U-Boot 移植手册

移植版本：U-Boot-1.2.0

移植时间：2009.12.06

ARM 技术交流网荣誉出品

首发网站：www.arm79.com

写在前面的话

为配合 ARM 技术交流网推出的 ARM 培训系列视频，我们编写了此教程。本文档从 12 月 6 日一直写到 12 月 13 日，每天晚上下班写 2 小时，终于写完。它系统地介绍了整个 u-boot 移植的过程：从 u-boot 起源开始，到代码的获取，到系统启动工作分析，再到 flash 驱动及其他相关驱动分析，最后到 u-boot 移植成功，进入 u-boot 命令行界面。

本移植手册面向的对象，是对 ARM 体系结构初步了解，并有一定的汇编基础的初学者，以及 ARM 的 fans。毫不夸张地说，本教程可以使得初学者从一无所知，直至完全掌握 u-boot 移植技术，至此，您真正地入门了！

然而，本移植手册希望达到的目的，不仅仅是 ARM 的入门。我们希望借助 u-boot 移植的过程，让大家对系统启动有一个清晰的认识，甚至能自己写出一个 boot 代码来进行项目设计。同时，我们也希望大家能对项目开发使用的 C 语言，有更深入的掌握和认识。因为在 u-boot 源代码中，很多思路是我们原先编写小程序中无法练习的。

——END——

☆翔子★　于陋室

2009.12.06

目　录

1. u-boot介绍

1.1 U-Boot的来源

U-Boot，全称 Universal Boot Loader，是遵循 GPL 条款的开放源码项目。它最早是由 DENX 软件工程中心的 Wolfgang Denk 基于 8xxrom 的源码创建的 PPCBOOT 工程，并且不断添加处理器的支持。后来，Sysgo Gmbh 把 ppcboot 移植到 ARM 平台上，创建了 ARMboot 工程。然后以 ppcboot 工程和 armboot 工程为基础，创建了 U-Boot 工程。

现在，U-Boot 作为通用的 Bootloader，已经成功地移植到包括 PowerPC、ARM、X86、MIPS 体系结构的上百种开发板，已经成为功能最多、灵活性最强并且开发最积极的开放源码 Bootloader。目前仍然由 DENX 的 Wolfgang Denk 维护。

U-boot 对我们来说，最大的优势就是，它加快了项目开发的进度，节省了工程师的时间。

1.2 U-Boot在系统中的地位

对于一个项目系统来说，u-boot 到底处于什么样地位呢？

目前，在专用的嵌入式板子上运行 GNU/Linux 系统，应该算比较流行的。对于一个嵌入式 Linux 系统，从软件的角度看通常可以分为四个层次：

1、引导加载程序。包括固化在固件(firmware)中的 boot 代码(可选)，和 BootLoader 两大部分。

2、Linux 内核。特定于嵌入式板子的定制内核以及内核的启动参数。

3、文件系统。包括根文件系统和建立于 Flash 内存设备之上文件系统。通常用 ramdisk 来作为 rootfs。

4、用户应用程序。特定于用户的应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有：MicroWindows 和 MiniGUI 等。

按照这个层次，U-Boot 处于引导加载程序阶段。引导加载程序是系统加电后运行的第一段软件代码。PC 机中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OS BootLoader（比如，LILO 和 GRUB 等）一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 BootLoader 读到系统的 RAM 中，然后将控制权交给 OS BootLoader。

BootLoader 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，也即开始启动操作系统。

而在嵌入式系统中，通常并没有像 BIOS，因此整个系统的加载启动任务就完全由 BootLoader 来完成。简单地说，BootLoader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间

的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

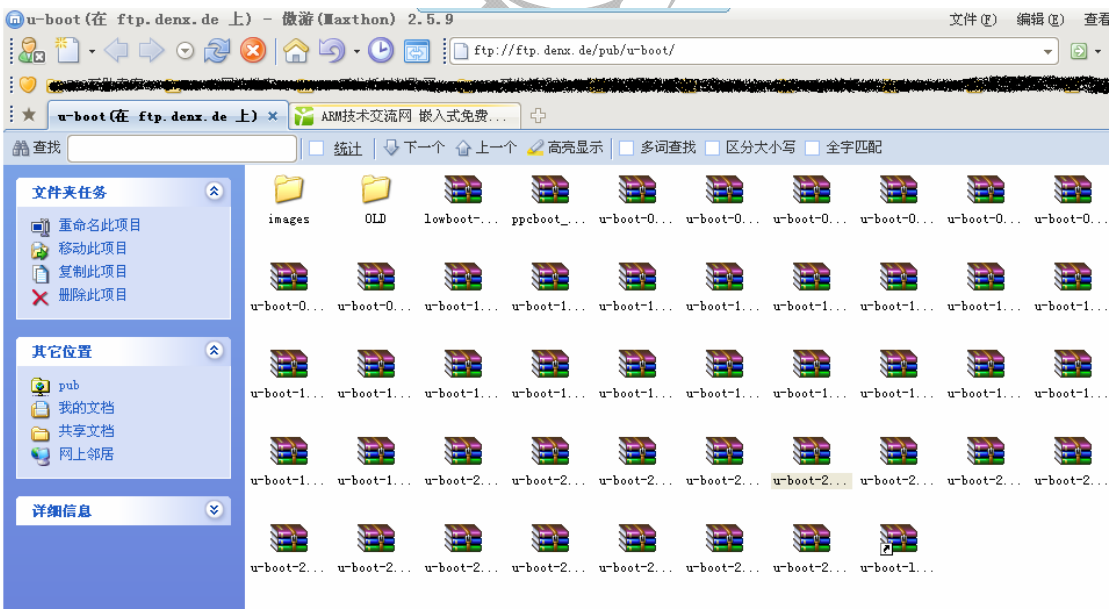
通常，BootLoader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 BootLoader 几乎是不可能的。尽管如此，我们仍然可以对 BootLoader 归纳出一些通用的概念来，以指导用户特定的 BootLoader 设计与实现。U-Boot 适应该“通用性”，成为较为流行的 bootloader。

2. U-Boot移植准备工作

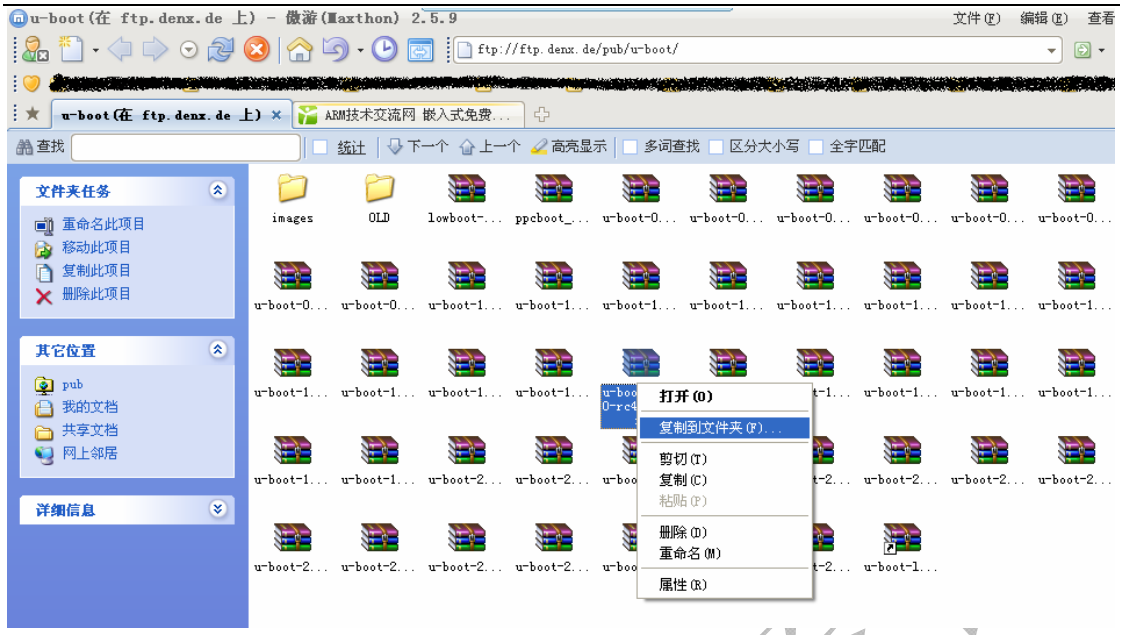
2.1 U-Boot源码的获取

U-Boot 的源码包可以从 sourceforge 网站下载，还可以订阅该网站活跃的 U-Boot Users 邮件论坛，这个邮件论坛对于 U-Boot 的开发和使用都很有帮助。

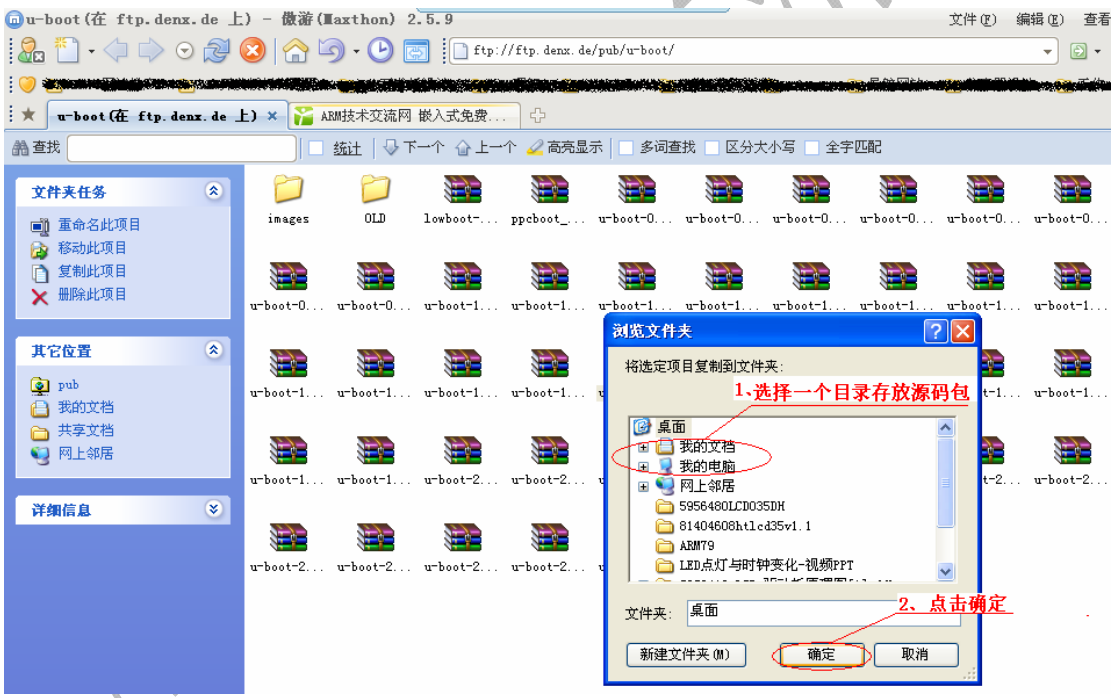
U-Boot软件包下载网站：http://sourceforge.net/project/u-boot。事实上，笔者更喜欢从这里下载源码：ftp://ftp.denx.de/pub/u-boot/ 。因为这里是一个FTP服务器，上面网址打开后，直接就是以文件夹中存放源码包的形式出现，我们选好相应的版本包后，鼠标点击右键，选择“复制到文件夹...”选项，即可下载至您的电脑里的任何位置，如下图所示：



鼠标点击右键，选择“复制到文件夹...”选项，如下图所示：



这时候会打开一个“浏览文件夹”的对话框，您可以选择一个目录存放源码包，然后点击“确定”，即可下载源码。

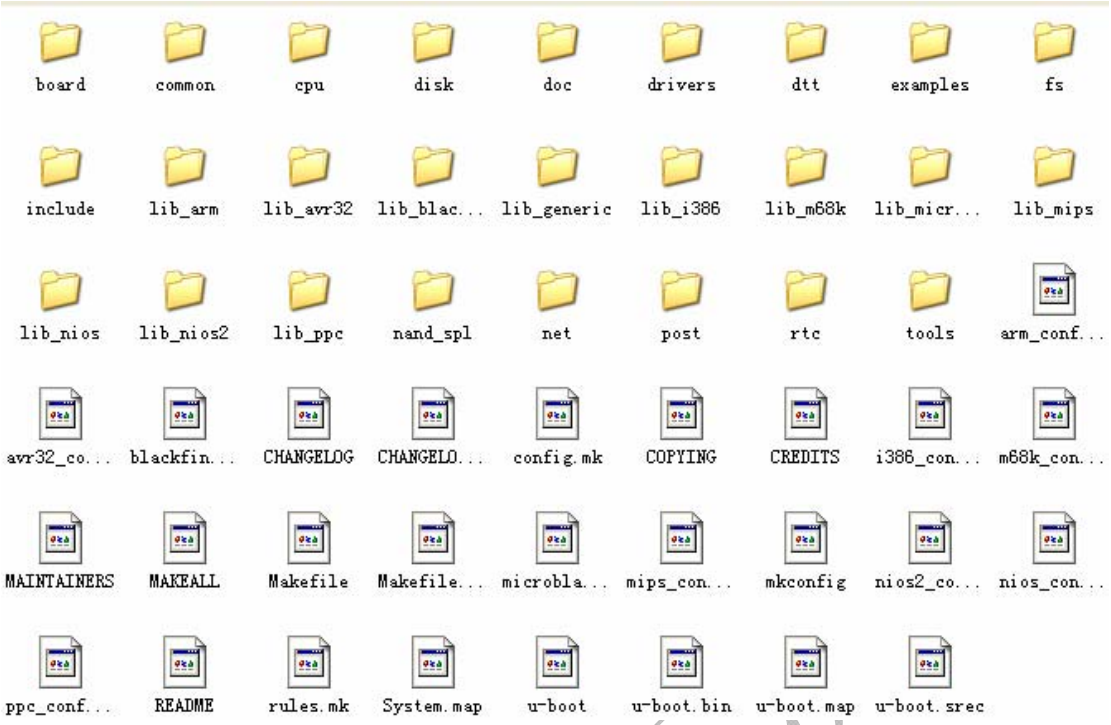


如果您对u-boot情有独钟，那么您还可以进入下列的邮件列表。U-Boot邮件列表网站：http://lists.sourceforge.net/lists/listinfo/u-boot-users/。另外，还有一个DENX相关的网站：http://www.denx.de/re/DPLG.html。

如果您正在进行某个项目，或许您更关注 U-Boot 代码，那么，让我们开始移植的旅途吧！

2.2 U-Boot体系结构

本次移植采用的是 U-Boot-1.2.0 版本，我们先来浏览一下源代码，如下图所示：



我们在这个源码里找到了 lib_arm、lib_avr32、lib_blackfin、lib_i386、lib_m68k、lib_mips、lib_ppc 等硬件平台的目录。说明此代码在这些平台上都是兼容的。我们用的是 arm 硬件平台，我们只关注 arm 相关的代码。因此，我们可以随时把带有 lib_xxx 的目录（文件夹）删掉，而只保留 lib_arm 和 lib_generic 文件夹。注意，在删除 U-Boot 源码包中的文件或者文件夹(目录)时候，务必在相应的目录下打开 makefile 文件核对一下，需要在 makefile 文件中修改或删除对应的条目。

这时候，您一定想知道每个文件夹都是干什么的。如果连文件夹是干什么的都不知道，那怎么能放心地删掉某些文件夹或者修改某些文件代码呢？我们就先来解释一下这些文件夹。

Board:存放和开发板有关的文件。U-boot 支持的每个开发板的文件，都会以子目录的形式存放在 baord 目录下。比如我们关注的 2440 开发板，与之最相近的目录，便是 SMDK2410 文件夹。或许，您可以将它更名为您自己的文件夹，比如将 SMDK2410 文件夹更名为 ARM79----这样，您可能会更有成就感！

Common:U-Boot 支持的所有命令，都在这个目录中实现。每个命令放在该目录下的一个文件中。一般情况下，我们如果修改该目录下的文件代码，无非是加一些调试信息，打开或关闭一些宏。对于该目录下的 C 代码，我们无须大幅度修改。除非您想自己增加自己的 u-boot 命令----同样也很有成就感哦！

Cpu:这个目录下，存放的是与 cpu 架构有关的目录。每个目录对应一个架构的 cpu。比如我们想移植 ARM9 的 S3C2440，就应该去找 ARM920T 的目录。其他目录实际上对我们是没有意义的。

Disk:这是要对磁盘的支持。我们只移植 u-boot 的话，那这个对我们也没有意义。

Doc:参考文档的意思，这是最没用的，也是最有用的。推荐想研究 u-boot 的同学抽时间阅读一下，有好处。

Driver:u-boot 支持的所有的驱动代码，默认是放在这个目录下的。如果您需要添加自己的驱动代码，也可以放在这里。然后再 makefile 中加入相应的.o 文件名。

Fs:这个目录下放的是 u-boot 支持的文件系统。目前 u-boot 已经能支持包括 cramfs、fat、fdos、jffs2 等文件系统。

Include:这个目录下存放的是头文件。U-boot 使用的头文件以及对各种硬件平台的系统配置文件都放在这里。对于每款特定的开发板，我们都需要修改系统配置文件，它存放在 include 目录下的 configs 子目录中。比如我们研究 2440 的移植，那么可能就对 SMDK2410.h 感兴趣。

Lib_xxx:这是与体系结构相关的库文件。

Net:此目录下存放的代码，是有关网络协议的实现的代码。比如 TFTP 协议的实现就在这里面。

Post:上电自检的目录。该目录，我对此一点都没有研究。

Tools:生成 u-boot 的工具的目录。比如创建 bin 镜像文件等。

3. U-Boot源码分析

3.1 源码入口的解释

可能大多数的同学上网查资料后都了解到，stage1 阶段的启动代码，主要就在 start.s 文件里。此 start.s 也是系统上电后执行的第一个代码。它全部由汇编编写。在讲述 start.s 之前，我们先来了解一下，系统怎么知道它要先去 start.s 里执行代码。

