完全理解ARM启动流程：Uboot-Kernel

Linux爱好者 2024-04-26 11:50 浙江

以下文章来源于TrustZone ，作者Hcoco

TrustZone
一个搞技术的读书人，妄图分享最干货的技术知识与世界运转的底层逻辑。

前言

bootloader是系统上电后最初加载运行的代码。它提供了处理器上电复位后最开始需要执行的初始化代码。

PC机上引导程序一般由BIOS开始执行，然后读取硬盘中位于MBR(Main Boot Record，主引导记录)中的Bootloader(例如LILO或GRUB),并进一步引导操作系统的启动。

嵌入式系统中通常没有像BIOS那样的固件程序，因此整个系统的加载启动就完全由bootloader来完成，它主要的功能是加载与引导内核映像。

一个嵌入式的存储设备通过通常包括四个分区：

- 第一个分区：存放的当然是u-boot
- 第二个分区：存放着u-boot要传给系统内核的参数
- 第三个分区：是系统内核（kernel）
- 第四个分区：则是根文件系统

进入开发板/sys/class/mtd/目录下，执行ls命令查看 ：

```
mtd0     mtd1     mtd2     mtd3     mtd4     mtd5
mtd0ro   mtd1ro   mtd2ro   mtd3ro   mtd4ro   mtd5ro
```

开发板存储设置被分成5个区：

```
cat mtd0/name    U-Boot
cat mtd1/name    U-Boot Env
cat mtd2/name    U-Boot Logo
cat mtd3/name    Kernel
cat mtd4/name    File System
```

Bootloader介绍

Bootloader的定义：Bootloader是在操作系统运行之前执行的一小段程序，通过这一小段程序，我们可以**初始化硬件设备、建立内存空间的映射表**，从而建立适当的系统软硬件环境，为最终调用操作系统内核做好准备。

意思就是说如果我们要想让一个操作系统在我们的板子上运转起来，我们就必须首先对我们的板子进行一些基本配置和初始化，然后才可以将操作系统引导进来运行。

具体在Bootloader中完成了哪些操作我们会在后面分析到，这里我们先来回忆一下PC的体系结构：PC机中的引导加载程序是由BIOS和位于硬盘MBR中的OS Boot Loader（比如LILO和GRUB等）一起组成的，BIOS在完成硬件检测和资源分配后，将硬盘MBR中的Boot Loader读到系统的RAM中，然后将控制权交给OS Boot Loader。

Boot Loader的主要运行任务就是将内核映象从硬盘上读到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。

在嵌入式系统中，通常并没有像BIOS那样的固件程序（注：有的嵌入式cpu也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由Boot Loader来完成。

比如在一个基于ARM7TDMI core的嵌入式系统中，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。（先想一下，通用PC和嵌入式系统为何会在此处存在如此的差异呢？）

Bootloader是基于特定硬件平台来实现的，因此几乎不可能为所有的嵌入式系统建立一个通用的Bootloader，不同的处理器架构都有不同的Bootloader，Bootloader不但依赖于cpu的体系结构，还依赖于嵌入式系统板级设备的配置。

对于2块不同的板子而言，即使他们使用的是相同的处理器，要想让运行在一块板子上的Bootloader程序也能运行在另一块板子上，一般也需要修改Bootloader的源程序。

Bootloader的启动方式

Bootloader的启动方式主要有网络启动方式、磁盘启动方式和Flash启动方式，当然还可以有其他启动方式，例如：MMC等。

1、网络启动方式

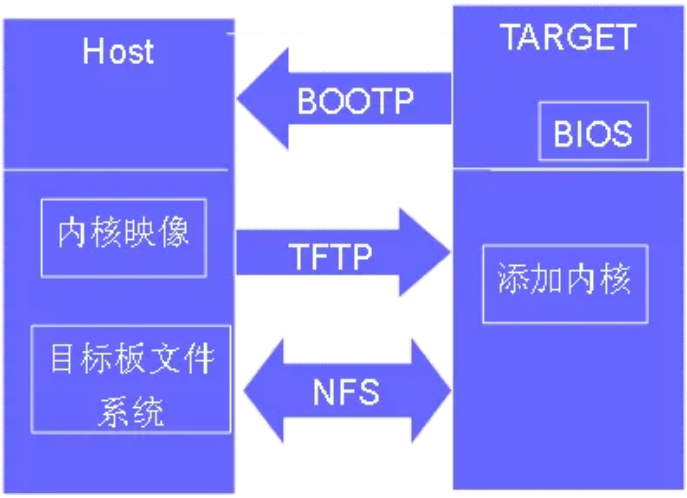


图1 Bootloader网络启动方式示意图

如图1所示，里面主机和目标板，他们中间通过网络来连接，首先目标板的DHCP/BIOS通过BOOTP服务来为Bootloader分配IP地址，配置网络参数，这样才能支持网络传输功能。

我们使用的u-boot可以直接设置网络参数，因此这里就不用使用DHCP的方式动态分配IP了。

接下来目标板的Bootloader通过TFTP服务将内核映像下载到目标板上，然后通过网络文件系统来建立主机与目标板之间的文件通信过程，之后的系统更新通常也是使用Boot Loader的这种工作模式。

工作于这种模式下的Boot Loader通常都会向它的终端用户提供一个简单的命令行接口。

2、磁盘启动方式

这种方式主要是用在台式机和服务器上的，这些计算机都使用BIOS引导，并且使用磁盘作为存储介质，这里面两个重要的用来启动linux的有LILO和GRUB，这里就不再具体说明了。

3、Flash启动方式

这是我们最常用的方式。Flash有NOR Flash和NAND Flash两种。NOR Flash可以支持随机访问，所以代码可以直接在Flash上执行，Bootloader一般是存储在Flash芯片上的。另外Flash上还存储着参数、内核映像和文件系统。

这种启动方式与网络启动方式之间的不同之处就在于，在网络启动方式中，内核映像和文件系统首先是放在主机上的，然后经过网络传输下载进目标板的，而这种启动方式中内核映像和文件系统则直接是放在Flash中的，这两点在我们u-boot的使用过程中都用到了。

u-boot是一种普遍用于嵌入式系统中的Bootloader。

第一、U-Boot介绍

U-boot的定义

U-boot，全称Universal Boot Loader，是由DENX小组的开发的遵循GPL条款的开放源码项目，它的主要功能是完成硬件设备初始化、操作系统代码搬运，并提供一个控制台及一个指令集在操作系统运行前操控硬件设备。

U-boot之所以这么通用，原因是他具有很多特点：开放源代码、支持多种嵌入式操作系统内核、支持多种处理器系列、较高的稳定性、高度灵活的功能设置、丰富的设备驱动源码以及较为丰富的开发调试文档与强大的网络技术支持。另外u-boot对操作系统和产品研发提供了灵活丰富的支持，主要表现在：可以引导压缩或非压缩系统内核，可以灵活设置/传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，支持多种文件系统，支持多种目标板环境参数存储介质，采用CRC32校验，可校验内核及镜像文件是否完好，提供多种控制台接口，使用户可以在不需要ICE的情况下通过串口/以太网/USB等接口下载数据并烧录到存储设备中去（这个功能在实际的产品中是很实用的，尤其是在软件现场升级的时候），以及提供丰富的设备驱动等。

U-boot源代码的目录结构

- board 中存放于开发板相关的配置文件，每一个开发板都以子文件夹的形式出现。
- Commom 文件夹实现u-boot行下支持的命令，每一个命令对应一个文件。
- cpu 中存放特定cpu架构相关的目录，每一款cpu架构都对应了一个子目录。
- Doc 是文档目录，有u-boot非常完善的文档。
- Drivers 中是u-boot支持的各种设备的驱动程序。
- Fs 是支持的文件系统，其中最常用的是JFFS2文件系统。
- Include 文件夹是u-boot使用的头文件，还有各种硬件平台支持的汇编文件，系统配置文件和文件系统支持的文件。
- Net 是与网络协议相关的代码，bootp协议、TFTP协议、NFS文件系统得实现。
- Tooles 是生成U-boot的工具。

对u-boot的目录有了一些了解后，分析启动代码的过程就方便多了，其中比较重要的目录就是/board、/cpu、/drivers和/include目录，如果想实现u-boot在一个平台上的移植，就要对这些目录进行深入的分析。

什么是《编译地址》？什么是《运行地址》？

1. 编译地址：32位的处理器，它的每一条指令是4个字节，以4个字节存储顺序，进行顺序执行，CPU是顺序执行的，只要没发生什么跳转，它会顺序进行执行行，编译器会对每一条指令分配一个编译地址，这是编译器分配的，在编译过程中分配的地址，我们称之为编译地址。

2. 运行地址：是指程序指令真正运行的地址，是由用户指定的，用户将运行地址烧录到哪里，哪里就是运行的地址。

比如有一个指令的编译地址是0x5，实际运行的地址是0x200，如果用户将指令烧到0x200上，那么这条指令的运行地址就是0x200，

当编译地址和运行地址不同的时候会出现什么结果？结果是不能跳转，编译后会产生跳转地址，如果实际地址和编译后产生的地址不相等，那么就不能跳转。

C语言编译地址：都希望把编译地址和实际运行地址放在一起的，但是汇编代码因为不需要做C语言到汇编的转换，可以认为的去写地址，所以直接写的就是他的运行地址这就是为什么任何bootloader刚开始会有一段汇编代码，因为起始代码编译地址和实际地址不相等，这段代码和汇编无关，跳转用的运行地址。

编译地址和运行地址如何来算呢?

- 1.假如有两个编译地址a=0x10，b=0x7，b的运行地址是0x300，那么a的运行地址就是b的运行地址加上两者编译地址的差值，a-b=0x10-0x7=0x3，a的运行地址就是0x300+0x3=0x303。

- 2.假设uboot上两条指令的编译地址为a=0x33000007和b=0x33000001，这两条指令都落在bank6上，现在要计算出他们对应的运行地址，要找出运行地址的始地址，这个是由用户烧录进去的，假设运行地址的首地址是0x0，则a的运行地址为0x7，b为0x1，就是这样算出来的。

为什么要分配编译地址？这样做有什么好处，有什么作用?

比如在函数a中定义了函数b，当执行到函数b时要进行指令跳转，要跳转到b函数所对应的起始地址上去，编译时，编译器给每条指令都分配了编译地址，如果编译器已经给分配了地址就可以直接进行跳转，查找b函数跳转指令所对应的表，进行直接跳转，因为有个编译地址和指令对应的一个表，如果没有分配，编译器就查找不到这个跳转地址，要进行计算，非常麻烦。

什么是《相对地址》?

以NOR Flash为例，NOR Falsh是映射到bank0上面，SDRAM是映射到bank6上面，uboot和内核最终是在SDRAM上面运行，最开始我们是从Nor Flash的零地址开始往后烧录，uboot中至少有一段代码编译地址和运行地址是不一样的，编译uboot或内核时，都会将编译地址放入到SDRAM中，他们最终都会在SDRAM中执行，刚开始uboot在Nor Flash中运行，运行地址是一个低端地址，是bank0中的一个地址，但编译地址是bank6中的地址，这样就会导致绝对跳转指令执行的失败，所以就引出了相对地址的概念。

那么什么是相对地址呢?

至少在bank0中uboot这段代码要知道不能用b+编译地址这样的方法去跳转指令，因为这段代码的编译地址和运行地址不一样，那如何去做呢?

