Bootloader是一小段程序, 在系统上电后开始执行, 初始化硬件, 准备好软件环境, 最后调用操作系统内核。
因此，将具体硬件问题交给了Bootloader来实现, 操作系统内核就可以实现通用性。
只要Bootloader配置好硬件环境，提供好接口, 各种操作系统都很容易在上面跑。

Bootlader的工作模式

(1) 启动加载模式, 上电后, 从某个存储设备上将操作系统加载至RAM中运行, 这个启动过程无需用户介入。
(2) 下载模式。上电后，进入下载模式供开发人员使用。开发人员可以利用Bootloader提供的功能, 如串口/ 网络/ USB/ nfs等实现调试开发板。

一般我们用的如U-boot, supervivi等, 都是默认上电后等待一段时间, 如果这段时间, 我按了下按钮或按键, bootloader将进入下载模式, 否则加载内核并启动。

- - - - - - - - - - - - - -

嵌入式Linux系统的软件角度划分 / 分区

1. 引导加载程序, 包括在固件(firmware)中的boot代码(可选, 比如x86结构的CPU会先运行固件上的BIOS实现自检,然后才运行硬盘第一个分区MBR中的Bootloader)以及Bootloader两部分。
2. Linux内核。其中Bootloader还要传递部分参数给内核供启动。
3. 文件系统。包括第一个挂载的根文件系统，以及其他可选择挂载的文件系统。里面包括了各种程序，库文件等。
4. 用户应用程序。也存储在文件系统中。

在嵌入式系统的固态存储设备中, 有相应的分区来存储它们。这样就是更新某一分区的内容而不影响其他分区(这也正是分区的主要思想所在)。

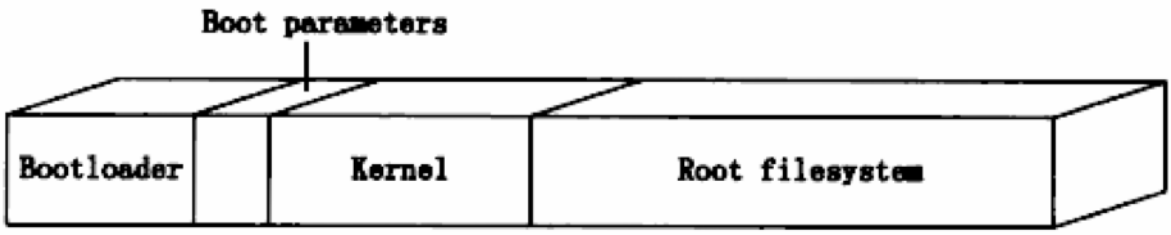


图 15.1　嵌入式 Linux 系统中的典型分区结构

例如我们的ARM开发板mini2440, 存储Bootloader, linux kernel , rootfs等在Nand Flash设备上，linux kernel提供了MTD技术, 抽象了Nand Flash设备的硬件接口, 我们在linux-kernle/arch/arm/mach-s3c2440/mach-mini2440.c 中进行分区配置。修改其中的mtd_partition结构体即可。

另外, 我们使用FriendlyARM提供的supervivi时, 默认的分区如下图所示。
这样, 我们在使用supervivi的各种下载功能时, 都是通过USB将镜像先下载到内存SDRAM中, 然后supervivi根据默认的分区表自动将内存中的镜像再烧写进Nand Flash中去。supervivi默认的分区可以在其命令行进行更改。

```
Enter your selection: q
Supervivi> part show
Number of partitions: 4
name            :       offset          size            flag
-----------------------------------------------------------------
vivi            :       0x00000000      0x00040000      0
param           :       0x00040000      0x00020000      0
kernel          :       0x00060000      0x00500000      0
root            :       0x00560000      0x03a9c000      0
Supervivi>
```

因此, supervivi的分区与Linux kernel的分区要严格移植, 否则linux将不知道从何处挂载根文件系统。

```
static struct mtd_partition smdk_default_nand_part[] = {
        [0] = {
                .name   = "supervivi",
                .size   = 0x00040000,
                .offset = 0x00000000,
        },
        [1] = {
                .name   = "param",
                .offset = 0x00040000,
                .size   = 0x00020000,
        },
        [2] = {
                .name   = "kernel",
                .offset = 0x00060000,
                .size   = 0x00500000,
        },
        [3] = {
                .name   = "root",
                .offset = 0x00560000,
                .size   = 0x03a9c000,
        }
};
```

