1 U-boot 概念

U-boot，它是常用的嵌入式操作系统启动程序，是著名的开源 bootloader 程序。可以启动 linux、android 等系统。它的最基本的作用是：

1、把操作系统镜像从介质如 flash、nand、SD 卡等加载到内存（可以用于从 u 盘启动系统）

2、在内存中把操作系统启动，启动时可以向操作系统传递启动配置信息。

3、当然它还有一个简单的控制台，利用串口与用户交互以提供一些额外的辅助功能，如在 OS 启动前查看内存、数据拷贝、查看 OS 镜像信息、检查坏块等。

uboot 是 Universal BootLoad。一个“通用”的启动代码载入器。Linux 本身不能自己把自己读取到内存中并且运行，所以他需要一个 loader（载入器）读入内存并且运行。

uboot 可以在很多种 cpu 架构上运行，同时也支持很多开发板，但是每种 cpu 架构之间有差别，或者开发板的资源不同，假如在某款开发板上能正常引导启动操作系统的话，并不意味着在其他款就能引导启动，建立一款统一的 bootloader 几乎是不可能的，但是经过大师们的努力，能够实现通过简单的配置改动，就可以实现引导启动很多操作系统（也就是 bootloader 移植，uboot 是 bootloader 中的一种）

uboot 类似台式机的 BIOS + grub 启动 Linux 的组合。

嵌入式系统因为构造很特殊，所以他的系统启动一般都是要在 NOR-flash（我们常见的 NAND-flash 存储器不能直接运行程序，而 NOR-flash 可以直接运行程序，和 BIOS 很类似。NOR-flash 成本很高）。这个前期启动需要做一些初始化工作，以及因为环境限制，程序运行有很大的功能限制。这使得原本就不支持自己启动的 Linux 内核更需要一个 loader 来提供前期的准备，这就是 bootloader 的主要用处。

选择 U-Boot 的理由：

① 开放源码；

② 支持多种嵌入式操作系统内核，如 Linux、VxWorks；

③ 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；

④ 较高的可靠性和稳定性；

⑤ 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等；

⑥ 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；

⑦ 较为丰富的开发调试文档与强大的网络技术支持

2 Bootloader 简介

2.1 bootlaoder 概念

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，特别是在嵌入式里。因此，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 完全开源，可以从网上下载 ftp://ftp.denx.de/pub/u-boot/；

2.2 bootlaoder 的启动方式

①自启动模式：根据设置启动内核；
②交互模式：设置配置信息；

CPU 上电或复位时，会从某个地址开始执行。ARM 结构的 CPU 从地址 0x0000000 开始，通常这个地址处就存放了 BL1+bootloader，这样一上电就可以执行（bootloader 相当于 BL2）。

从开发和用户使用角度来分析，Bootlaoder 可以分为以下两种操作模式（Operation Mode）。

自启动模式（Boot laoding）： bootloader 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。产品发布后，bootloader 工作在这种模式下。

交互模式或者**下载模式**（Downlaoding）：开发人员可以使用各种命令，通过串口连接或网络连接等通信手段从主机下载文件（比如内核映像、文件系统映像）到 RAM 中，可以被 bootloader 写到目标机上的固态存储介质中，或者直接进入系统的引导。

U-boot 能够同时支持这两种工作模式，而且允许用户在这两种模式之间进行切换。比如，U-Boot 在启动时处于正常的自启动模式，但是它会延时若干秒（时间可以设置），等待终端用户按下任意键，而将 U-boot 切换到交互模式。如果在指定时间内没有用户按键，则 U-Boot 继续启动 Linux 内核。

2.3 Bootloader 的结构

嵌入式 LINUX 系统从软件的角度通常可以分为 4 个层次：

Bootloader：

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。有些 CPU 在运行 Bootloader 之前先运行一段固化的程序，比如 x86 结构的 CPU 就是先运行 BIOS 中的固件，

然后才运行硬盘第一个分区（MBR）中的 Bootloader。在大多嵌入式系统中并没有固件，Bootloader 是上电后执行的第一个程序。

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。内核的启动参数可以是内核默认的，或是由 Bootloader 传递给它的。

文件系统：

包括根文件系统和建立于 FLASH 内存设备之上的文件系统。包含了 LINUX 系统能够运行所必须的应用程序、库等，比如可以给用户提供操作 LINUX 的控制界面和 shell 程序、动态连接和程序运行时需要的 glibc 或 uClibc 库等。