要去计算这个指令运行的真实地址，计算出来后再做跳转，应该是b+运行地址，不能出现b+编译地

址，而是b+运行地址，而运行地址是算出来的。

U-Boot工作过程

大多数 Boot Loader 都包含两种不同的操作模式:"启动加载"模式和"下载"模式,这种区别仅对于开发人员才有意义。

但从最终用户的角度看,Boot Loader 的作用就是:用来加载操作系统,而并不存在所谓的启动加载模式与下载工作模式的区别。

- (一)启动加载(Boot loading)模式:这种模式也称为"自主"(Autonomous)模式。

也即 Boot Loader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行,整个过程并没有用户的介入。

这种模式是 Boot Loader 的正常工作模式,因此在嵌入式产品发布的时侯,Boot Loader 显然必须工作在这种模式下。

- (二)下载(Downloading)模式:在这种模式下,目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机(Host)下载文件,比如:下载内核映像和根文件系统映像等。

从主机下载的文件通常首先被 Boot Loader保存到目标机的RAM 中,然后再被 BootLoader写到目标机上的FLASH类固态存储设备中。

Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用;此外,以后的系统更新也会使用 Boot Loader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。

这种工作模式通常在第一次安装内核与跟文件系统时使用。或者在系统更新时使用。进行嵌入式系统调试时一般也让bootloader工作在这一模式下。

UBoot 这样功能强大的 Boot Loader 同时支持这两种工作模式,而且允许用户在这两种工作模式之间进行切换。

大多数 bootloader 都分为阶段 1(stage1)和阶段 2(stage2)两大部分,uboot 也不例外。

依赖于 CPU 体系结构的代码(如 CPU 初始化代码等)通常都放在阶段 1 中且通常用汇编语言实现;而阶段 2 则通常用 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。

第二、U-Boot总体分析

系统启动的入口点。既然我们现在要分析u-boot的启动过程，就必须先找到u-boot最先实现的是哪些代码，最先完成的是哪些任务。

另一方面一个可执行的image必须有一个入口点，并且只能有一个全局入口点，所以要通知编译器这个入口在哪里。由此我们可以找到程序的入口点是在/board/ti/ti8168_dvr/u-boot.lds中指定的，其中ENTRY(_start)说明程序从_start开始运行，而他指向的是cpu/arm_cortexa8/start.o文件。

因为我们用的是cortex-a8的cpu架构，在复位后从地址0x00000000取它的第一条指令，所以我们将Flash映射到这个地址上，这样在系统加电后，cpu将首先执行u-boot程序。

u-boot的启动过程是多阶段实现的，分了两个阶段。

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stage1中，而且通常都是用汇编语言来实现，以达到短小精悍的目的。

而stage2则通常是用C语言来实现的，这样可以实现复杂的功能，而且代码具有更好的可读性和可移植性。

U-Boot启动内核的过程可以分为两个阶段，两个阶段的功能如下：

- （1）第一阶段的功能 Ø 硬件设备初始化 Ø 加载U-Boot第二阶段代码到RAM空间 Ø 设置好栈 Ø 跳转到第二阶段代码入口
- （2）第二阶段的功能 Ø 初始化本阶段使用的硬件设备 Ø 检测系统内存映射 Ø 将内核从Flash读取到RAM中 Ø 为内核设置启动参数 Ø 调用内核

代码真正开始是在_start，设置异常向量表，这样在cpu发生异常时就跳转到/arch/arm/lib/interrupts中去执行相应得中断代码。

在interrupts文件中大部分的异常代码都没有实现具体的功能，只是打印一些异常消息，其中关键的是reset中断代码，跳到reset入口地址。

reset复位入口之前有一些段的声明。

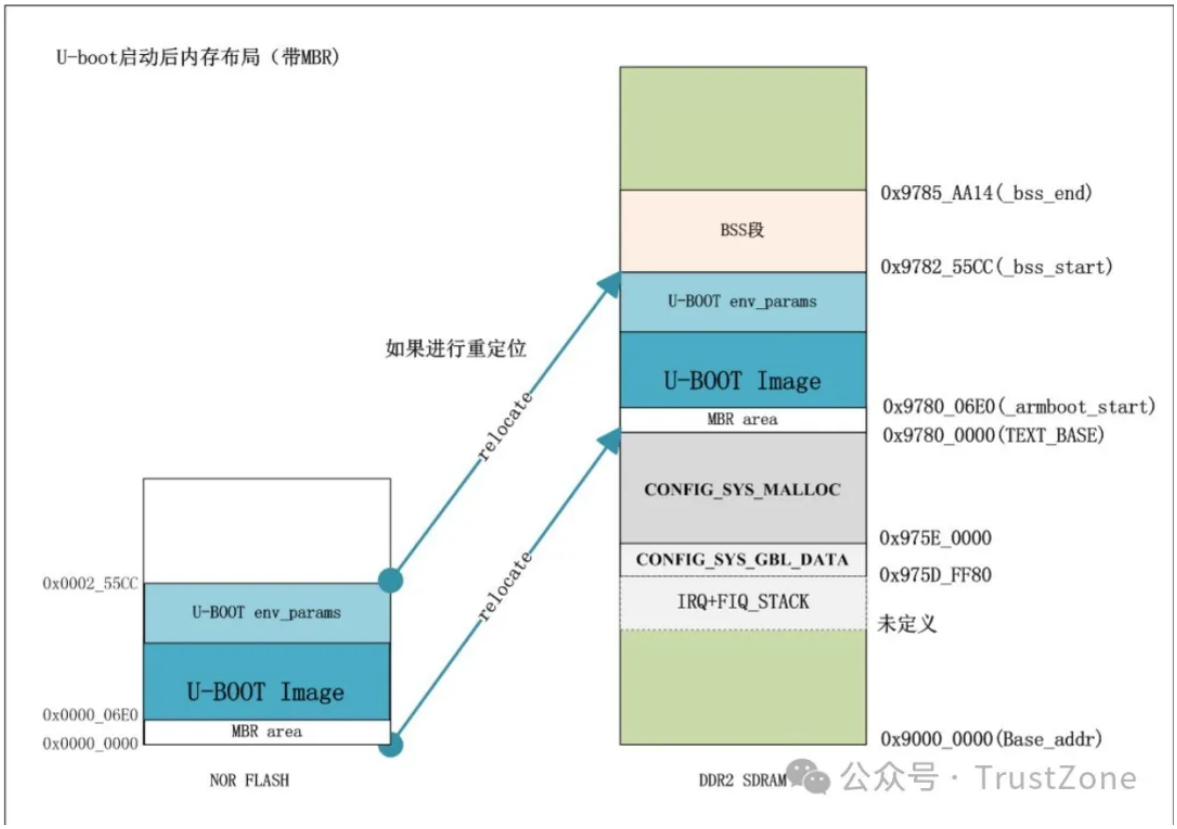
因为我们用的是 cortex-a8 的 cpu 架构，在CPU复位后从iROM地址0x00000000取它的第一条指令，执行iROM代码的功能是把flash中的前16K的代码加载到iRAM中，系统上电后将首先执行 u-boot 程序。

- 1．stage1：cpu/arm_cortexa8/start.S

- 2．当系统启动时, ARM CPU 会跳到 0x00000000去执行,一般 BootLoader 包括如下几个部分： 1> 建立异常向量表 2> 显示的切换到 SVC 且 32 指令模式 3> 设置异常向量表 4> 关闭 TLB，MMU，cache，刷新指令 cache 数据 cache 5> 关闭内部看门狗 6> 禁止所有的中断 7> 串口初始化 8> tzpc （TrustZone Protection Controller） 9> 配置系统时钟频率和总线频率 10> 设置内存区的控制寄存器 11> 设置堆栈 12> 代码的搬移阶段 代码的搬移阶段：为了获得更快的执行速度，通常把stage2加载到RAM空间中来执行，因此必须为加载Boot Loader的stage2准备好一段可用的RAM空间范围。空间大小最好是memory page大小(通常是4KB)的倍数，一般而言，1M的RAM空间已经足够了。 flash中存储的u-boot可执行文件中，代码段、数据段以及BSS段都是首尾相连存储的，所以在计算搬移大小的时候就是利用了用BSS段的首地址减去代码的首地址，这样算出来的就是实际使用的空间。程序用一个循环将代码搬移到0x81180000，即RAM底端1M空间用来存储代码。然后程序继续将中断向量表搬到RAM的顶端。由于stage2通常是C语言执行代码，所以还要建立堆栈去。在堆栈区之前还要将malloc分配的空间以及全局数据所需的空间空下来，他们的大小是由宏定义给出的，可以在相应位置修改。

  13> 跳到 C 代码部分执行

基本内存分布图(只供参考)：



-

3. 下来是u-boot启动的第二个阶段，是用c代码写的， 这部分是一些相对变化不大的部分，我们针对不同的板子改变它调用的一些初始化函数，并且通过设置一些宏定义来改变初始化的流程， 所以这些代码在移植的过程中并不需要修改，也是错误相对较少出现的文件。 在文件的开始先是定义了一个函数指针数组，通过这个数组，程序通过一个循环来按顺序进行常规的初始化，并在其后通过一些宏定义来初始化一些特定的设备。在最后程序进入一个循环，main_loop。这个循环接收用户输入的命令，以设置参数或者进行启动引导。

第三、代码分析

1 定义入口

由于一个可执行的 Image 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,该工作可通过修改连接器脚本来完成。

1. board/ti/ti8168_dvr/uboot.lds: ENTRY(_start) ==> arch/arm/cpu/cortex-a8/start.S: .globl _start

2. uboot 代码区(TEXT_BASE = 0x08070000) 定义在 board/ti/ti8168_dvr/config.mk

第一阶段对应的文件是 arch/arm/cpu/cortex-a8/start.S 和 arch/arm/cpu/cortex-a8/ti81xx/lowlevel_init.S。

U-Boot启动第一阶段流程如下：

根据cpu/cortex_a8/u-boot.lds中指定的连接方式：

看一下uboot.lds文件，在board/ti/ti8168_dvr/目录下面，uboot.lds是告诉编译器这些段改怎么划分。

GUN编译过的段，最基本的三个段是RO，RW，ZI，RO表示只读，对应于具体的指代码段，RW是数据段，ZI是归零段，就是全局变量的那段。

Uboot代码这么多，如何保证start.s会第一个执行，编译在最开始呢？就是通过uboot.lds链接文件进行

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;//起始地址
    . = ALIGN(4);//4字节对齐
    .text   : //test指代码段，上面3行标识是不占用任何空间的
    {
       arch/arm/cpu/arm_cortexa8/start.o     (.text) //这里把start.o放在第一位就表示把star
       arch/arm/cpu/arm_cortexa8/ti81xx/lowlevel_init.o      (.text)
       *(.text)
    }
    . = ALIGN(4);//前面的 “.” 代表当前值，是计算一个当前的值，是计算上面占用的整个空间，再加一个单元
    .rodata : { *(.rodata) }
    . = ALIGN(4);
    .data : { *(.data) }
    . = ALIGN(4);
    .got : { *(.got) }

    . = .;
    __u_boot_cmd_start = .;
    .u_boot_cmd : { *(.u_boot_cmd) }
    __u_boot_cmd_end = .;

    . = ALIGN(4);
    __bss_start = .;
    .bss (NOLOAD) : { *(.bss) . = ALIGN(4); }
    _end = .;
}
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
```

第一个链接的是cpu/arm_cortexa8/start.o，因此u-boot.bin的入口代码在cpu/arm_cortexa8/start.o中，其源代码在cpu/arm_cortexa8/start.S中。下面我们来分析cpu/arm_cortexa8/start.S的执行。

2. 硬件设备初始化

1> 设置异常向量

下面代码是系统启动后U-boot上电后运行的第一段代码，它是什么意思？

u-boot对应的第一阶段代码放在cpu/arm_cortexa8/start.S文件中，入口代码如下：

```
.globl _start                                   /*声明一个符号可被其它文件引用，相当于声明了一
_start:    b     reset                          /* 复位，b是不带返回的跳转(bl是带返回的跳转)
    ldr   pc, _undefined_instruction            /* 未定义指令向量 l---dr相当于mov操作*/
    ldr   pc, _software_interrupt               /*  软件中断向量 */
    ldr   pc, _prefetch_abort                   /*  预取指令异常向量 */
    ldr   pc, _data_abort                       /*  数据操作异常向量 */
    ldr   pc, _not_used                         /*  未使用    */
    ldr   pc, _irq                              /*  irq中断向量   */
    ldr   pc, _fiq                              /*  fiq中断向量   */

/*  中断向量表入口地址 */
_undefined_instruction:    .word undefined_instruction  /*就是在当前地址，即_undefined_ir
_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    //word伪操作用于分配一段字内存单元(分配的单元都是字对齐的)，并用伪
_pad:           .word 0x12345678 /* now 16*4=64 */
.global _end_vect
_end_vect:

    .balignl 16,0xdeadbeef
```

他们是系统定义的异常，一上电程序跳转到reset异常处执行相应的汇编指令，下面定义出的都是不同的异常，比如软件发生软中断时，CPU就会去执行软中断的指令，这些异常中断在CUP中地址是从0开始，每个异常占4个字节。

ldr pc, _undefined_instruction：表示把_undefined_instruction存放的数值存放到pc指针上。

_undefined_instruction: .word undefined_instruction：表示未定义的这个异常是由.word来定义的，它表示定义一个字，一个32位的数。

.word后面的数：表示把该标识的编译地址写入当前地址，标识是不占用任何指令的。把标识存放的数值copy到指针pc上面，那么标识上存放的值是什么？是由.word undefined_instruction来指定的，pc就代表你运行代码的地址，她就实现了CPU要做一次跳转时的工作。

以上代码设置了ARM异常向量表，各个异常向量介绍如下：

地址	异常	进入模式	描述				
0x00000000	复位	管理模式	复位电平有效时，产生复位异常，程序跳转到复位处理程序处执行				
0x00000004	未定义指令	未定义模式	遇到不能处理的指令时，产生未定义指令异常				
0x00000008	软件中断	管理模式	执行SWI指令产生，用于用户模式下的程序调用特权操作指令				

地址	异常	进入模式	描述				
0x00000000c	预存指令	中止模式	处理器预取指令的地址不存在，或该地址不允许当前指令访问，产生指令预取中止异常				
0x00000010	数据操作	中止模式	处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常				
0x00000014	未使用	未使用	未使用				
0x00000018	IRQ	IRQ	外部中断请求有效，且CPSR中的I位为0时，产生IRQ异常				
0x0000001c	FIQ	FIQ	快速中断请求引脚有效，且CPSR中的F位为0时，产生FIQ异常				

在cpu/arm_cortexa8/start.S中还有这些异常对应的异常处理程序。当一个异常产生时，CPU根据异常号在异常向量表中找到对应的异常向量，然后执行异常向量处的跳转指令，CPU就跳转到对应的异常处理程序执行。

其中复位异常向量的指令“b reset”决定了U-Boot启动后将自动跳转到标号“reset”处执行。

