this is auto-relocated! \*/*

*/\*board\_init\_r变开始进入Stage2，/u-boot/arch/arm/lib/board.c\*/*

#else /\* CONFIG\_RELOC\_TO\_TEXT\_BASE \*/

bl \_main

#endif

*/\*------------------------------------------------------------------------------\*/*

ENTRY(c\_runtime\_cpu\_setup)

*/\**

*\* If I-cache is enabled invalidate it*

*\*/*

#ifndef CONFIG\_SYS\_ICACHE\_OFF

mcr p15, 0, r0, c7, c5, 0 @ invalidate icache

mcr p15, 0, r0, c7, c10, 4 @ DSB

mcr p15, 0, r0, c7, c5, 4 @ ISB

#endif

*/\**

*\* Move vector table*

*\*/*

*/\* Set vector address in CP15 VBAR register \*/*

ldr r0, =\_stext

mcr p15, 0, r0, c12, c0, 0 @Set VBAR

bx lr

ENDPROC(c\_runtime\_cpu\_setup)

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* void save\_boot\_params(u32 r0, u32 r1, u32 r2, u32 r3)*

*\* \_\_attribute\_\_((weak));*

*\**

*\* Stack pointer is not yet initialized at this moment*

*\* Don't save anything to stack even if compiled with -O0*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

ENTRY(save\_boot\_params)

bx lr @ back to my caller

ENDPROC(save\_boot\_params)

.weak save\_boot\_params

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* cpu\_init\_cp15*

*\**

*\* Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on unless*

*\* CONFIG\_SYS\_ICACHE\_OFF is defined.*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

ENTRY(cpu\_init\_cp15)

*/\**

*\* Invalidate L1 I/D*

*\*/*

*/\*MRC，协处理器寄存器到ARM寄存器的数据传输指令。MRC将协处理器寄存器中的数据传送到*

*ARM处理器的寄存器中。若协处理器不能成功执行该操作，将产生未定义异常中断\*/*

mov r0, #0

*/\*set up for MCR\*/*

mcr p15, 0, r0, c8, c7, 0

*/\*invalidate TLBs，使TLBs无效\*/*

mcr p15, 0, r0, c7, c5, 0

*/\*invalidate icache，使icache无效\*/*

mcr p15, 0, r0, c7, c5, 6

*/\*invalidate BP array\*/*

*/\*如下两条命令，参考http://blog.csdn.net/itxiebo/article/details/50957808*

*了解即可，甚至可以直接跳过分析\*/*

dsb

isb

*/\**

*\* disable MMU stuff and caches*

*\*/*

mrc p15, 0, r0, c1, c0, 0

bic r0, r0, #0x00002000

*/\*clear bits 13 (--V-)\*/*

bic r0, r0, #0x00000007

*/\*clear bits 2:0 (-CAM)\*/*

orr r0, r0, #0x00000002

*/\*set bit 1 (--A-) Align\*/*

orr r0, r0, #0x00000800

*/\*set bit 11 (Z---) BTB\*/*

#ifdef CONFIG\_SYS\_ICACHE\_OFF

bic r0, r0, #0x00001000

*/\*clear bit 12 (I) I-cache\*/*

#else

orr r0, r0, #0x00001000

*/\*set bit 12 (I) I-cache\*/*

#endif

mcr p15, 0, r0, c1, c0, 0

mov pc, lr

*/\*back to my caller，返回调用处\*/*

ENDPROC(cpu\_init\_cp15)

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\**

*\* CPU\_init\_critical registers*

*\**

*\* setup important registers*

*\* setup memory timing*

*\**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

ENTRY(cpu\_init\_crit)

*/\**

*\* Jump to board specific initialization...*

*\* The Mask ROM will have already initialized*

*\* basic memory. Go here to bump up clock rate and handle*

*\* wake up conditions.*

*\*/*

b lowlevel\_init

*/\*go setup pll,mux,memory\*/*

*/\*u-boot/arch/arm/cpu/slsiap/s5p4418/low\_init.S\*/*

ENDPROC(cpu\_init\_crit)

