嵌入式 Linux 系统开发技术详解-基于 ARM

6.1 Bootloader

对于计算机系统来说，从开机上电到操作系统启动需要一个引导过程。嵌入式 Linux 系统同样离不开引导程序，这个引导程序就叫作 Bootloader。

6.1.1 Bootloader 介绍

Bootloader 是在操作系统运行之前执行的一段小程序。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表，从而建立适当的系统软硬件环境，为最终调用操作系统内核做好准备。

对于嵌入式系统，Bootloader 是基于特定硬件平台来实现的。因此，几乎不可能为所有的嵌入式系统建立一个通用的 Bootloader，不同的处理器架构都有不同的 Bootloader。Bootloader 不但依赖于 CPU 的体系结构，而且依赖于嵌入式系统板级设备的配置。对于 2 块不同的嵌入式板而言，即使它们使用同一种处理器，要想让运行在一块板子上的 Bootloader 程序也能运行在另一块板子上，一般也都需要修改 Bootloader 的源程序。

反过来，大部分 Bootloader 仍然具有很多共性，某些 Bootloader 也能够支持多种体系结构的嵌入式系统。例如，U-Boot 就同时支持 PowerPC、ARM、MIPS 和 X86 等体系结构，支持的板子有上百种。通常，它们都能够自动从存储介质上启动，都能够引导操作系统启动，并且大部分都可以支持串口和以太网接口。

本章将对各种 Bootloader 总结分类，分析它们的共同特点。以 U-Boot 为例，详细讨论 Bootloader 的设计与实现。

6.1.2 Bootloader 的启动

Linux 系统是通过 Bootloader 引导启动的。一上电，就要执行 Bootloader 来初始化系统。可以通过第 4 章的 Linux 启动过程框图回顾一下。

系统加电或复位后，所有 CPU 都会从某个地址开始执行，这是由处理器设计决定的。比如，X86 的复位向量在高地址端，ARM 处理器在复位时从地址 0x00000000 取第一条指令。嵌入

式系统的开发板都要把板上 ROM 或 Flash 映射到这个地址。因此，必须把 Bootloader 程序存储在相应的 Flash 位置。系统加电后，CPU 将首先执行它。

主机和目标机之间一般有串口可以连接，Bootloader 软件通常会通过串口来输入输出。例如：输出出错或者执行结果信息到串口终端，从串口终端读取用户控制命令等。

Bootloader 启动过程通常是多阶段的，这样既能提供复杂的功能，又有很好的可移植性。例如：从 Flash 启动的 Bootloader 多数是两阶段的启动过程。从后面 U-Boot 的内容可以详细分析这个特性。

大多数 Bootloader 都包含 2 种不同的操作模式：本地加载模式和远程下载模式。这 2 种操作模式的区别仅对于开发人员才有意义，也就是不同启动方式的使用。从最终用户的角度看，Bootloader 的作用就是用来加载操作系统，而并不存在所谓的本地加载模式与远程下载模式的区别。

因为 Bootloader 的主要功能是引导操作系统启动，所以我们详细讨论一下各种启动方式的特点。

1. 网络启动方式

这种方式开发板不需要配置较大的存储介质，跟无盘工作站有点类似。但是使用这种启动方式之前，需要把 Bootloader 安装到板上的 EPROM 或者 Flash 中。Bootloader 通过以太网接口远程下载 Linux 内核映像或者文件系统。第 4 章介绍的交叉开发环境就是以网络启动方式建立的。这种方式对于嵌入式系统开发来说非常重要。

使用这种方式也有前提条件，就是目标板有串口、以太网接口或者其他连接方式。串口一般可以作为控制台，同时可以用来下载内核影像和 RAMDISK 文件系统。串口通信传输速率过低，不适合用来挂接 NFS 文件系统。所以以太网接口成为通用的互连设备，一般的开发板都可以配置 10M 以太网接口。

对于 PDA 等手持设备来说，以太网的 RJ-45 接口显得大了些，而 USB 接口，特别是 USB 的迷你接口，尺寸非常小。对于开发的嵌入式系统，可以把 USB 接口虚拟成以太网接口来通讯。这种方式在开发主机和开发板两端都需要驱动程序。

另外，还要在服务器上配置启动相关网络服务。Bootloader 下载文件一般都使用 TFTP 网络

协议，还可以通过 DHCP 的方式动态配置 IP 地址。

DHCP/BOOTP 服务为 Bootloader 分配 IP 地址，配置网络参数，然后才能够支持网络传输功能。如果 Bootloader 可以直接设置网络参数，就可以不使用 DHCP。

TFTP 服务为 Bootloader 客户端提供文件下载功能，把内核映像和其他文件放在/tftpboot 目录下。这样 Bootloader 可以通过简单的 TFTP 协议远程下载内核映像到内存。如图 6.1 所示。

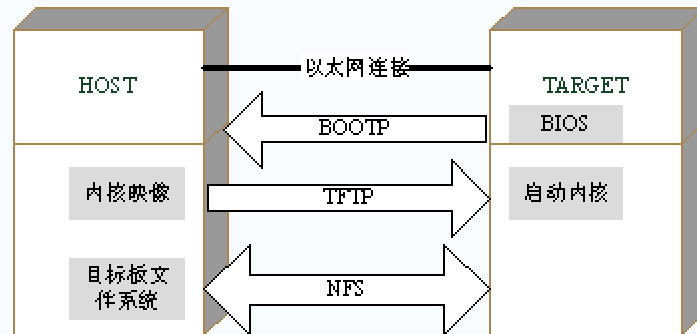

图 6.1　网络启动示意图

大部分引导程序都能够支持网络启动方式。例如：BIOS 的 PXE（Preboot Execution Environment）功能就是网络启动方式；U-Boot 也支持网络启动功能。

2. 磁盘启动方式

传统的 Linux 系统运行在台式机或者服务器上，这些计算机一般都使用 BIOS 引导，并且使用磁盘作为存储介质。如果进入 BIOS 设置菜单，可以探测处理器、内存、硬盘等设备，可以设置 BIOS 从软盘、光盘或者某块硬盘启动。也就是说，BIOS 并不直接引导操作系统。那么在硬盘的主引导区，还需要一个 Bootloader。这个 Bootloader 可以从磁盘文件系统中把操作系统引导起来。

Linux 传统上是通过 LILO（LInux LOader）引导的，后来又出现了 GNU 的软件 GRUB（GRand Unified Bootloader）。这 2 种 Bootloader 广泛应用在 X86 的 Linux 系统上。你的开发主机可能就使用了其中一种，熟悉它们有助于配置多种系统引导功能。

LILO 软件工程是由 Werner Almesberger 创建，专门为引导 Linux 开发的。现在 LILO 的维护者是 John Coffman，最新版本下载站点：http://lilo.go.dyndns.org。LILO 有详细的文档，例如 LILO 套件中附带使用手册和参考手册。此外，还可以在 LDP 的“LILO mini-HOWTO”

中找到 LILO 的使用指南。

GRUB 是 GNU 计划的主要 bootloader。GRUB 最初是由 Erich Boleyn 为 GNU Mach 操作系统撰写的引导程序。后来有 Gordon Matzigkeit 和 Okuji Yoshinori 接替 Erich 的工作，继续维护和开发 GRUB。GRUB 的网站 http://www.gnu.org/software/grub/上有对套件使用的说明文件，叫作《GRUB manual》。GRUB 能够使用 TFTP 和 BOOTP 或者 DHCP 通过网络启动，这种功能对于系统开发过程很有用。

除了传统的 Linux 系统上的引导程序以外，还有其他一些引导程序，也可以支持磁盘引导启动。例如：LoadLin 可以从 DOS 下启动 Linux；还有 ROLO、LinuxBIOS，U-Boot 也支持这种功能。