```
_TEXT_BASE:
    .word TEXT_BASE //0x07080000,在board/ti/ti8168_dvr/config.mk中,这段话表示，用户告诉编译

.globl _armboot_start
_armboot_start:
 .word _start /*_start 是uboot的第一行代码的标号,代表的是第一行代码的地址*/

.globl _bss_start
_bss_start:
 .word __bss_start //在cpu/arm_cortexa8/u-boot.lds中定义

.globl _bss_end
_bss_end:
 .word _end //在cpu/arm_cortexa8/u-boot.lds中定义

#ifdef CONFIG_USE_IRQ  // 这个宏没有定义,预编译不执行
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
 .word 0x0badc0de

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
 .word 0x0badc0de
#endif
```

2> CPU进入SVC模式

```
reset:
    /*
     * set the cpu to SVC32 mode
    */
    mrs r0, cpsr
    bic  r0, r0, #0x1f        /*工作模式位清零 */
    orr   r0, r0, #0xd3       /*工作模式位设置为“10011”（管理模式），并将中断禁止位和快中断禁止位
    msr cpsr, r0
```

以上代码将CPU的工作模式位设置为管理模式，即设置相应的CPSR程序状态字，并将中断禁止位和快中断禁止位置一，从而屏蔽了IRQ和FIQ中断。

操作系统先注册一个总的中断，然后去查是由哪个中断源产生的中断，再去查用户注册的中断表，查出来后就去执行用户定义的用户中断处理函数。

```
#if (CONFIG_OMAP34XX) // 这个宏没有定义,下面的代码不会预编译
    /* Copy vectors to mask ROM indirect addr */
    adr r0, _start  @ r0 <- current position of code
    add r0, r0, #4  @ skip reset vector
    mov r2, #64   @ r2 <- size to copy
    add r2, r0, r2  @ r2 <- source end address
    mov r1, #SRAM_OFFSET0 @ build vect addr
    mov r3, #SRAM_OFFSET1
    add r1, r1, r3
    mov r3, #SRAM_OFFSET2
```

```
    add r1, r1, r3
next:
    ldmia r0!, {r3 - r10}  @ copy from source address [r0]
    stmia r1!, {r3 - r10}  @ copy to   target address [r1]
    cmp r0, r2   @ until source end address [r2]
    bne next   @ loop until equal */
#if !defined(CONFIG_SYS_NAND_BOOT) && !defined(CONFIG_SYS_ONENAND_BOOT)
    /* No need to copy/exec the clock code - DPLL adjust already done
     * in NAND/oneNAND Boot.
     */
    bl cpy_clk_code  @ put dpll adjust code behind vectors
#endif /* NAND Boot */
#endif
```

3> CPU初始化

```
 /* the mask ROM code should have PLL and others stable */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif
```

cpu_init_crit这段代码在U-Boot正常启动时才需要执行，若将U-Boot从RAM中启动则应该注释掉这段代码。下面分析一下cpu_init_crit到底做了什么：

```
/*************************************************************************
 *
 * CPU_init_critical registers
 *
 * setup important registers
 * setup memory timing
 *
 *************************************************************************/
cpu_init_crit:
    /*
     * Invalidate L1 I/D
     */
    mov    r0, #0            @ set up for MCR
    mcr    p15, 0, r0, c8, c7, 0    @ invalidate TLBs  //将0写入c8，使TLB内容无效
    mcr    p15, 0, r0, c7, c5, 0    @ invalidate icache //将0写入c7，使Cache内容无效

    /*
     * 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 12 (Z---) BTB
    mcr    p15, 0, r0, c1, c0, 0 //修改CP15的c1寄存器来实现关闭MMU

    /*
     * 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.
     */
    mov    ip, lr            @ persevere link reg across call
    bl    lowlevel_init      @ go setup pll,mux,memory
```

```
    mov    lr, ip            @ restore link
    mov    pc, lr            @ back to my caller
```

代码中的c0，c1，c7，c8都是cortex-a8的协处理器CP15的寄存器。其中c7是cache控制寄存器，c8是TLB控制寄存器。325~327行代码将0写入c7、c8，使Cache，TLB内容无效。

第332~337行代码关闭了MMU。这是通过修改CP15的c1寄存器来实现的，先看CP15的c1寄存器的格式（仅列出代码中用到的位）：

CP15的c1寄存器格式（部分）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
.	.	V	I	.	.	R	S	B	.	.	.	.	C	A	M

各个位的意义如下：

V :表示异常向量表所在的位置，0：异常向量在0x00000000；1：异常向量在 0xFFFF0000I : 0：关闭ICaches；1： 开启ICachesR、S : 用来与页表中的描述符一起确定内存的访问权限B : 0：CPU为小字节序；1： CPU为大字节序C : 0：关闭DCaches；1：开启DCachesA : 0：数据访问时不进行地址对齐检查；1：数据访问时进行地址对齐检查。M : 0：关闭MMU；1：开启MMU

为什么要关闭catch和MMU呢？catch和MMU是做什么用的？

MMU是Memory Management Unit的缩写，中文名是内存管理单元，它是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路, 同时也负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权

- 概述：

  - 一，关catch　　catch和MMU是通过CP15管理的，刚上电的时候，CPU还不能管理他们　　上电的时候MMU必须关闭，指令catch可关闭，可不关闭，但数据catch一定要关闭　　否则可能导致刚开始的代码里面，去取数据的时候，从catch里面取，而这时候RAM中数据还没有catch过来，导致数据预取异常

  - 二：关MMU　　因为MMU是;把虚拟地址转化为物理地址得作用　　而目的是设置控制寄存器，而控制寄存器本来就是实地址（物理地址），再使能MMU，不就是多此一举了吗?

详细分析---　　Catch是cpu内部的一个2级缓存，它的作用是将常用的数据和指令放在cpu内部，MMU是用来把虚实地址转换为物理地址用的　　我们的目的:是设置控制的寄存器，寄存器都是实地址（物理地址），如果既要开启MMU又要做虚实地址转换的话，中间还多一步，多此一举了嘛?

先要把实地址转换成虚地址，然后再做设置，但对uboot而言就是起到一个简单的初始化的作用和引导操作系统，如果开启MMU的话，很麻烦，也没必要，所以关闭MMU.

说到catch就必须提到一个关键字Volatile，以后在设置寄存器时会经常遇到，他的本质：是告诉编译器不要对我的代码进行优化，作用是让编写者感觉不倒变量的变化情况（也就是说，让它执行速度加快吧）

优化的过程：是将常用的代码取出来放到catch中，它没有从实际的物理地址去取，它直接从cpu的缓存中去取，但常用的代码就是为了感觉一些常用变量的变化。

优化原因：如果正在取数据的时候发生跳变，那么就感觉不到变量的变化了，所以在这种情况下要用Volatile关键字告诉编译器不要做优化，每次从实际的物理地址中去取指令，这就是为什么关闭catch关闭MMU。但在C语言中是不会关闭catch和MMU的，会打开，如果编写者要感觉外界变化，或变化太快，从catch中取数据会有误差，就加一个关键字Volatile。

4> 初始化RAM控制寄存器

bl lowlevel_init下来初始化各个bank，把各个bank设置必须搞清楚，对以后移植复杂的uboot有很大帮助。

设置完毕后拷贝uboot代码到4k空间，拷贝完毕后执行内存中的uboot代码，其中的lowlevel_init就完成了内存初始化的工作，由于内存初始化是依赖于开发板的，因此lowlevel_init的代码一般放在

board下面相应的目录中。

对于ti8168_dvr，lowlevel_init在arch/arm/arm_cortexa8/ti81xx/level_init.S中定义如下：

```
/*****************************************************************************
 * lowlevel_init: - Platform low level init.
 * Corrupted Register : r0, r1, r2, r3, r4, r5, r6
 *****************************************************************************/
.globl lowlevel_init
lowlevel_init:
    /* The link register is saved in ip by start.S */
    mov r6, ip
    /* check if we are already running from RAM */ //判断程序是否已经在 RAM 中运行
    ldr r2, _lowlevel_init
    ldr r3, _TEXT_BASE
    sub r4, r2, r3
    sub r0, pc, r4
    /* require dummy instr or subtract pc by 4 instead i'm doing stack init */
    ldr sp, SRAM_STACK
mark1:
    ldr r5, _mark1
    sub r5, r5, r2 /* bytes between mark1 and lowlevel_init */
    sub r0, r0, r5 /* r0 <- _start w.r.t current place of execution */
    mov r10, #0x0 /* r10 has in_ddr used by s_init() */

#ifdef CONFIG_NOR_BOOT
    cmp r0, #0x08000000 /* check for running from NOR */
    beq ocmc_init_start /* if == then running from NOR */
    ands r0, r0, #0xC0000000 /* MSB 2 bits <> 0 then we are in ocmc or DDR */
    cmp r0, #0x40000000 /* if running from ocmc */
    beq nor_init_start /* if == skip ocmc init and jump to nor init */
    mov r10, #0x01 /* if <> we are running from DDR hence skip ddr init */
       /* by setting in_ddr to 1 */
    b s_init_start /* and jump to s_init */
#else
    ands r0, r0, #0xC0000000 /* MSB 2 bits <> 0 then we are in ocmc or DDR */
    cmp r0, #0x80000000
    bne s_init_start
    mov r10, #0x01
    b s_init_start
#endif

#ifdef CONFIG_NOR_BOOT
ocmc_init_start:
    /**** enable ocmc 0 ****/
    /* CLKSTCTRL */
    ldr r5, cm_alwon_ocmc_0_clkstctrl_addr
    mov r2, #0x2
    str r2, [r5]
    /* wait for gpmc enable to settle */
ocmc0_wait0:
    ldr r2, [r5]
    ands r2, r2, #0x00000100
    cmp r2, #0x00000100
    bne ocmc0_wait0
    /* CLKCTRL */
    ldr r5, cm_alwon_ocmc_0_clkctrl_addr
    mov r2, #0x2
    str r2, [r5]
    /* wait for gpmc enable to settle */
```

```
ocmc0_wait1:
    ldr r2, [r5]
    ands r2, r2, #0x00030000
    cmp r2, #0
    bne ocmc0_wait1

#ifdef CONFIG_TI816X
    /**** enable ocmc 1 ****/
    /* CLKSTCTRL */
    ldr r5, cm_alwon_ocmc_1_clkstctrl_addr
    mov r2, #0x2
    str r2, [r5]
    /* wait for gpmc enable to settle */
ocmc1_wait0:
    ldr r2, [r5]
    ands r2, r2, #0x00000100
    cmp r2, #0x00000100
    bne ocmc1_wait0
    /* CLKCTRL */
    ldr r5, cm_alwon_ocmc_1_clkctrl_addr
    mov r2, #0x2
    str r2, [r5]
    /* wait for gpmc enable to settle */
ocmc1_wait1:
    ldr r2, [r5]
    ands r2, r2, #0x00030000
    cmp r2, #0
    bne ocmc1_wait1
#endif

nor_init_start:
    /* gpmc init */
    bl  cpy_nor_gpmc_code /* copy nor gpmc init code to sram */
    mov r0, pc
    add r0, r0, #12  /* 12 is for next three instructions */
    mov lr, r0  /* gpmc init code in sram should return to s_init_start */
    ldr r0, sram_pc_start
    mov pc, r0 /* transfer ctrl to nor_gpmc_init() in sram */
#endif

s_init_start:
    mov r0, r10 /* passing in_ddr in r0 */
    bl s_init // 调用C语言初始化系统时钟和MUX
    /* back to arch calling code */
    mov pc, r6
```

5> 重定位

ti8168_dvr 开发板没有在这里进行重定位，但是我们还是要分析一下这里的重定位：

```
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
relocate:        @ relocate U-Boot to RAM
     adr r0, _start    @ 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 stack_setup
```

注释：

- （1） 使用相对寻址，以 PC 值为基地址计算出当前代码的开始地址，通过反汇编u-boot.bin，_start在FLASH上代表地址：0x0000_06E0 ，指令 addr r0 ，_start地址为 0x0000_0744 ，执行到该指令时，PC=0x0000_0744+8=0x0000_074c，这里，编译器将用指令SUB r0，PC，#6C ；因此 r0=0x0000_06E0，相当于把代码的开始地址送到了r0 中。事实上，在 0x0 地址，存在一条跳转指令，跳到了_start 标签处，而_start标签处才是 b reset 真正的跳转。所以，程序在 FLASH 上运行的真正 reset 跳转，是在_start=0x0000_06E0 地址对应的 b reset 指令。
- （2） 把_TEXT_BASE 地址对应的内存里的值(TEXT_BASE) 送到r1 寄存器，这里是 0x07080000
- （3） 比较 r0 寄存器和 r1 寄存器的值。
- （4） 如果 r0=r1，就跳去执行 stack_setup 程序段，设置内存中的栈空间。比较的目的就是看当前程序的运行是在内存里还是在FLASH里，如果是 debug 模式，那么 u-boot是在内存里运行，其开始地址就是TEXT_BASE ，即0x70800000 。上面的4 句代码，就是比较一下，看看当前程序代码在什么位置，如果已经在内存指定的0x07080000 这个位置了，就不必再进行重定位了，而直接进行堆栈设置。

6> 设置堆栈