用户应用程序：

特定于用户的应用程序，它们也存储在文件系统中。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：Qtopia 和 MiniGUI 等。

显然，在嵌入式系统的固态存储设备上有相应的分区来存储它们，如下图所示为一个典型的 Bootloader 分区结构：

Bootloader	Boot parameters	Kernel	Root filesystem

“Boot parameters”分区中存放一些可设置的参数，比如 IP 地址、串口波特率、要传递给内核的命令行参数等。正常启动过程中，Bootloader 首先运行，然后它将内核复制到内存中（也有些内核可以直接在固态存储设备上直接运行），并且在内存某个固定的地址设置好要传递给内核的参数，最后运行内核。内核启动之后，它会挂接（mount）根文件系统（“Root filesystem”），启动文件系统中的应用程序。

Kernel： 操作系统内核，是指大多数操作系统的核心部分。它由操作系统中用于管理存储器、文件、外设和系统资源的那些部分组成。操作系统内核通常运行进程，并提供进程间的通信。

2.4 常用 Bootloader 介绍

现在 Bootloader 种类繁多，比如 X86 上有 LILO、GRUB 等。对于 ARM 架构的 CPU，有 U-Boot、Vivi 等。它们各有特点，下面列出 Linux 下的引导程序代码 Bootloader 及其支持的体系结构：

开放源码的 linux 引导程序					
Bootloader	Monitor	描述	X86	ARM	PowerPC
LILO	否	linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 linux	是	否	否
ROLO	否	从 ROM 引导 linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BUIS 的 linux 引导程序	是	否	否
BLOB	是	LART 等硬件平台的引导程序	否	是	否
U-Boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是
Vivi	是	Mizi 公司针对 SAMSUNG 的 ARM CPU 设计的引导程序	否	是	否

Vivi 是 Mizi 公司针对 SAMSUNG 的 ARM 架构 CPU 专门设计的，基本上可以直接使用，命令简单方便。不过其初始版本只支持串口下载，速度较慢。在网上出现了各种改进版本：支持网络功能、USB 功能、烧写 YAFFS 文件系统映像等。

U-BOOT 则支持大多 CPU，可以烧写 EXT2、JFFS2 文件系统映像，支持串口下载、网络下载，并提供了大量的命令。相对于 Vivi，它的使用更复杂，但是可以用来更方便地调试程序。

2.5 Bootloader 启动的两个阶段

U-BOOT 运行过程

2.5.1 程序启动过程：

1.u-boot 上电后执行的第一个文件为 arch/arm/cpu/armv7/start.S 文件，start.S 文件准备好第二阶段的运行环境。

2.start.S 执行完毕最终会调用到 arch/arm/lib/board.c 中的 board_init_r 函数，函数对会对 flash、net、串口等进行初始化，最终会进入死循环，如下：

```
for (;;)
{
    main_loop ();
    //进入等待命令解析状态，直到启动内核后，u-boot 退出。
}
```

2.5.2 Bootloader 的两个阶段：

Bootloader 的启动过程可以分为单阶段（single Stage）、多阶段（Multi-stage）两种。通常多阶段的 Bootloader 能提供更为复杂的功能以及更好的可移植性。从固态存储设备上启动的 Bootloader 大多都是两阶段的启动过程。第一阶段使用汇编来实现，它完成一些依赖于 CPU 体系结构的初始化，并调用第二阶段的代码；第二阶段则通常使用 C 语言实现，这样可以实现更复杂的功能，而且代码会有更好的可读性和可移植性。

一般而言，这两个阶段完成的功能可以如下分类：

1.Bootloader 第一阶段的功能：

1) **硬件设备初始化。**
2) **为加载 Bootloader 的第二阶段代码准备 RAM 空间。**
3) **设置 CPU 速度、时钟频率及终端控制寄存器。**
4) **初始化内存控制器。**
5) **复制 Bootloader 的第二阶段代码到 RAM 空间中。**
6) **设置好栈。**
7) **跳转到第二阶段代码的 C 入口点。**