3. Flash 启动方式

大多数嵌入式系统上都使用 Flash 存储介质。Flash 有很多类型，包括 NOR Flash、NAND Flash 和其他半导体盘。其中，NOR Flash（也就是线性 Flash）使用最为普遍。

NOR Flash 可以支持随机访问，所以代码是可以直接在 Flash 上执行的。Bootloader 一般是存储在 Flash 芯片上的。另外，Linux 内核映像和 RAMDISK 也可以存储在 Flash 上。通常需要把 Flash 分区使用，每个区的大小应该是 Flash 擦除块大小的整数倍。图 6.2 是 Bootloader 和内核映像以及文件系统的分区表。

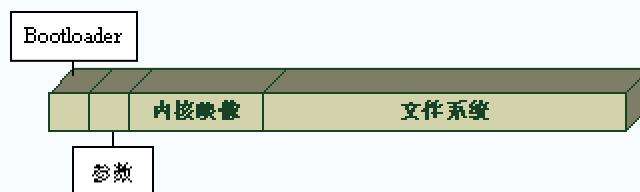

图 6.2 Flash 存储示意图

Bootloader 一般放在 Flash 的底端或者顶端，这要根据处理器的复位向量设置。要使 Bootloader 的入口位于处理器上电执行第一条指令的位置。

接下来分配参数区，这里可以作为 Bootloader 的参数保存区域。

再下来内核映像区。Bootloader 引导 Linux 内核，就是要从这个地方把内核映像解压到 RAM 中去，然后跳转到内核映像入口执行。

然后是文件系统区。如果使用 Ramdisk 文件系统，则需要 Bootloader 把它解压到 RAM 中。如果使用 JFFS2 文件系统，将直接挂接为根文件系统。这两种文件系统将在第 12 章详细讲解。

最后还可以分出一些数据区，这要根据实际需要和 Flash 大小来考虑了。

这些分区是开发者定义的，Bootloader 一般直接读写对应的偏移地址。到了 Linux 内核空间，可以配置成 MTD 设备来访问 Flash 分区。但是，有的 Bootloader 也支持分区的功能，例如: Redboot 可以创建 Flash 分区表，并且内核 MTD 驱动可以解析出 redboot 的分区表。

除了 NOR Flash，还有 NAND Flash、Compact Flash、DiskOnChip 等。这些 Flash 具有芯片价格低，存储容量大的特点。但是这些芯片一般通过专用控制器的 I/O 方式来访问，不能随机访问，因此引导方式跟 NOR Flash 也不同。在这些芯片上，需要配置专用的引导程序。通常，这种引导程序起始的一段代码就把整个引导程序复制到 RAM 中运行，从而实现自举启动，这跟从磁盘上启动有些相似。

6.1.3 Bootloader 的种类

嵌入式系统世界已经有各种各样的 Bootloader，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。

首先区分一下“Bootloader”和“Monitor”的概念。严格来说，“Bootloader”只是引导设备并且执行主程序的固件；而“Monitor”还提供了更多的命令行接口，可以进行调试、读写内存、烧写 Flash、配置环境变量等。“Monitor”在嵌入式系统开发过程中可以提供很好的调试功能，开发完成以后，就完全设置成了一个“Bootloader”。所以，习惯上大家把它们统称为 Bootloader。

表 6.1 列出了 Linux 的开放源码引导程序及其支持的体系结构。表中给出了 X86 ARM PowerPC 体系结构的常用引导程序，并且注明了每一种引导程序是不是“Monitor”。

表 6.1 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	否	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

对于每种体系结构，都有一系列开放源码 Bootloader 可以选用。

（1）X86

X86 的工作站和服务器上一般使用 LILO 和 GRUB。LILO 是 Linux 发行版主流的 Bootloader。不过 Redhat Linux 发行版已经使用了 GRUB，GRUB 比 LILO 有更有好的显示界面，使用配置也更加灵活方便。

在某些 X86 嵌入式单板机或者特殊设备上，会采用其他 Bootloader，例如：ROLO。这些 Bootloader 可以取代 BIOS 的功能，能够从 FLASH 中直接引导 Linux 启动。现在 ROLO 支持的开发板已经并入 U-Boot，所以 U-Boot 也可以支持 X86 平台。

（2）ARM

ARM 处理器的芯片商很多，所以每种芯片的开发板都有自己的 Bootloader。结果 ARM bootloader 也变得多种多样。最早有为 ARM720 处理器的开发板的固件，又有了armboot，StrongARM 平台的 blob，还有 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 U-Boot，所以 U-Boot 也支持 ARM/XSCALE 平台。U-Boot 已经成为 ARM 平台事实上的标准 Bootloader。

（3）PowerPC

PowerPC 平台的处理器有标准的 Bootloader，就是 ppcboot。PPCBOOT 在合并 armboot 等之后，创建了 U-Boot，成为各种体系结构开发板的通用引导程序。U-Boot 仍然是 PowerPC 平台的主要 Bootloader。

（4）MIPS

MIPS 公司开发的 YAMON 是标准的 Bootloader，也有许多 MIPS 芯片商为自己的开发板写了 Bootloader。现在，U-Boot 也已经支持 MIPS 平台。

（5）SH

SH 平台的标准 Bootloader 是 sh-boot。Redboot 在这种平台上也很好用。

（6）M68K

M68K 平台没有标准的 Bootloader。Redboot 能够支持 m68k 系列的系统。

值得说明的是 Redboot，它几乎能够支持所有的体系结构，包括 MIPS、SH、M68K 等体系结构。Redboot 是以 eCos 为基础，采用 GPL 许可的开源软件工程。现在由 core eCos 的开发人员维护，源码下载网站是 http://www.ecoscentric.com/snapshots。Redboot 的文档也相当完善，有详细的使用手册《RedBoot User’s Guide》。

6.2 U-Boot 编程

U-Boot 作为通用的 Bootloader，U-Boot 可以方便地移植到其他硬件平台上，其源代码也值得开发者们研究学习。

6.2.1 U-Boot 工程简介

最早，DENX 软件工程中心的 Wolfgang Denk 基于 8xxrom 的源码创建了 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 已经能够支持 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 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。

U-Boot 邮件列表网站：http://lists.sourceforge.net/lists/listinfo/u-boot-users/。

DENX 相关的网站：http://www.denx.de/re/DPLG.html。

6.2.2 U-Boot 源码结构

从网站上下载得到U-Boot源码包，例如：U-Boot-1.1.2.tar.bz2

解压就可以得到全部U-Boot源程序。在顶层目录下有18个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为3类。

- 第1类目录与处理器体系结构或者开发板硬件直接相关；
- 第2类目录是一些通用的函数或者驱动程序；
- 第3类目录是U-Boot的应用程序、工具或者文档。

表6.2列出了U-Boot顶层目录下各级目录存放原则。

表6.2 U-Boot的源码顶层目录说明

目　录	特　性	解 释 说 明
board	平台依赖	存放电路板相关的目录文件，例如：RPXlite(mpc8xx)、smdk2410(arm920t)、sc520_cdp(x86) 等目录
cpu	平台依赖	存放CPU相关的目录文件，例如：mpc8xx、ppc4xx、arm720t、arm920t、xscale、i386等目录
lib_ppc	平台依赖	存放对PowerPC体系结构通用的文件，主要用于实现PowerPC平台通用的函数
目　录	特　性	解 释 说 明
lib_arm	平台依赖	存放对ARM体系结构通用的文件，主要用于实现ARM平台通用的函数
lib_i386	平台依赖	存放对X86体系结构通用的文件，主要用于实现X86平台通用的函数
include	通用	头文件和开发板配置文件，所有开发板的配置文件都在configs目录下
common	通用	通用的多功能函数实现
lib_generic	通用	通用库函数的实现
Net	通用	存放网络的程序
Fs	通用	存放文件系统的程序
Post	通用	存放上电自检程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动
Disk	通用	硬盘接口程序
Rtc	通用	RTC的驱动程序
Dtt	通用	数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如helloworld
tools	工具	存放制作S-Record 或者 U-Boot格式的映像等工具，例如mkimage
Doc	文档	开发使用文档