```
/* Set up the stack */
stack_setup:
    ldr  r0, _TEXT_BASE    @ upper 128 KiB: relocated uboot
    sub r0, r0, #CONFIG_SYS_MALLOC_LEN @ malloc area
    sub r0, r0, #CONFIG_SYS_GBL_DATA_SIZE @ bdinfo  //跳过全局数据区
```

注释：　　这段代码在TEXT_BASE （0x0708_0000）的下面，也就是挨着这个地址往下，建立：动态内存区域和全局数据结构区域。只要将sp指针指向一段没有被使用的内存就完成栈的设置了。根据上面的代码可以知道U-Boot内存使用情况了。

使用SourceInsight 跟踪到 ti8168_dvr.h 文件中，有：

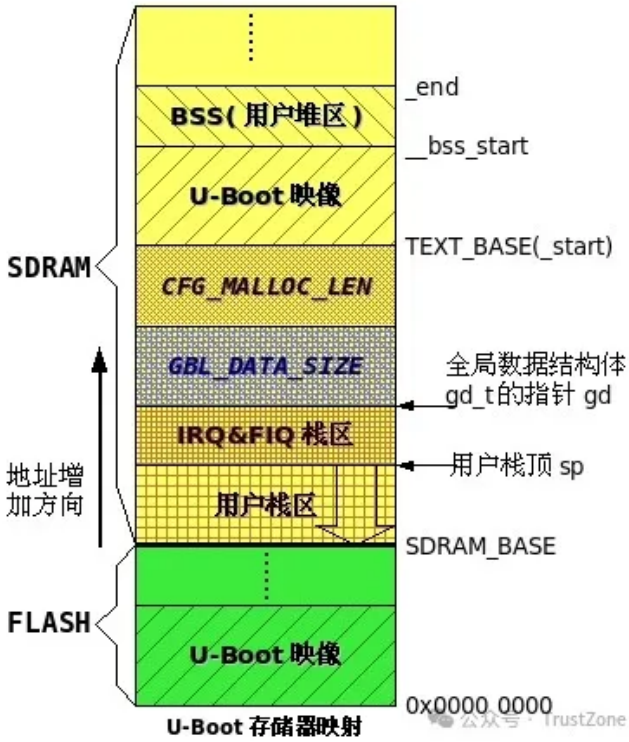
```
/*
  * Size of malloc() pool
 */
#define CONFIG_SYS_MALLOC_LEN    (CONFIG_ENV_SIZE + 32 * 1024)
/* size in bytes reserved for initial data */
#define CONFIG_SYS _GBL_DATA_SIZE 128
#define CONFIG_ENV_SIZE         0x2000
```

由此可见，从TEXT_BASE 往下的32kB+8KB 空间用作动态内存分配；再继续往下128 个字节做为全局数据结构指针。

```
#ifdef CONFIG_USE_IRQ
    sub r0, r0, #(CONFIG_STACKSIZE_IRQ + CONFIG_STACKSIZE_FIQ)
#endif
    sub  sp, r0, #12    @ leav e 3 words for abort-stack
    and  sp, sp, #~7    @ 8 byte alinged for (ldr/str)d
```

注释： 如果使用外部中断IRQ ，在全局数据结构指针继续往低地址方向分配，分配的空间大小由CONFIG_STACKSIZE_IRQ 和CONFIG_STACKSIZE_FIQ 在I.MX51_bbg_android.h 文件中定义。默认时，该头文件中，没有定义 CONFIG_USE_IRQ，也没有定义空间大小，因此意味着此时不对IRQ 和FIQ 进行空间预留。

只要将sp指针指向一段没有被使用的内存就完成栈的设置了。根据上面的代码可以知道U-Boot内存使用情况了，如下图所示:



7> 代码的搬移阶段

```
    ldr r2, _armboot_start
    ldr r3, _bss_start
    sub r2, r3, r2    @ r2 <- size of armboot
    add r2, r0, r2   @ r2  <- source end address
```

注释：

- （1） 把_armboot_start程序段的首地址读进 r2 寄存器。事实上，根据_armboot_start标号的定义：_armboot_star ： .word _start，_armboot_start是内存地址，这个地址中存放着TEXT_BASE+_start，即 0x90708_06E0 。所以_armboot_start所对应的内存地址是真正的内存中u-boot 第一条指令的地址。
- （2） 把_bss_start代码段首地址读进r3 寄存器
- （3） 寄存器 r3 的值-寄存器r2 的值，差值送回寄存器 r2 ，这个差值就是_armboot_start代码（u-boot 代码段）所占用的内存空间大小。
- （4） 计算出_armboot_start代码的结束地址。寄存器 r0 中保存着 FLASH上代码的开始地址，r2 中保存着u-boot 代码的容量，那么r0+r2?r2 后，r2 中保存的就是u-boot 在FLASH上的最后一句代码的地址。

```
copy_loop:     @ copy 32 bytes at a time
    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
#endif  /* CONFIG_SKIP_RELOCATE_UBOOT */
```

注释：上面4 句话用来实现 U-BOOT 代码从FLASH—>DDR2 MEM 的搬移。

- （1 ） r0=_start=0x0000_06E0 ，即指向 FLASH中代码开始地址；读 8 个字（32 字节），分别存到r3-r10 这个8 个32 为通用寄存器中。然后，r0+8 ?r0
- （2 ） r1=TEXT_BASE=0x9780_0000，指向内存基地址；把 r3-r10 这8 个寄存器中的数据写入内存中，因此一次stmia 指令传输8 个字，共 32 字节。然后，r1+8 ?r1
- （3 ） r2 为FLASH上代码结束地址。比较r0 和r2 ，即是否到达代码结尾。
- （4 ） 如果没有到达代码结尾，继续循环复制，直到完成。

8> 清除BSS段

```
/* Clear BSS (if any). Is below tx (watch load addr - need space) */
clear_bss:
    ldr    r0, _bss_start        @ find start of bss segment /* BSS段开始地址，在u-boot.
    ldr    r1, _bss_end        @ stop here /* BSS段结束地址，在u-boot.lds中指定*/
    mov    r2, #0x00000000        @ clear value /* 将bss段清零*/
clbss_l:
    str    r2, [r0]        @ clear BSS location
    cmp    r0, r1            @ are we at the end yet
    add    r0, r0, #4        @ increment clear index pointer
    bne    clbss_l            @ keep clearing till at end
```

注释：

这段代码，对bss 段进行初始化，从.lds 文件可以知道它的开始和结束位置。从对u-boot.bin的反汇编结果看，_bss_start=0x0708_55CC，这个地址恰好位于内存中 u-boot 映像的上方相邻地址；而bss 段的结束地址_bss_end=0x0708_AA14。前后地址相减，可以算出bss 段占用的空间是213KB 。

这段初始化的方式是把 bss 段全部写 0 ，寄存器 r0 所指示的目标地址指针按照+4递增方式循环，直到全部初始化完成。

初始值为0，无初始值的全局变量，静态变量将自动被放在BSS段。应该将这些变量的初始值赋为0，否则这些变量的初始值将是一个随机的值，若有些程序直接使用这些没有初始化的变量将引起未知的后果。

9> 跳转到第二阶段代码入口start_armboot处。

```
    ldr pc, _start_armboot @ jump to C code
```

```
_start_armboot: .word start_armboot
```

- 问题一：如果换一块开发板有可能改哪些东西？首先，cpu的运行模式，如果需要对cpu进行设置那就设置，管看门狗，关中断不用改，时钟有可能要改，如果能正常使用则不用改，关闭catch和MMU不用改，设置bank有可能要改。最后一步拷贝时看地址会不会变，如果变化也要改，执行内存中代码，地址有可能要改。

- 问题二：Nor Flash和Nand Flash本质区别：就在于是否进行代码拷贝，也就是下面代码所表述：无论是Nor Flash还是Nand Flash，核心思想就是将uboot代码搬运到内存中去运行，但是没有拷贝bss后面这段代码，只拷贝bss前面的代码，bss代码是放置全局变量的。Bss段代码是为了清零，拷贝过去再清零重复操作

---

3. U-Boot启动第二阶段代码分析：start_armboot

start_armboot函数在lib_arm/board.c中定义，是U-Boot第二阶段代码的入口。U-Boot启动第二阶段流程如下：

在分析start_armboot函数前先来看看一些重要的数据结构：

（1）gd_t结构体

U-Boot使用了一个结构体gd_t来存储全局数据区的数据，这个结构体在include/asm-arm/global_data.h中定义如下：

```
typedef    struct    global_data {
    bd_t        *bd;
    unsigned long    flags;
    unsigned long    baudrate;
    unsigned long    have_console;    /* serial_init() was called */
    unsigned long    env_addr;    /* Address  of Environment struct */
    unsigned long    env_valid;    /* Checksum of Environment valid? */
    unsigned long    fb_base;    /* base address of frame buffer */
#ifdef CONFIG_VFD
    unsigned char    vfd_type;    /* display type */
#endif
#ifdef CONFIG_FSL_ESDHC
```

```
    unsigned long    sdhc_clk;
#endif
#if 0
    unsigned long    cpu_clk;    /* CPU clock in Hz!        */
    unsigned long    bus_clk;
    phys_size_t    ram_size;    /* RAM size */
    unsigned long    reset_status;    /* reset status register at boot */
#endif
    void        **jt;        /* jump table */
} gd_t;
```

U-Boot使用了一个存储在寄存器中的指针gd来记录全局数据区的地址：

```
#define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("r8")
```

DECLARE_GLOBAL_DATA_PTR定义一个gd_t全局数据结构的指针，这个指针存放在指定的寄存器r8中。

这个声明也避免编译器把r8分配给其它的变量。

任何想要访问全局数据区的代码，只要代码开头加入“DECLARE_GLOBAL_DATA_PTR”一行代码，然后就可以使用gd指针来访问全局数据区了。

根据U-Boot内存使用图中可以计算gd的值：

gd = TEXT_BASE －CONFIG_SYS_MALLOC_LEN － sizeof(gd_t)

（2）bd_t结构体

bd_t在include/asm/u-boot.h中定义如下：

```
typedef struct bd_info {
    int                bi_baudrate;               /* 串口通讯波特率 */
    unsigned long     bi_ip_addr;          /* IP 地址*/
    struct environment_s        *bi_env;              /* 环境变量开始地址 */
    ulong             bi_arch_number;      /* 开发板的机器码 */
    ulong             bi_boot_params;       /* 内核参数的开始地址 */
    struct                          /* RAM配置信息 */
    {
              ulong start;
              ulong size;
    }bi_dram[CONFIG_NR_DRAM_BANKS];
} bd_t;
```

U-Boot启动内核时要给内核传递参数，这时就要使用gd_t，bd_t结构体中的信息来设置标记列表。

第一阶段调用start_armboot指向C语言执行代码区，首先它要从内存上的重定位数据获得不完全配置的全局数据表格和板级信息表格，即获得gd_t和bd_t，

这两个类型变量记录了刚启动时的信息，并将要记录作为引导内核和文件系统的参数，如bootargs等等，并且将来还会在启动内核时，由uboot交由kernel时会有所用。

（3）init_sequence数组

U-Boot使用一个数组init_sequence来存储对于大多数开发板都要执行的初始化函数的函数指针。

init_sequence数组中有较多的编译选项，去掉编译选项后init_sequence数组如下所示：

```
typedef int (init_fnc_t) (void);

int print_cpuinfo (void);

init_fnc_t *init_sequence[] = {
#if defined(CONFIG_ARCH_CPU_INIT)
    arch_cpu_init,        /* basic arch cpu dependent setup */
#endif
    board_init,         /*开发板相关的配置--board/samsung/mini2440/mini2440.c */
#if defined(CONFIG_USE_IRQ)
    interrupt_init,        /* set up exceptions */
#endif
    timer_init,         /*开发板相关的配置--board/samsung/mini2440/mini2440.c */
#ifdef CONFIG_FSL_ESDHC
    get_clocks,
#endif
    env_init,        /*初始化环境变量--common/env_flash.c 或common/env_nand.c*/
    init_baudrate,         /*初始化波特率-- lib_arm/board.c */
    serial_init,         /* 串口初始化-- drivers/serial/serial_s3c24x0.c */
    console_init_f,        /* 控制通讯台初始化阶段1-- common/console.c */
    display_banner,        /*打印U-Boot版本、编译的时间-- gedit lib_arm/board.c */
#if defined(CONFIG_DISPLAY_CPUINFO)
    print_cpuinfo,        /* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,        /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
    init_func_i2c,
#endif
    dram_init,         /*配置可用的RAM-- board/samsung/mini2440/mini2440.c */
#if defined(CONFIG_CMD_PCI) || defined (CONFIG_PCI)
    arm_pci_init,
#endif
    display_dram_config,  /* 显示RAM大小-- lib_arm/board.c */
    NULL,
};
```

其中的board_init函数在board/ti/ti8168_dvr/dvr.c中定义，该函数设置了软复位UART，以及一些GPIO寄存器的值，还设置了U-Boot机器码和内核启动参数地址：