在第一阶段进行的硬件初始化一般包括：关闭看门狗、关闭中断、设置 CPU 速度和时钟频率、初始化 RAM 等。这些并不都是必需

的，比如 S3C2410/S3C2440 的开发板所使用的 U-Boot 中，就将 CPU 的速度和时钟频率的设置放在第二阶段。

甚至，将第二阶段的代码复制到 RAM 空间中也不是必需的，对于 NOR-FLASH 等存储设备，完全可以在上面直接执行代码，只不过相比在 RAM 中执行效率大为降低。

2.Bootloader 第二阶段的功能：

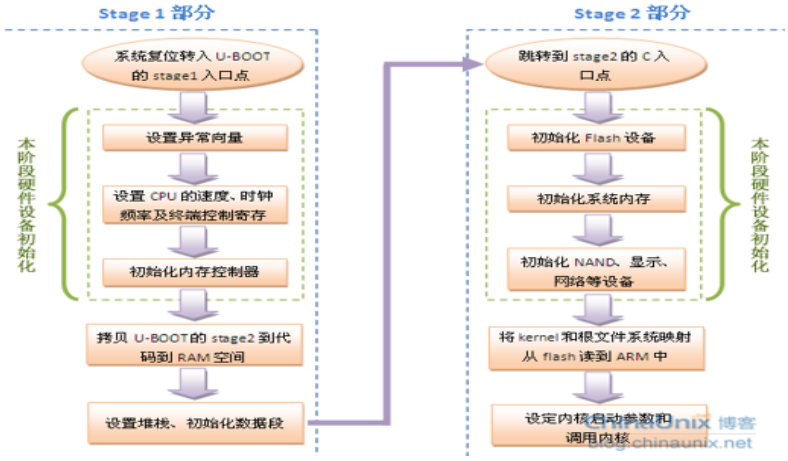
1) **初始化本阶段要使用的硬件设备。**
2) **检测系统内存映射（memory map）。**
3) **将内核映像和根文件系统映像从 FLASH 上读到 RAM 空间中。**
4) **为内核设置启动参数。**
5) **调用内核。**

所谓检测内存映射，就是确定板上使用了多少内存、它们的地址空间是什么。由于嵌入式开发中 Bootloader 大多都是针对某一类板子进行编写，所以可以根据板子的情况直接设置，不需要考虑可以适用于各类情况的复杂算法。

FLASH 上的内核映像有可能是经过压缩的，在读到 RAM 之后，还需要进行解压。当然对于有自解压功能的内核，不需要 Bootloader 来解压。

将根文件系统映像复制到 RAM 中，这不是必需的。这取决于是什么类型的根文件系统，以及内核访问它的方法。内核存放在适当的位置后，直接跳到它的入口点即可调用内核。

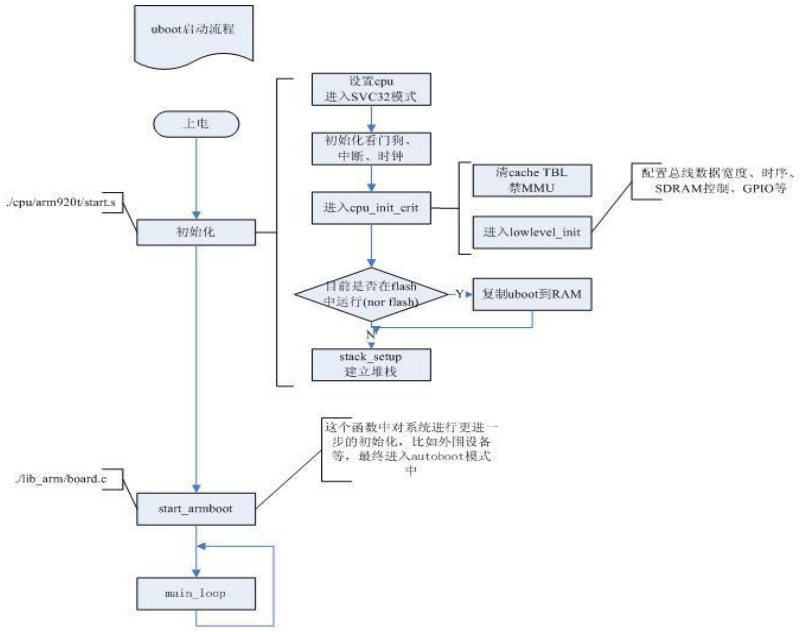
U-Boot 的启动顺序流程图分析：



Stage1阶段：运行依赖于 CPU 体系架构的代码，这部分主要完成 cpu 上电初始化过程，使用汇编语言编写，对应于 u-boot 是 arch/arm/cpu/armv7/start.S

```
[root@bogon armv7]# pwd
/whz/u-boot/uboot_tiny4412/arch/arm/cpu/armv7
[root@bogon armv7]# vim start.S
```