U-Boot 的源代码包含对几十种处理器、数百种开发板的支持。可是对于特定的开发板，配置编译过程只需要其中部分程序。这里具体以 S3C2410 arm920t 处理器为例，具体分析 S3C2410 处理器和开发板所依赖的程序，以及 U-Boot 的通用函数和工具。

6.2.3 U-Boot 的编译

U-Boot 的源码是通过 GCC 和 Makefile 组织编译的。顶层目录下的 Makefile 首先可以设置开发板的定义，然后递归地调用各级子目录下的 Makefile，最后把编译过的程序链接成 U-Boot 映像。

1．顶层目录下的 Makefile

它负责 U-Boot 整体配置编译。按照配置的顺序阅读其中关键的几行。

每一种开发板在 Makefile 都需要有板子配置的定义。例如 smdk2410 开发板的定义如下。

```
smdk2410_config :   unconfig
    @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

执行配置 U-Boot 的命令 make smdk2410_config，通过./mkconfig 脚本生成 include/config.mk 的配置文件。文件内容正是根据 Makefile 对开发板的配置生成的。

```
ARCH    = arm
CPU     = arm920t
BOARD   = smdk2410
SOC     = s3c24x0
```

上面的 include/config.mk 文件定义了 ARCH、CPU、BOARD、SOC 这些变量。这样硬件平台依赖的目录文件可以根据这些定义来确定。SMDK2410 平台相关目录如下。

board/smdk2410/

cpu/arm920t/

cpu/arm920t/s3c24x0/

lib_arm/

include/asm-arm/

include/configs/smdk2410.h

再回到顶层目录的 Makefile 文件开始的部分，其中下列几行包含了这些变量的定义。

```
# load ARCH, BOARD, and CPU configuration
include include/config.mk
export      ARCH CPU BOARD VENDOR SOC
```

Makefile 的编译选项和规则在顶层目录的 config.mk 文件中定义。各种体系结构通用的规则直接在这个文件中定义。通过 ARCH、CPU、BOARD、SOC 等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在 ppc_config.mk、arm_config.mk、mips_config.mk 等文件中。

顶层目录的 Makefile 中还要定义交叉编译器，以及编译 U-Boot 所依赖的目标文件。

```
ifeq ($(ARCH),arm)
CROSS_COMPILE = arm-linux-         //交叉编译器的前缀
#endif
export  CROSS_COMPILE
…
# U-Boot objects....order is important (i.e. start must be first)
OBJS  = cpu/$(CPU)/start.o                //处理器相关的目标文件
…
LIBS  = lib_generic/libgeneric.a          //定义依赖的目录，每个目录下先把目标文件连接成*.a 文件。
```

```
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
…
```

然后还有 U-Boot 映像编译的依赖关系。

```
ALL = u-boot.srec u-boot.bin System.map
all:        $(ALL)
u-boot.srec:    u-boot
            $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
u-boot.bin: u-boot
            $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
……
u-boot:         depend $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
            UNDEF_SYM='$(OBJDUMP) -x $(LIBS) \
            |sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
            $(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) \
                --start-group $(LIBS) $(PLATFORM_LIBS) --end-group \
                -Map u-boot.map -o u-boot
```

Makefile 缺省的编译目标为 all，包括 u-boot.srec、u-boot.bin、System.map。u-boot.srec 和 u-boot.bin 又依赖于 U-Boot。U-Boot 就是通过 ld 命令按照 u-boot.map 地址表把目标文件组装成 u-boot。

其他 Makefile 内容就不再详细分析了，上述代码分析应该可以为阅读代码提供了一个线索。

2．开发板配置头文件

除了编译过程 Makefile 以外，还要在程序中为开发板定义配置选项或者参数。这个头文件是 include/configs/<*board_name*>.h。<*board_name*>用相应的 BOARD 定义代替。

这个头文件中主要定义了两类变量。

一类是选项，前缀是 CONFIG_，用来选择处理器、设备接口、命令、属性等。例如：

```
#define    CONFIG_ARM920T          1
#define    CONFIG_DRIVER_CS8900   1
```

另一类是参数，前缀是 CFG_，用来定义总线频率、串口波特率、Flash 地址等参数。例如：

```
#define      CFG_FLASH_BASE       0x00000000
#define CFG_PROMPT             "=>"
```

3．编译结果

根据对 Makefile 的分析，编译分为 2 步。第 1 步配置，例如：make smdk2410_config；第 2 步编译，执行 make 就可以了。

编译完成后，可以得到 U-Boot 各种格式的映像文件和符号表，如表 6.3 所示。

表 6.3　　U-Boot 编译生成的映像文件

文 件 名 称	说　明	文 件 名 称	说　明
System.map	U-Boot 映像的符号表	u-boot.bin	U-Boot 映像原始的二进制格式
u-boot	U-Boot 映像的 ELF 格式	u-boot.srec	U-Boot 映像的 S-Record 格式

U-Boot 的 3 种映像格式都可以烧写到 Flash 中，但需要看加载器能否识别这些格式。一般 u-boot.bin 最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到 Flash 中就可以了。U-Boot 和 u-boot.srec 格式映像都自带定位信息。

4．U-Boot 工具

在 tools 目录下还有些 U-Boot 的工具。这些工具有的也经常用到。表 6.4 说明了几种工具的

用途。

表 6.4 U-Boot 的工具

工 具 名 称	说 明	工 具 名 称	说 明
bmp_logo	制作标记的位图结构体	img2srec	转换 SREC 格式映像
envcrc	校验 u-boot 内部嵌入的环境变量	mkimage	转换 U-Boot 格式映像
gen_eth_addr	生成以太网接口 MAC 地址	updater	U-Boot 自动更新升级工具

这些工具都有源代码，可以参考改写其他工具。其中 mkimage 是很常用的一个工具，Linux 内核映像和 ramdisk 文件系统映像都可以转换成 U-Boot 的格式。

6.2.4 U-Boot 的移植

U-Boot 能够支持多种体系结构的处理器，支持的开发板也越来越多。因为 Bootloader 是完全依赖硬件平台的，所以在新电路板上需要移植 U-Boot 程序。

开始移植 U-Boot 之前，先要熟悉硬件电路板和处理器。确认 U-Boot 是否已经支持新开发板的处理器和 I/O 设备。假如 U-Boot 已经支持一块非常相似的电路板，那么移植的过程将非常简单。

移植 U-Boot 工作就是添加开发板硬件相关的文件、配置选项，然后配置编译。

开始移植之前，需要先分析一下 U-Boot 已经支持的开发板，比较出硬件配置最接近的开发板。选择的原则是，首先处理器相同，其次处理器体系结构相同，然后是以太网接口等外围接口。还要验证一下这个参考开发板的 U-Boot，至少能够配置编译通过。

以 S3C2410 处理器的开发板为例，U-Boot-1.1.2 版本已经支持 SMDK2410 开发板。我们可以基于 SMDK2410 移植，那么先把 SMDK2410 编译通过。

我们以 S3C2410 开发板 fs2410 为例说明。移植的过程参考 SMDK2410 开发板，SMDK2410 在 U-Boot-1.1.2 中已经支持。

移植 U-Boot 的基本步骤如下。

（1）在顶层 Makefile 中为开发板添加新的配置选项，使用已有的配置项目为例。

```
smdk2410_config    :       unconfig
        @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