```
/*
 * Basic board specific setup
 */
int board_init(void)
{
    u32 regVal;
```

```
    /* Get Timer and UART out of reset */

    /* UART 软复位 */
    regVal = __raw_readl(UART_SYSCFG);
    regVal |= 0x2;
    __raw_writel(regVal, UART_SYSCFG);
    while( (__raw_readl(UART_SYSSTS) & 0x1) != 0x1);

    /* 关闭smart idle */
    regVal = __raw_readl(UART_SYSCFG);
    regVal |= (1<<3);
    __raw_writel(regVal, UART_SYSCFG);

    /* 8168 dvr 开发板的机器码 */
    gd->bd->bi_arch_number = MACH_TYPE_TI8168EVM;

    /* 内核启动参数地址 */
    gd->bd->bi_boot_params = PHYS_DRAM_1 + 0x100;

    /*通用存储控制模块 初始化*/
    gpmc_init();
    /*gpio 初始化*/
    gpio_clkctrl_enable();
    gpio_dvr_init();
    /*初始化模拟spi的IO*/
    #if defined(CONFIG_3WIRE_EEPROM)
    dvr_netra_eeprom_init();
    #endif

    //# RDK 支持8bit NAND
    gpmc_set_cs_buswidth(0, 0);

    return 0;
}
```

其中的dram_init函数在board/ti/ti8168_dvr/dvr.c中定义如下：

```
/*
 * Configure DRAM banks
 *
 * Description: sets uboots idea of sdram size
 */
int dram_init(void)
{
    /* Fill up board info */
    gd->bd->bi_dram[0].start = PHYS_DRAM_1;
    gd->bd->bi_dram[0].size = PHYS_DRAM_1_SIZE;

    gd->bd->bi_dram[1].start = PHYS_DRAM_2;
    gd->bd->bi_dram[1].size = PHYS_DRAM_2_SIZE;

    return 0;
}
```

ti8168 dvr使用2片1GB的DRAM组成了2GB的内存，接在存储控制器的DDR3，地址空间是0x80000000~0xFFFFFFFF。

在include/configs/ti8168_dvr.h中PHYS_DRAM_1和PHYS_DRAM_1_SIZE, 分别被定义为0x80000000和0xC0000000。

分析完上述的数据结构，下面来分析start_armboot函数：

```
void start_armboot (void)
{
	init_fnc_t **init_fnc_ptr;
	char *s;
	… …
	/* 计算全局数据结构的地址gd */
	gd = (gd_t*)(_armboot_start - CONFIG_SYS_MALLOC_LEN - sizeof(gd_t));
	… …
	memset ((void*)gd, 0, sizeof (gd_t));
	gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
	memset (gd->bd, 0, sizeof (bd_t));
	gd->flags |= GD_FLG_RELOC;

	monitor_flash_len = _bss_start - _armboot_start;

	/* 逐个调用init_sequence数组中的初始化函数  */
	for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
		if ((*init_fnc_ptr)() != 0) {
			hang ();
		}
	}

	/* armboot_start 在cpu/arm920t/start.S 中被初始化为u-boot.lds连接脚本中的_start */
	mem_malloc_init (_armboot_start - CONFIG_SYS_MALLOC_LEN,
			CONFIG_SYS_MALLOC_LEN);

	/* NOR Flash初始化 */
#ifndef CONFIG_SYS_NO_FLASH
	/* configure available FLASH banks */
	display_flash_config (flash_init ());
#endif /* CONFIG_SYS_NO_FLASH */

	… …
	/* NAND Flash 初始化*/
#if defined(CONFIG_CMD_NAND)
	puts ("NAND:  ");
	nand_init();		/* go init the NAND */
#endif
	… …
	/*配置环境变量，重新定位 */
	env_relocate ();
	… …
	/* 从环境变量中获取IP地址 */
	gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
	stdio_init (); /* get the devices list going. */
	jumptable_init ();
	… …
	console_init_r (); /* fully init console as a device */
	… …
	/* enable exceptions */
	enable_interrupts ();

#ifdef CONFIG_USB_DEVICE
	usb_init_slave();
#endif

	/* Initialize from environment */
	if ((s = getenv ("loadaddr")) != NULL) {
		load_addr = simple_strtoul (s, NULL, 16);
	}
#if defined(CONFIG_CMD_NET)
	if ((s = getenv ("bootfile")) != NULL) {
```

```
                copy_filename (BootFile, s, sizeof (BootFile));
        }
#endif
        … …
        /* 网卡初始化 */
#if defined(CONFIG_CMD_NET)
#if defined(CONFIG_NET_MULTI)
        puts ("Net:   ");
#endif
        eth_initialize(gd->bd);
… …
#endif

        /* main_loop() can return to retry autoboot, if so just run it again. */
        for (;;) {
                main_loop ();
        }
        /* NOTREACHED - no way out of command loop except booting */
}
```

main_loop函数在common/main.c中定义。

```
void main_loop (void)
{
#ifndef CONFIG_SYS_HUSH_PARSER
    static char lastcommand[CONFIG_SYS_CBSIZE] = { 0, };
    int len;
    int rc = 1;
    int flag;
#endif

#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    char *s;
    int bootdelay;
#endif
#ifdef CONFIG_PREBOOT
    char *p;
#endif
#ifdef CONFIG_BOOTCOUNT_LIMIT
    unsigned long bootcount = 0;
    unsigned long bootlimit = 0;
    char *bcs;
    char bcs_set[16];
#endif /* CONFIG_BOOTCOUNT_LIMIT */

#if defined(CONFIG_VFD) && defined(VFD_TEST_LOGO)
    ulong bmp = 0;        /* default bitmap */
    extern int trab_vfd (ulong bitmap);

#ifdef CONFIG_MODEM_SUPPORT
    if (do_mdm_init)
        bmp = 1;    /* alternate bitmap */
#endif
    trab_vfd (bmp);
#endif    /* CONFIG_VFD && VFD_TEST_LOGO */

#ifdef CONFIG_BOOTCOUNT_LIMIT
    bootcount = bootcount_load();
    bootcount++;
    bootcount_store (bootcount);
    sprintf (bcs_set, "%lu", bootcount);
    setenv ("bootcount", bcs_set);
    bcs = getenv ("bootlimit");
    bootlimit = bcs ? simple_strtoul (bcs, NULL, 10) : 0;
#endif /* CONFIG_BOOTCOUNT_LIMIT */

#ifdef CONFIG_MODEM_SUPPORT
    debug ("DEBUG: main_loop:   do_mdm_init=%d\n", do_mdm_init);
    if (do_mdm_init) {
        char *str = strdup(getenv("mdm_cmd"));
        setenv ("preboot", str);  /* set or delete definition */
        if (str != NULL)
            free (str);
        mdm_init(); /* wait for modem connection */
    }
#endif  /* CONFIG_MODEM_SUPPORT */

#ifdef CONFIG_VERSION_VARIABLE
    {
        extern char version_string[];

        setenv ("ver", version_string);  /* set version variable */
    }
#endif /* CONFIG_VERSION_VARIABLE */
```

```
#ifdef CONFIG_SYS_HUSH_PARSER
    u_boot_hush_start ();
#endif

#if defined(CONFIG_HUSH_INIT_VAR)
    hush_init_var ();
#endif

#ifdef CONFIG_AUTO_COMPLETE
    install_auto_complete(); //安装自动补全的函数，分析如下
#endif

#ifdef CONFIG_PREBOOT
    if ((p = getenv ("preboot")) != NULL) {
# ifdef CONFIG_AUTOBOOT_KEYED
        int prev = disable_ctrlc(1);    /* disable Control C checking */
# endif

# ifndef CONFIG_SYS_HUSH_PARSER
        run_command (p, 0);
# else
        parse_string_outer(p, FLAG_PARSE_SEMICOLON |
                    FLAG_EXIT_FROM_LOOP);
# endif

# ifdef CONFIG_AUTOBOOT_KEYED
        disable_ctrlc(prev);    /* restore Control C checking */
# endif
    }
#endif /* CONFIG_PREBOOT */

#if defined(CONFIG_UPDATE_TFTP)
    update_tftp ();
#endif /* CONFIG_UPDATE_TFTP */

#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;

    debug ("### main_loop entered: bootdelay=%d\n\n", bootdelay);

# ifdef CONFIG_BOOT_RETRY_TIME
    init_cmd_timeout ();
# endif    /* CONFIG_BOOT_RETRY_TIME */

#ifdef CONFIG_POST
    if (gd->flags & GD_FLG_POSTFAIL) {
        s = getenv("failbootcmd");
    }
    else
#endif /* CONFIG_POST */
#ifdef CONFIG_BOOTCOUNT_LIMIT
    if (bootlimit && (bootcount > bootlimit)) {
        printf ("Warning: Bootlimit (%u) exceeded. Using altbootcmd.\n",
                (unsigned)bootlimit);
        s = getenv ("altbootcmd");
    }
    else
#endif /* CONFIG_BOOTCOUNT_LIMIT */
        s = getenv ("bootcmd"); //获取引导命令。分析见下面。
```

```
    printf ("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

    if (bootdelay >= 0 && s && !abortboot (bootdelay)) {
                //如果延时大于等于零，并且没有在延时过程中接收到按键，则引导内核。abortboot函数的分
# ifdef CONFIG_AUTOBOOT_KEYED
        int prev = disable_ctrlc(1);    /* disable Control C checking */
# endif

# ifndef CONFIG_SYS_HUSH_PARSER
        run_command (s, 0);//运行引导内核的命令。这个命令是在配置头文件中定义的。run_command的分
# else
        parse_string_outer(s, FLAG_PARSE_SEMICOLON |
                    FLAG_EXIT_FROM_LOOP);
# endif

# ifdef CONFIG_AUTOBOOT_KEYED
        disable_ctrlc(prev);    /* restore Control C checking */
# endif
    }

# ifdef CONFIG_MENUKEY
    if (menukey == CONFIG_MENUKEY) {
        s = getenv("menucmd");
        if (s) {
# ifndef CONFIG_SYS_HUSH_PARSER
        run_command (s, 0);
# else
        parse_string_outer(s, FLAG_PARSE_SEMICOLON |
                    FLAG_EXIT_FROM_LOOP);
# endif
        }
    }
#endif /* CONFIG_MENUKEY */
#endif    /* CONFIG_BOOTDELAY */

#ifdef CONFIG_AMIGAONEG3SE
    {
        extern void video_banner(void);
        video_banner();
    }
#endif

    /*
     * Main Loop for Monitor Command Processing
     */
#ifdef CONFIG_SYS_HUSH_PARSER
    parse_file_outer();
    /* This point is never reached */
    for (;;);
#else
    for (;;) {
#ifdef CONFIG_BOOT_RETRY_TIME
        if (rc >= 0) {
            /* Saw enough of a valid command to
             * restart the timeout.
             */
            reset_cmd_timeout();
        }
#endif
        len = readline (CONFIG_SYS_PROMPT); //CONFIG_SYS_PROMPT的意思是回显字符，一般是“>”
        flag = 0;    /* assume no special flags for now */
        if (len > 0)
```

```
            strcpy (lastcommand, console_buffer); //保存输入的数据。
        else if (len == 0)
            flag |= CMD_FLAG_REPEAT; ;//如果输入数据为零，则重复执行上次的命令，如果上次输入的是
#ifdef CONFIG_BOOT_RETRY_TIME
        else if (len == -2) {
            /* -2 means timed out, retry autoboot
             */
            puts ("\nTimed out waiting for command\n");
# ifdef CONFIG_RESET_TO_RETRY
            /* Reinit board to run initialization code again */
            do_reset (NULL, 0, 0, NULL);
# else
            return;        /* retry autoboot */
# endif
        }
#endif

        if (len == -1)
            puts ("<INTERRUPT>\n");
        else
            rc = run_command (lastcommand, flag); //执行命令

        if (rc <= 0) {//执行失败，则清空记录
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
#endif /*CONFIG_SYS_HUSH_PARSER*/
}
```

U-Boot启动Linux过程

U-Boot使用标记列表（tagged list）的方式向Linux传递参数。]标记的数据结构式是tag，在U-Boot源代码目录arch/arminclude/asm/setup.h中定义如下：

```
struct tag_header {
       u32 size;       /* 表示tag数据结构的联合u实质存放的数据的大小*/
       u32 tag;        /* 表示标记的类型 */
};

struct tag {
       struct tag_header hdr;
       union {
              struct tag_core           core;
              struct tag_mem32      mem;
              struct tag_videotext   videotext;
              struct tag_ramdisk     ramdisk;
              struct tag_initrd  initrd;
              struct tag_serialnr       serialnr;
              struct tag_revision      revision;
              struct tag_videolfb     videolfb;
              struct tag_cmdline     cmdline;