我们知道，每个可执行的映像 Image，肯定会给编译器一个入口，而且是“有且只有一个全局的入口”。我们可以把这个入口放在 flash 的 0x0 地址上，然后让系统去找这个 0x0 即可。

实际上，我们可以通过编写链接文件(lds)和 mk 文件来告知编译器这些情况。Lds 文件可以决定一个可执行代码的各个段的存储位置、入口地址等，详情请参考附录中的文章《u-boot lds 文件详解》。这里来说的 Mk 文件，是在 board/ 下对应开发板子目录中的 mk 文件。它指定了 TEXT_BASE 的地址。

3.2 stage1:启动分析

终于开始 u-boot 源代码的讲述了！本文讲述的 u-boot-1.2.0 源码，是经笔者修改的代码。不过，笔者也会将它与完整的源码包进行比较分析。

首先是 start.s 文件，刚才说过了，这个是系统启动后运行的第一个代码，我们详细地分析如下：

3.2.1 中断向量表的设置

```
.globl _start
_start: b       reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort:    .word data_abort
_not_used:      .word not_used
_irq:           .word irq
_fiq:           .word fiq

    .balignl 16,0xdeadbeef
```

Start.s 文件一开始，就定义了_start 的全局变量。也即，在别的文件，照样能引用这个_start 变量。这段代码验证了我们之前学过的 arm 体系的理论知识：中断向量表放在从 0x0 开始的地方。其中，每个异常中断的摆放次序，是事先规定的。比如第一个必须是 reset 异常，第二个必须是未定义的指令异常等等。

需要注意的是，在这里，我们也可以理解：为何系统一上电，会自动运行代码。因为系统上电后，会从 0x0 地方取指令，而 0x0 处放置的是 reset 标签，直接就跳去 reset 标签处去启动系统了。

另外，这里使用了 ldr 指令。而 ldr 指令中的 label，分别用一个.word 伪操作来定义。比如：

```
_undefined_instruction: .word undefined_instruction
```

我们用 source insight 跟踪代码后，发现，undefined_instruction 在 start.s 的后面给出了具体的操作，如下：

```
undefined_instruction:
    get_bad_stack
```

```
    bad_save_user_regs
    bl  do_undefined_instruction
```

在跳转到中断服务子程序之前，先有两个宏代码，一个是对stack的操作，一个是用户regs的保存。然后才能跳转如中断服务子程序中执行。读者若不理解进入中断之前做的“现场保护”，请参考《ARM体系结构与编程》等相关书籍，自然能获得详细的答案。或者，您可以前往ARM技术交流网：www.arm79.com 发帖求助，我们必将在 24 小时内给予您满意的答复。

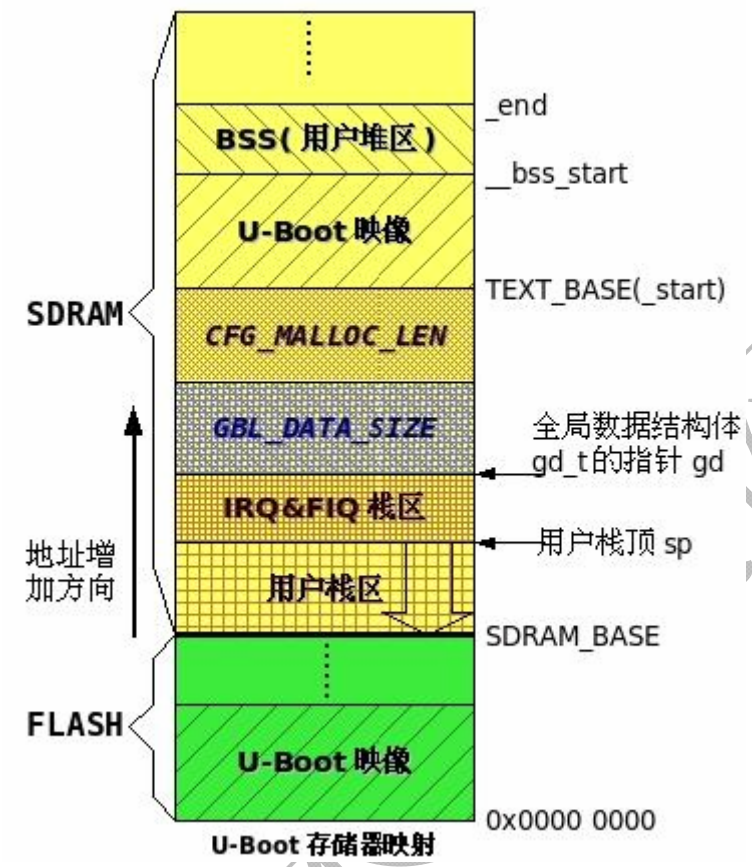
值得一提的是，当发生异常时，都将执行 u-boot-1.2.0\cpu\arm920t\ interrupts.c 中定义的中断函数。也就是说，start.s 中要跳转的这些中断子程序的代码，均在 u-boot-1.2.0\cpu\arm920t\ interrupts.c 中定义。

3.2.2 U-Boot存储器映射定义

该代码段主要是定义 u-boot 需要使用的一些映射区的 label，比如用户堆区、用户栈区、全局数据结构区等。笔者在下页给出了一个图示，把整个 u-boot 映射的所有区都列出来了，这个图非常经典，网上找的，大家可以好好研究一把。

```
_TEXT_BASE:
    .word  TEXT_BASE

.globl _armboot_start
_armboot_start:
    .word _start

/*
 * These are defined in the board-specific linker script.
 */
.globl _bss_start
_bss_start:
    .word __bss_start

.globl _bss_end
_bss_end:
    .word _end

#ifdef CONFIG_USE_IRQ
/* 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
```



U-Boot 存储器映射

从上图也可以清晰地发现，堆和栈是有区别的。而且可以看到，用户栈区是向下递减的，即地址减少的方向生长。

3.2.3 上电后CPU为SVC模式

```
reset:
    /*
     * set the cpu to SVC32 mode
     */
    mrs     r0,cpsr
    bic r0,r0,#0x1f
    orr r0,r0,#0xd3
    msr     cpsr,r0
```

这是系统复位后执行的“第一个代码段”（严格来说不是）。CPU 复位后，系统会立即被设置成 SVC 模式。记得之前有网友发帖咨询这个问题，问系统复位后，cpu 处于哪个处理器模式。这个代码，就回答了这个问题。

从这个代码中，我们也可以得到一个对寄存器操作的经验：读—修改--写。这里先把 cpsr 的值读到 r0 中，清除掉我们想修改的 bit 位，然后用 orr 指令来保证其他 bit 位不被改动，并达到修改寄存器低 5 位值的目的。最后用 msr 指令把 r0 的值给 cpsr 寄存器达到我们的修改目的。

3.2.4 关闭看门狗

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
    ldr      r0, =pWTCON
    mov      r1, #0x0
    str      r1, [r0]
```

根据 S3C2440 的 datasheet 文档，系统启动后，看门狗寄存器是被使能的，所以，如果不在预计的时间内“喂狗”，就有“被狗咬”的可能。别说啥了，赶紧先喂狗。上面这段代码即为喂狗代码。u-boot 代码编写者把它放在 CPU 上电修改 SVC 模式后的第一个代码，是可以理解的。

这个代码，也是修改寄存器的代码，它的思路依旧是：读—修改—写。

实际上，u-boot-1.2.0 代码在喂狗代码之前，还有一段代码，如下：

```
#if defined(CONFIG_S3C2400)
# define pWTCON         0x15300000
# define INTMSK     0x14400008  /*     Interupt-Controller     base
addresses */
# define CLKDIVN        0x14800014      /* clock divisor register */
#elif defined(CONFIG_S3C2410)
# define pWTCON         0x53000000     /* 喂狗寄存器*/
# define INTMSK     0x4A000008  /*     Interupt-Controller     base
addresses */
# define INTSUBMSK      0x4A00001C
# define CLKDIVN        0x4C000014      /* clock divisor register */
#endif
```

这是定义寄存器用的。比如根据 S3C2440 的 datasheet 文档，喂狗寄存器 pWTCON 的寄存器地址是 0x15300000，需要定义后才能使用。同理，这里还定义了时钟除数寄存器 CLKDIVN 和中断掩码的 INTMSK 寄存器的地址。在后续代码中会陆续用到。

3.2.5 关掉中断

```
    /*
     * mask all IRQs by setting all bits in the INTMR - default
     */
    mov    r1, #0xffffffff
    ldr r0, =INTMSK
    str r1, [r0]
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
    str r1, [r0]
# endif
```

从注释可以看出此段代码的作用：屏蔽掉所有的 irq 中断。为了屏蔽这些中

断，我们只要把 INTMSK 的所有的 bit 位都置 1 即可。INTMSK 寄存器共 32bit 位，每个 bit 对应着不同的中断源。事实上，笔者认为这个代码是多余的，只是为了“心里更踏实”而已。因为 S3C2440 的 datasheet 文档里明确指出，cpu 在复位的时候，这个寄存器的值就是 0XFFFFFFFF，以防止发生异常中断。

3.2.6 修改时钟除数寄存器

```
    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    mov    r1, #0     /* 原先的值是 3 ,现在是 1:1:1*/
    str r1, [r0]
```

在 u-boot-1.2.0 源码中，给 CLKDIVN 寄存器赋值的是#0x3，表示 FCLK:HCLK:PCLK = 1:2:4，这里笔者将其比例改为 1:1:1，没啥特殊的目的，调试代码的时候试验用的，后来调试完毕，就没有再修改了。

3.2.7 调用cpu_init_crit

```
#ifndef    CONFIG_SKIP_LOWLEVEL_INIT
    bl    cpu_init_crit
#endif
```

此段代码指明：若未定义 CONFIG_SKIP_LOWLEVEL_INIT，就执行 cpu_init_crit。我们当然不会跳过底层的初始化。因为 LOWLEVEL_INIT 会对我们的 SDRAM 进行初始化，这对我们的 cpu 是必要的。根据 source insight 的索引，我们转到了 cpu_init_crit 的代码中：

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
    /*
     * flush v4 I/D caches
     */
    mov    r0, #0
    mcr    p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
    mcr    p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
    /*
     * disable MMU stuff and caches
     */
    mrc    p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002300    @ clear bits 13, 9:8 (--V- --RS)
    bic r0, r0, #0x00000087    @ clear bits 7, 2:0 (B--- -CAM)
    orr r0, r0, #0x00000002    @ set bit 2 (A) Align
    orr r0, r0, #0x00001000    @ set bit 12 (I) I-Cache
    mcr    p15, 0, r0, c1, c0, 0
    /*
```

```
     * before relocating, we have to setup RAM timing
     * because memory timing is board-dependend, you will
     * find a lowlevel_init.S in your board directory.
     */
    mov    ip, lr
    bl   lowlevel_init
    mov    lr, ip
    mov    pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */
```

非常符合我们的思维，我们无效掉了指令 cache 和数据 cache，并禁止 MMU 与 cache。为什么会有这一步呢？笔者曾经深受 cache 的伤害。在调试代码的时候，下载完修改的 bin 文件后，如果只按复位键，而不关掉板子重新上电，就会造成 cache 中可能残留之前对 cache 操作的数据。我们称之为“脏数据”，它会映像我们的调试结果，造成假象。

当然，在这里无效 cache 和 MMU 肯定还有别的原因。比如在初始化阶段，可以认为我们只有一个任务在跑，没有必要，也不允许使用地址变换。因此最好应该无效掉 MMU。

由于在 cpu_init_cri 子程序中又一次调用子程序 lowlevel_init，因此，需要事先保护好 lr 寄存器的内容。当返回时候，再恢复它。在进入 lowlevel_init 之前，有必要详细说一下 mov ip, lr，这个语句的 ip。

为了使单独编译的 C 语言程序和汇编程序之间能相互调用，必须为子程序间的调用规定一定的规则。这就是 ATPCS 规则。它规定了一些子程序间调用的基本规则。在寄存器的使用规则里，寄存器 R12 作用子程序间的 scratch 寄存器，记做 ip。mov ip, lr 语句的 ip 由此而来。笔者认为，这里使用别的通用寄存器来代替 ip，实现的功能也是一样的。详情请参考《ARM 体系结构与编程》第 6 章 ATPCS 介绍。

3.2.8 调用lowlevel_init

这个函数在 u-boot-1.2.0\board\smdk2410\lowlevel_init.S 文件中。这是对 SDRAM 的初始化。

```
_TEXT_BASE:
    .word  TEXT_BASE

.globl lowlevel_init
lowlevel_init:
    /* memory control configuration */
    /* make r0 relative the current location so that it */
    /* reads SMRDATA out of FLASH rather than memory ! */
    ldr      r0, =SMRDATA
    ldr r1, _TEXT_BASE
    sub    r0, r0, r1
    ldr r1, =BWSCON    /* Bus Width Status Controller */
```

```
    add     r2, r0, #13*4
0:
    ldr     r3, [r0], #4
    str     r3, [r1], #4
    cmp     r2, r0
    bne     0b

    /* everything is fine now */
    mov    pc, lr
    .ltorg
/* the literal pools origin */

SMRDATA:
    .word
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(
B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B
7_BWSCON<<28))
    .word
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+
(B0_Tah<<4)+(B0_Tacp<<2)+(B0_PMC))
    .word
((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+
(B1_Tah<<4)+(B1_Tacp<<2)+(B1_PMC))
    .word
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+
(B2_Tah<<4)+(B2_Tacp<<2)+(B2_PMC))
    .word
((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+
(B3_Tah<<4)+(B3_Tacp<<2)+(B3_PMC))
    .word
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+
(B4_Tah<<4)+(B4_Tacp<<2)+(B4_PMC))
    .word
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+
(B5_Tah<<4)+(B5_Tacp<<2)+(B5_PMC))
    .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
    .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
    .word
((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<1
6)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30
```

该段代码是对 SDRAM 控制器相关的寄存器赋值，赋值过程中，采用了一个

巧妙的做法，把 SDRAM 控制器初始化需要用到的 13 个寄存器的值先保存在文字池(literal pools)中，然后通过 LDR 伪指令以及.ltorg 来访问这个文字池，获取寄存器的值赋值到对应的寄存器地址中去。

很多同学对此代码的两个地址不理解：SMRDATA 与_TEXT_BASE。不理解这两个地址相减之后，到底是一个什么值。为什么要相减呢？

其实编译器进行编译，是按照链接文件进行的。也就是说，编译的时候所有的地址都是相对于这个 TEXT_BASE 计算出来的。而我们的程序是存放在 Flash 中的，ARM 上电后，假设从 nandflash 模式启动，那么它会把 Nandflash 的前 4K 加载到内存中开始运行，当然是从 0x0 这个地址开始运行，所以要求我们的代码在还没有搬移到 TEXT_BASE（0x38f00000）这个位置以前是不能使用这些 label 的，只能找到一个相对于 0x0 的地址出来，才能得到真正的数据。而且这时候，我们编译出来的 bin 文件是存放在 0x0000000 的，而不是存放在 0x38f00000 的。嘿嘿，说的有点乱，不知道有没有把笔者的意思表达出来。

关于SDRAM初始化，笔者在之前的免费课程的讲解中，已经讲述的非常详细。ARM技术交流网也有关于SDRAM初始化的专题讨论：u-boot里SDRAM初始化中的地址问题，网址：http://www.arm79.com/read.php?tid=218。另外，可参考ARM技术交流网的免费课程录音，网址：http://www.arm79.com/thread.php?fid=32。

3.2.9 代码的搬移

```
#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

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

copy_loop:
    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 */
```

在 SDRAM 初始化完毕后，我们开始搬移代码，把代码从原先的 0x0 开始的位置搬移到内存中的适当的位置继续执行。为啥要搬移代码？原因可能如下：

1、运行速度的考虑。

flash 的读写速度远小于 SDRAM 的读写速度，搬移到 SDRAM 后，可提高

运行效率。

2、空间的考虑。

如果是 nandflash 启动模式，那么只有 4KB 的空间供用户使用，实际的代码是永远大于 4KB 的，因此需要重新开辟空间来进行代码的运行工作。

一时间只能想起这两个原因。欢迎广大网友补充。

有些版本的 u-boot 的代码搬移用 C 语言来实现：`bl CopyCode2Ram`，也是可以的。因为这时候，我们完全搭建好了 C 环境。

在这段代码中，还有一个子程序：`beq stack_setup`，用来设置栈空间的，我们在下节中讲解。

3.2.10 栈空间的设置

```
stack_setup:
    ldr r0, _TEXT_BASE      /* upper 128 KiB: relocated uboot    */
    sub     r0, r0, #CFG_MALLOC_LEN         /* malloc area   */
    sub     r0, r0, #CFG_GBL_DATA_SIZE      /* bdinfo        */
#ifdef CONFIG_USE_IRQ
    sub                                 r0,                                 r0,
#(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
    sub     sp, r0, #12             /* leave 3 words for abort-stack   */
```

这段代码是用来分配各个栈空间的。包括分配动态内存区，全局数据区，IRQ 和 FIQ 的栈空间等。

3.2.11 BSS段的清零

```
clear_bss:
    ldr r0, _bss_start          /* find start of bss segment          */
    ldr r1, _bss_end            /* stop here                          */
    mov     r2, #0x00000000     /* clear */

clbss_l:
        str r2, [r0]            /* clear loop...                      */
    add     r0, r0, #4
    cmp     r0, r1
    ble clbss_l
```

本段代码先设置了 BSS 段的起始地址与结束地址，然后循环清楚所有的 BSS 段。至此，所有的 cpu 初始化工作（stage1 阶段）已经全部结束了。后面的代码，将通过 `ldr pc, _start_armboot`，进入 C 代码执行。这个 C 入口的函数，是在 u-boot-1.1.6\lib_arm\board.c 文件中。它标志着后续将全面启动 C 语言程序，同时它也是整个 u-boot 的主函数。