进入进行分析（每一个内核文件夹都有一个 start.S 文件，cpu 上电首先从 start.S 运行）



stage2阶段:一般用 C 语言来实现，这样可以实现复杂的功能，并且具有良好的可移植性和可读性

（参考网址 http://blog.chinaunix.net/uid-23193900-id-3184107.html ）

C 语言代码部分 arch/arm/lib/board.c 中的 board_init_f 是 C 语言开始的函数也是整个启动代码中 C 语言的主函数，同时还是整个 u-boot 的主函数，该函数完成如下操作：

1）调用一系列的初始化函数。

2）初始化 Flash 设备。

3）初始化系统内存分配函数。

4）如果目标系统拥有 NAND 设备，则初始化 NAND 设备。

5）如果目标系统有显示设备，则初始化该类设备。

6）初始化相关网络设备，填写 IP、MAC 地址等。

7）进去命令循环（即整个 boot 的工作循环），接受用户从串口输入的命令，然后进行相应的工作。（为了方便开发，至少要初始化一个串口以便程序员与 Bootloader 进行交互）。

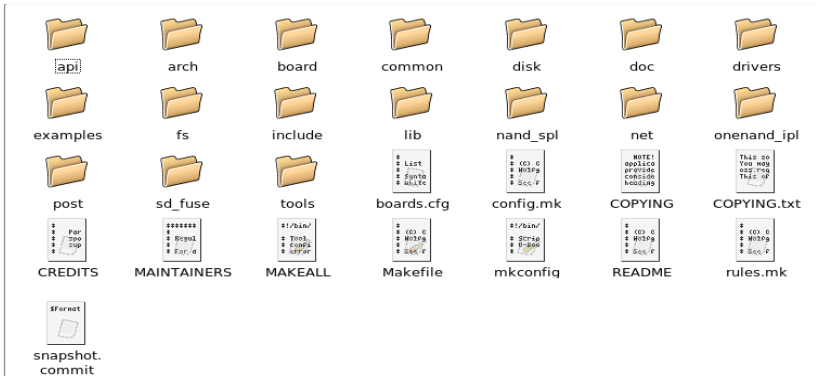
用 for 循环对结构体 **init_fnc_t *init_sequence[]**数组中的函数指针逐个调用。

```
void board_init_f(ulong bootflag)
{
   bd_t *bd;
   init_fnc_t **init_fnc_ptr;
   gd_t *id;
   ulong addr, addr_sp;

   /* Pointer is writable since we allocated a register for it */
   gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);

   /* compiler optimization barrier needed for GCC >= 3.4 */
   __asm__ __volatile__("": : :"memory");

   memset((void*)gd, 0, sizeof (gd_t));

   gd->mon_len = _bss_end_ofs;

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

   debug ("monitor len: %08lX\n", gd->mon_len);
   /*
    * Ram is setup, size stored in gd !!
    */
```

函数指针数组

```
init_fnc_t *init_sequence[] = {
#if defined(CONFIG_ARCH_CPU_INIT)
   arch_cpu_init,    /* basic arch cpu dependent setup */
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
   board_early_init_f,
#endif
   timer_init,    /* initialize timer */
#ifdef CONFIG_FSL_ESDHC
   get_clocks,
#endif
   env_init,      /* initialize environment */
#if defined(CONFIG_S5P6450) && !defined(CONFIG_S5P6460_IP_TEST
   init_baudrate,    /* initialze baudrate settings */
   serial_init,      /* serial communications setup */
#endif
   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
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
   init_func_i2c,
#endif
   dram_init,     /* configure available RAM banks */
#if defined(CONFIG_CMD_PCI) || defined(CONFIG_PCI)
   arm_pci_init,
#endif
   NULL,
};
```

U-BOOT 目录介绍：



U-BOOT 的目录可以分成三类：

- 第一类目录与处理器体系结构或者开发板硬件直接相关；
- 第二类目录是一些与体系无关的通用的函数或者驱动程序；
- 第三类目录是 u-boot 的应用程序、工具或者文档；

