

U-Boot 命令

刘通平 Homepage: <http://www.cs.umass.edu/~tonyliu/>

U-Boot 运行稳定后，可以用它的内部命令来查看目标系统的信息，设置环境变量等。U-Boot 在硬件初始化完成后将进入 `main_loop()` 函数，`main_loop()` 函数将进入一个无限循环，当用户输入命令后，首先将调用 `run_command()` 函数进行处理。在 `run_command()` 函数中，将调用 `find_cmd()` 函数把用户从终端输入的命令进行比较，当 `find_cmd()` 返回值不为 0 时证明系统支持用户输入的命令，在对命令进行检验后最后将调用命令处理函数。

`find_cmd()` 函数将从系统默认的命令表中查询一个匹配的命令，查看源代码发现，`find_cmd()` 将把 `__u_boot_cmd_start` 开始到 `__u_boot_cmd_end` 结束的所有命令一一和用户输入的命令进行比较。而关于 `__u_boot_cmd_start` 和 `__u_boot_cmd_end` 的定义是在板级相关的链接文件中定义的，比如对于 fads 的板子，在 `board/fads/u-boot.lds` 中有以下定义：

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

从以上定义可知，`.u_boot_cmd` 是真正的命令部分，而 `.u_boot_cmd` 的定义在 `command.h` 中，

```
#define Struct_Section __attribute__((unused,section(".u_boot_cmd")))
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
    cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, \
usage}
```

因此，即 `U_BOOT_CMD` 可以把一个命令加入到全局的命令表中，查询 `U_BOOT_CMD` 可知，`common` 目录下的所有 `cmd_***` 文件和 `command.c` 文件都包含有 `U_BOOT_CMD` 宏定义。因此可看出，实际上 `command.c` 中定义了和命令本身相关的命令处理程序，而其它的 `cmd_***` 文件则是某一类型的命令处理程序。这些强大的命令处理程序构成了 U-Boot 强大的命令处理功能。

下面将介绍 U-Boot 中最重要的命令。因为 u-boot 是高度可配置，因此你还可以对这些命令进行裁减。如果你不清楚当前的 U-Boot 中支持哪些命令，你可以利用 `help` 列出当前能用的所有命令。

对于大部分命令，你并不需要输入命令的全名，u-boot 还支持命令的匹配。比如，`help` 可缩写成 `h`。

关于 u-boot 的命令接口，这里需要注意以下几点：

- 一些命令的行为取决于 u-boot 的配置和 u-boot 环境变量的设置。
- 所有除数字外的 u-boot 命令都可以 16 进制的格式表示。

4.5.1 信息查询命令

4.5.1.1 bdfinfo – 打印板级信息

bdfinfo (缩写: bdi) 将打印板子信息, 包括内存地址, 时钟频率, MAC 地址等等, 板级信息最终将传给 Linux 内核。如下例:

```
=> bdi
memstart    = 0x00000000
memsize     = 0x04000000
flashstart  = 0x40000000
flashsize   = 0x00800000
flashoffset = 0x00030000
sramstart   = 0x00000000
sramsize    = 0x00000000
immr_base   = 0xFFF00000
bootflags   = 0x00000001
intfreq     =    50 MHz
busfreq     =    50 MHz
ethaddr     = 00:D0:93:00:28:81
IP addr     = 10.0.0.99
baudrate    = 115200 bps
=>
```

4.5.1.2. coninfo – 打印串口设备信息

coninfo (缩写: conin) 将打印串口设备信息, 包括设备名, 标志符和当前使用情况。如下例:

```
=> conin
List of available devices:
serial 80000003 SIO stdin stdout stderr
=>
```

serial (设备名) 是一种提供输入 (标识符“**I**”) 和输出 (标识符“**O**”) 功能的系统设备 (标识符“**S**”), 现在已经分配给 `stdin`, `stdout`, `stderr` 三种标准 I/O 流。