3.3 stage2:C代码分析

上节提到，start_armboot 函数不仅标志着后续将全面启动 C 语言程序，同时它也是整个 u-boot 的主函数。那么该函数完成什么操作呢？

3.3.1 为gd与bd分配空间

```
    /* Pointer is writable since we allocated a register for it */
    gd     =     (gd_t*)(_armboot_start     -     CFG_MALLOC_LEN     -
sizeof(gd_t));
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__("": : :"memory");

    memset ((void*)gd, 0, sizeof (gd_t));
    gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
    memset (gd->bd, 0, sizeof (bd_t));
```

如同使用变量之前，需要声明定义一样，这里使用全局变量 gd 和 bd 之前，我们需要先设置它的地址，并用 memset 函数为它分配合适的空间。u-boot 的注释告知我们，gd 和 bd 是一个可写的指针，实际上不过是一个地址而已。

代码中的这句话：__asm__ __volatile__("": : :"memory");目的就是告诉编译器内存被修改过了。更详细的关于 C 程序中内嵌汇编的文档，请参考附录中的文献《ARM GCC 内嵌（inline）汇编手册》。

3.3.2 执行初始化列表函数

```
    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }
```

这是一个 for 语句，却完成了板子初始化列表函数的功能。我们先来看一下 for 语句的初始值：init_sequence。用 source insight 跟踪后发现，它是一个指针数组：

```
init_fnc_t *init_sequence[] = {
    cpu_init,       /* basic cpu dependent setup */
    board_init,         /* basic board dependent setup */
    interrupt_init,     /* set up exceptions */
    env_init,       /* initialize environment */
    init_baudrate,          /* initialze baudrate settings */
    serial_init,        /* serial communications setup */
    console_init_f,         /* stage 1 init of console */
    display_banner,         /* say that we are here */
```

```
#if defined(CONFIG_DISPLAY_CPUINFO)
    print_cpuinfo,      /* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,         /* display board info */
#endif
    dram_init,      /* configure available RAM banks */
    display_dram_config,
    NULL,
};
```

指针数组的每个成员都对应着一个函数名（函数指针），指向的是 init_fnc_t 类型的函数。For 语句每次都会判断当前的函数是不是 NULL，如果是，则跳出 for 语句，完成当前的板子初始化列表函数的功能。

可能大家都注意到了一个类型：init_fnc_t，它表示什么意思呢？我们看到了在初始化列表函数之前，有一个新的数据类型，它是个 typedef 语句：

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

可能有的同学对此不太理解，为啥非得用一个 typedef 呢？笔者认为，可以不用 typedef，但是用了 init_fnc_t 后，团队中别的成员来看代码的时候，会很轻松地知道，这是一个初始化(init)的函数(fnc)，增加了代码的可读性。如果您对 typedef 用法还不是很理解，那就赶紧咯，复习下 typedef 的用法。我们在附录 C 中给出了《typedef 用法小结》，附录 D 中给出了《u-boot 中 typedef 应用解析》，以上两篇文档均摘自互联网资料，可供参考。

现在，我们对每个初始化列表函数，都进行分析，由于代码量太大，我们不一一列出代码，大家可以参考 u-boot-1.2.0 的源码包。

Cpu_init 函数，并没有做实质性的工作，而且我们现在暂时没有定义 CONFIG_USE_IRQ，因此，代码执行到这里，直接就 return 0;

Board_init 函数，是初始化与硬件平台有关的函数。它的工作很明显：时钟的设置，引脚 IO 口的设置，并且在这里把数据 cache 和指令 cache 也打开了。以上工作的完成，标志着板子已经准备好工作了。当然，考虑到可能系统会发生意外中断，所以我们还需要初始化中断，让中断也准备好工作，因此 u-boot 代码中下一步就开始了中断的初始化。

interrupt_init 函数，这实际上是定时器的中断初始化。和我们之前的培训课程相符的是，我们看到了中断初始化中的那几个熟悉的寄存器，首先是两个配置寄存器 TCFG0 和 TCFG1。晕倒，怎么代码中只有 TCFG0 的设置，没有 TCFG1 的设置？很明显，TCFG1 采用的是默认值。然后配置寄存器的下载值，最后打开启动开关，启动定时器工作。

env_init 函数，这是对我们板子的环境做出初始化配置。顺便提一下，我们修改的配置文件里，用的是 nand flash 来存放环境变量的值。

```
#define CFG_ENV_IS_IN_NAND      1
#define CFG_ENV_OFFSET          0x40000
#define CFG_ENV_SIZE64          0xc000    /* Total Size of
Environment Sector */
#define CFG_ENV_SIZE            0x20000  /*    Total    Size    of
```

```
Environment Sector */
```

因此，我们在进入 u-boot 命令行之后，运行的关于环境变量的操作，只要它被保存，saveenv，肯定是 save 在 nandflash 中的某个位置。

init_baudrate 函数，初始化波特率。我们心里要很明确，初始化波特率，目的只有一个：让串口打印调试信息。因此，下一个函数，肯定是串口的初始化函数。所以，我们可以在调试的时候，先算好波特率的值，直接赋值给 gd->bd->bi_baudrate，注释掉该函数中的其他代码。调试完毕，再恢复出原先的代码。这样，我们可以不用考虑别的因素导致串口打印不出信息。

serial_init 函数，串口的初始化函数。这里调用了另一个函数来配置串口寄存器：serial_setbrg();在这个函数中，我们看到了关于串口的 5 个寄存器的配置。关于每个寄存器的更详细的配置信息，请参考 ARM 技术交流网推出的串口课程讲解部分。

console_init_f 函数，这个函数的功能只有一个，就是指出我们目前是使用串口的，因此有此句：`gd->have_console = 1;`然后直接返回 0。

display_banner 函数。OK，现在串口初始化完毕，我们可以打印信息了。这是 u-boot 代码中第一次打印信息。我们可以在这里加入我们自己的代码，比如笔者移植的 u-boot 代码中，就加入了如下“欢迎”的代码信息：

```
    printf ("\n\n");
    printf("*********************************************
****\n");
    printf("*                                           *\n");
    printf("*              ARM 技术交流网欢迎您!          *\n");
    printf("*                                  www.arm79.com
*\n");
    printf("*                                           *\n");
    printf("*********************************************
****\n");
```

出现打印信息后，可以说，u-boot 移植已经成功了一半。有了打印信息，我们可以随时用打印信息来调试。“点灯大法”，你可以睡觉去了。

初始化列表函数中，还有几个函数，比较简单，我这里就不说了。随后开始的是一系列外设的初始化。

3.3.3 配置可用的flash区：flash_init

当您跟踪到 flash_init 函数的时候，您会发现，这里只兼容 AMD 系列的 flash 芯片，比如 LV400 及 LV800。如果您的开发板上刚好就是 AMD 的芯片，那么恭喜，您可能就不需要修改 flash ID 号了。可惜，笔者用的开发板上用的是 EON 生产的 flash 芯片。笔者只好把 AMD 的所有代码，都改成 EON 的代码。比如，笔者嫌麻烦，直接补上

```
#define EN29LV160AB_ID 0x2249001c
```

再来一个：

```
#define CONFIG_EON_29LV160AB    1
```

后面再修改 FLASH_BANK_SIZE、CFG_MAX_FLASH_SECT、

PHYS_FLASH_1 等信息，来配置笔者的板子上可用的 flash 区域。

3.3.4 初始化内存分配函数

mem_malloc_init 函数，这是非常关键的一步，请大家引起注意。我们必须配置好内存的起始地址和结束地址，然后把这块区域清零，以便后续来使用它。

3.3.5 nand flash的初始化

这部分代码，可能隐含是不执行的。如果您想使用它，需要自行打开，然后添加自己的 nand flash 驱动的代码。笔者自己没有写 nand flash 的代码，而是直接 copy 别人的代码，拿过来改一改。如果想验证自己修改或者自己写的 nand flash 的驱动是否正确，可以试着从 nand flash 中读取或写入一个数据，并用串口打印出来（笔者修改的 nand flash 驱动代码，将在 ARM 技术交流网上公布，需要的可以随时下载）。后面的代码，一直到 main_loop 函数，我们都不需要修改。main_loop 函数是进入命令循环的函数，它接受用户从串口输入的命令，然后执行相应的工作，这也是整个 u-boot 的工作循环。

注意，它并没有使用中断来触发命令的操作，而是用循环来做这部分的工作：

```
/* main_loop() can return to retry autoboot, if so just run it again.
*/
    for (;;) {
        main_loop ();
    }
```

至此，u-boot 代码的分析接近尾声。如果您对 u-boot 代码中某一部分代码感兴趣，并有不解之处，欢迎前往 ARM 技术交流网进行学习讨论！

4. U-Boot移植过程参考

4.1 移植准备

我们采用的是 u-boot-1.2.0 版本。该版本源码包的获取，请参考本文档“2.1 章节 u-boot 源码的获取”。下面，您只需要一台电脑，一块开发板，即可亲身体验 u-boot 移植啦！

4.2 U-Boot移植过程分析

本章节将详细给出整个 u-boot 移植的过程，您只需要按照此过程操作，即可轻松地移植，并定制属于您自己的 u-boot-1.2.0 版本到您的开发板上！

说明：交叉编译工具的制作，请自行完成！事实上，许多开发板厂商都给出了详细的制作过程供用户参考。

4.2.1 修改Makefile文件

我们建议，除非您只是体验一次 u-boot，而非研究 u-boot。否则，请抽时间浏览一下 u-boot 根目录下的 readme 文档。这将对您理解 u-boot 大有帮助。

请点击您的鼠标，打开 makefile 文件。如果您是在 linux 环境下开发，使用 vi makefile 命令可打开该文件。使用 ctrl + F 键，查找“smdk2400_config”，找到后，您会看到如下代码：

```
smdk2400_config    :    unconfig
    @$(MKCONFIG) $(@:_config=) arm arm920t smdk2400 NULL s3c24x0
```

我们解释一下代码：

arm，就表示现在用的是 CPU 的架构是 arm 体系结构。

arm920t，指明这是 cpu 的内核类型，它对应于 cpu/arm920t 目录。

Smdk2400，这是开发板的型号，它的目录在 board/smdk2400 目录下。您也可以自己命名您的开发板。比如：ARM79。

NULL，表示开发者或者经销商是谁（vender）。

S3c24x0，表示开发板上的 cpu 是啥。对于我们的开发板，当然是 S3C2440 了。

根据以上的解释，我们可以自己模仿着建立自己的编译项：

```
arm79_config    :    unconfig
    @$(MKCONFIG) $(@:_config=) arm arm920t arm79 NULL s3c24x0
```

OK，修改完毕，可以保存、退出 makefile。

4.2.2 建立自己的开发板文件

为了使得 u-boot 具有自己的特征，我们需要在 board 目录下建立自己的文件：

1、复制 board/smdk2410，并更名为 board/arm79。

2、复制 board/smdk2410/smdk2410.c，并更名为 board/arm79/arm79.c

OK，我们的开发板是自己花钱买的，现在开发板上面跑的 u-boot，我们也可以假装是自己写的代码了。

4.2.3 建立自己的配置文件

配置文件在：include/configs/smdk2410.h。大家还希望用别人的配置文件吗？当然不想！所以，改过来！复制 include/configs/smdk2410.h，并更名为：include/configs/arm79.h。这时候，可以暂时保留 arm79.h 中的配置信息。一会再来修改它。我们现在有更重要的事情要做。

4.2.4 修改交叉编译工具的路径

交叉编译工具，您可以使用开发板公司为您提供的制作包即可。修改交叉编译工具的路径，请参考每个开发板公司的用户手册。这里无法给出一个定性的答

案。一般都是在/etc/profile 文件下修改，增加一个.bin 目录。

4.2.5 测试编译u-boot-1.2.0 版本

其实，u-boot 虽然号称经典，但是有些版本在某些特定的 arm 平台或者 powerpc 平台是编译不通过的。笔者在实习时候，在公司产品上移植了一个 u-boot 版本，就是不行的。换成 u-boot-1.2.0 版本，可以编译通过。因此，笔者本次移植也采用了 u-boot-1.2.0 版本。

```
cd u-boot-1.2.0        /* 切换到 u-boot 目录下 */
make arm79_config
```

这时候，命令行界面上会显示：Configuring for arm79 board…然后您再敲入 make，回车。如果您的交叉编译工具安装正确的话，这时候就开始编译了，大约几分钟后，您就会看到窗口中出现了.bin 文件的打印信息，回到您的 u-bot 根目录下，您就会发现，那里多出了一个 u-boot.bin 文件。

当然，当您在调试的时候，或许您还想得到 u-boot 的反汇编代码，那么，请再次打开 makefile 文件，用 ctrl + F 键，查找到“u-boot.bin”所在的行，大约在第 239 行（如果您之前没有在 makefile 中修改别的信息的话）：

```
ALL = $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U_BOOT_NAND)
```

这行的代码，是指定编译后，输出啥文件的。可以看到，编译结果，会输出 u-boot.srec 文件，u-boot.bin 文件，system.map 文件，等等。这时候，您如果想让它输出 u-boot 的反汇编文件，只要这样做：

```
ALL = $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(obj)u-boot.dis $(U_BOOT_NAND)
```

对比一下，发现我们现在增加了“$(obj)u-boot.dis”。对了，这就是指定编译结果，要输出 u-boot 反汇编文件。

4.2.6 修改配置文件

之前已经提到，笔者的配置文件已经改为 arm79.h，目录在 include/configs/arm79.h。由于配置文件修改较多，而且是根据具体开发板进行配置的，因此笔者直接给出了修改完的配置文件，并作出详细的注释，希望对您有所帮助！

```
#ifndef __CONFIG_H
#define __CONFIG_H

/*
 * High Level Configuration Options
 * (easy to change)
 */
#define CONFIG_ARM920T      1   /* This is an ARM920T Core  */
#define  CONFIG_S3C2410     1   /* in a SAMSUNG S3C2410 SoC     */
#define CONFIG_SMDK2410         1   /* on a SAMSUNG SMDK2410 Board
```

```
*/

/* input clock of PLL */
#define CONFIG_SYS_CLK_FREQ   12000000     /*  输入时钟 12M */


#define USE_920T_MMU            1
#undef CONFIG_USE_IRQ                   /*  暂时不使用 IRQ */

/*
 * Size of malloc() pool
 */
#define CFG_MALLOC_LEN          (CFG_ENV_SIZE + 128*1024)
#define CFG_GBL_DATA_SIZE   128 /* size in bytes reserved for initial data */

/*
 *  网卡的配置信息
 */
#define CONFIG_DRIVER_DM9000        1
#define CONFIG_DM9000_BASE          0x20000300
#define DM9000_IO               CONFIG_DM9000_BASE
#define DM9000_DATA             (CONFIG_DM9000_BASE + 4)
#define CONFIG_DM9000_USE_16BIT

/*
 * select serial console configuration
 */
#define CONFIG_SERIAL1                  1  /*  使用串口  */

/***********************************************************
 * RTC
 ***********************************************************/
#define  CONFIG_RTC_S3C24X0      1

/* allow to overwrite serial and ethaddr */
#define CONFIG_ENV_OVERWRITE

#define CONFIG_BAUDRATE         38400      /*  波特率使用 38400 */

/***********************************************************
 * Command definition
 ***********************************************************/
#define CONFIG_COMMANDS \
                (CONFIG_CMD_DFL      | \
```

```
                CFG_CMD_LOADS | \
                CFG_CMD_LOADB | \
                CFG_CMD_CACHE  | \
                CFG_CMD_NAND | \
                CFG_CMD_FLASH | \
                CFG_CMD_PING | \
                /*CFG_CMD_EEPROM |*/ \
                /*CFG_CMD_I2C      |*/ \
                /*CFG_CMD_USB    |*/ \
                CFG_CMD_REGINFO   | \
                CFG_CMD_DATE     | \
                CFG_CMD_ELF)

/* this must be included AFTER the definition of CONFIG_COMMANDS (if any) */
#include <cmd_confdefs.h>

#define CONFIG_BOOTDELAY  3          /*  进入命令行的等待时间 3s */
/*#define CONFIG_BOOTARGS       "root=ramfs                devfs=mount
console=ttySA0,9600" */
/*#define CONFIG_ETHADDR    08:00:3e:26:0a:5b */
#define CONFIG_NETMASK              255.255.255.0
#define CONFIG_IPADDR         10.0.0.110
#define CONFIG_SERVERIP       10.0.0.1
/*#define CONFIG_BOOTFILE   "elinos-lart" */
/*#define CONFIG_BOOTCOMMAND    "tftp; bootm" */

#if (CONFIG_COMMANDS & CFG_CMD_KGDB)
#define CONFIG_KGDB_BAUDRATE    9600          /* speed to run kgdb serial
port */
/* what's this ? it's not used anywhere */
#define CONFIG_KGDB_SER_INDEX    1          /* which serial port to use */
#endif

/*
 * Miscellaneous configurable options
 */
#define  CFG_LONGHELP                /* undef to save memory         */
#define  CFG_PROMPT       "[arm79-uboot-1.2.0]# "   /*     Monitor      Command
Prompt   */
#define  CFG_CBSIZE         256       /* Console I/O Buffer Size      */
#define  CFG_PBSIZE  (CFG_CBSIZE+sizeof(CFG_PROMPT)+16)  /*  Print  Buffer
Size */
#define  CFG_MAXARGS         16       /* max number of command args   */
#define CFG_BARGSIZE          CFG_CBSIZE    /* Boot Argument Buffer Size */
```

```
#define CFG_MEMTEST_START0x30000000 /* memtest works on  */
#define CFG_MEMTEST_END         0x33F00000 /* 63 MB in DRAM   */

#undef    CFG_CLKS_IN_HZ       /* everything, incl board info, in Hz */

#define  CFG_LOAD_ADDR        0x33000000 /* default load address      */

/* the PWM TImer 4 uses a counter of 15625 for 10 ms, so we need */
/* it to wrap 100 times (total 1562500) to get 1 sec. */
#define  CFG_HZ                  1562500

/* valid baudrates */
#define CFG_BAUDRATE_TABLE    { 9600, 19200, 38400, 57600, 115200 }

/*-----------------------------------------------------------------------
 * Stack sizes
 *
 * The stack sizes are set up in start.S using the settings below
 */
#define CONFIG_STACKSIZE    (128*1024)  /* regular stack */
#ifdef CONFIG_USE_IRQ
#define CONFIG_STACKSIZE_IRQ   (4*1024)      /* IRQ stack */
#define CONFIG_STACKSIZE_FIQ   (4*1024)      /* FIQ stack */
#endif

/*-----------------------------------------------------------------------
 * Physical Memory Map
 */
#define CONFIG_NR_DRAM_BANKS    1          /* we have 1 bank of DRAM */
#define PHYS_SDRAM_1            0x30000000 /* SDRAM Bank #1 */
#define PHYS_SDRAM_1_SIZE  0x04000000 /* 64 MB */

#define PHYS_FLASH_1       0x00000000 /* Flash Bank #1 */

#define CFG_FLASH_BASE        PHYS_FLASH_1

/*-----------------------------------------------------------------------
 * FLASH and environment organization
 */
#if 0
#define CONFIG_AMD_LV400    1     /* uncomment this if you have a LV400 flash
*/
#define CONFIG_AMD_LV800    1     /* uncomment this if you have a LV800 flash
```

```
*/
#endif
#define CONFIG_EON_29LV160AB  1

/**
   * added by www.arm79.con
   */
#define CFG_MAX_FLASH_BANKS 1     /* max number of memory banks */
#ifdef CONFIG_EON_29LV160AB
#define PHYS_FLASH_SIZE       0x00200000 /* 2MB */
#define CFG_MAX_FLASH_SECT    (35)/* max number of sectors on one chip */
#define CFG_ENV_ADDR          (CFG_FLASH_BASE  +  0x1F0000)  /*  addr  of
environment */
#endif


#ifdef CONFIG_AMD_LV800
#define PHYS_FLASH_SIZE       0x00200000 /* 1MB */
#define CFG_MAX_FLASH_SECT    (19)/* max number of sectors on one chip */
#define CFG_ENV_ADDR          (CFG_FLASH_BASE  +  0x1F0000)  /*  addr  of
environment */
#endif
#ifdef CONFIG_AMD_LV400
#define PHYS_FLASH_SIZE       0x00080000 /* 512KB */
#define CFG_MAX_FLASH_SECT    (11)/* max number of sectors on one chip */
#define CFG_ENV_ADDR          (CFG_FLASH_BASE  +  0x070000)  /*  addr  of
environment */
#endif

/* timeout values are in ticks */
#define CFG_FLASH_ERASE_TOUT(5*CFG_HZ) /* Timeout for Flash Erase */
#define CFG_FLASH_WRITE_TOUT(5*CFG_HZ) /* Timeout for Flash Write */

//#define      CFG_ENV_IS_IN_FLASH    1
#define CFG_ENV_IS_IN_NAND          1
#define CFG_ENV_OFFSET              0x40000
#define CFG_ENV_SIZE64              0xc000  /* Total Size of Environment Sector
*/
#define CFG_ENV_SIZE                0x20000     /* Total Size of Environment Sector
*/

/**
   * added by www.arm79.con
   */
```

```
#define CFG_NAND_BASE            0
#define CFG_MAX_NAND_DEVICE 1
#define NAND_MAX_CHIPS           1


#endif   /* __CONFIG_H */
```