              /*
               * Acorn specific
               */
              struct tag_acorn  acorn;
              /*
```

```
                * DC21285 specific
                */
               struct tag_memclk      memclk;
        } u;
};
```

U-Boot使用命令bootm来启动已经加载到内存中的内核。而bootm命令实际上调用的是do_bootm函数。

对于Linux内核，do_bootm函数会调用do_bootm_linux函数来设置标记列表和启动内核。

do_bootm_linux函数在arch/arm/bootm.c 中定义如下：

```
int do_bootm_linux(int flag, int argc, char *argv[], bootm_headers_t *images)
{
        bd_t       *bd = gd->bd;
        char       *s;
        int   machid = bd->bi_arch_number;
        void       (*theKernel)(int zero, int arch, uint params);

#ifdef CONFIG_CMDLINE_TAG
        char *commandline = getenv ("bootargs");   /* U-Boot环境变量bootargs */
#endif
        … …
        theKernel = (void (*)(int, int, uint))images->ep; /* 获取内核入口地址 */
        … …
#if   defined (CONFIG_SETUP_MEMORY_TAGS) || \
        defined (CONFIG_CMDLINE_TAG) || \
        defined (CONFIG_INITRD_TAG) || \
        defined (CONFIG_SERIAL_TAG) || \
        defined (CONFIG_REVISION_TAG) || \
        defined (CONFIG_LCD) || \
        defined (CONFIG_VFD)
        setup_start_tag (bd);                                        /* 设置ATAG_CORE标志 *
        … …
#ifdef CONFIG_SETUP_MEMORY_TAGS
       setup_memory_tags (bd);                                 /* 设置内存标记 */
#endif
 #ifdef CONFIG_CMDLINE_TAG
       setup_commandline_tag (bd, commandline);       /* 设置命令行标记 */
#endif
        … …
       setup_end_tag (bd);                                    /* 设置ATAG_NONE标志 */
#endif
       /* we assume that the kernel is in place */
       printf ("\nStarting kernel ...\n\n");
        … …
       cleanup_before_linux ();          /* 启动内核前对CPU作最后的设置 */

       theKernel (0, machid, bd->bi_boot_params);       /* 调用内核 */

       /* does not return */
       return 1;
}
```

其中的setup_start_tag，setup_memory_tags，setup_end_tag函数在lib_arm/bootm.c中定义如下：

（1）setup_start_tag函数

```
static void setup_start_tag (bd_t *bd)
{
       params = (struct tag *) bd->bi_boot_params;  /* 内核的参数的开始地址 */

       params->hdr.tag = ATAG_CORE;
       params->hdr.size = tag_size (tag_core);

       params->u.core.flags = 0;
       params->u.core.pagesize = 0;
       params->u.core.rootdev = 0;

       params = tag_next (params);
}
```

标记列表必须以ATAG_CORE开始，setup_start_tag函数在内核的参数的开始地址设置了一个ATAG_CORE标记。

（2）setup_memory_tags函数

```
static void setup_memory_tags (bd_t *bd)
{
       int i;
       /*设置一个内存标记 */
       for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
              params->hdr.tag = ATAG_MEM;
              params->hdr.size = tag_size (tag_mem32);

              params->u.mem.start = bd->bi_dram[i].start;
              params->u.mem.size = bd->bi_dram[i].size;

              params = tag_next (params);
       }
}
```

setup_memory_tags函数设置了一个ATAG_MEM标记，该标记包含内存起始地址，内存大小这两个参数。

（3）setup_end_tag函数

```
static void setup_end_tag (bd_t *bd)
{
       params->hdr.tag = ATAG_NONE;
       params->hdr.size = 0;
}
```

标记列表必须以标记ATAG_NONE结束，setup_end_tag函数设置了一个ATAG_NONE标记，表示标记列表的结束。

U-Boot设置好标记列表后就要调用内核了。但调用内核前，CPU必须满足下面的条件：

（1） CPU寄存器的设置

Ø r0=0

Ø r1=机器码

Ø r2=内核参数标记列表在RAM中的起始地址

（2） CPU工作模式

Ø 禁止IRQ与FIQ中断

Ø CPU为SVC模式

（3） 使数据Cache与指令Cache失效

do_bootm_linux中调用的cleanup_before_linux函数完成了禁止中断和使Cache失效的功能。

cleanup_before_linux函数在cpu/arm920t/cpu.中定义：

```
int cleanup_before_linux (void)
{
        /*
         * this function is called just before we call linux
         * it prepares the processor for linux
         *
         * we turn off caches etc ...
         */
        disable_interrupts ();          /* 禁止FIQ/IRQ中断 */

        /* turn off I/D-cache */
        icache_disable();                /* 使指令Cache失效 */
        dcache_disable();               /* 使数据Cache失效 */
        /* flush I/D-cache */
        cache_flush();                     /* 刷新Cache */

        return 0;
}
```

由于U-Boot启动以来就一直工作在SVC模式，因此CPU的工作模式就无需设置了。

代码将内核的入口地址“images->ep”强制类型转换为函数指针。

根据ATPCS规则，函数的参数个数不超过4个时，使用r0~r3这4个寄存器来传递参数。

因此第128行的函数调用则会将0放入r0，机器码machid放入r1，内核参数地址bd->bi_boot_params放入r2，从而完成了寄存器的设置，最后转到内核的入口地址。

到这里，U-Boot的工作就结束了，系统跳转到Linux内核代码执行。

do_bootm_linux中：

```
void       (*theKernel)(int zero, int arch, uint params);
    … …
    theKernel = (void (*)(int, int, uint))images->ep;
    … …
    theKernel (0, machid, bd->bi_boot_params);




int do_bootm_linux(int flag, int argc, char * const argv[],
           bootm_headers_t *images)
{
    /* No need for those on ARM */
    if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
        return -1;

        // 当flag为BOOTM_STATE_OS_PREP，则说明只需要做准备动作boot_prep_linux
    if (flag & BOOTM_STATE_OS_PREP) {
        boot_prep_linux(images);
        return 0;
    }

        // 当flag为BOOTM_STATE_OS_GO ，则说明只需要做跳转动作
        if (flag & (BOOTM_STATE_OS_GO | BOOTM_STATE_OS_FAKE_GO)) {
        boot_jump_linux(images, flag);
        return 0;
    }

    boot_prep_linux(images); // 以全局变量bootm_headers_t images为参数传递给boot_prep_linu
    boot_jump_linux(images, flag);// 以全局变量bootm_headers_t images为参数传递给boot_jump
    return 0;
}
```

boot_prep_linux用于实现跳转到linux前的准备动作。而boot_jump_linux用于跳转到linux中。都是以全局变量bootm_headers_t images为参数，这样就可以直接获取到前面步骤中得到的kernel镜像、ramdisk以及fdt的信息了。

- boot_prep_linux首先要说明一下LMB的概念。LMB是指logical memory blocks，主要是用于表示内存的保留区域，主要有fdt的区域，ramdisk的区域等等。boot_prep_linux主要的目的是修正LMB，并把LMB填入到fdt中。

实现如下：

```
static void boot_prep_linux(bootm_headers_t *images)
{
    char *commandline = getenv("bootargs");

    if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len) {
#ifdef CONFIG_OF_LIBFDT
        debug("using: FDT\n");
        if (image_setup_linux(images)) {
            printf("FDT creation failed! hanging...");
            hang();
        }
#endif
    }
```

- boot_jump_linux以arm为例：arch/arm/lib/bootm.c

```
static void boot_jump_linux(bootm_headers_t *images, int flag)
{
    unsigned long machid = gd->bd->bi_arch_number; // 从bd中获取machine-id，machine-id在
    char *s;
    void (*kernel_entry)(int zero, int arch, uint params); // kernel入口函数，也就是kern
    unsigned long r2;
    int fake = (flag & BOOTM_STATE_OS_FAKE_GO); // 伪跳转，并不真正地跳转到kernel中

    kernel_entry = (void (*)(int, int, uint))images->ep;
         // 将kernel_entry设置为images中的ep（kernel的入口地址），后面直接执行kernel_entry也就
        // 这里要注意这种跳转的方法

    debug("## Transferring control to Linux (at address %08lx)" \
        "...\n", (ulong) kernel_entry);
    bootstage_mark(BOOTSTAGE_ID_RUN_OS);
    announce_and_cleanup(fake);

        // 把images->ft_addr（fdt的地址）放在r2寄存器中
    if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
        r2 = (unsigned long)images->ft_addr;
    else
        r2 = gd->bd->bi_boot_params;