4.5.1.3. flinfo – 打印 Flash 信息

flinfo (缩写: fli) 将打印当前系统的 Flash 信息。如下例:

```
=> fli
```

```
Bank # 1: FUJITSU AM29LV160B (16 Mbit, bottom boot sect)
Size: 4 MB in 35 Sectors
Sector Start Addresses:
 40000000 (R0) 40008000 (R0) 4000C000 (R0) 40010000 (R0) 40020000 (R0)
 40040000      40060000      40080000      400A0000      400C0000
 400E0000      40100000      40120000      40140000      40160000
 40180000      401A0000      401C0000      401E0000      40200000
 40220000      40240000      40260000      40280000      402A0000
 402C0000      402E0000      40300000      40320000      40340000
 40360000      40380000      403A0000      403C0000      403E0000
=>
```

4.5.1.4. iminfo – 打印映像的头部信息

iminfo (缩写: imi) 可用于打印 Linux 内核映像的头部信息, 包括映像名, 映像类型, 大小和检查 CRC 校验和。如下例:

```
=> imi 100000
## Checking Image at 00100000 ...
Image Name:   Linux-2.4.4
Created:      2002-04-07 21:31:59 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    605429 Bytes = 591 kB = 0 MB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
=>
```

4.5.1.5. help – 打印帮助信息

help (缩写: imi) 命令将打印帮助信息, 当不指定具体命令时将打印出当前 u-boot 映像所能支持的所有命令。

4.5.2 内存控制命令

4.5.2.1 bdfinfo – 打印和设置偏移量

base 命令 (缩写: ba) 用于打印和设置内存中的地址偏移量。如下例:

```
=> base
```

Base Address: 0x00000000

=> md 0 c

```
00000000: feffffff 00000000 7cbd2b78 7cdc3378      .....|. +x|. 3x
00000010: 3cfb3b78 3b000000 7c0002e4 39000000      <. ;x;... |... 9...
00000020: 7d1043a6 3d000400 7918c3a6 3d00c000      }. C. =... y... =...
```

4.5.2.2. crc32 – 计算校验和

crc32 (缩写: crc) 用于计算一段内存空间的校验和, 当该命令带 3 个参数时表示把计算出来的校验和写到第 3 个参数所在的存储单元中。

=> crc 100004 3FC 100000

CRC32 for 00100004 ... 001003ff ==> d433b05b

=> md 100000 4

```
00100000: d433b05b ec3827e4 3cb0bacf 00093cf5      .3. [. 8'. <..... <
```

=>

4.5.2.3. cmp – 内存比较

cmp 命令可用于比较两段内存的内容是否相同, 该命令将比较指定的长度的内存 (如果存在第 3 个参数) 或在第一个不相等处退出。如下例:

=> cmp 100000 40000000 400

word at 0x00100004 (0x50ff4342) != word at 0x40000004 (0x50504342)

Total of 1 word were the same

=> md 100000 C

```
00100000: 27051956 50ff4342 6f6f7420 312e312e      '.. VP. CBoot 1. 1.
```

```
00100010: 3520284d 61722032 31203230 3032202d      5 (Mar 21 2002 -
```

```
00100020: 2031393a 35353a30 34290000 00000000      19:55:04).....
```

=> md 40000000 C

```
40000000: 27051956 50504342 6f6f7420 312e312e      '.. VPPCBoot 1. 1.
```

```
40000010: 3520284d 61722032 31203230 3032202d      5 (Mar 21 2002 -
```

```
40000020: 2031393a 35353a30 34290000 00000000      19:55:04).....
```

=>

注意: 像大部分内存比较指令一样, 可以指定比较单元的大小: 32 位 (长字), 16 位 (短字) 或 8 位 (字节), 不同的比较单元将会产生不同的结果。系统默认按 4 字节的长字为单位进行比较。如下例:

=> cmp.l 100000 40000000 400

word at 0x00100004 (0x50ff4342) != word at 0x40000004 (0x50504342)

Total of 1 word were the same

=> cmp.w 100000 40000000 800

halfword at 0x00100004 (0x50ff) != halfword at 0x40000004 (0x5050)

```
Total of 2 halfwords were the same
=> cmp.b 100000 40000000 1000
byte at 0x00100005 (0xff) != byte at 0x40000005 (0x50)
Total of 5 bytes were the same
=>
```

4.5.2.4. cp – 数据拷贝

cp 命令应该是最常用的命令之一，用 cp 命令可进行烧录 Flash，当然，也可进行内存中数据的拷贝。

在 U-Boot 中敲入相应的帮助命令，可出现以下信息：

```
=> help cp
cp [.b, .w, .l] source target count
- copy memory
```