4.2.7 修改start.s文件

这是系统启动运行的第一个文件。大部分代码是不需要修改的，毕竟S3C2410和S3C2440的启动时差别不大的。笔者修改了下时钟：

```
    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr   r0, =CLKDIVN
    mov     r1, #0          /*  原先的值是 3 ,现在是 1:1:1*/
    str   r1, [r0]
```

事实上，没有必要修改这个。笔者也是调试的时候修改的，调试结束，也就没有再改回去。其他地方就不需要修改了：SDRAM初始化部分，代码搬移部分，都可以直接用。

4.2.8 修改board/arm79/arm79.c

这个文件是由原来的board/smdk2410/smdk2410.c来的。笔者修改了这段：

```
#if FCLK_SPEED==0          /* Fout = 203MHz, Fin = 12MHz for Audio */
#define M_MDIV     0xC3
#define M_PDIV0x4
#define M_SDIV0x1
#elif FCLK_SPEED==1        /* Fout = 75MHz */
#define M_MDIV     42   /* 42*/
#define M_PDIV0x2       /* 0x3 */
#define M_SDIV0x2
#endif
```

这段代码修改了MPLL的时钟。它是为了迎合波特率计算公式的设置的。然后在该文件里的board_init函数，笔者把UPLLCON的配置和MPLLCON的配置顺序颠倒下。可能这是2410与2440的区别。S3C2440的datasheet文档中明确规定，必须先初始化UPLLCON，然后延迟一段时间后才能初始化MPLLCON。代码如下：

```
    /* configure UPLL */
    clk_power->UPLLCON = ((U_M_MDIV << 12) + (U_M_PDIV << 4) +
U_M_SDIV);


    /* some delay between MPLL and UPLL */
     delay(0xffff);
```

```
    delay(0xffff);
    delay(0xffff);

    /* configure MPLL */
    clk_power->MPLLCON = ((M_MDIV << 12) + (M_PDIV << 4) + M_SDIV);

    /* some delay between MPLL and UPLL */
     delay(0xffff);
    delay(0xffff);
    delay(0xffff);
```

另外，笔者修改了该函数里的 IO 口的初始化配置部分，这是根据笔者开发板上面的硬件结构修改的代码：

```
    /* set up the I/O ports */
    gpio->GPACON = 0x007FFFFF;
    gpio->GPBCON = 0x00055555;
    gpio->GPBUP = 0x000007FF;
    gpio->GPCCON = 0xAAAAAAAA;
    gpio->GPCUP = 0x0000FFFF;
    gpio->GPDCON = 0xAAAAAAAA;
    gpio->GPDUP = 0x0000FFFF;
    gpio->GPECON = 0xAAAAAAAA;
    gpio->GPEUP = 0x0000FFFF;
    gpio->GPFCON = 0x000055AA;
    gpio->GPFUP = 0x000000FF;
    gpio->GPGCON = 0xFF94FFBA;
    gpio->GPGUP = 0x0000FFEF;
    gpio->GPGDAT = gpio->GPGDAT & (~(1<<4)) | (1<<4) ;
    gpio->GPHCON = 0x002AFAAA;
    gpio->GPHUP = 0x000007FF;
```

4.2.9 修改cpu/arm920t/s3c24x0/speed.c

修改该文件，是因为 u-boot 版本中没有 S3C2440 对应的版本，只有 2410 的版本。而 2410 与 2440 在计算 MPLL 的公式上有区别。2440 芯片的 MPLL 计算公式中，多了一个“乘以 2”。代码修改的是 get_PLLCLK 函数：

```
static ulong get_PLLCLK(int pllreg)
{
    S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
    ulong r, m, p, s;

    if (pllreg == MPLL)
    r = clk_power->MPLLCON;
    else if (pllreg == UPLL)
```

```
    r = clk_power->UPLLCON;
    else
    hang();

    if (pllreg == MPLL)
        m = 2*(((r & 0xFF000) >> 12) + 8);
    else if (pllreg == UPLL)
    m = ((r & 0xFF000) >> 12) + 8;
    else
    hang();

    p = ((r & 0x003F0) >> 4) + 2;
    s = r & 0x3;

    return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
}
```

笔者承认，这个代码修改的很不成功。大家可以看到，笔者只是增加了：

```
    if (pllreg == MPLL)
        m = 2*(((r & 0xFF000) >> 12) + 8);
```

而这段代码，根本不具移植性。假设以后出了新的产品，升级版，那么这个代码无法移植，需要重新修改。最好的代码修改思路应该是，在 return 语句上修改：如果当前是 2440 的芯片，就 return 乘以 2 的时钟；如果是 2410 芯片，就不乘以 2；或者 2442 的芯片等等。这样，有几个版本的 CPU，只要增加这里的代码兼容性即可。

4.2.10 修改board.c文件

由于在修改的时候，还未编写 nand flash 驱动的代码，所以这时候最好屏蔽掉 nand_init 函数。本文件中的其他函数不需要修改。

4.2.11 重新编译u-boot

现在，我们可以试一下之前修改的 u-boot 是否可行。我们执行命令：cd u-boot-1.2.0 进入 u-boot 根目录，然后 make 一下，执行编译。当生成 u-boot.bin 文件后，把它用 JTAG 软件烧到 nor flash 或者 nand flash 中，启动开发板，如果之前的修改工作正确的话，就会出现如下界面：

```
************************************************
*                                              *
*                ARM 技术交流网欢迎您!         *
*                  www.arm79.com               *
*                                              *
************************************************
```

```
U-Boot 1.2.0 (Dec  2 2009 - 16:51:34)

U-Boot code: 33F80000 -> 33FA0A4C  BSS: -> 33FA5DB4
DRAM:  64 MB
Nor  Flash:  2 MB
Nand Flash:  256 MiB
In:      serial
Out:     serial
Err:     serial
[arm79-uboot-1.2.0]#
[arm79-uboot-1.2.0]#
```

说明您的 u-boot 移植工作基本完成。但是，我们还需要验证一下它是否可以执行我们需要的命令。所以，我们在这里一边介绍 u-boot 命令，一边演示。

5. U-Boot命令

5.1 Help命令

和 linux 的 shell 命令一样，我们同样可以用 help，甚至只用一个“？”就能查当前 u-boot 版本中支持的所有命令。只是可能用 help 打印出来的命令，并不一定被正确执行，这与具体开发板关系密切。我们来看一下 u-boot-1.2.0 的 help 命令能打印出啥来：

```
[arm79-uboot-1.2.0]# help
?           - alias for 'help'
autoscr - run script from memory
base        - print or set address offset
bdinfo    - print Board Info structure
boot        - boot default, i.e., run 'bootcmd'
bootd      - boot default, i.e., run 'bootcmd'
bootelf - Boot from an ELF image in memory
bootm      - boot application image from memory
bootp      - boot image via network using BootP/TFTP protocol
bootvx    - Boot vxWorks from an ELF image
cmp         - memory compare
coninfo - print console devices and information
cp          - memory copy
crc32      - checksum calculation
date        - get/set/reset date & time
dcache    - enable or disable data cache
echo        - echo args to console
erase      - erase FLASH memory
```

```
flinfo    - print FLASH memory information
go        - start application at address 'addr'
help      - print online help
icache    - enable or disable instruction cache
iminfo    - print header information for application image
imls      - list all images found in flash
itest     - return true/false on integer compare
loadb     - load binary file over serial line (kermit mode)
loads     - load S-Record file over serial line
loady     - load binary file over serial line (ymodem mode)
loop      - infinite loop on address range
md        - memory display
mm        - memory modify (auto-incrementing)
mtest     - simple RAM test
mw        - memory write (fill)
nand      - NAND sub-system
nboot     - boot from NAND device
nfs       - boot image via network using NFS protocol
nm        - memory modify (constant address)
ping      - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset     - Perform RESET of the CPU
run       - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv    - set environment variables
sleep     - delay execution for some time
tftpboot- boot image via network using TFTP protocol
version - print monitor version
[arm79-uboot-1.2.0]#
```

以上就是 u-boot-1.2.0 支持的命令了。我们查看这些命令列表，发现它们都只列出了命令，并没有列出具体的参数选项。这会让我们很郁闷的，比如，我们敲入 nand 命令：

```
[arm79-uboot-1.2.0]# nand
Usage:
nand      - NAND sub-system

[arm79-uboot-1.2.0]#
```

晕倒，nand 命令执行完，并没有给我们任何有用的信息，那么，这个 nand 命令就是多余的吗？不是。help 命令也提供了更详细的帮助信息，比如，我们想知道当前 nand 命令的详细选项信息，我们只要这么做：

```
[arm79-uboot-1.2.0]# help nand
nand info                       - show available NAND devices
```

```
nand device [dev]         - show or set current device
nand read[.jffs2]         - addr off|partition size
nand write[.jffs2]        - addr off|partiton size - read/write `size' bytes starting
      at offset `off' to/from memory address `addr'
nand erase [clean] [off size] - erase `size' bytes from
      offset `off' (entire device if not specified)
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
nand markbad off - mark bad block at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
nand lock [tight] [status] - bring nand to lock state or display locked pages
nand unlock [offset] [size] - unlock section

[arm79-uboot-1.2.0]#
```

OK，现在，我们看到了有关 nand 命令的所有的选项以及更详细的信息。比如我们想查看当前 nand flash 中的信息，那么，我们就可以执行：

```
[arm79-uboot-1.2.0]# nand info

Device 0: NAND 256MiB 3,3V 8-bit, sector size 128 KiB
[arm79-uboot-1.2.0]#
```

当然，刚才列出的 nand 的这么多命令，并不是每个命令都是可用的。得看我们的开发板支持否。所以我们举了 nand info 命令来说明，因为所有开发板肯定支持这个命令的。

5.2 flinfo命令

由于上一节中举例 nand flash 的命令，所以本节赶紧说一下 nor flash 的命令：flinfo。我们敲入 flinfo：

```
[arm79-uboot-1.2.0]# flinfo

Bank # 1: EON: 1x EN29LV160AB (16Mbit)
  Size: 2 MB in 35 Sectors
  Sector Start Addresses:
    00000000 (RO) 00004000 (RO) 00006000 (RO) 00008000 (RO) 00010000 (RO)
    00020000 (RO) 00030000      00040000      00050000      00060000
    00070000      00080000      00090000      000A0000      000B0000
    000C0000      000D0000      000E0000      000F0000      00100000
    00110000      00120000      00130000      00140000      00150000
    00160000      00170000      00180000      00190000      001A0000
    001B0000      001C0000      001D0000      001E0000
001F0000 (RO)
[arm79-uboot-1.2.0]#
```

打印出的是 nor flash 的所有信息。

5.3 version与date命令

现在来看下最基本的版本显示信息：version

```
[arm79-uboot-1.2.0]# version

U-Boot 1.2.0 (Dec  2 2009 - 16:51:34)
[arm79-uboot-1.2.0]#
```

它显示出了当前我们使用的 u-boot 版本是 u-boot-1.2.0。顺带也说一下打印当前时间的命令 date：

```
[arm79-uboot-1.2.0]# date
Date: 2008-12-25 (Thursday)    Time: 14:02:33
[arm79-uboot-1.2.0]#
```

可惜，这个时间是错的，我们还需要对相关的代码进行修改。

5.4 coninfo命令

这是打印所有控制设备的信息，如：

```
[arm79-uboot-1.2.0]# coninfo
List of available devices:
serial      80000003 SIO stdin stdout stderr
[arm79-uboot-1.2.0]#
```

OK，以上是刚才提到 nand flash 想起的几个命令。现在我们开始进行 u-boot 最常用的几个命令：printenv，setenv，saveenv 等等。

5.5 printenv命令

环境变量的设置，最常用的 u-boot 命令了。敲入它，我们可以知道当前的环境的一些设置：

```
[arm79-uboot-1.2.0]# printenv
bootdelay=3
baudrate=38400
netmask=255.255.255.0
ipaddr=192.168.0.104
serverip=192.168.0.101
dns=192.168.0.1
stdin=serial
stdout=serial
stderr=serial

Environment size: 150/131068 bytes
```

[arm79-uboot-1.2.0]#

这些环境设置包括开发板 IP 地址，服务器 IP 地址，dns 的设置，掩码设置等等。其实这些环境变量中，有一部分是在配置文件中定义的，另外的一些，是可以使用 setenv 命令来设置的。然后我们可以保存到 nand flash 中。下次开机的时候，就不要再去设置了。比如，上面打印出来的 dns 的设置，就是笔者添加的。

5.6 setenv命令

刚才提到，我们可以使用 setenv 命令来设置 u-boot 的环境变量。那么就来体验一下吧！从 printenv 打印的信息看出，我们的 IP 是 192.168.0.104，现在，我们要用 setenv 命令改为：192.168.0.104，试试看：

```
[arm79-uboot-1.2.0]# setenv ipaddr 192.168.0.105
[arm79-uboot-1.2.0]# printenv
bootdelay=3
baudrate=38400
netmask=255.255.255.0
serverip=192.168.0.101
dns=192.168.0.1
stdin=serial
stdout=serial
stderr=serial
ipaddr=192.168.0.105

Environment size: 150/131068 bytes
[arm79-uboot-1.2.0]#
```

当我们 printenv 之后，我们发现，在环境变量打印信息的后面出现了刚刚设置的 192.168.0.105。现在我们已经把 IP 改为这个值了。注意，这时候的 IP 地址，是在内存中的。内存，掉电丢失。所以如果现在掉电，那刚设置的 IP 地址自然也丢掉了。赶紧保存下咯，防止突发掉电，导致我们设置的 IP 丢了。

5.7 saveenv命令

我们把刚才设置的 IP 地址保存到 nand flash 中：

```
[arm79-uboot-1.2.0]# saveenv
Saving Environment to NAND...
Erasing Nand...Writing to Nand... done
[arm79-uboot-1.2.0]#
```