    if (!fake) {
             kernel_entry(0, machid, r2);
        // 这里通过调用kernel_entry，就跳转到了images->ep中了，也就是跳转到kernel中了，具体则是k
        // 参数0则传入到r0寄存器中，参数machid传入到r1寄存器中，把images->ft_addr（fdt的地址）放
        // 满足了kernel启动的硬件要求
    }
}
```

Linux内核启动过程

Linux 的启动过程可以分为两部分:

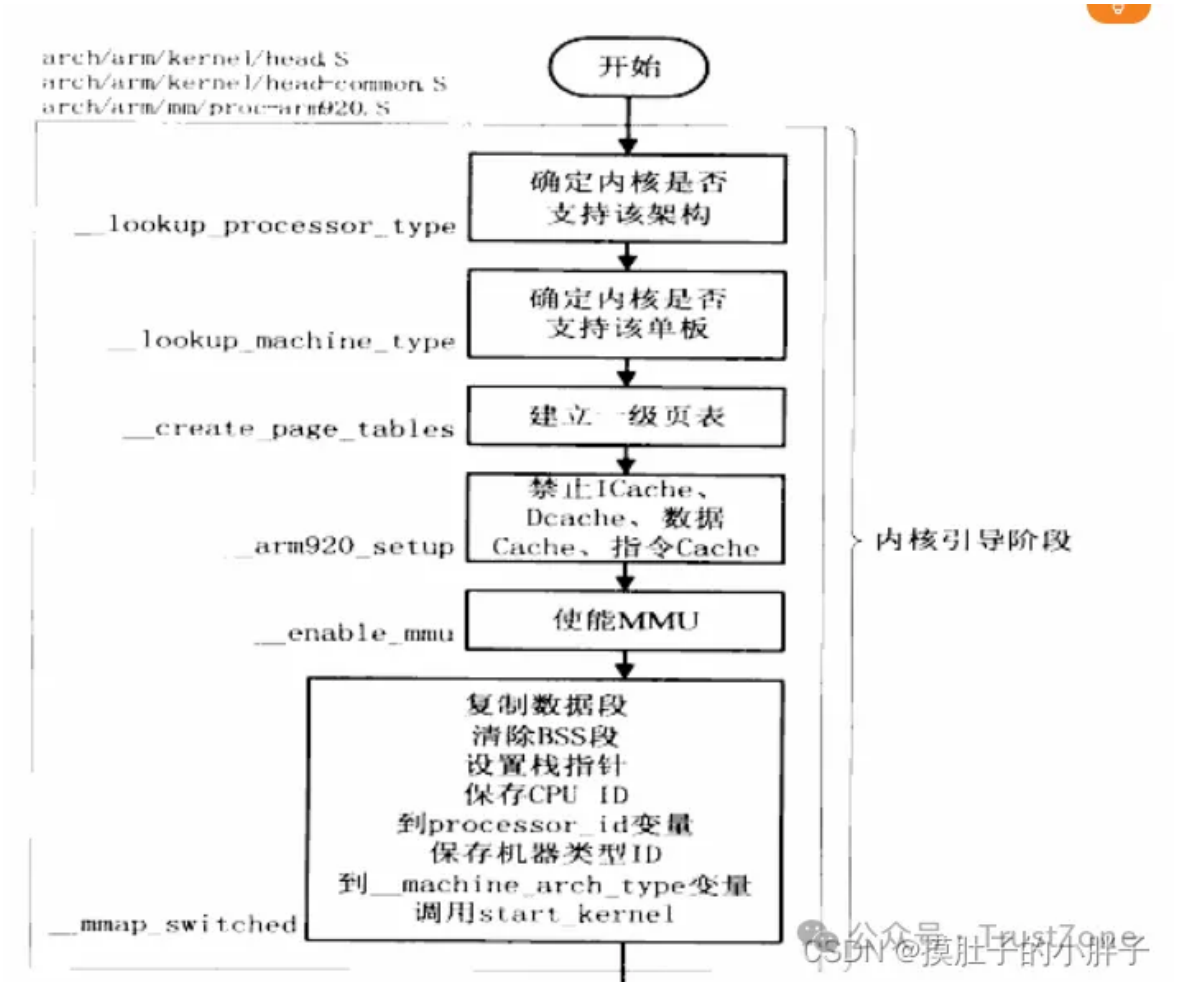
第一阶段：引导阶段通常使用汇编语言编写：

- 首先检查内核是否支持当前架构的处理器，
- 然后检查是否支持当前开发板。通过检查后，就为调用下一阶段的start_kernel 函数作准备了。这主要分如下两个步骤。
  - 连接内核时使用的是虚拟地址，所以要设置页表、使能MMU
  - 做一些调用C函数 start_kernel之前的常规工作，包括复制数据段、清除BSS段、调用start_kernel函数。

第二阶段第二阶段的关键代码主要使用C语言编写。

它进行内核初始化的全部工作， 最后调用**rest_init函数启动 init过程，创建系统第一个进程: init进程。**

在第二阶段，仍有部分架构/开发板相关的代码。



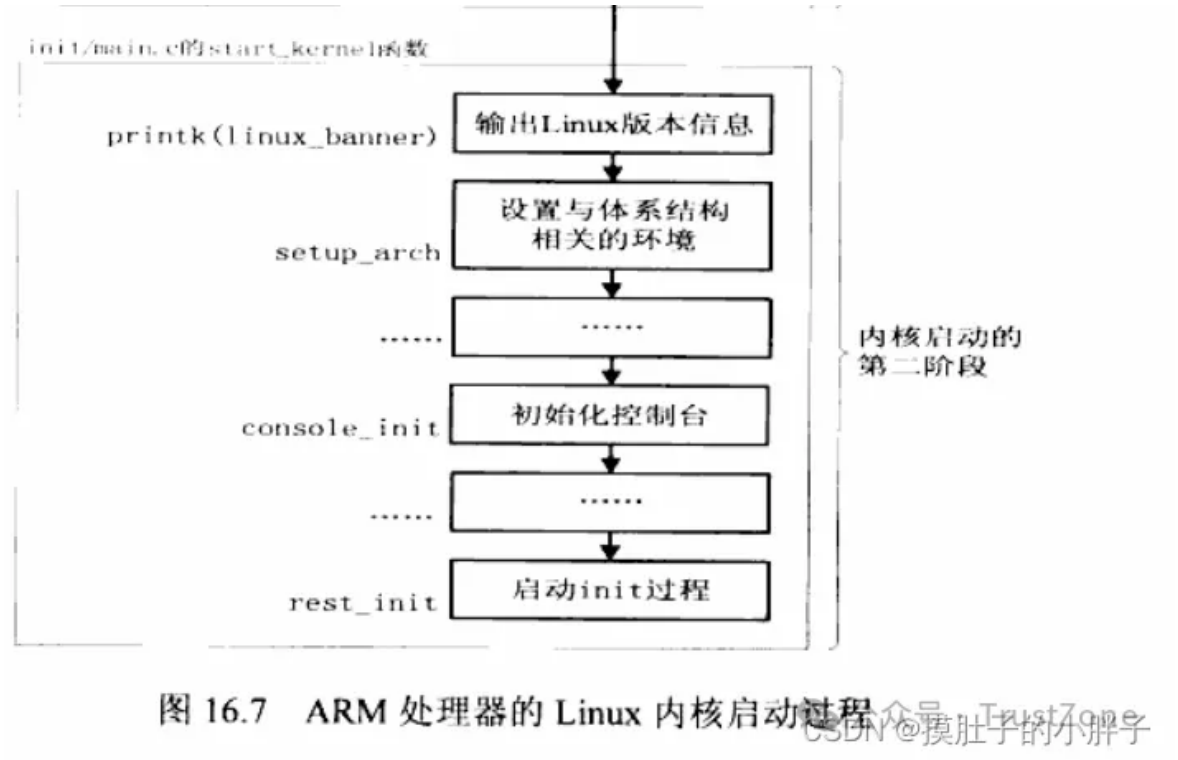
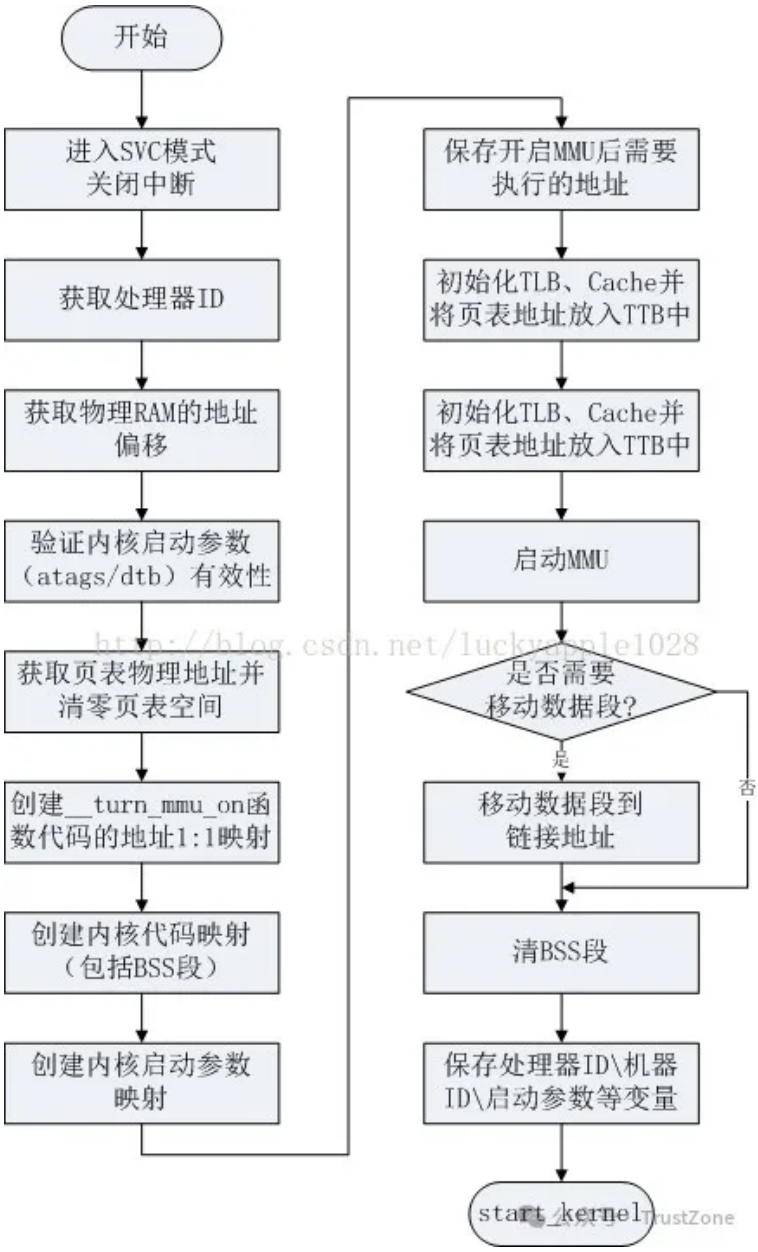


图 16.7 ARM 处理器的 Linux 内核启动过程

关注公众号：TrustZone
CSDN @摸肚子的小胖子

在内核启动时执行自解压完成后，会跳转到解压后的地址处运行，在我的环境中就是地址0x00008000处，然后内核启动并执行初始化。

首先给出你内核启动的汇编部分的总流程如下：



内核启动程序的入口：参见arch/arm/kernel/vmlinux.lds（由arch/arm/kernel/vmlinux.lds.S生成）。

arch/arm/kernel/vmlinux.lds：

```
ENTRY(stext)
jiffies = jiffies_64;
SECTIONS
{
......
. = 0xC0000000 + 0x00008000;
.head.text : {
_text = .;
*(.head.text)
}
.text : { /* Real text segment */
_stext = .; /* Text and read-only data */
```

此处的TEXT_OFFSET表示内核起始地址相对于RAM地址的偏移值，定义在arch/arm/Makefile中，值为0x00008000：

```
textofs-y := 0x00008000
......
# The byte offset of the kernel image in RAM from the start of RAM.
TEXT_OFFSET := $(textofs-y)
```

PAGE_OFFSET表示内核虚拟地址空间的起始地址，定义在arch/arm/include/asm/memory.h中：

```
#ifdef CONFIG_MMU

/*
* PAGE_OFFSET - the virtual address of the start of the kernel image
* TASK_SIZE - the maximum size of a user space task.
* TASK_UNMAPPED_BASE - the lower boundary of the mmap VM area
*/
#define PAGE_OFFSET UL(CONFIG_PAGE_OFFSET)
```

CONFIG_PAGE_OFFSET定义在arch/arm/Kconfig中，采用默认值0xC0000000。

```
config PAGE_OFFSET
hex
default 0x40000000 if VMSPLIT_1G
default 0x80000000 if VMSPLIT_2G
default 0xC0000000
```

所以，可以看出**内核的链接地址采用的是虚拟地址，地址值为0xC0008000。**

内核启动程序的入口在linux/arch/arm/kernel/head.S中，head.S中定义了几个比较重要的变量，在看分析程序前先来看一下：

```
/*
* swapper_pg_dir is the virtual address of the initial page table.
* We place the page tables 16K below KERNEL_RAM_VADDR. Therefore, we must
* make sure that KERNEL_RAM_VADDR is correctly set. Currently, we expect
* the least significant 16 bits to be 0x8000, but we could probably
* relax this restriction to KERNEL_RAM_VADDR >= PAGE_OFFSET + 0x4000.
*/
#define KERNEL_RAM_VADDR (PAGE_OFFSET + TEXT_OFFSET)
#if (KERNEL_RAM_VADDR & 0xffff) != 0x8000
#error KERNEL_RAM_VADDR must start at 0xXXXX8000
#endif

#ifdef CONFIG_ARM_LPAE
/* LPAE requires an additional page for the PGD */
#define PG_DIR_SIZE 0x5000
#define PMD_ORDER 3
#else
#define PG_DIR_SIZE 0x4000
#define PMD_ORDER 2
#endif

.globl swapper_pg_dir
.equ swapper_pg_dir, KERNEL_RAM_VADDR - PG_DIR_SIZE

.macro pgtbl, rd, phys
```

```
add \rd, \phys, #TEXT_OFFSET - PG_DIR_SIZE
.endm
```

其中KERNEL_RAM_VADDR表示内核启动地址的虚拟地址，即前面看到的链接地址0xC0008000，同时内核要求这个地址的第16位必须是0x8000。

然后由于没有配置ARM LPAE，则采用一级映射结构，页表的大小为16KB，页大小为1MB。

最后swapper_pg_dir表示初始页表的起始地址，这个值等于内核起始虚拟地址-页表大小=0xC0004000（内核起始地址下16KB空间存放页表）。虚拟地址空间如下图：

需要说明一下：在我的环境中，内核在自解压阶段被解压到了0x00008000地址处，由于内核入口链接地址采用的是虚拟地址0xC0008000，这两个地址并不相同；并且此时MMU并没有被使能，所以无法进行虚拟地址到物理地址的转换，程序开始执行后在打开MMU前的将使用位置无关码。

在知道了内核的入口位置后，来看一下此时的设备和寄存器的状态：

```
/*
* Kernel startup entry point.
* ---------------------------
*
* This is normally called from the decompressor code. The requirements
* are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
* r1 = machine nr, r2 = atags or dtb pointer.
*
* This code is mostly position independent, so if you link the kernel at
* 0xc0008000, you call this at __pa(0xc0008000).
*
* See linux/arch/arm/tools/mach-types for the complete list of machine
* numbers for r1.
*
* We're trying to keep crap to a minimum; DO NOT add any machine specific
* crap here - that's what the boot loader (or in extreme, well justified
* circumstances, zImage) is for.
*/
.arm

__HEAD
ENTRY(stext)
```

注释中说明了，此时的MMU关闭、D-cache关闭、r0 = 0、r1 = 机器码、r2 = 启动参数atags或dtb的地址（我的环境中使用的是atags），同时内核支持的机器码被定义在了linux/arch/arm/tools/mach-types中。

4. U-Boot添加命令的方法及U-Boot命令执行过程

下面以添加menu命令（启动菜单）为例讲解U-Boot添加命令的方法。

（1） 建立common/cmd_menu.c

习惯上通用命令源代码放在common目录下，与开发板专有命令源代码则放在board/<board_dir>目录下，并且习惯以“cmd_<命令名>.c”为文件名。

（2）定义“menu”命令

在cmd_menu.c中使用如下的代码定义“menu”命令：

```
_BOOT_CMD(
       menu,    3,    0,    do_menu,
       "menu - display a menu, to select the items to do something\n",
       " - display a menu, to select the items to do something"
);
```

其中U_BOOT_CMD命令格式如下：

```
U_BOOT_CMD(name,maxargs,rep,cmd,usage,help)
```

各个参数的意义如下：

- 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）

> 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/ti8168_dvr.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。

> U_BOOT_CMD宏在include/command.h中定义：

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
       cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage
```

“##”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。

其中的cmd_tbl_t在include/command.h中定义如下：

```
struct cmd_tbl_s {
       char       *name;          /* 命令名 */
       int          maxargs;       /* 最大参数个数 */
       int          repeatable;    /* 是否自动重复 */
       int          (*cmd)(struct cmd_tbl_s *, int, int, char *[]);  /*  响应函数 */
       char       *usage;         /* 简短的帮助信息 */
#ifdef    CONFIG_SYS_LONGHELP
       char              *help;           /*  较详细的帮助信息 */
#endif
#ifdef CONFIG_AUTO_COMPLETE
       /* 自动补全参数 */
       int          (*complete)(int argc, char *argv[], char last_char, int maxv, cha
#endif
};

typedef struct cmd_tbl_s  cmd_tbl_t;
```