也就是说，内存拷贝时支持三种拷贝方式：按字节拷贝，按字拷贝和按长字拷贝。由于按字拷贝和按长字拷贝需要进行换算，因此更常用的方法是按字拷贝。实际上，在 U-Boot 的 cmd_mem.c 中的 do_mem_cp() 函数中，最后都是调用 flash_write ((uchar *)addr, dest, count*size) 进行烧录 Flash 的，其中字节数的不同会影响 size 的大小，但对于 flash_write 而言，最后获得的字节数都是一样，而且调用的函数也是一样，因此，如果转换成字和长字的方式并不会带来额外的好处。

常用的拷贝命令如下：

```
cp.b 0x10000000 0x0c000000 0x20000
```

4.5.2.5. md – 内存显示

md 也是常见内存和 Flash 的命令之一，md 命令将以 16 进制显示指定地址的内容。

在 U-Boot 中敲入相应的帮助命令，将出现以下信息：

```
=> help cp
md [.b, .w, .l] address [# of objects]
- memory display
```

即显示地址的内容可按字节，双字节和四字节显示。

4.5.3 Flash 操作命令

4.5.3.1. cp – 数据拷贝

该命令的详细介绍参见“内存控制命令”一节。

4.5.3.2. flinfo – 打印 Flash 信息

该命令的详细介绍参见“信息查询命令”一节。

4.5.3.3. erase – 擦除 Flash

erase 命令可擦除 Flash 中的内容，erase 命令是最常用的命令之一。因为每次 Flash 烧录前都必须先对 Flash 进行擦除，这样 Flash 才可能写入数据。所谓的擦除操作会把 Flash 的内容都置为 0xFFFFFFFF。

关于 Flash 的擦除命令，利用 help erase 可得到更详细的命令说明：

=> help erase

erase start end

- erase FLASH from addr 'start' to addr 'end'

erase N:SF[-SL]

- erase sectors SF-SL in FLASH bank # N

erase bank N

- erase FLASH bank # N

erase all

- erase all FLASH banks

erase start end

通过指定擦除的起始地址和结束地址来擦除一块 Flash 空间。这种方法是最常用的方法，但同时也是很容易出错的一种方法。

对 Flash 的擦除操作有块擦除和芯片擦除，块擦除是把某一擦除块的内容都变为 0xFF，芯片擦除是把整块 Flash 的内容都变为 0xFF。若要进从 ÷ 行块擦除操作，必须指定的擦除的起始地址为可擦除块的起始地址，擦除的 Flash 结束地址为 Flash 的某擦除块的结束地址。因此，若要对 Flash 进行块擦除，必须先了解相应的 Flash 芯片的块大小。

然后，必须指定结束地址为块的结束地址，比如 Flash 的块大小为 128K 时，很多人常犯这种错误：

```
erase 0x0 0x20000
```

因此，U-Boot 通常会提示以下错误：Error: end address not on sector boundary。

由于 0x20000 为 128K，因此第一个可擦除块的结束地址为 0x1ffff，正确的命令应该是：

```
erase 0x0 0x1FFFF
```

erase N:SF[-SL]

通过指定 bank 号和擦除块的范围进行 Flash 擦除。此处，需要指定正确的 bank 号。一个 bank 通常指连接到 CPU 片选的一块存储区域，通常 bank 0 指的是内存，bank 1 指的是 Flash。但有的板子包括两个 bank 的 Flash。这些取决于实际的系统，不同的系统配置不一致。

若要擦除 Flash 的第一块和第二块区域，可使用如下命令：

```
erase 1:0-1
```

erase bank N

erase all

这两条命令常用于整块 Flash 的擦除。

4.5.3.4. protect – 打开和关闭写保护

为了防止 Flash 的重要区域被不小心写坏，通常 Flash ROM 芯片有写保护功能，在对起始的块进行擦除和读写之前需要打开写保护。

U-Boot 支持的写保护命令的详细说明可通过以下命令查询：

=> help protect

```
protect on start end
    - protect FLASH from addr 'start' to addr 'end'
protect on N:SF[-SL]
    - protect sectors SF-SL in FLASH bank # N
protect on bank N
    - protect FLASH bank # N
protect on all
    - protect all FLASH banks
protect off start end
    - make FLASH from addr 'start' to addr 'end' writable
protect off N:SF[-SL]
    - make sectors SF-SL writable in FLASH bank # N
protect off bank N
    - make FLASH bank # N writable
protect off all
    - make all FLASH banks writable
```