打印信息提示：写到 nand flash，已经 done 了。可是我们还是不放心。主动重启一下，看看是不是真的保存到 nand flash 中了：

```
[arm79-uboot-1.2.0]# saveenv
Saving Environment to NAND...
Erasing Nand...Writing to Nand... done
[arm79-uboot-1.2.0]# reset
```

```
************************************************
*                                              *
*                ARM 技术交流网欢迎您!                *
*                   www.arm79.com                *
*                                              *
************************************************

U-Boot 1.2.0 (Dec 14 2009 - 15:34:28)

U-Boot code: 33F80000 -> 33FA0870  BSS: -> 33FA5BD8
DRAM:  64 MB
Nor  Flash:  2 MB
Nand Flash:  256 MiB
In:      serial
Out:     serial
Err:     serial
[arm79-uboot-1.2.0]# printenv
bootdelay=3
baudrate=38400
netmask=255.255.255.0
serverip=192.168.0.101
dns=192.168.0.1
ipaddr=192.168.0.105
stdin=serial
stdout=serial
stderr=serial

Environment size: 150/131068 bytes
[arm79-uboot-1.2.0]#
```

果然已经保存到了 nand flash 中，因为我们重启后，刚才设置的 IP 地址没有丢掉。

5.8 cmp命令

这是对两段内存地址开始的数据进行比较：

```
[arm79-uboot-1.2.0]# cmp 0x30002000 0x30004000 30
word at 0x30002010 (0x73cc33cc) != word at 0x30004010 (0x7bcc33cc)
Total of 4 words were the same
[arm79-uboot-1.2.0]#
```

最后的参数 30，表示从这两个地址开始的 30 个单元进行比较。

5.9 cp命令

很熟悉的命令，linux 里面我们最经常用 cp 来复制文件到另一个地方了。甚至用它来更改文件的名字。在 u-boot 里，cp 照样是复制的意思，我们看如下演示：

```
[arm79-uboot-1.2.0]# help cp
cp [.b, .w, .l] source target count
    - copy memory

[arm79-uboot-1.2.0]# cp 0x30000000 0x30000100 16
[arm79-uboot-1.2.0]#
```

那如何来验证我们的复制功能是否正常呢？好办，我们已经介绍了 cmp 命令，刚好派上用场：

```
[arm79-uboot-1.2.0]# cmp 0x30000000 0x30000100 16
Total of 22 words were the same
[arm79-uboot-1.2.0]#
```

u-boot 移植成功后，cp 命令必须是对的！我们用 cmp 命令验证后，打印出：Total of 22 words were the same。正确！

5.10 mm命令

MM 命令，和刚才介绍的几个命令，实际上本质是一样的，都是去检测内存，然后打印内存单元的数据。mm 命令是说，按顺序显示指定地址往后的内存单元的内容，地址会自动递增，同时我们还可以去修改这些内存单元内容：

```
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: ea000012 ?
```

我们敲入 mm 0x30000000，想查看内存首地址开始的单元是啥内容，如果确认就是这个内容，我们直接按回车，显示下个单元的内容：

```
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: ea000012 ?
30000004: e59ff014 ?
```

不断地敲回车，地址不断递增显示。上面，我们敲了两个回车，因此打印了 0x30000000 和 0x30000004 的地址单元内容。如果多敲几个回车，显示如下：

```
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: ea000012 ?
30000004: e59ff014 ?
30000008: e59ff014 ?
3000000c: e59ff014 ?
30000010: e59ff014 ?
30000014: e59ff014 ?
```

那我想修改内存中的数据！ok，刚才演示过了，在敲入 mm 0x30000000，回车后，显示：

```
[arm79-uboot-1.2.0]# mm 0x30000000
```

30000000: ea000012 ?

这时候，不敲回车了，我们直接输入 FFFF，大家也试试看，会是啥结果呢？我们来试下，笔者这时候直接敲入 FFFF：

```
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: ea000012 ? ffff
30000004: e59ff014 ?
```

然后按“ctrl + C”，终止当前的命令执行。可以多按几个，反正无所谓。然后，我们使用 mm 命令，查看刚才修改过的内存：

```
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: 0000ffff ?
```

看到了！内存地址 0x30000000 单元的数据，已经被笔者修改成 0x0000ffff，而不是刚开始的：ea000012。好了，这条 mm 命令就介绍到这里，大家回去可以体验一下！

5.11 mtest命令

不知道您用过 windows XP 下的 memtest 小软件吗？可以测试内存条的好坏。它的原理就是不断地去读写内存条的每个地址单元，来验证该内存单元是否是好的。Mtest 命令，就是实现这个功能，呵呵，u-boot 其实蛮强大的！我们来测试 0x30000000，这个地址。看看这个地址是不是好的（当然，肯定是好的，我们刚才一直在使用了）。

```
[arm79-uboot-1.2.0]# mtest 0x30000000
Pattern 00000001    Writing...    Reading...
```

大家在测试这条命令的时候，会发现，这个命令执行之后，Writing... Reading...在不断地跳动，表示不断地进行读写操作。

5.12 mw命令

向内存写入数据的命令，我们 help 一下，发现很简单：

```
[arm79-uboot-1.2.0]# help mw
mw [.b, .w, .l] address value [count]
        - write memory

[arm79-uboot-1.2.0]#
```

直接写地址，然后写入该地址的数据即可。演示吧！我们执行 mw 命令，往地址 0x30000000 写入一个数据，然后用 mm 命令去查看到底有没有写进去：

```
[arm79-uboot-1.2.0]# mw 0x30000000 eeeeee
[arm79-uboot-1.2.0]# mm 0x30000000
30000000: 00eeeeee ?
[arm79-uboot-1.2.0]#
```

执行结果符合我们的预想：我们在地址 0x30000000 写入了 eeeeee，然后 mm 一下，打印出来了 eeeeee 这个数据。实际上，我们修改或者写入内存单元的数据，还有一个更直观的命令 nm，我们来介绍下。

5.13 nm命令

啥也不说了，我们 help 一下，然后直接演示：

```
[arm79-uboot-1.2.0]# help nm
nm [.b, .w, .l] address
    - memory modify, read and keep address

[arm79-uboot-1.2.0]# nm 0x30000000
30000000: 00eeeeee ? 1234567
30000000: 01234567 ? abcdefg
30000000: 00abcdef ?
```

Help nm 执行的结果告诉我们，直接输入地址即可。所以我们想修改 0x30000000 的地址，很简单，直接用 nm 0x30000000。回车之后，出现：

30000000: 00eeeeee ?

这个 00eeeeee 是我们刚才修改的结果。我们不按回车了，直接输入 1234567，回车，OK！又打印出来 30000000 的地址，然后显示 01234567 的数据。随后，我们又把这个单元的数据改为 abcdefg，显示出来，居然少了一个 g，呵呵，看来 u-boot 的代码还是有 bug 的。经典的代码有点 bug，正常，原谅一下！

5.14 md命令

打印指定内存中的数据：

```
[arm79-uboot-1.2.0]# md 0x100
00000100: ee080f17 ee110f10 e3c00c23 e3c00087       ........#.......
00000110: e3800002 e3800a01 ee010f10 e1a0c00e       ................
00000120: eb0000b7 e1a0e00c e1a0f00e 00000000       ................
00000130: 00000000 00000000 00000000 00000000       ................
00000140: e51fd104 e24dd806 e24dd088 e58de000       ......M...M.....
00000150: e14fe000 e58de004 e3a0d013 e169f00d       ..O..........i.
00000160: e1a0e00f e1b0f00e e24dd048 e88d1fff       ........H.M.....
00000170: e51f2134 e2422806 e2422088 e892000c       4!...(B.. B.....
00000180: e28d0048 e28d5034 e1a0100e e885000f       H...4P..........
00000190: e1a0000d eb00010b 00000000 00000000       ................
000001a0: e51fd164 e24dd806 e24dd088 e58de000       d.....M...M.....
000001b0: e14fe000 e58de004 e3a0d013 e169f00d       ..O..........i.
000001c0: e1a0e00f e1b0f00e e24dd048 e88d1fff       ........H.M.....
000001d0: e51f2194 e2422806 e2422088 e892000c       .!...(B.. B.....
000001e0: e28d0048 e28d5034 e1a0100e e885000f       H...4P..........
000001f0: e1a0000d eb0000fc 00000000 00000000       ................
[arm79-uboot-1.2.0]#
```

5.15 bdinfo命令

打印开发板信息，我们在 u-boot 命令行下敲入 bdinfo 之后，出现如下信息：

```
[arm79-uboot-1.2.0]# bdinfo
arch_number = 0x0000016A
env_t       = 0x00000000
boot_params = 0x30000100
DRAM bank   = 0x00000000
-> start    = 0x30000000
-> size     = 0x04000000
ethaddr     = 00:00:00:00:00:00
ip_addr     = 192.168.0.104
baudrate    = 38400 bps
[arm79-uboot-1.2.0]#
```

从上面的打印信息，我们看到了，打印出了开发板配置的很多地址，比如内存开始地址为 0x30000000 大小为 64M，即 0x04000000。显示 MAC 地址的时候，我们发现，怎么会是全零呢？ethaddr = 00:00:00:00:00:00 很明显，这是我们没有配置 MAC 地址，在源代码中，您会发现，由于开发板上只有 DM9000A 芯片，我们的 MAC 地址没有 EEPROM 可读，只能自己去设置一个 MAC，保证在局域网内不和其他网卡的 MAC 冲突即可。

5.16 ping命令

从这条命令开始，我们后续慢慢会接触到网络相关的比较常用的命令，在附录 E 中，我们给出了一些网络协议，希望大家有空也去了解下。我们先看 ping 命令：

```
[arm79-uboot-1.2.0]# ping 192.168.0.101
dm9000 i/o: 0x20000300, id: 0x90000a46
MAC: 08:ef:3e:26:0a:5b
host 192.168.0.101 is alive
[arm79-uboot-1.2.0]#
```

我们的开发板 IP 是 192.168.0.104，PC 的 IP 是 192.168.0.101，因此用的是 ping 192.168.0.101，ping 完结果，我们看到了 host 192.168.0.101 is alive，说明开发板与主机已经 ping 通。

5.17 TFTP命令

事实上，我们进行项目开发的时候，更经常使用 TFTP 来下载一些升级版本，而不是 usb 下载。TFTP 命令，晕倒，我突然忘了它的参数形式，没关系，我们先 help 一下：

```
[arm79-uboot-1.2.0]# help tftp
tftpboot [loadAddress] [bootfilename]
```

```
[arm79-uboot-1.2.0]#
```

完整的命令实际上是 tftpboot，后面加地址和 bin 文件的名字。我们这里可以做一个试验：用 tftpboot 命令，把已经编译好的 u-boot.bin 文件重新下载进去，看看是啥结果。注意，实际上，我们在用 tftp 协议，借网口来使 u-boot 自己升级自己-----很爽！我们可以自己升级自己的版本了，可以抛弃 usb 下载了！

```
[arm79-uboot-1.2.0]# tftp 0x30000000 u-boot3.bin
dm9000 i/o: 0x20000300, id: 0x90000a46
MAC: 08:ef:3e:26:0a:5b
TFTP from server 192.168.0.101; our IP address is 192.168.0.104
Filename 'u-boot3.bin'.
Load address: 0x30000000
Loading: T ##############################
done
Bytes transferred = 133232 (20870 hex)
[arm79-uboot-1.2.0]#
```

从打印信息，我们可以看到，服务器是 192.168.0.101，也就是我们的 PC，下载到 our IP：192.168.0.104。下载地址是 0x30000000，就是我们内存首地址。下载的字节长度是 133232 字节，大概 130KB。我们已经把自己编译的 u-boot 下载到内存里了，那怎么知道是对是错呢？？需要验证。这就是下节介绍的 go 命令。

5.18 go命令

GO 命令，功能是直接跳转到可执行文件的入口地址，执行可执行文件。刚才我们下载了 u-boot.bin 文件到 0x30000000 地址去了，那么现在，我们 go 一下，执行这个 u-boot.bin 文件，看看会是啥样的效果，为了查看方便，笔者把刚才的 tftpboot 下载的打印信息也贴出来了，以便您确认：确实是从刚才 tftp 下载来的 u-boot.bin 文件：

```
[arm79-uboot-1.2.0]# tftp 0x30000000 u-boot3.bin
dm9000 i/o: 0x20000300, id: 0x90000a46
MAC: 08:ef:3e:26:0a:5b
TFTP from server 192.168.0.101; our IP address is 192.168.0.104
Filename 'u-boot3.bin'.
Load address: 0x30000000
Loading: T ##############################
done
Bytes transferred = 133232 (20870 hex)
[arm79-uboot-1.2.0]# go 0x30000000
## Starting application at 0x30000000 ...


**************************************************
*                                                *
```

```
*                    ARM 技术交流网欢迎您!                    *
*                      www.arm79.com                          *
*                                                             *
***************************************************

U-Boot 1.2.0 (Dec 14 2009 - 15:34:28)

U-Boot code: 33F80000 -> 33FA0870  BSS: -> 33FA5BD8
DRAM:  64 MB
Nor  Flash:  2 MB
Nand Flash:  256 MiB
In:     serial
Out:    serial
Err:    serial
[arm79-uboot-1.2.0]#
```

哈哈，是不是很有成就感!! 我们 go 到 0x30000000 之后，就又开始执行我们的 u-boot 了。但是这里请注意，这个 u-boot 和刚才我们的启动开发板执行的 u-boot 是不一样的。这个 u-boot 是直接下载到内存执行。我们启动开发板时运行的 u-boot，则是事先烧写在 flash 里面的。笔者烧写在 nor flash 里。

现在，我们顺便来执行一下 reset 命令，重启一下系统，然后继续我们的 u-boot 使用之旅吧！事实上，不用重启，照样可用!!!

5.19 reset命令

我们 reset 一下，重新启动开发板，看看效果。开发板重启后，再一次进入到我们熟悉的“ARM 技术交流网欢迎您!”的界面。

```
 [arm79-uboot-1.2.0]# reset
***************************************************
*                                                 *
*                    ARM 技术交流网欢迎您!                    *
*                      www.arm79.com                          *
*                                                             *
***************************************************

U-Boot 1.2.0 (Dec 14 2009 - 15:34:28)

U-Boot code: 33F80000 -> 33FA0870  BSS: -> 33FA5BD8
DRAM:  64 MB
Nor  Flash:  2 MB
Nand Flash:  256 MiB
In:     serial
Out:    serial
```

```
Err:   serial
[arm79-uboot-1.2.0]#
```

5.20 set命令

这个 set 命令和 setenv 命令可有本质的区别哈~！set 命令，使得 u-boot 命令更灵活。它直接可以用我们自己好记的单词来操作一些命令，举个例子：假设我们感觉 printenv，这个命令好难记，可能一不小心会写错哦~，没关系，用 set 吧：

```
[arm79-uboot-1.2.0]# set aa ping 192.168.0.101
```

这条命令就相当于告诉 u-boot，我想把 ping 192.168.0.101 用 aa 代替，以后我想 ping 这个 IP，我只要 run aa 即可。不要再敲 ping 192.168.0.101 这么复杂的命令了。实际上，这也是一个环境变量的范围哦~我们可以 printenv 一下就知道了：

```
[arm79-uboot-1.2.0]# set aa ping 192.168.0.101
[arm79-uboot-1.2.0]# printenv
bootdelay=3
baudrate=38400
netmask=255.255.255.0
serverip=192.168.0.101
dns=192.168.0.1
stdin=serial
stdout=serial
stderr=serial
ipaddr=192.168.0.101
aa=ping 192.168.0.101

Environment size: 198/131068 bytes
[arm79-uboot-1.2.0]#
```

呵呵，它已经进入了 printenv 的势力范围了。

5.21 run命令

Run 命令，实际上刚才已经提过了。它会运行 u-boot 已经定义好的命令。比如刚才我们用 set 定义好 aa 这个命令。OK，我们来执行 aa：

```
[arm79-uboot-1.2.0]# run aa
dm9000 i/o: 0x20000300, id: 0x90000a46
MAC: 08:ef:3e:26:0a:5b
host 192.168.0.101 is alive
[arm79-uboot-1.2.0]#
```

u-boot 很牛 X 吧，呵呵。执行 run aa，实际上执行的是先前定义好的 ping 192.168.0.101。OK，u-boot 命令暂且介绍到这里。这是 v1.0 版本，后续还会有 v1.2，v1.3 版本和大家见面！

附　录：

附A、U-Boot的lds文件详解

（声明：摘自百度空间，参考文献中给出了此文的网址）

对于.lds 文件，决定一个可执行程序的各个段的存储位置，以及入口地址，这也是链接定位的作用。这里以 u-boot 的 lds 为例说明 uboot 的链接过程。

首先看一下 GNU 官方网站上对.lds 文件形式的完整描述：

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

secname 和 contents 是必须的，前者用来命名这个段，后者用来确定代码中的什么部分放在这个段，以下是对这个描述中的一些关键字的解释。

1、secname：段名

2、contents：决定哪些内容放在本段，可以是整个目标文件，也可以是目标文件中的某段（代码段、数据段等）

3、start：是段的重定位地址，本段连接（运行）的地址，如果代码中有位置无关指令，程序运行时这个段必须放在这个地址上。start 可以用任意一种描述地址的符号来描述。

4、AT（ldadr）：定义本段存储（加载）的地址，如果不使用这个选项，则加载地址等于运行地址，通过这个选项可以控制各段分别保存于输出文件中不同的位置。

例：

```
/* nand.lds */
SECTIONS {
firtst 0x00000000 : { head.o init.o }
second 0x30000000 : AT(4096) { main.o }
}
```

以上，head.o 放在 0x00000000 地址开始处，init.o 放在 head.o 后面，他们的运行地址也是 0x00000000，即连接和存储地址 相同（没有 AT 指定）；main.o 放在 4096（0x1000，是 AT 指定的，存储地址）开始处，但它的运行地址在 0x30000000，运行之前需要 从 0x1000（加载地址处）复制到 0x30000000（运行地址处），此过程也就需要读取 flash，把程序拷贝到相应位置才能运行。这就是存储地址和运行地址的不同，称为加载时域和运行时域，可以在.lds 连接脚本文件中分别指定。