参考上面 2 行，添加下面 2 行。

```
fs2410_config   :       unconfig
        @./mkconfig $(@:_config=) arm arm920t fs2410 NULL s3c24x0
```

（2）创建一个新目录存放开发板相关的代码，并且添加文件。

board/fs2410/config.mk

board/fs2410/flash.c

board/fs2410/fs2410.c

board/fs2410/Makefile

board/fs2410/memsetup.S

board/fs2410/u-boot.lds

（3）为开发板添加新的配置文件

可以先复制参考开发板的配置文件，再修改。例如：

```
$cp include/configs/smdk2410.h  include/configs/fs2410.h
```

如果是为一颗新的 CPU 移植，还要创建一个新的目录存放 CPU 相关的代码。

（4）配置开发板

```
$ make fs2410_config
```

（5）编译 U-Boot

执行 make 命令，编译成功可以得到 U-Boot 映像。有些错误是跟配置选项是有关系的，通常打开某些功能选项会带来一些错误，一开始可以尽量跟参考板配置相同。

（6）添加驱动或者功能选项

在能够编译通过的基础上，还要实现 U-Boot 的以太网接口、Flash 擦写等功能。对于 FS2410 开发板的以太网驱动和 smdk2410 完全相同，所以可以直接使用。CS 8900 驱动程序文件如下。

drivers/cs8900.c

drivers/cs8900.h

对于 Flash 的选择就麻烦多了，Flash 芯片价格或者采购方面的因素都有影响。多数开发板大小、型号不都相同。所以还需要移植 Flash 的驱动。每种开发板目录下一般都有 flash.c 这个文件，需要根据具体的 Flash 类型修改。例如：

board/fs2410/flash.c

（7）调试 U-Boot 源代码，直到 U-Boot 在开发板上能够正常启动。

调试的过程可能是很艰难的，需要借助工具，并且有些问题可能困扰很长时间。

6.2.5 添加 U-Boot 命令

U-Boot 的命令为用户提供了交互功能，并且已经实现了几十个常用的命令。如果开发板需要很特殊的操作，可以添加新的 U-Boot 命令。

U-Boot 的每一个命令都是通过 U_Boot_CMD 宏定义的。这个宏在 include/command.h 头文件中定义，每一个命令定义一个 cmd_tbl_t 结构体。

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

这样每一个 U-Boot 命令有一个结构体来描述。结构体包含的成员变量：命令名称、最大参数个数、重复数、命令执行函数、用法、帮助。

从控制台输入的命令是由 common/command.c 中的程序解释执行的。find_cmd() 负责匹配输入的命令，从列表中找出对应的命令结构体。

基于 U-Boot 命令的基本框架，来分析一下简单的 icache 操作命令，就可以知道添加新命令的方法。

（1）定义 CACHE 命令。在 include/cmd_confdefs.h 中定义了所有 U-Boot 命令的标志位。

```
#define CFG_CMD_CACHE       0x00000010ULL   /* icache, dcache       */
```

如果有更多的命令，也要在这里添加定义。

（2）实现 CACHE 命令的操作函数。下面是 common/cmd_cache.c 文件中 icache 命令部分的代码。

```
#if (CONFIG_COMMANDS & CFG_CMD_CACHE)
static int on_off (const char *s)
{       //这个函数解析参数，判断是打开 cache，还是关闭 cache
        if (strcmp(s, "on") == 0) {   //参数为 “on”
                return (1);
        } else if (strcmp(s, "off") == 0) {   //参数为 “off”
                return (0);
    }
    return (-1);
}

int do_icache ( cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{     //对指令 cache 的操作函数
      switch (argc) {
      case 2:              /* 参数个数为 1，则执行打开或者关闭指令 cache 操作 */
            switch (on_off(argv[1])) {
            case 0:      icache_disable();        //打开指令 cache
                  break;
            case 1:      icache_enable ();        //关闭指令 cache
                  break;
            }
           /* FALL TROUGH */
      case 1:           /* 参数个数为 0，则获取指令 cache 状态*/
           printf ("Instruction Cache is %s\n",
                   icache_status() ? "ON" : "OFF");
           return 0;
```

```
    default:  //其他缺省情况下，打印命令使用说明
        printf ("Usage:\n%s\n", cmdtp->usage);
        return 1;
    }
    return 0;
}
……
U_Boot_CMD( //通过宏定义命令
  icache,   2,   1,    do_icache,  //命令为 icache，命令执行函数为 do_icache()
  "icache  - enable or disable instruction cache\n",   //帮助信息
  "[on, off]\n"
  "    - enable or disable instruction cache\n"
);
……
#endif
```

U-Boot 的命令都是通过结构体__U_Boot_cmd_##name 来描述的。根据 U_Boot_CMD 在 include/command.h 中的两行定义可以明白。

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

还有，不要忘了在 common/Makefile 中添加编译的目标文件。

（3）打开 CONFIG_COMMANDS 选项的命令标志位。这个程序文件开头有#if 语句需要预处理是否包含这个命令函数。CONFIG_COMMANDS 选项在开发板的配置文件中定义。例如：SMDK2410 平台在 include/configs/smdk2410.h 中有如下定义。

```
/*********************************************************
```

```
 * Command definition
 ***********************************************************/
#define CONFIG_COMMANDS \
            (CONFIG_CMD_DFL  | \
            CFG_CMD_CACHE     | \
            CFG_CMD_REGINFO    | \
            CFG_CMD_DATE      | \
            CFG_CMD_ELF)
```

按照这 3 步，就可以添加新的 U-Boot 命令。

6.3 U-Boot 的调试

新移植的 U-Boot 不能正常工作，这时就需要调试了。调试 U-Boot 离不开工具，只有理解 U-Boot 启动过程，才能正确地调试 U-Boot 源码。

6.3.1 硬件调试器

硬件电路板制作完成以后，这时上面还没有任何程序，就叫作裸板。首要的工作是把程序或者固件加载到裸板上，这就要通过硬件工具来完成。习惯上，这种硬件工具叫作仿真器。

仿真器可以通过处理器的 JTAG 等接口控制板子，直接把程序下载到目标板内存，或者进行 Flash 编程。如果板上的 Flash 是可以拔插的，就可以通过专用的 Flash 烧写器来完成。在第 4 章介绍过目标板跟主机之间的连接，其中 JTAG 等接口就是专门用来连接仿真器的。

仿真器还有一个重要的功能就是在线调试程序，这对于调试 Bootloader 和硬件测试程序很有用。

从最简单的 JTAG 电缆，到 ICE 仿真器，再到可以调试 Linux 内核的仿真器。

复杂的仿真器可以支持与计算机间的以太网或者 USB 接口通信。

对于 U-Boot 的调试，可以采用 BDI2000。BDI2000 完全可以反汇编地跟踪 Flash 中的程序，也可以进行源码级的调试。

使用 BDI2000 调试 U-boot 的方法如下。

（1）配置 BDI2000 和目标板初始化程序，连接目标板。

（2）添加 U-Boot 的调试编译选项，重新编译。

U-Boot 的程序代码是位置相关的，调试的时候尽量在内存中调试，可以修改连接定位地址 TEXT_BASE。TEXT_BASE 在 board/<board_name>/config.mk 中定义。另外，如果有复位向量也需要先从链接脚本中去掉。链接脚本是 board/<board_name>/
u-boot.lds。

添加调试选项，在 config.mk 文件中查找，DBGFLAGS，加上-g 选项。然后重新编译 U-Boot。

（3）下载 U-Boot 到目标板内存。

通过 BDI2000 的下载命令 LOAD，把程序加载到目标板内存中。然后跳转到 U-Boot 入口。

（4）启动 GDB 调试。

启动 GDB 调试，这里是交叉调试的 GDB。GDB 与 BDI2000 建立链接，然后就可以设置断点执行了。

```
$ arm-linux-gdb u-boot
(gdb)target remote 192.168.1.100:2001
(gdb)stepi
(gdb)b start_armboot
(gdb)c
```