1. board:与开发板相关部分代码，主要用来配置单板关系的外设。比如：Makefile 和 u-boot.lds 等和实际开发板的硬件和地址相关的代码；
2. common:与体系无关代码的文件，实现各种命令的 C 文件。比如：环境、命令、控制台的实现等；

3. arch:与芯片体系相关的文件，其中有常见的 arm 和 i386 文件夹。在其的子目录都是以 u-boot 所支持的 CPU 为名，比如有子目录 armv7、mips、mpc8260 和 nios 等。每个特定的子目录中都包括 cpu.c 和 interrupt.c 和 start.S。其三部分对应的功能描述如下：
   a) cpu.c:初始化 cpu、设置指令 cache 和数据 cache 等；
   b) interrupt.c:设置系统的各种终端和异常，比如快速中断、开关中断、时钟中断、软件中断、预取中止和未定义指令等；
   c) start.S:是 u-boot 启动时执行的第一个文件，他主要是设置系统堆栈和工作方式，为进入 C 程序奠定基础；
4. disk:硬盘、U 盘等处理代码；
5. doc:开发、使用文档；
6. drivers:通用设备驱动程序，比如各种网卡、支持 CFI 的 flash、串口和 USB 总线等。
7. example:程序范例。 一些独立运行的应用程序的例子；
8. fs:支持的文件系统：支持文件系统的文件，u-boot 现在支持 cramfs、fat、fdos、jffs2、yaffs 和 registerfs。
9. include: 头文件，还有对各种硬件平台支持的汇编文件，系统的配置文件和对文件系统支持的文件。比如在 configs 下和单板相关的头文件；
10. nand_spl：nand-flash 启动的一些相关代码，如果要从 nand 启动需要修改 boot 里面的代码。
11. net:网络协议，存放网络协议的程序、TFP 协议栈。
12. onenand_ipl:速度更快。(可以当 nand-flash 也可兼容 nor-flash)。
13. post：电源管理、电源检测相关函数。例如电脑 cpu 风扇掉了，如果没有该函数，容易烧 cpu；如果有 post 函数，可以检测到是否有风扇，没有风扇就不启动 cpu，即保护了 cpu；
14. tools：创建 S-Record 格式文件和 U-BOOT images 的工具。Bmp_logo 制作 logo 工具等

3 U-BOOT 应用：主要用来启动内核

3.1 U-boot 的目的

u-boot 的目的主要是用来引导内核，在引导内核启动期间，u-boot 还提供了下载内核镜像的功能，对 Exynos 4412 芯片提供了四种启动方式，分别是：

1、 常用的 NAND-FLASH 启动
2、 SD/MMC 卡启动
3、 eMMC 启动
4、 USB 设备启动

在上面四种模式下，可以通过 u-boot 命令方式把 SD 卡上的内核烧写到 eMMC 上，在从 eMMC 上读取内核镜像在去启动内核。

3.2 Bootloader 与内核的交互

Bootloader 与内核的交互是单向的，Bootloader 将各类参数传给内核。由于它们不能同时运行（uboot 和内核），传递办法只有一个：Bootloader 将参数放在某个约定的地方之后，再启动内核，内核启动后从这个地方获得参数。

除了约定好参数存放的地址外,还要规定参数的结构。Linux2.4.x 以后的内核都期望以标记列表（tagged list）的形式来传递启动参数。标记，就是一种数据结构；标记列表，就是挨着存放的多个标记列表以 tag_header 结构和一个联合（union）组成。tag_header 结构表示标记的类型及长度，(比如是表示内存还是表示命令行参数等)。对于不同类型的标记使用不同的联合,比如表示内存时使用 tag_mem32（内存大小），表示命令行时使用 tag_cmdline。

数据结构 tag 和 tag.header 定义在 linux 内核源码的 /include/asm/setup.h 头文件中（内核才有）。

3.3 U-boot 的重要数据结构（了解）

u-boot 的主要功能是用于引导 OS 的，但是本身也提供许多强大的功能，可以通过输入命令行来完成许多操作。所以它本身也是一个很完备的系统。u-boot 的大部分操作都是围绕它自身的数据结构，这些数据结构是通用的，但是不同的板子初始化这些数据是不一样的。所以 u-boot 的通用代码是依赖于这些重要的数据结构的。这里说的数据结构其实就是一些全局变量。