实际保护的级别取决于实际的硬件和相应的驱动程序。对于 U-Boot 而言，通常只自动对起始的第一块区域和第二块区域加上写保护，因为 U-Boot 默认把 U-Boot 自身放在 Flash 的第一块区域，而把相应环境变量放在 Flash 的第二块区域。

因此，对 Flash 进行擦除和烧录前，需要执行相应的 `protect off` 命令关闭相应的保护机制，当擦除和烧录完成后再执行 `protect on` 命令打开相应的保护机制。

4.5.4 运行控制命令

4.5.4.1. bootm – 加载和启动映像

`bootm` 用于加载并启动 U-Boot 能辨识的操作系统映像，即 `bootm` 加载的映像必须是用 `mkimage` 工具打过包的映像，`bootm` 不能启动直接的内核映像，因为 `bootm` 必须从映像的头获取映像的一些信息，比如操作系统的类型，映像是否压缩，映像的加载地址和压缩地址等。更详细的映像头信息可以查看 `mkimage` 工具的说明。而 `bootm` 的详细用法可通过 `help bootm` 获得。

=> help bootm

```
bootm [addr [arg ...]]
```

- boot application image stored in memory passing arguments 'arg ...'; when booting a Linux kernel, 'arg' can be the address of an initrd image

Bootm 用于将内核映像加载到指定的地址，如果需要还要进行解压映像。然后根据操作系统和体系结构的不同给内核传递不同的内核参数，最后启动内核。

bootm 可以有两个参数，第一个参数为内核映像的地址，它可以是 RAM 地址或者 Flash 地址。第二个参数是可选参数，即 initrd 映像的地址，当采用 Ramdisk 作为根文件系统时需要使用 bootm 的第二个参数。当需要加载 initrd 映像时，首先 U-Boot 把内核映像加载到指定地址，然后再把 Ramdisk 映像加载到指定地址，同时把 Ramdisk 映像的大小和地址告知内核。

4.5.4.2 go – 启动映像

go 命令是另外一种启动映像的命令，和 bootm 命令相比，go 命令将直接跳转到某个地址然后执行放在该地址处的映像。go 命令并不会设置环境变量，也不能解析映像。因此，go 命令更经常用于加载 standalone 的应用程序，非常适于加载类似于硬件测试程序和 bootstrap 代码等，因为这种应用程序并不需要复杂的环境。

Go 命令的说明可通过“help go”来查看 go 命令的帮助信息：

```
=> help go
```

```
go addr [arg ...]
```

- start application at address 'addr', passing 'arg' as arguments

4.5.5 数据下载命令

4.5.5.1 tftpboot – 通过 tftp 协议下载映像

tftpboot 命令通过 tftp 协议下载映像，该命令是最常用的数据下载命令。若需要使用 tftpboot 命令，必须配置 dhcp 服务器和 tftp 服务器，关于这两个服务器的配置，请参见具体环境设置章节。需要注意的是 tftpboot 命令只会到指定的目录下下载映像，即开发主机上 /etc/xinetd.d/tftp 下 server_args 中配置的目录下，比如“-s /tftpboot”将指定默认的下载目录为“/tftpboot”。因此，若想通过 tftpboot 命令下载映像，需要先把相应映像拷贝到具体的目录下。

tftpboot 命令的详细解析可通过“help tftpboot”查看其帮助信息：

```
=> help tftpboot
```

```
tftpboot [loadAddress] [bootfilename]
```

tftpboot 命令的第一个参数为加载地址，这个地址指的是内存地址，不能是 flash 地址。通常不能直接通过 tftpboot 命令烧录到 Flash 中，必须先通过 tftpboot 命令下载到内存中，然后通过“cp”命令进行烧录。如果指定的为 Flash 地址时，U-Boot 或其它 Flash 上的程序通常会有一些莫名其妙的行为，比如 U-Boot 被莫名其妙的被修改几个字节，导致 U-Boot 不能正常工作。