6.3.2 软件跟踪

假如 U-Boot 没有任何串口打印信息，手头又没有硬件调试工具，那样怎么知道 U-Boot 执行到什么地方了呢？可以通过开发板上的 LED 指示灯判断。

开发板上最好设计安装八段数码管等 LED，可以用来显示数字或者数字位。

U-Boot 可以定义函数 show_boot_progress (int status)，用来指示当前启动进度。在 include/common.h 头文件中声明这个函数。

```
#ifdef CONFIG_SHOW_BOOT_PROGRESS
void    show_boot_progress (int status);
#endif
```

CONFIG_SHOW_BOOT_PROGRESS 是需要定义的。这个在板子配置的头文件中定义。CSB226 开发板对这项功能有完整实现，可以参考。在头文件 include/configs/csb226.h 中，有下列一行。

```
#define CONFIG_SHOW_BOOT_PROGRESS       1
```

函数 show_boot_progress (int status)的实现跟开发板关系密切，所以一般在 board 目录下的文件中实现。看一下 CSB226 在 board/csb226/csb226.c 中的实现函数。

```
/** 设置 CSB226 板的 0、1、2 三个指示灯的开关状态
 * csb226_set_led: - switch LEDs on or off
 * @param led:   LED to switch (0,1,2)
 * @param state: switch on (1) or off (0)
 */
void csb226_set_led(int led, int state)
{
    switch(led) {
        case 0: if (state==1) {
                    GPCR0 |= CSB226_USER_LED0;
                } else if (state==0) {
                    GPSR0 |= CSB226_USER_LED0;
                }
                break;
        case 1: if (state==1) {
```

```
                        GPCR0 |= CSB226_USER_LED1;
                } else if (state==0) {
                        GPSR0 |= CSB226_USER_LED1;
                }
                break;
        case 2: if (state==1) {
                        GPCR0 |= CSB226_USER_LED2;
                } else if (state==0) {
                        GPSR0 |= CSB226_USER_LED2;
                }
                break;
    }
    return;
}
/** 显示启动进度函数，在比较重要的阶段，设置三个灯为亮的状态（1, 5, 15）*/
void show_boot_progress (int status)
{
    switch(status) {
        case  1: csb226_set_led(0,1); break;
        case  5: csb226_set_led(1,1); break;
        case 15: csb226_set_led(2,1); break;
    }
    return;
}
```

这样，在 U-Boot 启动过程中就可以通过 show_boot_progresss 指示执行进度。比如 hang()函数是系统出错时调用的函数，这里需要根据特定的开发板给定显示的参数值。

```
void hang (void)
{
	puts ("### ERROR ### Please RESET the board ###\n");
#ifdef CONFIG_SHOW_BOOT_PROGRESS
	show_boot_progress(-30);
#endif
	for (;;);
}
```

6.3.3 U-Boot 启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 6.3 所示。

看一下 board/smsk2410/u-boot.lds 这个链接脚本，可以知道目标程序的各部分链接顺序。第一个要链接的是 cpu/arm920t/start.o，那么 U-Boot 的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

1．cpu/arm920t/start.S

这个汇编程序是 U-Boot 的入口程序，开头就是复位向量的代码。

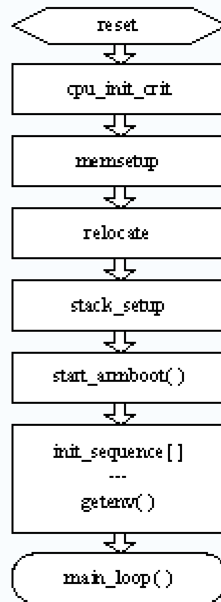

图 6.3 U-Boot 启动代码流程图

```
_start: b       reset       //复位向量
        ldr   pc, _undefined_instruction
        ldr   pc, _software_interrupt
        ldr   pc, _prefetch_abort
        ldr   pc, _data_abort
        ldr   pc, _not_used
        ldr   pc, _irq      //中断向量
        ldr   pc, _fiq      //中断向量
…
 /* the actual reset code  */
reset:          //复位启动子程序
        /* 设置 CPU 为 SVC32 模式 */
        mrs   r0,cpsr
        bic   r0,r0,#0x1f
        orr   r0,r0,#0xd3
        msr   cpsr,r0
/* 关闭看门狗 */
```

```
/* 这些初始化代码在系统重起的时候执行，运行时热复位从 RAM 中启动不执行 */
#ifdef CONFIG_INIT_CRITICAL
        bl      cpu_init_crit
#endif

relocate:                               /* 把 U-Boot 重新定位到 RAM */
        adr     r0, _start              /* r0 是代码的当前位置 */
        ldr     r1, _TEXT_BASE          /* 测试判断是从 Flash 启动，还是 RAM */
        cmp     r0, r1                  /* 比较 r0 和 r1，调试的时候不要执行重定位 */
        beq     stack_setup             /* 如果 r0 等于 r1，跳过重定位代码 */
        /* 准备重新定位代码 */
        ldr     r2, _armboot_start
        ldr     r3, _bss_start
        sub     r2, r3, r2              /* r2 得到 armboot 的大小 */
        add     r2, r0, r2              /* r2 得到要复制代码的末尾地址 */
copy_loop: /* 重新定位代码 */
        ldmia r0!, {r3-r10}     /*从源地址[r0]复制 */
        stmia r1!, {r3-r10}     /* 复制到目的地址[r1] */
        cmp     r0, r2          /* 复制数据块直到源数据末尾地址[r2] */
        ble     copy_loop

        /* 初始化堆栈等 */
stack_setup:
        ldr     r0, _TEXT_BASE                  /* 上面是 128 KiB 重定位的 u-boot */
        sub     r0, r0, #CFG_MALLOC_LEN         /* 向下是内存分配空间 */
        sub     r0, r0, #CFG_GBL_DATA_SIZE /* 然后是 bdinfo 结构体地址空间 */
#ifdef CONFIG_USE_IRQ
```

```
        sub     r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
        sub     sp, r0, #12       /* 为 abort-stack 预留 3 个字 */
clear_bss:
        ldr     r0, _bss_start        /* 找到 bss 段起始地址 */
        ldr     r1, _bss_end          /*  bss 段末尾地址    */
        mov     r2, #0x00000000       /* 清零 */
clbss_l:str r2, [r0]          /* bss 段地址空间清零循环...  */
        add     r0, r0, #4
        cmp     r0, r1
        bne     clbss_l
        /* 跳转到 start_armboot 函数入口，_start_armboot 字保存函数入口指针 */
        ldr     pc, _start_armboot
_start_armboot: .word start_armboot      //start_armboot 函数在 lib_arm/board.c 中实现
/* 关键的初始化子程序 */
cpu_init_crit:
……  //初始化 CACHE，关闭 MMU 等操作指令
        /* 初始化 RAM 时钟。
        * 因为内存时钟是依赖开发板硬件的，所以在 board 的相应目录下可以找到 memsetup.S
文件。
        */
        mov     ip, lr
        bl      memsetup          //memsetup 子程序在 board/smdk2410/memsetup.S 中实现
        mov     lr, ip
        mov     pc, lr
```