#endif

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 213
* 214
* 215
* 216
* 217
* 218
* 219
* 220
* 221
* 222
* 223
* 224
* 225
* 226
* 227
* 228
* 229
* 230
* 231
* 232
* 233
* 234
* 235
* 236
* 237
* 238
* 239
* 240
* 241
* 242
* 243
* 244
* 245
* 246
* 247
* 248
* 249
* 250
* 251
* 252
* 253
* 254
* 255
* 256
* 257
* 258
* 259
* 260
* 261
* 262
* 263
* 264
* 265
* 266
* 267
* 268
* 269
* 270
* 271
* 272
* 273
* 274
* 275
* 276
* 277
* 278
* 279
* 280
* 281
* 282
* 283
* 284
* 285
* 286
* 287
* 288
* 289
* 290
* 291
* 292
* 293
* 294
* 295
* 296
* 297
* 298
* 299
* 300
* 301
* 302
* 303
* 304
* 305
* 306
* 307
* 308
* 309
* 310
* 311
* 312
* 313
* 314
* 315
* 316
* 317
* 318
* 319
* 320
* 321
* 322
* 323
* 324
* 325
* 326
* 327
* 328
* 329
* 330
* 331
* 332
* 333
* 334
* 335
* 336
* 337
* 338
* 339
* 340
* 341
* 342
* 343
* 344
* 345
* 346
* 347
* 348
* 349
* 350
* 351
* 352
* 353
* 354
* 355
* 356
* 357
* 358
* 359
* 360
* 361
* 362
* 363

回顾全文，我们总结一下start.S都干了些什么？

1. 首先定义程序入口\_stext
2. 然后定义了异常向量表
3. 一堆初始化，e.g. PLL, MUX, Memory等
4. Copy代码从ROM到RAM
5. MMU on
6. 最后跳转到C入口

参考博客：   
<http://blog.chinaunix.net/uid-22891435-id-380150.html>

完

**u-boot分析 五 (u-boot如何向kernel传启动参数)**

2016年03月26日 12:36:49 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：1871 标签： [u-boot](http://so.csdn.net/so/search/s.do?q=u-boot&t=blog)[kernel](http://so.csdn.net/so/search/s.do?q=kernel&t=blog)[启动参数](http://so.csdn.net/so/search/s.do?q=%E5%90%AF%E5%8A%A8%E5%8F%82%E6%95%B0&t=blog)[传递](http://so.csdn.net/so/search/s.do?q=%E4%BC%A0%E9%80%92&t=blog) 更多

个人分类： [u-boot](https://blog.csdn.net/itxiebo/article/category/6140192)

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50986253

**u-boot分析 五 (u-boot如何向kernel传启动参数)**

u-boot分析系列文章，前面的文章中，咱们已经分析了：

1. u-boot目录结构，
2. u-boot.lds脚本，
3. 以及start.S程序入口（stage1）。

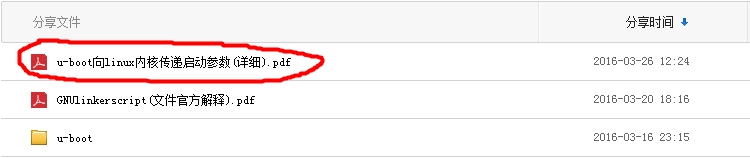
按理说，应该接着分析u-boot启动程序的stage2部分（即C语言程序部分）。但是，我想先跳过stage2的分析，先解决一个更有意思的问题：

u-boot是如何启动kernel(linux)的呢？

自认不会比下面的文章写得更好了，那就直接分享吧。

注意，本文章有一定难度，建议先囫囵吞枣看上三篇，再结合code过一遍（博主真是苦口婆心啊，哈哈哈！），理解便是自然了。

下载地址：   
<http://pan.baidu.com/s/1geJ18MB>



完

**u-boot分析 六 (从code flow看u-boot到kernel的启动过程)**

2016年03月27日 04:11:14 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：3334

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50990074

**u-boot分析 六 (从code flow看u-boot到kernel的启动过程)**

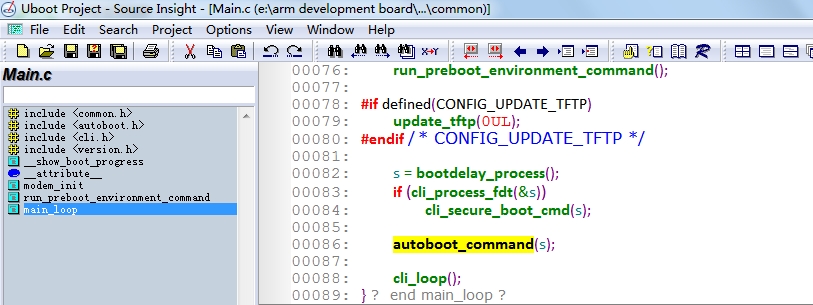
此文章绝对是精华……

本篇文章重点：

介绍u-boot是如何从start.S一步一步的run到linux kernel的入口的。

博主提醒你，要搞明白这篇文章：

1. 必须要有上一篇的基础，即uboot是如何传参数给linux kernel的[《u-boot分析 五》](http://blog.csdn.net/itxiebo/article/details/50986253)；
2. 建议下载博主在[《u-boot分析 一》](http://blog.csdn.net/itxiebo/article/details/50920094)中分享的[u-boot源码](http://pan.baidu.com/s/1kUhYmEj)，因为本文章就是在讲code flow。