4.5.5.2 bootp – 通过 bootp 协议下载映像

bootp 命令将通过 bootp 协议下载映像。Bootp 是 bootstrap Protocol 的简称，Bootp 协议将使用 TCP/IP 网络协议的 UDP 67/68 两个通讯端口，客户机从服务器得到 IP 地址、服务器的 IP 地址、启动映像文件名、网关 IP 等信息。

tftpboot 命令的详细解析可通过“help bootp”查看其帮助信息：

```
=> help bootp
    bootp [loadAddress] [bootfilename]
```

bootp 命令的行为和 tftpboot 命令差不多，只不过是用的不同的协议。在实际的源码中，不管是 bootp 命令（调用 do_bootp 函数）还是 tftpboot 命令（调用 do_tftpb 函数）最终都将调用 netboot_common() 函数，只不过是 proto 不同，而 proto 仅在 size = NetLoop(proto) 中被调用。

4.5.6 环境变量设置命令

4.5.6.1 printenv – 打印当前的环境变量

printenv 可以通过 help printenv 得到该命令的帮助信息，如下所示：

```
=> help printenv
    printenv
        - print values of all environment variables
    printenv name ...
        - print value of environment variable 'name'
```

从上面可以看出，printenv 可以打印所有环境变量的设置或者一部分环境变量的设置，即通过给定参数打印具体的环境变量，比如“printenv bootargs”，将打印类似于如下信息：

```
bootargs =mem=32M console=ttyS0,115200n8 ip=dhcp root=/dev/nfs
```

反之，如果使用“printenv”将打印当前设置的所有环境变量信息：

```
bootcmd=bootp;
bootdelay=5
baudrate=115200
stdin=serial
stdout=serial
stderr=serial
rootpath=/opt/eldk/ppc_8xx
ipaddr=10.0.0.99
serverip=10.0.0.2
```

4.5.6.2 setenv – 设置环境变量

```
=> help setenv
setenv name value ...
    - set environment variable 'name' to 'value ...'
setenv name
    - delete environment variable 'name'
```

环境变量的设置使用“setenv”命令，可通过 setenv 设置和删除某个环境变量，当 setenv 只带一个参数的话，就是删除该环境变量。当 setenv 带有多个参数时，将把第二个参数作为环境变量，而其它的参数作为该环境变量的说明，比如：

```
=> setenv bootargs mem=32M console=ttyS0,115200n8 ip=dhcp root=/dev/nfs
```

将把环境变量“bootargs”设置成 “mem=32M console=ttyS0,115200n8 ip=dhcp root=/dev/nfs”。当环境变量设置成功后，可通过 printenv 进行查看，比如对于上例，可通过：

```
=>printenv bootargs
bootargs= mem=32M console=ttyS0,115200n8 ip=dhcp root=/dev/nfs
```

当“setenv”只带一个参数时，将会删除该环境变量设置，比如对于刚设置环境变量“bootargs”，可使用以下命令删除：

```
=>setenv bootargs
=>printenv bootargs
## Error: "foo" not defined
```

关于 setenv，需要注意以下两点：

(1) 若想把某个环境变量设置成多条语句时，必须使用双引号””。比如，在开发阶段，需要经常设置 bootcmd，让其执行多条命令，其正确的设置如下所示：

```
setenv bootcmd “tftpboot 0x10000000 uImage; bootm 0x10000000;”
```

如果调用

```
setenv bootcmd tftpboot 0x10000000 uImage; bootm 0x10000000
```

那么系统将首先设置成功了这条启动命令“tftpboot 0x10000000 uImage;”

接着因为遇到“;”号，系统认为环境变量的设置已经完成，因此将执行用户输入的下一条指令，即系统将开始执行“bootm 0x10000000”指令。这显然违反了我们的初衷。

(2) setenv 并不会把相应环境变量存储到 Flash 中，每次重新上电后，刚才设置的环境变量将会丢失。若想相应环境变量存储到 Flash 中，必须调用 saveenv 命令进行保存环境变量的设置。

4.5.6.3 saveenv – 设置环境变量

=> help saveenv

saveenv - No help available.

saveenv 将把 RAM 中的环境变量设置保存到 Flash 中，这样才能保证系统重新启动后设置的环境变量仍然有效。该命令没有参数，不能指定只保存那个环境变量，而是把所有环境变量都保存一遍。