2. lib_arm/board.c

start_armboot 是 U-Boot 执行的第一个 C 语言函数，完成系统初始化工作，进入主循环，处理用户输入的命令。

```
void start_armboot (void)
{
	DECLARE_GLOBAL_DATA_PTR;
	ulong size;
	init_fnc_t **init_fnc_ptr;
	char *s;
	/* 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));
	monitor_flash_len = _bss_start - _armboot_start;
	/* 顺序执行 init_sequence 数组中的初始化函数 */
	for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
		if ((*init_fnc_ptr)() != 0) {
			hang ();
		}
	}
	/*配置可用的 Flash */
	size = flash_init ();
	display_flash_config (size);
	/* _armboot_start 在 u-boot.lds 链接脚本中定义 */
```

```
    mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);
    /* 配置环境变量，重新定位 */
    env_relocate ();
    /* 从环境变量中获取 IP 地址 */
    gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
    /* 以太网接口 MAC 地址 */
    ……
    devices_init ();      /* 获取列表中的设备 */
    jumptable_init ();
    console_init_r ();    /* 完整地初始化控制台设备 */
    enable_interrupts (); /* 使能例外处理 */
    /* 通过环境变量初始化 */
    if ((s = getenv ("loadaddr")) != NULL) {
        load_addr = simple_strtoul (s, NULL, 16);
    }
    /* main_loop()总是试图自动启动，循环不断执行 */
    for (;;) {
        main_loop ();     /* 主循环函数处理执行用户命令 -- common/main.c */
    }
    /* NOTREACHED - no way out of command loop except booting */
}
```

3. init_sequence[]

init_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

```
init_fnc_t *init_sequence[] = {
```

```
        cpu_init,               /* 基本的处理器相关配置 -- cpu/arm920t/cpu.c */
        board_init,             /* 基本的板级相关配置 -- board/smdk2410/smdk2410.c */
        interrupt_init,         /* 初始化例外处理 -- cpu/arm920t/s3c24x0/interrupt.c */
        env_init,               /* 初始化环境变量 -- common/cmd_flash.c */
        init_baudrate,          /* 初始化波特率设置 -- lib_arm/board.c */
        serial_init,            /* 串口通讯设置 -- cpu/arm920t/s3c24x0/serial.c */
        console_init_f,         /* 控制台初始化阶段 1 -- common/console.c */
        display_banner,         /* 打印 u-boot 信息 -- lib_arm/board.c */
        dram_init,              /* 配置可用的 RAM -- board/smdk2410/smdk2410.c */
        display_dram_config,    /* 显示 RAM 的配置大小 -- lib_arm/board.c */
        NULL,
};
```

6.3.4 U-Boot 与内核的关系

U-Boot 作为 Bootloader，具备多种引导内核启动的方式。常用的 go 和 bootm 命令可以直接引导内核映像启动。U-Boot 与内核的关系主要是内核启动过程中参数的传递。

1．go 命令的实现

```
/* common/cmd_boot.c  */
int do_go (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
        ulong addr, rc;
        int     rcode = 0;
        if (argc < 2) {
                printf ("Usage:\n%s\n", cmdtp->usage);
                return 1;
        }
```

```
        addr = simple_strtoul(argv[1], NULL, 16);
        printf ("## Starting application at 0x%08lX ...\n", addr);
        /*
         * pass address parameter as argv[0] (aka command name),
         * and all remaining args
         */
        rc = ((ulong (*)(int, char *[]))addr) (--argc, &argv[1]);
        if (rc != 0) rcode = 1;

        printf ("## Application terminated, rc = 0x%lX\n", rc);
        return rcode;
}
```

go 命令调用 do_go()函数，跳转到某个地址执行的。如果在这个地址准备好了自引导的内核映像，就可以启动了。尽管 go 命令可以带变参，实际使用时一般不用来传递参数。

2．bootm 命令的实现

```
/* common/cmd_bootm.c */
int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
        ulong iflag;
        ulong addr;
        ulong data, len, checksum;
        ulong   *len_ptr;
        uint   unc_len = 0x400000;
        int    i, verify;
```

```
char   *name, *s;
int    (*appl)(int, char *[]);
image_header_t *hdr = &header;

s = getenv ("verify");
verify = (s && (*s == 'n')) ? 0 : 1;
if (argc < 2) {
        addr = load_addr;
} else {
        addr = simple_strtoul(argv[1], NULL, 16);
}
SHOW_BOOT_PROGRESS (1);
printf ("## Booting image at %08lx ...\n", addr);
/* Copy header so we can blank CRC field for re-calculation */
memmove (&header, (char *)addr, sizeof(image_header_t));
if (ntohl(hdr->ih_magic) != IH_MAGIC)
{
        puts ("Bad Magic Number\n");
        SHOW_BOOT_PROGRESS (-1);
        return 1;
}
SHOW_BOOT_PROGRESS (2);
data = (ulong)&header;
len  = sizeof(image_header_t);

checksum = ntohl(hdr->ih_hcrc);
hdr->ih_hcrc = 0;
```

```
	if(crc32 (0, (char *)data, len) != checksum) {
		puts ("Bad Header Checksum\n");
		SHOW_BOOT_PROGRESS (-2);
		return 1;
	}
	SHOW_BOOT_PROGRESS (3);
	/* for multi-file images we need the data part, too */
	print_image_hdr ((image_header_t *)addr);
	data = addr + sizeof(image_header_t);
	len  = ntohl(hdr->ih_size);
	if(verify) {
		puts ("   Verifying Checksum ... ");
		if(crc32 (0, (char *)data, len) != ntohl(hdr->ih_dcrc)) {
			printf ("Bad Data CRC\n");
			SHOW_BOOT_PROGRESS (-3);
			return 1;
		}
		puts ("OK\n");
	}
	SHOW_BOOT_PROGRESS (4);
	len_ptr = (ulong *)data;
……
	switch (hdr->ih_os) {
	default:			/* handled by (original) Linux case */
	case IH_OS_LINUX:
	    do_bootm_linux  (cmdtp, flag, argc, argv,
			     addr, len_ptr, verify);
	    break;
```

```
    ……
}
```

bootm 命令调用 do_bootm 函数。这个函数专门用来引导各种操作系统映像，可以支持引导Linux、vxWorks、QNX 等操作系统。引导 Linux 的时候，调用 do_bootm_linux()函数。