下表导读：

u-boot从start.S开始执行，汇编语言编写的stage1结束后，跳转到可读行更高、采用C语言实现的stage2。   
在Stage2结束部分，呼叫main\_loop()函数，main\_loop()函数借u-boot command传递参数和启动内核……



注意：

1. 如果只看code，u-boot启动kernel的flow是这样的，   
   start.S –>board\_init\_r() –>main\_loop() –> autoboot\_command()–>   
   do\_bootm() –>do\_bootm\_states() –>bootm\_os\_get\_boot\_func() –>   
   boot\_os[os] –>do\_bootm\_linux() –> boot\_jump\_linux()
2. Step 10中boot\_prep\_linux()准备的参数tag，便是上一篇文章介绍的内容。

完

**u-boot分析 七 (添加u-boot命令，学习u-boot命令实现原理)**

2016年03月27日 12:13:35 [itxiebo](https://me.csdn.net/itxiebo) 阅读数：5368

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/itxiebo/article/details/50991049

**u-boot分析 七**

(添加u-boot命令，学习u-boot命令实现原理)

本文目标：

理解u-boot命令的实现原理。

上一篇文章分析了u-boot是如何启动[kernel](https://www.baidu.com/s?wd=kernel&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的，其中就涉及到***bootm***命令，考虑到文章主题需要，当时并没有对bootm命令做过多的解释。然而，u-boot命令行又是u-boot及其重要的一部分。这篇文章，我们就通过实战来理解u-boot命令的实现原理。

我们要做的很简单，就是添加一条uboot command，希望在开发板上，uboot命令行中输入***itxiebo***时，能够从串口打印出一句log。

**一、具体实现步骤：**

1. 在./common文件夹下新建cmd\_itxiebo.c，并在此文件中添加如下内容

#include <common.h>

#include <command.h>

static int do\_itxiebo(cmd\_tbl\_t \*cmdtp, int flag, int argc, char \* const argv[])

{

printf("do\_itxiebo command is ready now!");

return 0;

}

U\_BOOT\_CMD(

itxiebo, 2, 0, do\_itxiebo,

"itxiebo - this is a itxiebo command, do nothing",

"- this is a itxiebo command, do nothing"

);

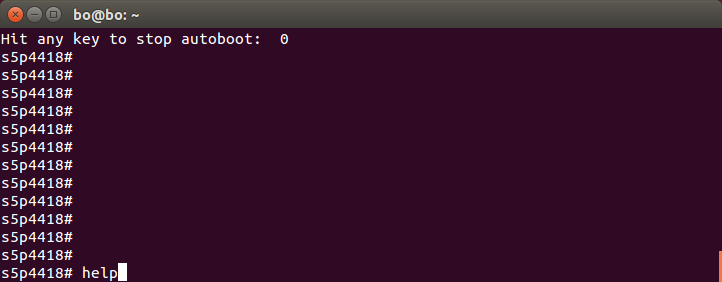
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

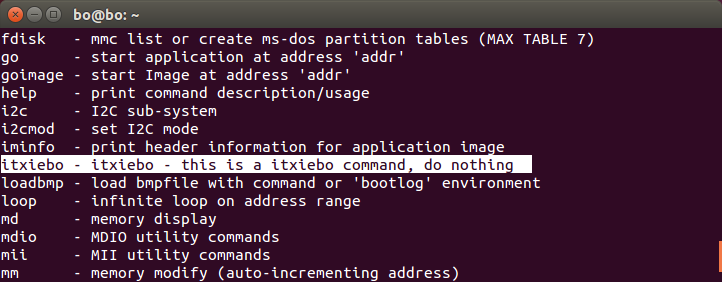
2.在./common/Makefile中添加：

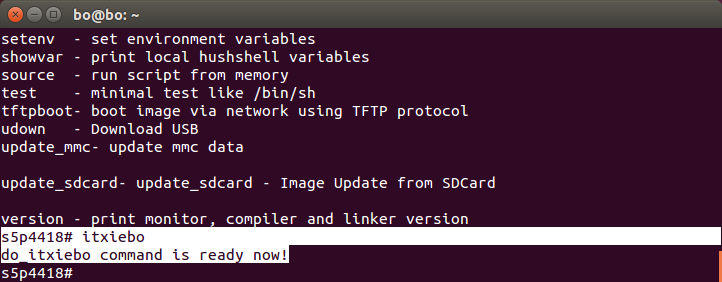
obj-y += cmd\_itxiebo.o

* 1

3.在linux环境下，重新编译u-boot，得到u-boot.bin，并升级到自己的开发板中（如果你没有开发板，没有关系，你只需要明白我们要在开发板中验证自己添加的命令itxiebo）