编写好的.lds 文件，在用 arm-linux-ld 连接命令时带-Tfilename 来调用执行，如
arm-linux-ld －Tnand.lds x.o y.o －o xy.o。也用-Ttext 参数直接指定连接地址，如
arm-linux-ld －Ttext 0x30000000 x.o y.o －o xy.o。
既然程序有了两种地址，就涉及到一些跳转指令的区别。
ARM 汇编中，常有两种跳转方法：b 跳转指令、ldr 指令向 PC 赋值。
要特别注意这两条指令的意思：
（1）b step：b 跳转指令是相对跳转，依赖当前 PC 的值，偏移量是通过该指令本身的 bit[23:0]算出来的，这使得使用 b 指令的程序不依赖于要跳到的代码的位置，只看指令本身。
（2）ldr pc, =step ：该指令是一个伪指令编译后会生成以下代码：

```
ldr pc, 0x30008000
<0x30008000>
                step
```

是从内存中的某个位置（step）读出数据并赋给 PC，同样依赖当前 PC 的值，但是偏移量是 step 的连接地址（运行时的地址），所以可以用它实现从 Flash 到 RAM 的程序跳转。
（3） 此外，有必要回味一下 adr 伪指令，U-boot 中那段 relocate 代码就是通过 adr 实现当前程序是在 RAM 中还是 flash 中：

```
relocate:                   /* 把 U-Boot 重新定位到 RAM */
        adr r0, _start      /* r0 是代码的当前位置 */
```

/* adr 伪指令，汇编器自动通过当前 PC 的值算出这条指令中“_start"的值，执行到_start 时 PC 的值放到 r0 中：
当此段在 flash 中执行时 r0 = _start = 0；当此段在 RAM 中执行时_start = _TEXT_BASE(在 board/smdk2410/config.mk 中指定的值为 0x33F80000，即 u-boot 在把代码拷贝到 RAM 中去 执行的代码段的开始) */

```
    ldr r1, _TEXT_BASE   /* 测试判断是从 Flash 启动，还是 RAM */
/* 此句执行的结果 r1 始终是 0x33FF80000，因为此值是链接指定的 */
    cmp r0, r1           /* 比较 r0 和 r1，调试的时候不要执行重定位 */
```

结合 u-boot.lds 谈谈连接脚本

```
OUTPUT_FORMAT("elf32&shy;littlearm",        "elf32&shy;littlearm",
"elf32&shy;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(_start)
                            ;指定输出可执行文件的起始代码段为_start.
SECTIONS
{
        . = 0x00000000                  ; 定位当前地址为 0 地址
        . = ALIGN(4)                    ; 代码以 4 字节对齐
```

```
        .text :                                 ; 指定代码段
        {
           cpu/arm920t/start.o (.text)   ; 代码的第一个代码部分
          *(.text)                              ; 其它代码部分
        }
        . = ALIGN(4)
        .rodata : { *(.rodata) }          ; 指定只读数据段
        . = ALIGN(4);
        .data : { *(.data) }                ; 指定读/写数据段
        . = ALIGN(4);
        .got : { *(.got) }                   ; 指定 got 段, got 段式是 uboot
自定义的一个段, 非标准段
         __u_boot_cmd_start = .       ; 把__u_boot_cmd_start 赋值
为当前位置, 即起始位置
        .u_boot_cmd : { *(.u_boot_cmd) }                ; 指 定
u_boot_cmd 段, uboot 把所有的 uboot 命令放在该段.
         __u_boot_cmd_end = .              ;把__u_boot_cmd_end
赋值为当前位置,即结束位置
        . = ALIGN(4);
         __bss_start = .                        ; 把__bss_start 赋值为
当前位置,即 bss 段的开始位置
        .bss : { *(.bss) }                      ; 指定 bss 段
         _end = .                               ; 把_end 赋值为当前位
置,即 bss 段的结束位置
}
```

附B、ARM GCC 内嵌（inline）汇编手册

关于这篇文档

对于基于 ARM 的 RISC 处理器，GNU C 编译器提供了在 C 代码中内嵌汇编的功能。这种非常酷的特性提供了 C 代码没有的功能，比如手动优化软件关键部分的代码、使用相关的处理器指令。

这里设想了读者是熟练编写 ARM 汇编程序读者，因为该片文档不是 ARM 汇编手册。同样也不是 C 语言手册。

这篇文档假设使用的是 GCC 4 的版本，但是对于早期的版本也有效。

GCC asm 声明

让我们以一个简单的例子开始。就像 C 中的声明一样，下面的声明代码可能出现在你的代码中。

```
/* NOP 例子 */
```

```
asm("mov r0,r0");
```

该语句的作用是将 r0 移动到 r0 中。换句话讲他并不干任何事。典型的就是 NOP 指令，作用就是短时的延时。

请接着阅读和学习这篇文档，因为该声明并不像你想象的和其他的 C 语句一样。内嵌汇编使用汇编指令就像在纯汇编程序中使用的方法一样。可以在一个 asm 声明中写多个汇编指令。但是为了增加程序的可读性，最好将每一个汇编指令单独放一行。

```
asm(
"mov r0, r0\n\t"
"mov r0, r0\n\t"
"mov r0, r0\n\t"
"mov r0, r0"
);
```

换行符和制表符的使用可以使得指令列表看起来变得美观。你第一次看起来可能有点怪异，但是当 C 编译器编译 C 语句的是候，它就是按照上面（换行和制表）生成汇编的。到目前为止，汇编指令和你写的纯汇编程序中的代码没什么区别。但是对比其它的 C 声明，asm 的常量和寄存器的处理是不一样的。通用的内嵌汇编模版是这样的。

```
asm(code : output operand list : input operand list : clobber list);
```

汇编和 C 语句这间的联系是通过上面 asm 声明中可选的 output operand list 和 input operand list。Clobber list 后面再讲。

下面是将 C 语言的一个整型变量传递给汇编，逻辑左移一位后在传递给 C 语言的另外一个整型变量。

```
/* Rotating bits example */
asm("mov %[result], %[value], ror #1" : [result] "=r" (y) : [value] "r" (x));
```

每一个 asm 语句被冒号（:）分成了四个部分。

- 汇编指令放在第一部分中的“”中间。

```
"mov %[result], %[value], ror #1"
```

- 接下来是冒号后的可选择的 output operand list，每一个条目是由一对[]（方括号）和被他包括的符号名组成，它后面跟着限制性字符串，再后面是圆括号和它括着的 C 变量。这个例子中只有一个条目。

```
[result] "=r" (y)
```

- 接着冒号后面是输入操作符列表，它的语法和输入操作列表一样

```
[value] "r" (x)
```

- 破坏符列表，在本例中没有使用

就像上面的 NOP 例子，asm 声明的 4 个部分中，只要最尾部没有使用的部分都可以省略。但是有有一点要注意的是，上面的 4 个部分中只要后面的还要使用，前面的部分没有使用也不能省略，必须空但是保留冒号。下面的一个例子就是设置 ARM Soc 的 CPSR 寄存器，它有 input 但是没有 output operand。

```
asm("msr cpsr,%[ps]" : : [ps]"r"(status))
```

即使汇编代码没有使用，代码部分也要保留空字符串。下面的例子使用了一个特别的破坏符，目的就是告诉编译器内存被修改过了。这里的破坏符在下面的优化部分在讲解。

```
asm(""::"memory");
```

为了增加代码的可读性，你可以使用换行，空格，还有 C 风格的注释。

```
asm("mov %[result], %[value], ror #1"

    : [result]"=r" (y) /* Rotation result. */
    : [value]"r" (x) /* Rotated value. */
    : /* No clobbers */
);
```

在代码部分%后面跟着的是后面两个部分方括号中的符号，它指的是相同符号操作列表中的一个条目。

%[result]表示第二部分的 C 变量 y，%[value]表示三部分的 C 变量 x；

符号操作符的名字使用了独立的命名空间。这就意味着它使用的是其他的符号表。简单一点就是说你不必关心使用的符号名在 C 代码中已经使用了。在早期的 C 代码中，循环移位的例子必须要这么写：

```
asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value))
```

在汇编代码中操作数的引用使用的是%后面跟一个数字，%1 代表第一个操作数，%2 代码第二个操作数，往后的类推。这个方法目前最新的编译器还是支持的。但是它不便于维护代码。试想一下，你写了大量的汇编指令的代码，要是你想插入一个操作数，那么你就不得不从新修改操作数编号。

优化 C 代码

有两种情况决定了你必须使用汇编。1st，C 限制了你更加贴近底层操作硬件，比如，C 中没有直接修改程序状态寄存器（PSR）的声明。2nd 就是要写出更加优

化的代码。毫无疑问 GNU C 代码优化器做的很好，但是他的结果和我们手工写的汇编代码相差很远。

这一部分有一点很重要，也是被别人忽视最多的就是：我们在 C 代码中通过内嵌汇编指令添加的汇编代码，也是要被 C 编译器的优化器处理的。让我们下面做个试验来看看吧。

下面是代码实例。

```
1 int test(int x){
2     int y;
3     asm("mov %[result], %[value], ror #1"
4                 :[result]"=r"(y)
5                 :[value]"r"(x)
6         );
7     return y;
8 }
```

```
bigtree@just:~/embedded/basic-C$ arm-linux-gcc -c test.c
bigtree@just:~/embedded/basic-C$ arm-linux-objdump -D
test.o
```

```
14:   e51b3010        ldr     r3, [fp, #-16]
18:   e1a030e3        mov     r3, r3, ror #1
1c:   e50b3014        str     r3, [fp, #-20]
```

编译器选择 r3 作为循环移位使用。它也完全可以选择为每一个 C 变量分配寄存器。Load 或者 store 一个值并不显式的进行。下面是其它编译器的编译结果。

```
E420A0E1 mov r2, r4, ror #1 @ y, x
```

编译器为每一个操作数选择一个相应的寄存器，将操作过的值 cache 到 r4 中，然后传递该值到 r2 中。这个过程你能理解不？

有的时候这个过程变得更加糟糕。有时候编译器甚至完全抛弃你嵌入的汇编代码。C 编译器的这种行为，取决于代码优化器的策略和嵌入汇编所处的上下文。如果在内嵌汇编语句中不使用任何输出部分，那么 C 代码优化器很有可能将该内嵌语句完全删除。比如 NOP 例子，我们可以使用它作为延时操作，但是对于编译器认为这影响了程序的执行速速，认为它是没有任何意义的。

上面的解决方法还是有的。那就是使用 volatile 关键字。它的作用就是禁止优化器优化。将 NOP 例子修改过后如下：

```
/* NOP example, revised */

asm volatile("mov r0, r0");
```

下面还有更多的烦恼等着我们。一个设计精细的优化器可能重新排列代码。看下面的代码：

```
i++;
```

```
if (j == 1)
x += 3;
i++;
```

优化器肯定是要从新组织代码的，两个 i++并没有对 if 的条件产生影响。更进一步的来讲，i 的值增加 2，仅仅使用一条 ARM 汇编指令。因而代码要重新组织如下：

```
if (j == 1)
   x += 3;
i += 2;
```

这样节省了一条 ARM 指令。结果是：这些操作并没有得到许可。

这些将对你的代码产生很到的影响，这将在下面介绍。下面的代码是 c 乘 b，其中 c 和 b 中的一个或者两个可能会被中断处理程序修改。进入该代码前先禁止中断，执行完该代码后再开启中断。

```
asm volatile("mrs r12, cpsr\n\t"
   "orr r12, r12, #0xC0\n\t"
   "msr cpsr_c, r12\n\t" ::: "r12", "cc");
c *= b; /* This may fail. */
asm volatile("mrs r12, cpsr\n"
   "bic r12, r12, #0xC0\n"
   "msr cpsr_c, r12" ::: "r12", "cc");
```

但是不幸的是针对上面的代码，优化器决定先执行乘法然后执行两个内嵌汇编，或相反。这样将会使得我们的代码变得毫无意义。

我们可以使用 clobber list 帮忙。上面例子中的 clobber list 如下：

```
"r12", "cc"
```

上面的 clobber list 将会将向编译器传达如下信息，修改了 r12 和程序状态寄存器的标志位。Btw，直接指明使用的寄存器，将有可能阻止了最好的优化结果。通常你只要传递一个变量，然后让编译器自己选择适合的寄存器。另外*寄存器名*，*cc*（condition registor 状态寄存器标志位），*memory* 都是在 clobber list 上有效的关键字。它用来向编译器指明，内嵌汇编指令改变了内存中的值。这将强迫编译器在执行汇编代码前存储所有缓存的值，然后在执行完汇编代码后重新加载该值。这将保留程序的执行顺序，因为在使用了带有 *memory* clobber 的 asm 声明后，所有变量的内容都是不可预测的。

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" :: : "r12", "cc", "memory");
c *= b; /* This is safe. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc", "memory");
```

使所有的缓存的值都无效，只是局部最优（suboptimal）。你可以有选择性的添加 dummy operand 来人工添加依赖。

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" : "=X" (b) :: "r12", "cc");
c *= b; /* This is safe. */
asm volatile("mrs r12
```

上面的第一个 asm 试图修改变量先 b，第二个 asm 试图修改 c。这将保留三个语句的执行顺序，而不要使缓存的变量无效。

理解优化器对内嵌汇编的影响很重要。如果你读到这里还是云里雾里，最好是在看下个主题之前再把这段文章读几遍^_^。

Input and output operands

前面我们学到，每一个 input 和 output operand，由被方括号[]中的符号名，限制字符串，圆括号中的 C 表达式构成。

这些限制性字符串有哪些，为什么我们需要他们？你应该知道每一条汇编指令只接受特定类型的操作数。例如：跳转指令期望的跳转目标地址。不是所有的内存地址都是有效的。因为最后的 opcode 只接受 24 位偏移。但矛盾的是跳转指令和数据交换指令都希望寄存器中存储的是 32 位的目标地址。在所有的例子中，C 传给 operand 的可能是函数指针。所以面对传给内嵌汇编的常量、指针、变量，编译器必须要知道怎样组织到汇编代码中。

对于 ARM 核的处理器，GCC 4 提供了一下的限制。

Constraint	Usage in ARM state	Usage in Thumb state
f	Floating point registers f0 .. f7	Not available
h	Not available	Registers r8..r15
G	Immediate floating point constant	Not available

H	Same a G, but negated	Not available
I	Immediate value in data processing instructions e.g. ORR R0, R0, #operand	Constant in the range 0 .. 255 e.g. SWI operand
J	Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand
l	Same as r	Registers r0..r7 e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand
O	Not available	Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand
r	General register r0 .. r15 e.g. SUB operand1, operand2, operand3	Not available
w	Vector floating point registers s0 .. s31	Not available
X	Any operand	

限制字符可能要单个 modifier 指示。要是没有 modifier 指示的默认为 read-only operand。

Modifier	Specifies
=	Write-only operand, usually used for all output operands
+	Read-write operand, must be listed as an output operand
&	A register that should be used for output only

Output operands 必须为 write-only，相应 C 表达式的值必须是左值。Input operands 必须为 read-only。C 编译器是没有能力做这个检查。

比较严格的规则是：不要试图向 input operand 写。但是如果你想要使用相同的 operand 作为 input 和 output。限制性 modifier（+）可以达到效果。例子如下：

```
asm("mov %[value], %[value], ror #1" : [value] "+r" (y))
```

和上面例子不一样的是，最后的结果存储在 input variable 中。

可能 modifier + 不支持早期的编译器版本。庆幸的是这里提供了其他解决办法，该方法在最新的编译器中依然有效。对于 input operators 有可能使用单一的数字 n 在限制字符串中。使用数字 n 可以告诉编译器使用的第 n 个 operand，operand 都是以 0 开始计数。下面是例子：

```
asm("mov %0, %0, ror #1" : "=r" (value) : "0" (value))
```

限制性字符串“0”告诉编译器，使用和第一个 output operand 使用同样 input register。

请注意，在相反的情况下不会自动实现。如果我没告诉编译器那样做，编译器也有可能为 input 和 output 选择相同的寄存器。第一个例子中就为 input 和 output 选择了 r3。

在多数情况下这没有什么，但是如果在 input 使用前 output 已经被修改过了，这将是致命的。在 input 和 output 使用不同寄存器的情况下，你必须使用&modifier 来限制 output operand。下面是代码示例：

```
asm volatile("ldr %0, [%1]" "\n\t"
             "str %2, [%1, #4]" "\n\t"
             : "=&r" (rdv)
             : "r" (&table), "r" (wdv)
             : "memory");
```

在以张表中读取一个值然后在写到该表的另一个位置。

其他

内嵌汇编作为预处理宏

要是经常使用使用部分汇编，最好的方法是将它以宏的形式定义在头文件中。使用该头文件在严格的 ANSI 模式下会出现警告。为了避免该类问题，可以使用__asm__代替 asm，__volatile__代替 volatile。这可以等同于别名。下面就是个例程：

```
#define BYTESWAP(val) \
  __asm__ __volatile__ ( \
    "eor r3, %1, %1, ror #16\n\t" \
    "bic r3, r3, #0x00FF0000\n\t" \
```

```
        "mov %0, %1, ror #8\n\t" \
        "eor %0, %0, r3, lsr #8" \
        : "=r" (val) \
        : "0"(val) \
        : "r3", "cc" \
    );
```

C 桩函数

宏定义包含的是相同的代码。这在大型 routine 中是不可以接受的。这种情况下最好定义个桩函数。

```
unsigned long ByteSwap(unsigned long val)
{
asm volatile (
        "eor r3, %1, %1, ror #16\n\t"
        "bic r3, r3, #0x00FF0000\n\t"
        "mov %0, %1, ror #8\n\t"
        "eor %0, %0, r3, lsr #8"
        : "=r" (val)
        : "0"(val)
        : "r3"
);
return val;
}
```

替换 C 变量的符号名

默认的情况下，GCC 使用同函数或者变量相同的符号名。你可以使用 asm 声明，为汇编代码指定一个不同的符号名

```
unsigned long value asm("clock") = 3686400
```