4.6 U-Boot 环境变量

U-Boot 的环境变量是保存在非易失性介质中（比如 Flash）的一块数据，当 U-Boot 启动时将被拷贝到 RAM 中，这些被 CRC 保护的环境变量将用于配置系统，并可能给内核传递启动参数。

当 U-Boot 启动时，首先将调用 env_init() 进行初始化，从配置的环境变量所在地址 CFG_ENV_ADDR（该地址在 environment.h 中定义，被定义成 CFG_FLASH_BASE + CFG_ENV_OFFSET）处读取指定大小的数据来计算循环冗余校验值 CRC，并判断计算出来的 CRC 值是否和存放在环境变量块开头的 CRC 值相等。若两个 CRC 值相等，证明数据没有破坏，因此将把保存在 Flash 上的地址作为环境变量的地址（即把 gd->env_addr 设置为 offsetof(env_t, data)）。否则，如果检验出错，证明数据已经出错误，可能是由于不正确的 Flash 读写导致了数据的不可靠，因此系统将采用默认的环境变量设置，默认的环境变量设置指的是系统编译就已经确定的配置，环境变量的默认设置主要通过一些宏的方式进行设置的。

默认的环境变量设置都在 default_environment 中设置的，如下所示：

```
uchar default_environment[] = {
#ifdef CONFIG_BOOTARGS
    "bootargs=" CONFIG_BOOTARGS "\0"
#endif
#ifdef CONFIG_BOOTCOMMAND
    "bootcmd=" CONFIG_BOOTCOMMAND "\0"
#endif
#ifdef CONFIG_RAMBOOTCOMMAND
    "ramboot=" CONFIG_RAMBOOTCOMMAND "\0"
#endif
#ifdef CONFIG_NFSBOOTCOMMAND
    "nfsboot=" CONFIG_NFSBOOTCOMMAND "\0"
#endif
#ifdef CONFIG_BOOTDELAY
    "bootdelay=" MK_STR(CONFIG_BOOTDELAY) "\0"
#endif
...
}
```

从上面可以看出，一个环境变量的设置取决于相应的宏是否已经被定义了，比如只有定义的宏 `CONFIG_BOOTARGS` 才会定义环境变量“bootargs”。由于不同的处理器和目标板可能会采取不同的设置，因此通过这种方式可做到和目标板无关，各个目标板可以定义自己的环境变量参数。比如 `include/configs/omap1510inn.h` 中这样定义 `CONFIG_BOOTARGS`:

```
#define CONFIG_BOOTARGS "console=ttyS0,115200n8    noinitrd    root=/dev/nfs  
ip=bootp"
```

而在 `MPC8260` 对应的配置文件中定义却不同：

```
#define CONFIG_BOOTARGS    "root=/dev/mtdblock2"
```

环境变量初始化完成后，调用 `printenv` 时就可把相应的参数打印出来。而其中有些环境变量比如 `bootargs`，可能就会传入内核，作为内核的启动参数。

4.6.1 环境变量的设置

环境变量的设置有两种方法，一种方法是通过修改静态配置，即通过修改 U-Boot 的文件中相关环境变量的配置。不过此类方法在实际中显然非常不实际，每次需要修改一次环境变量都需要重新编译整个 U-Boot。

另一种方法是利用 U-Boot 提供的“`setenv`”动态修改环境变量的设置，当设置完成后再通过“`saveenv`”把相应环境变量设置保存到 `Flash` 等非易失性存储设备上。

4.6.2 重要的环境变量

在嵌入式 Linux 开发中，必须注意以下一些常用的环境变量：

4.2.1 bootcmd

`bootcmd` 环境变量用于设置启动时自动加载的命令序列，但该环境变量只有在设置了 `CONFIG_BOOTDELAY` 时才会生效，即实际上 `main_loop()` 函数只有在 `CONFIG_BOOTDELAY` 被定义了而且大于 0 时，才会对“`bootcmd`”进行解析。

利用 `bootcmd` 环境变量，可以节省一些手工进行的工作，比如每次都希望从 `tftp` 根目录下下载内核映像，然后再启动相应的内核，那么我们把这工作可写成“`bootcmd`”环境变量：

```
=>setenv bootcmd "tftpboot 0x10000000 uImage; bootm 0x10000000"
```