3.3.1 gd 全局数据变量指针

它保存了 u-boot 运行需要的全局数据，类型定义：

```
typedef struct global_data
{
        bd_t    *bd;                    //board data pointor 板子数据指针
        unsigned long flags;            //指示标志，如设备已经初始化标志等。
        unsigned long baudrate;         //串口波特率
        unsigned long have_console;     /*  串口初始化标志*/
        unsigned long reloc_off;        /*重定位偏移，就是实际定向的位置与编译连接时指定
                                                的位置之差，一般为 0 */
        unsigned long env_addr;         /*  环境参数地址*/
        unsigned long env_valid;        /*  环境参数 CRC 检验有效标志  */
        unsigned long fb_base;          /* base address of frame buffer */
        #ifdef CONFIG_VFD
          unsigned char vfd_type;       /* display type */
        #endif
        void    **jt;                       /*  跳转表，1.1.6 中用来函数调用地址登记  */
      } gd_t;
```

3.3.2 bd 板子数据指针

包含板子很多重要的参数。 类型定义如下：

```
typedef struct bd_info
{
        int   bi_baudrate;                  /*  串口波特率  */
        unsigned long bi_ip_addr;           /* IP  地址  */
        unsigned char bi_enetaddr[6];       /* MAC 地址*/
        struct environment_s          *bi_env;
        ulong        bi_arch_number;        /* unique id for this board */
        ulong        bi_boot_params;        /*  启动参数  */
         struct                                  /* RAM  配置  */
         {
              ulong start;
              ulong size;
            }bi_dram[CONFIG_NR_DRAM_BANKS];
} bd_t;
```

3.4 环境变量

参数解释：

Bootdelay	定义执行自动启动的等候秒数
baudrate	定义串口控制台的波特率
netmask	定义以太网接口的掩码
ethaddr	定义以太网接口的 MAC 地址
bootfile	定义缺省的下载文件
bootargs	定义传递给 Linux 内核的命令行参数
bootcmd	定义自动启动时执行的几条命令
serverip	定义 tftp 服务器端的 IP 地址、
ipaddr	定义本地的 IP 地址
stdin	定义标准输入设备，一般是串口
stdout	定义标准输出设备，一般是串口
stderr	定义标准出错信息输出设备，一般是串口

2.5.4 命令结构构体类型定义

定义在 command.h 中：

```
struct cmd_tbl_s {
    char    *name;          /* 命令名     */
    int     maxargs;        /* 参数个数：最多能支持几个参数 */
    int     repeatable;     /* 命令是否可重复   */
                            /* Implementation function  命令执行函数*/
    int     (*cmd)(struct cmd_tbl_s *, int, int, char *[]);
    char    *usage;         /* Usage message (short) */
    #ifdef CFG_LONGHELP
    char    *help;          /* Help    message (long) */
    #endif
    #ifdef CONFIG_AUTO_COMPLETE     /* do auto completion on the arguments */
    int     (*complete)(int argc, char *argv[], char last_char, int maxv, char *cmdv[]);
    #endif
};

typedef struct cmd_tbl_s cmd_tbl_t;
```

定义 section 属性的结构体。

编译的时候会单独生成一个名为.u_boot_cmd 的 section 段。

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

定义命令结构体变量

```
U_BOOT_CMD(
            dcache,     2,     1,        do_dcache,
            "dcache    - enable or disable data cache\n",
            "[on, off]\n"
            "        - enable or disable data (writethrough) cache\n"
          );
```

其实就是定义了一个 cmd_tbl_t 类型的结构体变量，这个结构体变量名为__u_boot_cmd_dcache。

其中变量的五个域初始化为括号的内容。分别指明了命令名、参数个数、重复数、执行命令的函数、命令提示。

每个命令都对应这样一个变量，同时这个结构体变量的 section 属性为.u_boot_cmd.也就是说每个变量编译结束。

在目标文件中都会有一个.u_boot_cmd 的 section。一个 section 是连接时的一个输入段，如.text,.bss,.data 等都是 section 名。最后由链接程序把所有的.u_boot_cmd 段连接在一起，这样就组成了一个命令结构体数组。

u-boot.lds 中相应脚本如下:

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

可以看到所有的命令结构体变量集中在__u_boot_cmd_start 开始到__u_boot_cmd_end 结束的连续地址范围内，这样形成一个 cmd_tbl_t 类型的数组，run_command 函数就是在这个数组中查找命令的。