一个cmd_tbl_t结构体变量包含了调用一条命令的所需要的信息。

其中Struct_Section在include/command.h中定义如下：

```
#define Struct_Section  __attribute__ ((unused,section (".u_boot_cmd")))
```

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：

```
. = .;
__u_boot_cmd_start = .;          /*将 __u_boot_cmd_start指定为当前地址 */
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;           /*  将__u_boot_cmd_end指定为当前地址  */
```

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。

因此“menu”命令的定义经过宏展开后如下：

cmd_tbl_t __u_boot_cmd_menu **attribute** ((unused,section (".u_boot_cmd"))) = {menu, 3, 0, do_menu, "menu - display a menu, to select the items to do something\n", " - display a menu, to select the items to do something"}

实质上就是用U_BOOT_CMD宏定义的信息构造了一个cmd_tbl_t类型的结构体。编译器将该结构体放在“u_boot_cmd”段，执行命令时就可以在“u_boot_cmd”段查找到对应的cmd_tbl_t类型结构体。

（3）实现命令的函数

在cmd_menu.c中添加“menu”命令的响应函数的实现。具体的实现代码略：

```
int do_menu (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
       /* 实现代码略 */
}
```

（4）将common/cmd_menu.c编译进u-boot.bin

在common/Makefile中加入如下代码：

```
COBJS-$(CONFIG_BOOT_MENU) += cmd_menu.o
```

在include/configs/ti8168_dvr.h加入如代码：

```
#define CONFIG_BOOT_MENU 1
```

重新编译下载U-Boot就可以使用menu命令了

（5）menu命令执行的过程

在U-Boot中输入“menu”命令执行时，U-Boot接收输入的字符串“menu”，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_cmd_start与__u_boot_cmd_end间查找命令，并返回menu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menu命令的响应函数do_menu，从而完成了命令的执行。

（6）例子：USB下载，命令很简单。

```
#include <common.h>
#include <command.h>
extern char console_buffer[];
extern int readline (const char *const prompt);
extern char awaitkey(unsigned long delay, int* error_p);
extern void download_nkbin_to_flash(void);

 /**
 * Parses a string into a number.  The number stored at ptr is
 * potentially suffixed with K (for kilobytes, or 1024 bytes),
 * M (for megabytes, or 1048576 bytes), or G (for gigabytes, or
 * 1073741824).  If the number is suffixed with K, M, or G, then
 * the return value is the number multiplied by one kilobyte, one
 * megabyte, or one gigabyte, respectively.
 *
 * @param ptr where parse begins
 * @param retptr output pointer to next char after parse completes (output)
 * @return resulting unsigned int
 */

static unsigned long memsize_parse2 (const char *const ptr, const char **retptr)
{
      unsigned long ret = simple_strtoul(ptr, (char **)retptr, 0);
      int sixteen = 1;
      switch (**retptr) {
      case 'G':
      case 'g':
          ret <<= 10;
      case 'M':
      case 'm':
          ret <<= 10;
      case 'K':
      case 'k':
          ret <<= 10;
          (*retptr)++;
          sixteen = 0;
      default:
          break;
      }
      if (sixteen)
          return simple_strtoul(ptr, NULL, 16);

      return ret;
}

void param_menu_usage()
{
     printf("\r\n##### Parameter Menu #####\r\n");
     printf("[v] View the parameters\r\n");
     printf("[s] Set parameter \r\n");
     printf("[d] Delete parameter \r\n");
     printf("[w] Write the parameters to flash memeory \r\n");
     printf("[q] Quit \r\n");
     printf("Enter your selection: ");
}

void param_menu_shell(void)
{
     char c;
     char cmd_buf[256];
     char name_buf[20];
     char val_buf[256];
```

```
       while (1)
      {
           param_menu_usage();
          c = awaitkey(-1, NULL);
          printf("%c\n", c);
          switch (c)
          {
              case 'v':
              {
                  strcpy(cmd_buf, "printenv ");
                  printf("Name(enter to view all paramters): ");
                  readline(NULL);
                  strcat(cmd_buf, console_buffer);
                  run_command(cmd_buf, 0);
                  break;
              }

               case 's':
              {
                  sprintf(cmd_buf, "setenv ");
                  printf("Name: ");
                  readline(NULL);
                  strcat(cmd_buf, console_buffer);
                  printf("Value: ");
                  readline(NULL);
                  strcat(cmd_buf, " ");
                  strcat(cmd_buf, console_buffer);
                  run_command(cmd_buf, 0);
                  break;
              }

               case 'd':
              {
                  sprintf(cmd_buf, "setenv ");
                  printf("Name: ");
                  readline(NULL);
                  strcat(cmd_buf, console_buffer);
                  run_command(cmd_buf, 0);
                  break;
              }

               case 'w':
              {
                  sprintf(cmd_buf, "saveenv");
                  run_command(cmd_buf, 0);
                  break;
              }

               case 'q':
              {
                  return;
                  break;
              }
          }
      }
}

void main_menu_usage(void)
{
      printf("\r\n##### 100ask Bootloader for OpenJTAG #####\r\n");
      printf("[n] Download u-boot to Nand Flash\r\n");
```

```
     if (bBootFrmNORFlash())
          printf("[o] Download u-boot to Nor Flash\r\n");
     printf("[k] Download Linux kernel uImage\r\n");
     printf("[j] Download root_jffs2 image\r\n");
    //    printf("[c] Download root_cramfs image\r\n");

     printf("[y] Download root_yaffs image\r\n");
     printf("[d] Download to SDRAM & Run\r\n");
     printf("[z] Download zImage into RAM\r\n");
     printf("[g] Boot linux from RAM\r\n");
     printf("[f] Format the Nand Flash\r\n");
     printf("[s] Set the boot parameters\r\n");
     printf("[b] Boot the system\r\n");
     printf("[r] Reboot u-boot\r\n");
     printf("[q] Quit from menu\r\n");
     printf("Enter your selection: ");
}

void menu_shell(void)
{
     char c;
     char cmd_buf[200];
     char *p = NULL;
     unsigned long size;
     unsigned long offset;
     struct mtd_info *mtd = &nand_info[nand_curr_device];

     while (1)
     {
         main_menu_usage();
         c = awaitkey(-1, NULL);
         printf("%c\n", c);

         switch (c)
         {
            case 'n':
            {
                 strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase bootloader; nand
                 run_command(cmd_buf, 0);
                 break;
            }
            case 'o':
            {
                if (bBootFrmNORFlash())
                {
                     strcpy(cmd_buf, "usbslave 1 0x30000000; protect off all; erase 0
                     run_command(cmd_buf, 0);
                 }
                 break;
             }

            case 'k':
             {
                 strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase kernel; nand writ
                 run_command(cmd_buf, 0);
                 break;
             }
             case 'j':
             {
                 strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase root; nand write.
                 run_command(cmd_buf, 0);
                 break;
```

```
            }
#if 0
            case 'c':
            {
                strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase root; nand write.
                run_command(cmd_buf, 0);
                break;
            }
#endif
            case 'y':
            {
                strcpy(cmd_buf, "usbslave 1 0x30000000; nand erase root; nand write.
                run_command(cmd_buf, 0);
                break;
            }
            case 'd':
            {
                extern volatile U32 downloadAddress;
                extern int download_run;

                download_run = 1;
                strcpy(cmd_buf, "usbslave 1");
                run_command(cmd_buf, 0);

                download_run = 0;
                sprintf(cmd_buf, "go %x", downloadAddress);
                run_command(cmd_buf, 0);
                break;
            }
            case 'z':
            {
                strcpy(cmd_buf, "usbslave 1 0x30008000");
                run_command(cmd_buf, 0);
                break;
            }
            case 'g':
            {
                extern void do_bootm_rawLinux (ulong addr);
                do_bootm_rawLinux(0x30008000);
            }
            case 'b':
            {
                printf("Booting Linux ...\n");
                strcpy(cmd_buf, "nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0
                run_command(cmd_buf, 0);
                break;
            }
            case 'f':
            {
                strcpy(cmd_buf, "nand erase ");
                printf("Start address: ");
                readline(NULL);
                strcat(cmd_buf, console_buffer);
                printf("Size(eg. 4000000, 0x4000000, 64m and so on): ");
                readline(NULL);
                p = console_buffer;
                size = memsize_parse2(p, &p);
                sprintf(console_buffer, " %x", size);
                strcat(cmd_buf, console_buffer);
                run_command(cmd_buf, 0);
                break;
            }
```

```
            case 's':
            {
                param_menu_shell();
                break;
            }
            case 'r':
            {
                strcpy(cmd_buf, "reset");
                run_command(cmd_buf, 0);
                break;
            }

             case 'q':
            {
                return;
                 break;
            }
         }

      }
}

int do_menu (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
     menu_shell();
     return 0;
}

U_BOOT_CMD(
      menu, 3, 0, do_menu,
      "menu - display a menu, to select the items to do something\n",
      " - display a menu, to select the items to do something"
);
```

TFTP下载

```
#include <common.h>
#include <command.h>

/**功能：等待键盘输入***/
static char awaitkey(unsigned long delay, int* error_p)
{
     int i;
     char c;
     if (delay == -1) {
         while (1) {
             if (tstc()) /* we got a key press */
                 return getc();
         }
     }
     else {
          for (i = 0; i < delay; i++) {
              if (tstc()) /* we got a key press */
                  return getc();
              udelay (10*1000);
         }
     }
     if (error_p)
        *error_p = -1;
     return 0;
}

/*****提示符，功能说明****/
void main_menu_usage(void)
{
   printf("\r\n######## Hotips TFTP DownLoad for SMDK2440 ########\r\n");
   printf("\r\n");
   printf("[1] 下载 u-boot.bin       写入 Nand Flash\r\n");
   printf("[2] 下载 Linux(uImage)    内核镜像写入 Nand Flash\r\n");
   printf("[3] 下载 yaffs2(fs.yaffs) 文件系统镜像写入 Nand Flash\r\n");
   printf("[4] 下载 Linux(uImage)    内核镜像到内存并运行\r\n");
   printf("[5] 重启设备\r\n");
   printf("[q] 退出菜单\r\n");
   printf("\r\n");
   printf("输入选择: ");
}

/***do_menu()的调用函数，命令的具体实现***/
void menu_shell(void)
{
     char c;
     char cmd_buf[200];
     while (1)
     {
       main_menu_usage();
       c = awaitkey(-1, NULL);
       printf("%c\n", c);
       switch (c)
       {
         case '1':
         {
           strcpy(cmd_buf, "tftp 0x32000000 u-boot.bin; nand erase 0x0 0x60000; nand
           run_command(cmd_buf, 0);
           break;
         }
         case '2':
         {
           strcpy(cmd_buf, "tftp 0x32000000 uImage; nand erase 0x80000 0x200000; nand
```

```
            run_command(cmd_buf, 0);
            break;
          }
          case '3':
          {
            strcpy(cmd_buf, "tftp 0x32000000 fs.yaffs; nand erase 0x280000; nand write
            run_command(cmd_buf, 0);
            break;
          }
          case '4':
          {
            strcpy(cmd_buf, "tftp 0x32000000 uImage; bootm 0x32000000");
            run_command(cmd_buf, 0);
            break;
          }
          case '5':
          {
            strcpy(cmd_buf, "reset");
            run_command(cmd_buf, 0);
            break;
          }
          case 'q':
          {
            return;
             break;
          }
        }
      }
}

int do_menu (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
      menu_shell();
      return 0;
}

U_BOOT_CMD(
   menu, 1, 0, do_menu,
   "Download Menu",
   "U-boot Download Menu by Hotips\n"
);
```

对比两种下载方式我们清楚命令的添加和执行方式了

参考资料

- https://www.cnblogs.com/sky-heaven/p/13825134.html
- https://www.cnblogs.com/cslunatic/p/3655970.html
- https://www.cnblogs.com/cslunatic/archive/2013/05/11/3072811.html
- https://www.cnblogs.com/cslunatic/archive/2013/04/01/2992717.html
- 《ARM Linux内核源码剖析》
- 《ARM11 数据手册》
- https://blog.csdn.net/weixin_45264425/article/details/128090150
- http://blog.chinaunix.net/uid-20799298-id-99666.html

- https://blog.csdn.net/ooonebook/article/details/53495021
- https://blog.csdn.net/ooonebook/article/details/53070065#t15
- https://blog.csdn.net/weixin_45264425/article/details/125951565

推荐阅读 点击标题可跳转

1、微软正式开源专为Windows打造的Sudo

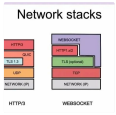
2、端口号都说不明白，别说自己懂网络！

3、Linux 性能调优工具的 9 张图

喜欢此内容的人还喜欢

全网最全网络基础思维导图（38张）

Linux爱好者



100 个网络基础知识

Linux爱好者

你管这破玩意叫网络

Linux爱好者