4.升级完后，在开发板启动内核之前，按space键进入u-boot命令行模式。   


5.在串口终端中输入help命令，回车，可以发现itxiebo命令行已经添加成功。   


6.在串口终端中输入itxiebo命令，回车执行该命令，发现我们添加的打印log，可以正确打印出来：   


**二、命令原理分析**

实战见效果了，我们继续分析。不难发现，只要能搞清上面代码中的“U\_BOOT\_CMD”，就能弄明白uboot命令实现原理。

U\_BOOT\_CMD格式如下：

U\_BOOT\_CMD(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help)

* 1

其中，各个参数解释如下：

| **参数名** | **说明** |
| --- | --- |
| \_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 |
| \_rep | 是否自动重复（按Enter键是否会重复执行） |
| \_cmd | 该命令对应的响应函数 |
| \_usage | 简短的使用说明（字符串） |
| \_help | 较详细的使用说明（字符串） |

说明：   
在内存中保存命令的help字段会占用一定的内存，通过配置U-Boot可以选择是否保存help字段。若在include/configs/s5p4418\_urbetter.h 中定义了CONFIG\_SYS\_LONGHELP宏，则在U-Boot中使用help命令查看某个命令的帮助信息时将显示\_usage和\_help字段的内容，否则就只显示usage字段的内容，而不显示\_help字段的内容。

另外，在include/command.h中，对U\_BOOT\_CMD的define 如下：

#define ll\_entry\_declare(\_type, \_name, \_list) \

\_type \_u\_boot\_list\_2\_##\_list##\_2\_##\_name \_\_aligned(4) \_\_attribute\_\_((unused, section(*".u\_boot\_list\_2\_"*#\_list*"\_2\_"*#\_name)))

#define U\_BOOT\_CMD\_COMPLETE(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help, \_comp) \

ll\_entry\_declare(cmd\_tbl\_t, \_name, cmd) = \

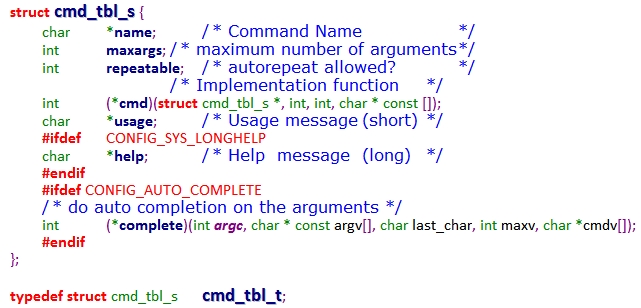
U\_BOOT\_CMD\_MKENT\_COMPLETE(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help, \_comp);

#define U\_BOOT\_CMD(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help) \

U\_BOOT\_CMD\_COMPLETE(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help, NULL)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

最后，同样在include/command.h中定义了一个结构体cmd\_tbl\_t：



如此，我们便可以将U\_BOOT\_COM解析，如下

U\_BOOT\_CMD(

itxiebo, 2, 0, do\_itxiebo,

"itxiebo - this is a itxiebo command, do nothing",

"- this is a itxiebo command, do nothing"

);

解析为：

cmd\_tbl\_t \_u\_boot\_list\_2\_do\_itxiebo\_2\_itxiebo \_\_aligned(4) \_\_attribute\_\_((unused,section(".u\_boot\_list\_2\_"do\_itxiebo"\_2\_"itxiebo)))

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

其中，“u\_boot\_list”，[《u-boot分析 三》](http://blog.csdn.net/itxiebo/article/details/50938753)分析u-boot.lds时解释过。也就是说咱们新增的itxiebo command会被储存在u\_boot\_list段内。

.u\_boot\_list : {

KEEP(\*(SORT(.u\_boot\_list\*)))*;*

*/\*.data段结束后，紧接着存放u-boot自有的一些function，例如u-boot command等\*/*

}

* 1
* 2
* 3
* 4

**三、itxiebo命令的执行过程：**

在U-Boot中输入(串口终端)“itxiebo”命令执行时，U-Boot接收输入的字符串“itxiebo”，传递给run\_command()函数。run\_command()函数调用common/command.c中实现的find\_cmd()函数在u\_boot\_list段内查找命令，并返回itxiebo命令的cmd\_tbl\_t结构。然后run\_command()函数使用返回的cmd\_tbl\_t结构中的函数指针调用itxiebo命令的响应函数do\_itxiebo，从而完成了命令的执行。

参考：   
<http://www.cnblogs.com/sdphome/archive/2011/08/19/2146327.html>

完事儿。