否则，我们只能每次手动输入一些命令，进行下载映像并启动映像。

4.2.2 bootargs

bootargs 用以设置内核启动参数，即用以设置传递给内核的参数。bootargs 的设置将被传递给 Linux 内核。

在 U-Boot 中，可以通过宏 CONFIG_BOOTARGS 来静态指定内核参数，这样需要重新编译 U-Boot 才能使新的宏 CONFIG_BOOTARGS 生效。

另外一种方法是通过“setenv bootargs ****”来重新设定内核参数，当动态的设定后将修改环境变量中相应的设置。当内核启动时，将对这些启动参数进行处理，即拷贝到相应的内存地址处。对于 ARM 体系结构的 Linux 操作系统而言，lib_arm 目录下的 armlinux.c 文件中的函数 do_bootm_linux() 将对启动参数进行分析，不过，必须定义了宏 CONFIG_CMDLINE_TAG 时才会对命令行进行分析。若想不对命令行进行解析，可以在对应的头文件（比如 include/configs/omap1510inn.h）去除相应的定义即可。

- (1) 首先调用 `commandline = getenv("bootargs")` 获得系统设置的命令行参数。
- (2) 调用 `setup_commandline_tag()` 对内核命令行参数进行处理。在 `setup_commandline_tag()` 将把真正的命令行（判断有效性，并去除头部空格后）拷贝到全局变量 `params` 中。因为内核命令行参数是需要传递给内核的，因此必须放入一块内核在解析命令行参数之前不会使用的内存中，这点必须和操作系统初始化代码的假设一致。

关于全局变量 `params`，其初始化是在 `setup_start_tag` 中进行初始化的：

```
params = (struct tag *) bd->bi_boot_params;
```

而相应的 `bi_boot_params` 是在板级对应的文件中初始化的，比如对于 omap5912 而言，在 `board/omap5912osk/omap5912osk.c` 中的 `board_init()` 函数中进行初始化的：

```
gd->bd->bi_boot_params = 0x10000100;
```

这里，有以下两点是需要注意的：

- (1) 由于在 U-Boot 中并没有操作系统，因此并无一个内存管理机制来管理内存，此处采取直接给定起始地址的方式来使用内存是没有问题的。但需要保证内存的地址不会互相冲突。
- (2) 这里指定的地址必须和操作系统默认的内核参数起始地址相一致，这点比较容易理解。比如 U-Boot 把内核参数放入 A 地址处，那么操作系统在 B 地址处进行解析内核参数无疑会出错。就 Linux 而言，比如在 `arch/arm/mach-omap/innovator.c` 中通过以下语句指定了内核参数的地址：

```
MACHINE_START(INNOVATOR, "TI-Innovator/OMAP1510")
    MAINTAINER("MontaVista Software, Inc.")
    BOOT_MEM(0x10000000, 0xe0000000, 0xe0000000)
    BOOT_PARAMS(0x10000100)
    FIXUP(fixup_innovator)
    MAPIO(innovator_map_io)
    INITIRQ(innovator_init_irq)
MACHINE_END
```

4.2.3 ipaddr 和 ethaddr

这两个环境变量用于设置目标机的地址信息，`ipaddr` 用以设置 IP 地址信息，而 `ethaddr` 用于设置以太网地址。在嵌入式处理板上的以太网控制器并没有相应的以太网地址信息，因此此处的以太网地址可以随意指定，但需要保证此处设定的以太网地址和 DHCP 服务器中设定的一致。

假如在 DHCP 服务器的配置文件 `/etc/dhcpd.conf` 中，有以下一些配置：

```
host target_test
{
    hardware ethernet 08:00:2b:4c:59:23;
    fixed-address 192.168.1.10;
}
```

那么，此处的设置应该和这些配置相匹配，因此可设置如下：

```
setenv ipaddr 192.168.1.10
setenv ethaddr 08:00:2b:4c:59:23
```

4.2.4 serverip

`serverip` 用以设置服务器的 IP 地址，也就是开发主机的 IP 地址。这里，目标板相当于客户端，向 `tftp` 的服务器端请求服务。

若该项设置不正确，则 `tftpboot` 命令没法从开发主机下载文件。