3．do_bootm_linux 函数的实现

```
/* lib_arm/armlinux.c */
void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
             ulong addr, ulong *len_ptr, int verify)
{
    DECLARE_GLOBAL_DATA_PTR;
    ulong len = 0, checksum;
    ulong initrd_start, initrd_end;
    ulong data;
    void (*theKernel)(int zero, int arch, uint params);
    image_header_t *hdr = &header;
    bd_t *bd = gd->bd;
#ifdef CONFIG_CMDLINE_TAG
    char *commandline = getenv ("bootargs");
#endif
    theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
    /* Check if there is an initrd image */
    if(argc >= 3) {
        SHOW_BOOT_PROGRESS (9);
        addr = simple_strtoul (argv[2], NULL, 16);
```

```
printf ("## Loading Ramdisk Image at %08lx ...\n", addr);
/* Copy header so we can blank CRC field for re-calculation */
memcpy (&header, (char *) addr, sizeof (image_header_t));
if (ntohl (hdr->ih_magic) != IH_MAGIC) {
        printf ("Bad Magic Number\n");
        SHOW_BOOT_PROGRESS (-10);
        do_reset (cmdtp, flag, argc, argv);
}
data = (ulong) & header;
len = sizeof (image_header_t);
checksum = ntohl (hdr->ih_hcrc);
hdr->ih_hcrc = 0;
if(crc32 (0, (char *) data, len) != checksum) {
       printf ("Bad Header Checksum\n");
       SHOW_BOOT_PROGRESS (-11);
       do_reset (cmdtp, flag, argc, argv);
}
SHOW_BOOT_PROGRESS (10);
print_image_hdr (hdr);
data = addr + sizeof (image_header_t);
len = ntohl (hdr->ih_size);
if(verify) {
       ulong csum = 0;
       printf ("   Verifying Checksum ... ");
       csum = crc32 (0, (char *) data, len);
       if (csum != ntohl (hdr->ih_dcrc)) {
              printf ("Bad Data CRC\n");
              SHOW_BOOT_PROGRESS (-12);
```

```
                do_reset (cmdtp, flag, argc, argv);
            }
            printf ("OK\n");
        }
        SHOW_BOOT_PROGRESS (11);
        if ((hdr->ih_os != IH_OS_LINUX) ||
            (hdr->ih_arch != IH_CPU_ARM) ||
            (hdr->ih_type != IH_TYPE_RAMDISK)) {
            printf ("No Linux ARM Ramdisk Image\n");
            SHOW_BOOT_PROGRESS (-13);
            do_reset (cmdtp, flag, argc, argv);
        }
        /* Now check if we have a multifile image */
    } else if ((hdr->ih_type == IH_TYPE_MULTI) && (len_ptr[1])) {
         ulong tail = ntohl (len_ptr[0]) % 4;
         int i;
         SHOW_BOOT_PROGRESS (13);
         /* skip kernel length and terminator */
         data = (ulong) (&len_ptr[2]);
         /* skip any additional image length fields */
         for (i = 1; len_ptr[i]; ++i)
              data += 4;
        /* add kernel length, and align */
        data += ntohl (len_ptr[0]);
        if (tail) {
             data += 4 - tail;
        }
        len = ntohl (len_ptr[1]);
```

```
	} else {
		/* no initrd image */
		SHOW_BOOT_PROGRESS (14);
		len = data = 0;
	}
	if (data) {
		initrd_start = data;
		initrd_end = initrd_start + len;
	} else {
		initrd_start = 0;
		initrd_end = 0;
	}
	SHOW_BOOT_PROGRESS (15);
	debug ("## Transferring control to Linux (at address %08lx) ...\n",
		(ulong) theKernel);
#if defined (CONFIG_SETUP_MEMORY_TAGS) || \
	defined (CONFIG_CMDLINE_TAG) || \
	defined (CONFIG_INITRD_TAG) || \
	defined (CONFIG_SERIAL_TAG) || \
	defined (CONFIG_REVISION_TAG) || \
	defined (CONFIG_LCD) || \
	defined (CONFIG_VFD)
	setup_start_tag (bd);
#ifdef CONFIG_SERIAL_TAG
	setup_serial_tag (&params);
#endif
#ifdef CONFIG_REVISION_TAG
	setup_revision_tag (&params);
```

```
#endif
#ifdef CONFIG_SETUP_MEMORY_TAGS
	setup_memory_tags (bd);
#endif
#ifdef CONFIG_CMDLINE_TAG
	setup_commandline_tag (bd, commandline);
#endif
#ifdef CONFIG_INITRD_TAG
	if (initrd_start && initrd_end)
		setup_initrd_tag (bd, initrd_start, initrd_end);
#endif
	setup_end_tag (bd);
#endif
	/* we assume that the kernel is in place */
	printf ("\nStarting kernel ...\n\n");
	cleanup_before_linux ();

	theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
}
```

do_bootm_linux()函数是专门引导 Linux 映像的函数，它还可以处理 ramdisk 文件系统的映像。这里引导的内核映像和 ramdisk 映像，必须是 U-Boot 格式的。U-Boot 格式的映像可以通过 mkimage 工具来转换，其中包含了 U-Boot 可以识别的符号。

6.4 使用 U-Boot

U-Boot 是“Monitor”。除了 Bootloader 的系统引导功能，它还有用户命令接口，提供了一些复杂的调试、读写内存、烧写 Flash、配置环境变量等功能。掌握 U-Boot 的使用，将极大地方便嵌入式系统的开发。

6.4.1 烧写 U-Boot 到 Flash

新开发的电路板没有任何程序可以执行，也就不能启动，需要先将U-Boot 烧写到 Flash 中。

如果主板上的 EPROM 或者 Flash 能够取下来，就可以通过编程器烧写。例如：计算机 BIOS 就存储在一块 256KB 的 Flash 上，通过插座与主板连接。

但是多数嵌入式单板使用贴片的 Flash，不能取下来烧写。这种情况可以通过处理器的调试接口，直接对板上的 Flash 编程。

处理器调试接口是为处理器芯片设计的标准调试接口，包含 BDM、JTAG 和 EJTAG 3 种接口标准。JTAG 接口在第 4 章已经介绍过；BDM（Background Debug Mode）主要应用在 PowerPC8xx 系列处理器上；EJTAG 主要应用在 MIPS 处理器上。这 3 种硬件接口标准定义有所不同，但是功能基本相同，下面都统称为 JTAG 接口。

JTAG 接口需要专用的硬件工具来连接。无论从功能、性能角度，还是从价格角度，这些工具都有很大差异。关于这些工具的选择，将在第 6.4.1 节详细介绍。

最简单方式就是通过 JTAG 电缆，转接到计算机并口连接。这需要在主机端开发烧写程序，还需要有并口设备驱动程序。开发板上电或者复位的时候，烧写程序探测到处理器并且开始通信，然后把 Bootloader 下载并烧写到 Flash 中。这种方式速率很慢，可是价格非常便宜。一般来说，平均每秒钟可以烧写 100～200 个字节。

烧写完成后，复位实验板，串口终端应该显示 U-Boot 的启动信息。

6.4.2 U-Boot 的常用命令

U-Boot 上电启动后，敲任意键可以退出自动启动状态，进入命令行。

```
U-Boot 1.1.2 (Apr 26 2005 - 12:27:13)

U-Boot code: 11080000 -> 1109614C  BSS: -> 1109A91C

RAM Configuration:

Bank #0: 10000000 32 MB

Micron StrataFlash MT28F128J3 device initialized
```

```
Flash: 32 MB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
U-Boot>
```

在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。掌握这些命令的使用，才能够顺利地进行嵌入式系统的开发。

输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
=> 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'
bootm   - boot application image from memory
bootp   - boot image via network using BootP/TFTP protocol
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
dhcp    - invoke DHCP client to obtain IP/boot params
```

```
echo     - echo args to console
erase    - erase FLASH memory
flinfo   - print FLASH memory information
go       - start application at address 'addr'
help     - print online help
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
loop     - infinite loop on address range
md       - memory display
mm       - memory modify (auto-incrementing)
mtest    - simple RAM test
mw       - memory write (fill)
nfs      - boot image via network using NFS protocol
nm       - memory modify (constant address)
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
=>
```

U-Boot 还提供了更加详细的命令帮助，通过 help 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下这些命令的功能和参数。

```
=> 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 命令可以引导启动存储在内存中的程序映像。这些内存包括 RAM 和可以永久保存的 Flash。

第 1 个参数 addr 是程序映像的地址，这个程序映像必须转换成 U-Boot 的格式。

第 2 个参数对于引导 Linux 内核有用，通常作为 U-Boot 格式的 RAMDISK 映像存储地址；也可以是传递给 Linux 内核的参数（缺省情况下传递 bootargs 环境变量给内核）。

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

bootp 命令通过 bootp 请求，要求 DHCP 服务器分配 IP 地址，然后通过 TFTP 协议下载指定的文件到内存。

第 1 个参数是下载文件存放的内存地址。

第 2 个参数是要下载的文件名称，这个文件应该在开发主机上准备好。

```
=> help cmp
cmp [.b, .w, .l] addr1 addr2 count
    - compare memory
```

cmp 命令可以比较 2 块内存中的内容。.b 以字节为单位；.w 以字为单位；.l 以长字为单位。注意：cmp.b 中间不能保留空格，需要连续敲入命令。

第 1 个参数 addr1 是第一块内存的起始地址。

第 2 个参数 addr2 是第二块内存的起始地址。

第 3 个参数 count 是要比较的数目，单位按照字节、字或者长字。

=> help cp

cp [.b, .w, .l] source target count

- copy memory

cp 命令可以在内存中复制数据块，包括对 Flash 的读写操作。

第 1 个参数 source 是要复制的数据块起始地址。

第 2 个参数 target 是数据块要复制到的地址。这个地址如果在 Flash 中，那么会直接调用写 Flash 的函数操作。所以 U-Boot 写 Flash 就使用这个命令，当然需要先把对应 Flash 区域擦干净。