这个声明告诉编译器使用了符号名 clock 代替了具体的值。

替换 C 函数的符号名

为了改变函数名，你需要一个原型声明，因为编译器不接受在函数定义中出现 asm 关键字。

```
extern long Calc(void) asm ("CALCULATE")
```

调用函数 *calc()*将会创建调用函数 *CALCULATE* 的汇编指令。

强制使用特定的寄存器

局部变量可能存储在一个寄存器中。你可以利用内嵌汇编为该变量指定一个特定的寄存器。

```
void Count(void) {
register unsigned char counter asm("r3");

... some code...
asm volatile("eor r3, r3, r3");
... more code...
}
```

汇编指令“eor r3, r3, r3”，会将 r3 清零。Waring：该例子在到多数情况下是有问题的，因为这将和优化器相冲突。因为 GCC 不会预留其它寄存器。要是优化器认为该变量在以后一段时间没有使用，那么该寄存器将会被再次使用。但是编译器并没有能力去检查是否和编译器预先定义的寄存器有冲突。如果你用这种方式指定了太多的寄存器，编译器将会在代码生成的时候耗尽寄存器的。

临时使用寄存器

如果你使用了寄存器，而你没有在 input 或 output operand 传递，那么你就必须向编译器指明这些。下面的例子中使用 r3 作为 scratch 寄存器，通过在 clobber list 中写 r3，来让编译器得知使用该寄存器。由于 ands 指令跟新了状态寄存器的标志位，使用 cc 在 clobber list 中指明。

```
asm volatile(
    "ands r3, %1, #3" "\n\t"
    "eor %0, %0, r3" "\n\t"
    "addne %0, #4"
    : "=r" (len)
    : "0" (len)
    : "cc", "r3"
```

```
);
```

最好的方法是使用桩函数并且使用局部临时变量。

寄存器的用途

比较好的方法是分析编译后的汇编列表，并且学习 C 编译器生成的代码。下面的列表是编译器将 ARM 核寄存器的典型用途，知道这些将有助于理解代码。

Register	Alt. Name	Usage
r0	a1	First function argument Integer function result Scratch register
r1	a2	Second function argument Scratch register
r2	a3	Third function argument Scratch register
r3	a4	Fourth function argument Scratch register
r4	v1	Register variable
r5	v2	Register variable
r6	v3	Register variable
r7	v4	Register variable
r8	v5	Register variable
r9	v6 rfp	Register variable Real frame pointer
r10	sl	Stack limit
r11	fp	Argument pointer
r12	ip	Temporary workspace
r13	sp	Stack pointer
r14	lr	Link register Workspace
r15	pc	Program counter

附C、typedef用法小结

在 C 语言的情况下，与 C++稍有出入。

这两天在看程序的时候,发现很多地方都用到 typedef,在结构体定义,还有一些数组等地方都大量的用到.但是有些地方还不是很清楚,今天下午,就想好好研究一下.上网搜了一下,有不少资料.归纳一下:

来源一:Using typedef to Curb Miscreant Code

Typedef 声明有助于创建平台无关类型，甚至能隐藏复杂和难以理解的语法。不管怎样，使用 typedef 能为代码带来意想不到的好处，通过本文你可以学习用 typedef 避免缺欠，从而使代码更健壮。

typedef 声明，简称 typedef，为现有类型创建一个新的名字。比如人们常常使用 typedef 来编写更美观和可读的代码。所谓美观，意指 typedef 能隐藏笨拙的语法构造以及平台相关的数据类型，从而增强可移植性和以及未来的可维护性。本文下面将竭尽全力来揭示 typedef 强大功能以及如何避免一些常见的陷阱。

如何创建平台无关的数据类型，隐藏笨拙且难以理解的语法?

使用 typedefs 为现有类型创建同义字。

定义易于记忆的类型名

typedef 使用最多的地方是创建易于记忆的类型名，用它来归档程序员的意图。类型出现在所声明的变量名字中，位于 "typedef" 关键字右边。例如：

```
typedef int size;
```

此声明定义了一个 int 的同义字，名字为 size。注意 typedef 并不创建新的类型。它仅仅为现有类型添加一个同义字。你可以在任何需要 int 的上下文中使用 size：

```
void measure(size * psz);
size array[4];
size len = file.getlength();
std::vector vs;
```

typedef 还可以掩饰符合类型，如指针和数组。例如，你不用象下面这样重复定义有 81 个字符元素的数组：

```
char line[81];
char text[81];
```

定义一个 typedef，每当要用到相同类型和大小的数组时，可以这样：

```
typedef char Line[81];
Line text, secondline;
getline(text);
```

同样，可以象下面这样隐藏指针语法：

```
typedef char * pstr;
int mystrcmp(pstr, pstr);
```

这里将带我们到达第一个 typedef 陷阱。标准函数 strcmp()有两个‘const char *'类型的参数。因此，它可能会误导人们象下面这样声明 mystrcmp()：

```
int mystrcmp(const pstr, const pstr);
```

这是错误的，按照顺序，‘const pstr'被解释为‘char * const'（一个指向 char 的常量指针），而不是‘const char *'（指向常量 char 的指针）。这个问题很容易解决：

typedef const char * cpstr;

int mystrcmp(cpstr, cpstr); // 现在是正确的

记住：不管什么时候，只要为指针声明 typedef，那么都要在最终的 typedef 名称中加一个 const，以使得该指针本身是常量，而不是对象。

代码简化

上面讨论的 typedef 行为有点像 #define 宏，用其实际类型替代同义字。不同点是 typedef 在编译时被解释，因此让编译器来应付超越预处理器能力的文本替换。例如：

typedef int (*PF) (const char *, const char *);

这个声明引入了 PF 类型作为函数指针的同义字，该函数有两个 const char * 类型的参数以及一个 int 类型的返回值。如果要使用下列形式的函数声明，那么上述这个 typedef 是不可或缺的：

PF Register(PF pf)（）;

Register() 的参数是一个 PF 类型的回调函数，返回某个函数的地址，其署名与先前注册的名字相同。做一次深呼吸。下面我展示一下如果不用 typedef，我们是如何实现这个声明的：

int (*Register (int (*pf)(const char *, const char *)))

(const char *, const char *);

很少有程序员理解它是什么意思，更不用说这种费解的代码所带来的出错风险了。显然，这里使用 typedef 不是一种特权，而是一种必需。持怀疑态度的人可能会问："OK，有人还会写这样的代码吗？"，快速浏览一下揭示 signal()函数的头文件 ，一个有同样接口的函数。

typedef 和存储类关键字（storage class specifier）

这种说法是不是有点令人惊讶，typedef 就像 auto，extern，mutable，static，和 register 一样，是一个存储类关键字。这并不是说 typedef 会真正影响对象的存储特性；它只是说在语句构成上，typedef 声明看起来象 static，extern 等类型的变量声明。下面将带到第二个陷阱：

typedef register int FAST_COUNTER; // 错误

编译通不过。问题出在你不能在声明中有多个存储类关键字。因为符号 typedef 已经占据了存储类关键字的位置，在 typedef 声明中不能用 register（或任何其它存储类关键字）。

促进跨平台开发

typedef 有另外一个重要的用途，那就是定义机器无关的类型，例如，你可以定义一个叫 REAL 的浮点类型，在目标机器上它可以 i 获得最高的精度：

typedef long double REAL;

在不支持 long double 的机器上，该 typedef 看起来会是下面这样：

typedef double REAL;

并且,在连 double 都不支持的机器上,该 typedef 看起来会是这样:、

typedef float REAL;

你不用对源代码做任何修改，便可以在每一种平台上编译这个使用 REAL 类型的应用程序。唯一要改的是 typedef 本身。在大多数情况下，甚至这个微小的变动完全都可以通过奇妙的条件编译来自动实现。不是吗?标准库广泛地使用 typedef 来创建这样的平台无关类型：size_t，ptrdiff 和 fpos_t 就是其中的例子。此外，象 std::string 和 std::ofstream 这样的 typedef 还隐藏了长长的，难以理解的模板特化语法，例如：basic_string，allocator> 和 basic_ofstream>。

作者简介

Danny Kalev 是一名通过认证的系统分析师，专攻 C++ 和形式语言理论的软件工程师。1997 年到 2000 年期间,他是 C++ 标准委员会成员。最近他以优异成绩完成了他在普通语言学研究方面的硕士论文。业余时间他喜欢听古典音乐，阅读维多利亚时期的文学作品，研究 Hittite、Basque 和 Irish Gaelic 这样的自然语言。其它兴趣包括考古和地理。Danny 时常到一些 C++ 论坛并定期为不同的 C++ 网站和杂志撰写文章。他还在教育机构讲授程序设计语言和应用语言课程。

来源二：(http://www.ccfans.net/bbs/dispbbs.asp?boardid=30&id=4455)

C 语言中 typedef 用法

1. 基本解释

typedef 为 C 语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int,char 等）和自定义的数据类型（struct 等）。

在编程中使用 typedef 目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

至于 typedef 有什么微妙之处，请你接着看下面对几个问题的具体阐述。

2. typedef & 结构的问题

当用下面的代码定义一个结构时，编译器报了一个错误，为什么呢？莫非 C 语言不允许在结构中包含指向它自己的指针吗？请你先猜想一下，然后看下文说明：

```
typedef struct tagNode
{
char *pItem;
```

```
pNode  pNext;
} *pNode;
```

答案与分析：

1、typedef 的最简单使用

```
typedef  long  byte_4;
```

给已知数据类型 long 起个新名字，叫 byte_4。

2、 typedef 与结构结合使用

```
typedef  struct  tagMyStruct
{
int  iNum;
long  lLength;
} MyStruct;
```

这语句实际上完成两个操作：

1) 定义一个新的结构类型

```
struct  tagMyStruct
{
int  iNum;
long  lLength;
};
```

分析：tagMyStruct 称为“tag”，即“标签”，实际上是一个临时名字，struct 关键字和 tagMyStruct 一起，构成了这个结构类型，不论是否有 typedef，这个结构都存在。

我们可以用 struct tagMyStruct varName 来定义变量，但要注意，使用 tagMyStruct varName 来定义变量是不对的，因为 struct 和 tagMyStruct 合在一起才能表示一个结构类型。

2) typedef 为这个新的结构起了一个名字，叫 MyStruct。

```
typedef  struct  tagMyStruct  MyStruct;
```

因此，MyStruct 实际上相当于 struct tagMyStruct，我们可以使用 MyStruct varName 来定义变量。

答案与分析

C 语言当然允许在结构中包含指向它自己的指针，我们可以在建立链表等数据结构的实现上看到无数这样的例子，上述代码的根本问题在于 typedef 的应用。

根据我们上面的阐述可以知道：新结构建立的过程中遇到了 pNext 域的声明，类型是 pNode，要知道 pNode 表示的是类型的新名字，那么在类型本身还没有建立完成的时候，这个类型的新名字也还不存在，也就是说这个时候编译器根本不认识 pNode。

解决这个问题的方法有多种：

1)、

```
typedef struct tagNode
{
char *pItem;
struct tagNode *pNext;
} *pNode;
```

2)、

```
typedef struct tagNode *pNode;
struct tagNode
{
char *pItem;
pNode pNext;
};
```

注意：在这个例子中，你用 typedef 给一个还未完全声明的类型起新名字。C 语言编译器支持这种做法。

3)、规范做法：

```
struct tagNode
{
char *pItem;
struct tagNode *pNext;
};
typedef struct tagNode *pNode;
```

3. typedef & #define 的问题

有下面两种定义 pStr 数据类型的方法，两者有什么不同？哪一种更好一点？

```
typedef char *pStr;
#define pStr char *;
```

答案与分析：

通常讲，typedef 要比#define 要好，特别是在有指针的场合。请看例子：

```
typedef char *pStr1;
#define pStr2 char *;
pStr1 s1, s2;
pStr2 s3, s4;
```

在上述的变量定义中，s1、s2、s3 都被定义为 char *，而 s4 则定义成了 char，不是我们所预期的指针变量，根本原因就在于#define 只是简单的字符串替换而 typedef 则是为一个类型起新名字。

#define 用法例子：

```
#define f(x) x*x
main( )
{
int a=6，b=2，c；
c=f(a) / f(b)；
printf("%d \\n"，c)；
}
```

以下程序的输出结果是：36。

因为如此原因，在许多 C 语言编程规范中提到使用#define 定义时，如果定义中包含表达式，必须使用括号，则上述定义应该如下定义才对：

```
#define f(x) (x*x)
```

当然，如果你使用 typedef 就没有这样的问题。

4. typedef & #define 的另一例

下面的代码中编译器会报一个错误，你知道是哪个语句错了吗？

```
typedef char * pStr;
char string[4] = "abc";
const char *p1 = string;
const pStr p2 = string;
p1++;
p2++;
```

答案与分析：

是 p2++出错了。这个问题再一次提醒我们：typedef 和#define 不同，它不是简单的文本替换。上述代码中 const pStr p2 并不等于 const char * p2。const pStr p2 和 const long x 本质上没有区别，都是对变量进行只读限制，只不过此处变量 p2 的数据类型是我们自己定义的而不是系统固有类型而已。因此，const pStr p2 的含义是：限定数据类型为 char *的变量 p2 为只读，因此 p2++错误。

#define 与 typedef 引申谈

1) #define 宏定义有一个特别的长处：可以使用 #ifdef ,#ifndef 等来进行逻辑判断，还可以使用#undef 来取消定义。

2) typedef 也有一个特别的长处：它符合范围规则，使用 typedef 定义的变量类型其作用范围限制在所定义的函数或者文件内（取决于此变量定义的位置），而宏定义则没有这种特性。

5. typedef & 复杂的变量声明

在编程实践中，尤其是看别人代码的时候，常常会遇到比较复杂的变量声明,使用 typedef 作简化自有其价值，比如：

下面是三个变量的声明，我想使用 typdef 分别给它们定义一个别名，请问该如何做？

>1：int *(*a[5])(int, char*);

>2：void (*b[10]) (void (*)());

>3. double(*)() (*pa)[9];

答案与分析：

对复杂变量建立一个类型别名的方法很简单，你只要在传统的变量声明表达式里用类型名替代变量名，然后把关键字 typedef 加在该语句的开头就行了。

```
>1：int *(*a[5])(int, char*);
//pFun 是我们建的一个类型别名
typedef int *(*pFun)(int, char*);
//使用定义的新类型来声明对象，等价于 int* (*a[5])(int, char*);
pFun a[5];
>2：void (*b[10]) (void (*)());
//首先为上面表达式蓝色部分声明一个新类型
typedef void (*pFunParam)();
//整体声明一个新类型
typedef void (*pFun)(pFunParam);
//使用定义的新类型来声明对象，等价于 void (*b[10]) (void (*)());
pFun b[10];
>3. double(*(*pa)[9])();
//首先为上面表达式蓝色部分声明一个新类型
typedef double(*pFun)();
//整体声明一个新类型
typedef pFun (*pFunParam)[9];
//使用定义的新类型来声明对象，等价于 double(*(*pa)[9])();
pFunParam pa;
```

附D、U-Boot中typedef应用解析

typedef 在 C 中真是一个神奇的东西，没有点事例真是很难理解:(回头看看那超烂的大学 C 教程，很多地方没写清楚，遇到问题时看不懂代码。
u-boot 中有这么一段代码。

```
/*这里定义了一个新的数据类型 init_fnc_t，
 *这个数据类型是参数为空，返回值为 int 的函数。
 */
typedef int (init_fnc_t) (void);
```

```
/*init_sequence 是一个指针数组，指向的是 init_fnc_t 类型的函数*/
init_fnc_t *init_sequence[] = {
    cpu_init, /* basic cpu dependent setup */
    board_init, /* basic board dependent setup */
    interrupt_init, /* set up exceptions */
    env_init, /* initialize environment */
    init_baudrate, /* initialze baudrate settings */
    serial_init, /* serial communications setup */
    console_init_f, /* stage 1 init of console */
    display_banner, /* say that we are here */
    dram_init, /* configure available RAM banks */
    display_dram_config,
#if defined(CONFIG_VCMA9) || defined (CONFIG_CMC_PU2)
    checkboard,
#endif
    NULL,
};

/*init_fnc_ptr 为指向函数指针的指针*/
init_fnc_t **init_fnc_ptr;
/*init_fnc_ptr 初始化指向 init_sequence 指针数组，下面的循环遇到
NULL 结束*/
for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr)
{
    if ((*init_fnc_ptr)() != 0) {/*(*init_fnc_ptr)()为 C 中调用指
针指向的函数*/
        hang ();
    }
}
```

自己写了 2 个 test 程序
一个 typedef int (test_fnc_t) (void);
一个 typedef int (*test_fnc_t) (void);

```
#include<stdio.h>

int test0 (void);
int test1 (void);

typedef int (*test_fnc_t) (void);

test_fnc_t test_sequence[] = {
    test0,
    test1,
    NULL,
};

//int _tmain(int argc, _TCHAR* argv[])

int main()
{
    test_fnc_t *test_fnc_ptr;

    for (test_fnc_ptr = test_sequence; *test_fnc_ptr;
++test_fnc_ptr) {
        if ((*test_fnc_ptr)() != 0) {
            printf("error here!");
        }
    }
```

```
    return 0;
}

int test0 (void)
{
    printf("test0\n");
    return 0;
}

int test1 (void)
{
    printf("test1\n");
    return 0;
}
```

```
#include<stdio.h>

int test0 (void);
int test1 (void);

typedef int (test_fnc_t) (void);

test_fnc_t *test_sequence[] = {
    test0,
    test1,
    NULL,
};

//int _tmain(int argc, _TCHAR* argv[])
```

```
int main()
{
    test_fnc_t **test_fnc_ptr;

    for (test_fnc_ptr = test_sequence; *test_fnc_ptr;
++test_fnc_ptr) {
        if ((*test_fnc_ptr)() != 0) {
            printf("error here!");
        }
    }

    return 0;
}

int test0 (void)
{
    printf("test0\n");
    return 0;
}

int test1 (void)
{
    printf("test1\n");
    return 0;
}
```