第 3 个参数 count 是要复制的数目，根据 cp.b cp.w cp.l 分别以字节、字、长字为单位。

=> help crc32

crc32 address count [addr]

- compute CRC32 checksum [save at addr]

crc32 命令可以计算存储数据的校验和。

第 1 个参数 address 是需要校验的数据起始地址。

第 2 个参数 count 是要校验的数据字节数。

第 3 个参数 addr 用来指定保存结果的地址。

```
=> help echo
echo [args..]
	- echo args to console; \c suppresses newline
```

echo 命令回显参数。

```
=> 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 命令可以擦 Flash。

参数必须指定 Flash 擦除的范围。

按照起始地址和结束地址，start 必须是擦除块的起始地址；end 必须是擦除末尾块的结束地址。这种方式最常用。举例说明：擦除 0x20000 – 0x3ffff 区域命令为 erase 20000 3ffff。

按照组和扇区，N 表示 Flash 的组号，SF 表示擦除起始扇区号，SL 表示擦除结束扇区号。另外，还可以擦除整个组，擦除组号为 N 的整个 Flash 组。擦除全部 Flash 只要给出一个 all 的参数即可。

```
=> help flinfo
flinfo
	- print information for all FLASH memory banks
```

```
flinfo N
       - print information for FLASH memory bank # N
```

flinfo 命令打印全部 Flash 组的信息，也可以只打印其中某个组。一般嵌入式系统的 Flash 只有一个组。

```
=> help go
go addr [arg ...]
      - start application at address 'addr'
        passing 'arg' as arguments
```

go 命令可以执行应用程序。
第 1 个参数是要执行程序的入口地址。
第 2 个可选参数是传递给程序的参数，可以不用。

```
=> help iminfo
iminfo addr [addr ...]
      - print header information for application image starting at
         address 'addr' in memory; this includes verification of the
         image contents (magic number, header and payload checksums)
```

iminfo 可以打印程序映像的开头信息，包含了映像内容的校验（序列号、头和校验和）。
第 1 个参数指定映像的起始地址。
可选的参数是指定更多的映像地址。

```
=> help loadb
loadb [ off ] [ baud ]
     - load binary file over serial line with offset 'off' and baudrate 'baud'
```

loadb 命令可以通过串口线下载二进制格式文件。

=> help loads

loads [off]

- load S-Record file over serial line with offset 'off'

loads 命令可以通过串口线下载 S-Record 格式文件。

=> help mw

mw [.b, .w, .l] address value [count]

- write memory

mw 命令可以按照字节、字、长字写内存，.b .w .l 的用法与 cp 命令相同。

第 1 个参数 address 是要写的内存地址。

第 2 个参数 value 是要写的值。

第 3 个可选参数 count 是要写单位值的数目。

=> help nfs

nfs [loadAddress] [host ip addr:bootfilename]

nfs 命令可以使用 NFS 网络协议通过网络启动映像。

=> help nm

nm [.b, .w, .l] address

- memory modify, read and keep address

nm 命令可以修改内存，可以按照字节、字、长字操作。

参数 address 是要读出并且修改的内存地址。

=> help printenv

printenv

第 2、3 参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

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

printenv 命令打印环境变量。

可以打印全部环境变量，也可以只打印参数中列出的环境变量。

```
=> 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
```

protect 命令是对 Flash 写保护的操作，可以使能和解除写保护。

第 1 个参数 on 代表使能写保护；off 代表解除写保护。

第 2、3 参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

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

rarboot 命令可以使用 TFTP 协议通过网络启动映像。也就是把指定的文件下载到指定地址，然后执行。

第 1 个参数是映像文件下载到的内存地址。

第 2 个参数是要下载执行的映像文件。

```
=> help run
run var [...]
    - run the commands in the environment variable(s) 'var'
```

run 命令可以执行环境变量中的命令，后面参数可以跟几个环境变量名。

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

setenv 命令可以设置环境变量。

第 1 个参数是环境变量的名称。

第 2 个参数是要设置的值，如果没有第 2 个参数，表示删除这个环境变量。

```
=> help sleep
sleep N
    - delay execution for N seconds (N is _decimal_ !!!)
```

sleep 命令可以延迟 N 秒钟执行，N 为十进制数。

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

tftpboot 命令可以使用 TFTP 协议通过网络下载文件。按照二进制文件格式下载。另外使用这个命令，必须配置好相关的环境变量。例如 serverip 和 ipaddr。

第 1 个参数 loadAddress 是下载到的内存地址。

第 2 个参数是要下载的文件名称，必须放在 TFTP 服务器相应的目录下。

这些 U-Boot 命令为嵌入式系统提供了丰富的开发和调试功能。在 Linux 内核启动和调试过程中，都可以用到 U-Boot 的命令。但是一般情况下，不需要使用全部命令。比如已经支持以太网接口，可以通过 tftpboot 命令来下载文件，那么还有必要使用串口下载的 loadb 吗？反过来，如果开发板需要特殊的调试功能，也可以添加新的命令。

在建立交叉开发环境和调试 Linux 内核等章节时，在 ARM 平台上移植了 U-Boot，并且提供了具体 U-Boot 的操作步骤。

6.4.3 U-Boot 的环境变量

有点类似 Shell，U-Boot 也使用环境变量。可以通过 printenv 命令查看环境变量的设置。

```
U-Boot> printenv
bootdelay=3
baudrate=115200
netmask=255.255.0.0
ethaddr=12:34:56:78:90:ab
bootfile=uImage
bootargs=console=ttyS0,115200 root=/dev/ram rw initrd=0x30800000,8M
bootcmd=tftp 0x30008000 zImage;go 0x30008000
serverip=192.168.1.1
```

```
ipaddr=192.168.1.100
stdin=serial
stdout=serial
stderr=serial

Environment size: 337/131068 bytes
U-Boot>
```

表 6.5 是常用环境变量的含义解释。通过 printenv 命令可以打印出这些变量的值。

表 6.5　　U-Boot 环境变量的解释说明

环 境 变 量	解 释 说 明
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

U-Boot 的环境变量都可以有缺省值，也可以修改并且保存在参数区。U-Boot 的参数区一般有 EEPROM 和 Flash 两种设备。

环境变量的设置命令为 setenv，在 6.2.2 节有命令的解释。

举例说明环境变量的使用。

```
=>setenv serverip  192.168.1.1
=>setenv ipaddr  192.168.1.100
```

```
=>setenv rootpath  "/usr/local/arm/3.3.2/rootfs"
=>setenv bootargs  "root=/dev/nfs rw nfsroot=\$(serverip):\$(rootpath) ip=
\$(ipaddr) "
=>setenv kernel_addr 30000000
=>setenv nfscmd  "tftp \$(kernel_addr) uImage; bootm \$(kernel_addr) "
=>run nfscmd
```

上面定义的环境变量有 serverip ipaddr rootpath bootargs kernel_addr。环境变量 bootargs 中还使用了环境变量，bootargs 定义命令行参数，通过 bootm 命令传递给内核。环境变量 nfscmd 中也使用了环境变量，功能是把 uImage 下载到指定的地址并且引导起来。可以通过 run 命令执行 nfscmd 脚本。