附E: Ping命令使用的ARP协议

如果您详细查看 u-boot 代码的 net.c 文件，您会发现，里面的 ping 命令，使用了 arp 协议。所以我们简单地来介绍下 arp 协议。如需深入研究，请查看网络

协议 IEEE 802.3 的官方文档。

ARP 协议原理简述

ARP 协议（Address Resolution Protocol 地址解析协议），在局域网中，网络中实际传输的是“帧”，帧里面有目标主机的 MAC 地址。在以太网中，一个注意要和另一个主机进行直接通信，必须要知道目标主机的 MAC 地址。这个 MAC 地址就是标识我们的网卡芯片唯一性的地址。但这个目标 MAC 地址是如何获得的呢？这就用到了我们这里讲到的地址解析协议。所有“地址解析”，就是主机在发送帧前将目标 IP 地址转换成 MAC 地址的过程。ARP 协议的基本功能就是通过目标设备的 IP 地址，查询目标设备的 MAC 地址，以保证通信的顺利进行。所以在第一次通信前，我们知道目标机的 IP 地址，想要获知目标机的 MAC 地址，就要发送 ARP 报文（即 ARP 数据包）。它的传输过程简单的说就是：我知道目标机的 IP 地址，那么我就向网络中所有的机器发送一个 ARP 请求，请求中有目标机的 IP 地址，请求的意思是目标机要是收到了此请求，就把你的 MAC 地址告诉我。如果目标机不存在，那么此请求自然不会有人回应。若目标机接收到了此请求，它就会发送一个 ARP 应答，这个应答是明确发给请求者的，应答中有 MAC 地址。我接到了这个应答，我就知道了目标机的 MAC 地址，就可以进行以后的通信了。因为每次通信都要用到 MAC 地址。

ARP 报文被封装在以太网帧头部中传输，如图为 ARP 请求报文的头部格式。

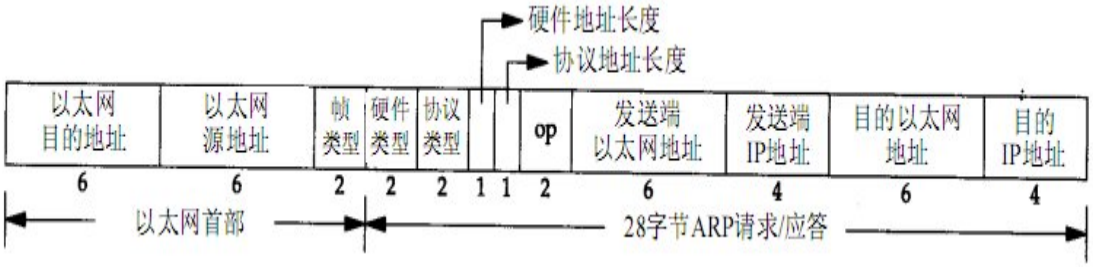


图 用于以太网的 ARP 请求或应答分组格式

注意，以太网的传输存储是“大端格式”，即先发送高字节后发送低字节。例如，两个字节的数据，先发送高 8 位后发送低 8 位。所以接收数据的时候要注意存储顺序。

整个报文分成两部分，以太网首部和 ARP 请求/应答。下面挑重点讲述。
“以太网目的地址”字段：若是发送 ARP 请求，应填写广播类型的 MAC 地址 FF-FF-FF-FF-FF-FF，意思是让网络上的所有机器接收到；
“帧类型”字段：填写 08-06 表示次报文是 ARP 协议；
“硬件类型”字段：填写 00-01 表示以太网地址，即 MAC 地址；
“协议类型”字段：填写 08-00 表示 IP，即通过 IP 地址查询 MAC 地址；
“硬件地址长度”字段：MAC 地址长度为 6（以字节为单位）；
“协议地址长度”字段：IP 地址长度为 4（以字节为单位）；
“操作类型”字段：ARP 数据包类型，0 表示 ARP 请求，1 表示 ARP 应答；
“目的以太网地址”字段：若是发送 ARP 请求，这里是需要目标机填充的。

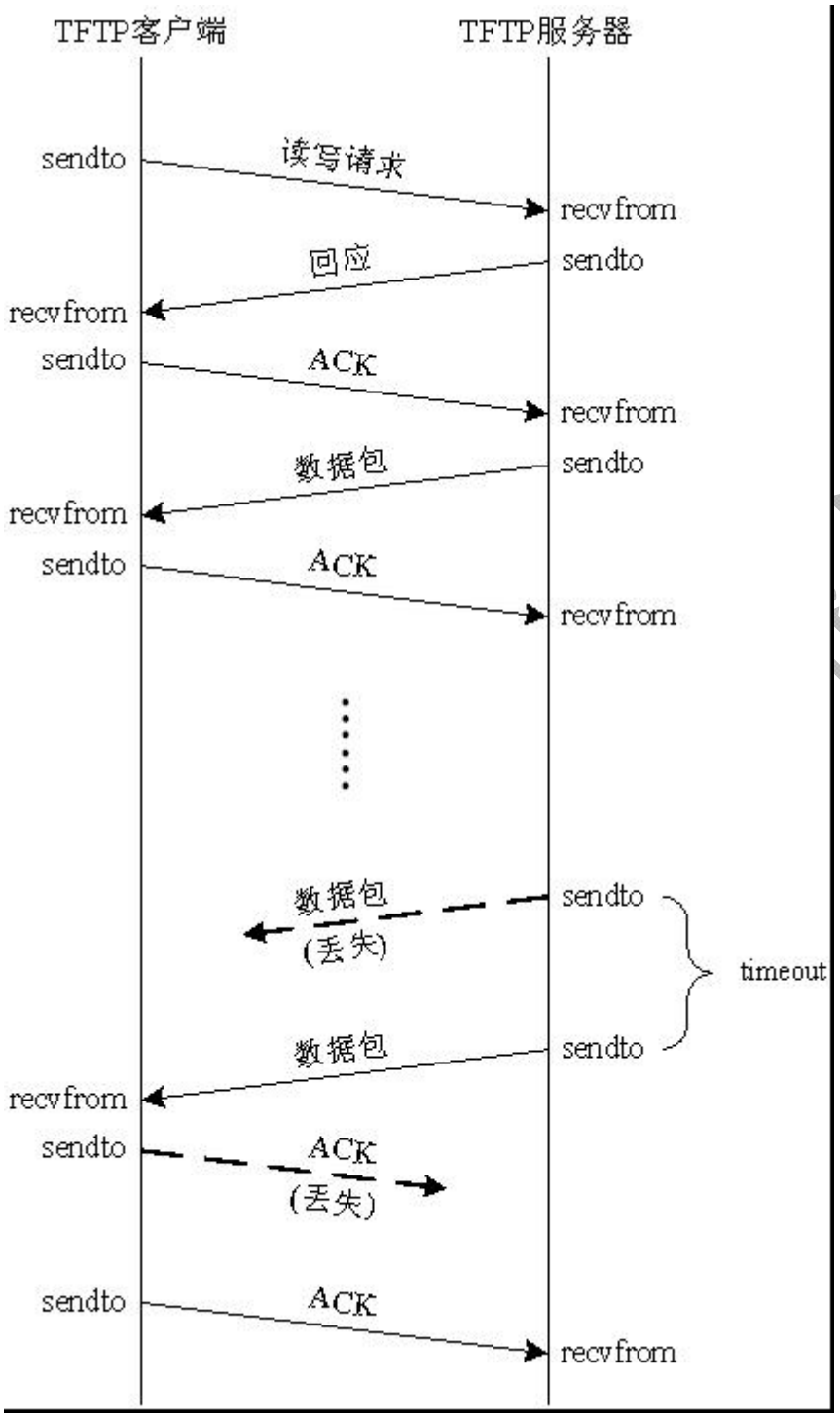
附F：TFTP协议详解

1. 目的

TFTP 是一个传输文件的简单协议，通常使用 UDP 协议而实现，但 tftp 并没有要求实现的具体协议，在特殊需求的场合可以同 tcp 实现。此协议设计的时候是进行小文件传输的。因此它不具备通常的 FTP 的许多功能，它只能从文件服务器上获得或写入文件，不能列出目录，不进行认证，它传输 8 位数据。传输中有三种模式：netascii，这是 8 位的 ASCII 码形式，另一种是 octet，这是 8 位源数据类型；最后一种 mail 已经不再支持，它将返回的数据直接返回给用户而不是保存为文件。

2. 概况

任何传输起自一个读取或写入文件的请求，这个请求也是连接请求。如果服务器批准此请求，则服务器打开连接，数据以定长 512 字节传输。每个数据包包括一块数据，服务器发出下一个数据包以前必须得到客户对上一个数据包的确认。如果一个数据包的大小小于 512 字节，则表示传输结构。如果数据包在传输过程中丢失，发出方会在超时后重新传输最后一个未被确认的数据包。通信的双方都是数据的发出者与接收者，一方传输数据接收应答，另一方发出应答接收数据。大部分的错误会导致连接中断，错误由一个错误的数据包引起。这个包不会被确认，也不会被重新发送，因此另一方无法接收到。如果错误包丢失，则使用超时机制。错误主要是由下面三种情况引起的：不能满足请求，收到的数据包内容错误，而这种错误不能由延时或重发解释，对需要资源的访问丢失(如硬盘满)。TFTP 只在一种情况下不中断连接，这种情况是源端口不正确，在这种情况下，指示错误的包会被发送到源机。这个协议限制很多，这是都是为了实现起来比较方便而进行的。通过下边的图片来了解 tftp 协议的通信流程：



- TFTP 协议概述
  1. 简单文件传送协议(Trivial File Trhansfer Protocol)
  2. 最初用于引导无盘系统，被设计用来传输小文件
  3. 基于 UDP 协议实现，但也可以由其他协议实现
  4. 不具备 FTP 的许多功能
  5. 只能从服务器获取或写入文件，不能列出目录
  6. 不进行认证

- 数据传输模式
  1. netascii：文本模式
  2. octet：二进制模式
  3. mail：已经不再支持
- **协议结构**

基本 TFTP 协议头结构：

16 bits	String	16 bits	String	16 bits
Opcode	Filename	0	Mode	0

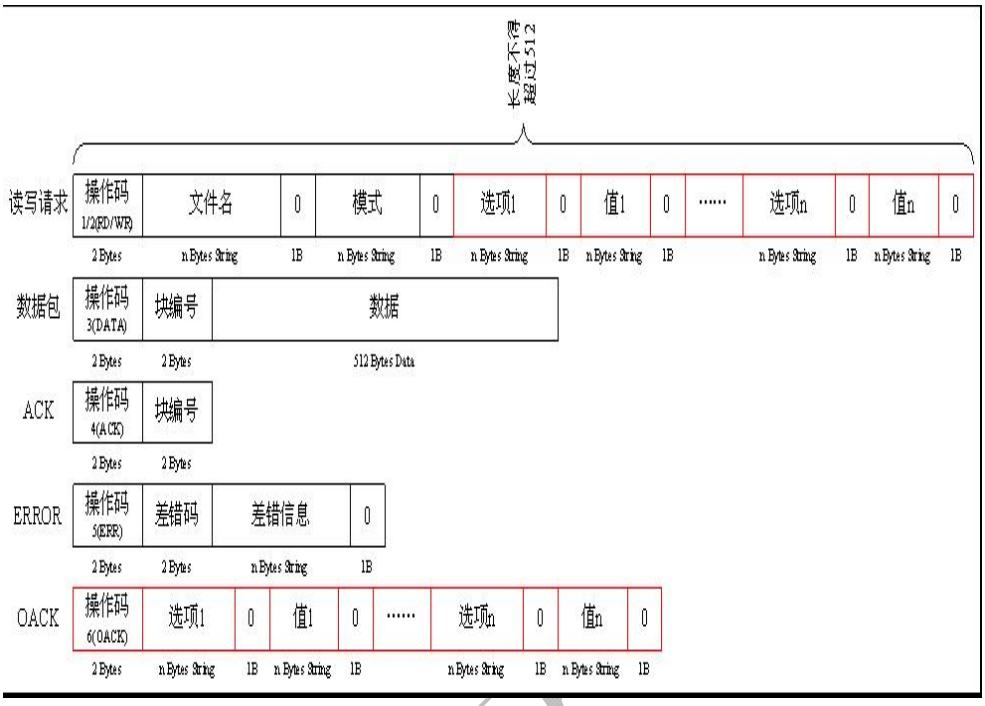
Opcode：操作代码或命令。以下为 TFTP 命令：

Opcode	Command	Description
1	Read Request	Request to read a file
2	Write Request	Request to write to a file
3	File Data	Transfer of file data
4	Data Acknowledge	Acknowledgement of file data
5	Error	Error indication

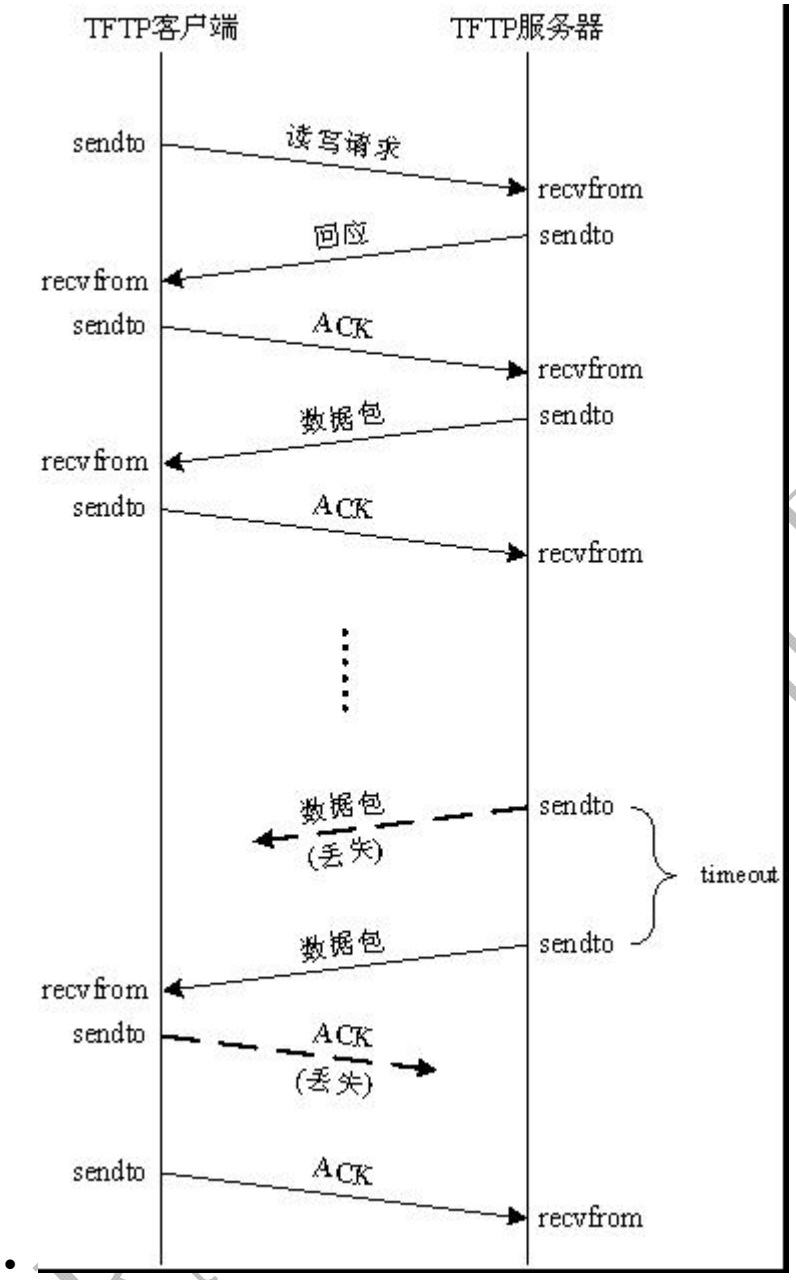
Filename：传送的字段名称。

Mode：数据模式。协议传输的文件数据格式。可以是 NetASCII，也可以是标准 ASCII，八位二进制数据或邮件标准 ASCII。

我们通过图片来详细看看 tftp 报文的格式：



- 在看看 tftp 的简单超时处理机制：



好了，tftp 的内容就这些了，下边是一个完整的 tftp 通信的报文：

参考文献：

1. 百度空间《U-BOOT 介绍以及常用 U-bot 命令介绍》，网址：
http://hi.baidu.com/nomorerain/blog/item/137b66ef20161e10fdfa3cb6.html
2. cu 博客《u-boot 介绍》，网址：
http://blog.chinaunix.net/u1/47395/showart_1932442.html
3. cu 博客《u-boot 中 typedef 应用解析》，网址：
http://blog.chinaunix.net/u/17660/showart_284498.html
4. 百度空间《u-boot lds 文件详解》，网址：
http://hi.baidu.com/kinylei/blog/item/e598fc3217bfadf2184cff64.html
5. 百度空间《单片机驱动 DM9000 网卡芯片（详细调试过程）【下】》，网址：
http://hi.baidu.com/mikenoodle/blog/item/a271def982bc6a51242df279.html
6. linux 时代网站，网址：
http://linux.chinaunix.net/techdoc/net/2009/05/04/1109928.shtml

感　　谢

在文档撰写过程中，我得到了众多网友的帮助和支持，感谢所有帮助过我的网友！本文档命令为《U-Boot 移植手册》，非常普通的名字，确参考了无数个文档，以及一些相关书籍，感谢众多网友写的关于 u-boot 的文章，才有了我的 U-Boot 移植手册。感谢我女朋友每天晚上一碗夜宵，使得我下班后有更充足的精力撰写文档；感谢 ARM 技术交流网的各位版主，没有他们，我们无法向大家提供完善的 ARM 技术交流平台。