Bootloader的两个阶段

第一阶段:

用汇编实现, 严重依赖硬件, 完成一些硬件上的初始化, 并调用第二阶段代码。

- 硬件设备初始化(关闭WATCHDOG/中断,设置CPU时钟频率,RAM初始化等)
- 为加载第二阶段代码准备RAM空间
- 复制第二阶段代码至RAM空间
- 设置好栈
- 跳转到第二阶段的C语言函数入口

第二阶段:

用c语言实现, 实现更复杂的功能。

- 初始化本阶段要使用的硬件设备
- 检测系统内存映射(确定开发板使用多少内存, 地址空间是多少)
- 将内核映像和根文件系统映像从Flash读到RAM空间
- 为内核设置启动参数
- 调用内核

调用内核时要满足的条件:

1. CPU寄存器设置。
   - R0 = 0
   - R1 = 机器类型ID; (ARM架构CPU,其ID参见linux/arch/arm/tools/mach-types)
   - R2 = 启动参数标记列表(tag list)在RAM中的起始基地址
2. CPU工作模式。
   - 必须禁止中断(IRQs & FIQs)
   - CPU必须为SVC管理模式
3. Cache/MMU设置。
   - MMU必须关闭。
   - Date Cache必须关闭。
   - 指令 Cache不关心, 可关闭可打开。

Bootloader最后调用内核的函数:

```
void (*theKernel) (int zero, int arch, u32 params_addr) = (void (*) (int, int, u32))
KERNEL_RAM_BASE;

theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

Bootloader与内核的交互

Bootloader与内核的相互是单向的, Bootloader将各类参数按照特定的数据结构存放在内存中某一地址, 并通过CPU寄存器传递地址给内核，内核启动后在指定地址读取对应的数据结构 。
参数以标记列表(tagged list)数据结构存放。
详见<u-boot_内核参数传递>
(icloud/arm/文件夹中. icloud: iguoxiaopeng@gmail.com)

U-Boot源码结构

u-boot根目录下的子目录分类:

1. 平台相关的或开发板相关。
   - board(开发板相关)
   - cpu(CPU相关)
   - lib_i386及其类似子目录(某一CPU架构相关)
2. 通用的函数
   - include(头文件, 配置文件(include/configs/))
   - lib_generic(通用的库函数)
   - common(通用的函数)
3. 通用的设备驱动程序
   - disk(硬盘接口驱动)
   - drivers(各类具体设备驱动)
   - dtt(传感器驱动)
   - fs(文件系统)
   - nand_spl(支持Nand Flash启动的驱动)
   - net(网络协议)
   - post(上电自检程序)
   - rtc(实时时钟驱动)
4. U-boot工具, 示例程序, 文档。
   - doc(开发/使用文档)
   - example(测试程序)
   - tools(常用的制作工具)

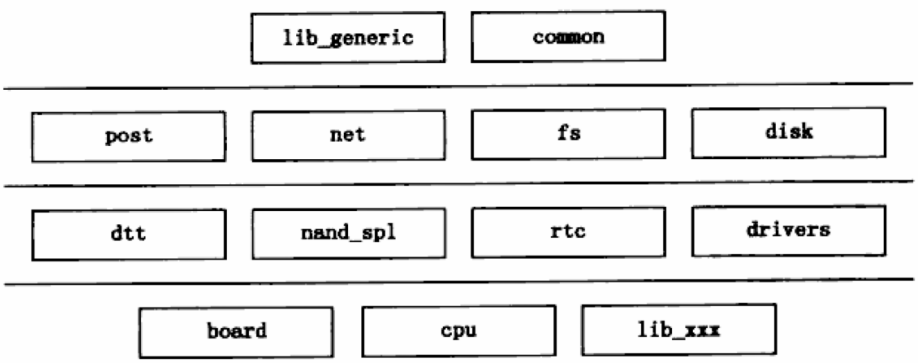


图 15.2　U-Boot 顶层目录的层次结构

U-BOOT编译流程

1. 得到U-Boot源代码
2. 设置配置文件, 即执行make <board_name>_config命令
3. make

其中make <board_name>_config是顶层Makefile文件定义的一个Make命令.
例如:

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

执行make <borad_name>_config等价与执行顶层目录下的mkconfig脚本。
./mkconfig smdk2400 arm arm920t smdk2400 NULL s3c24x0

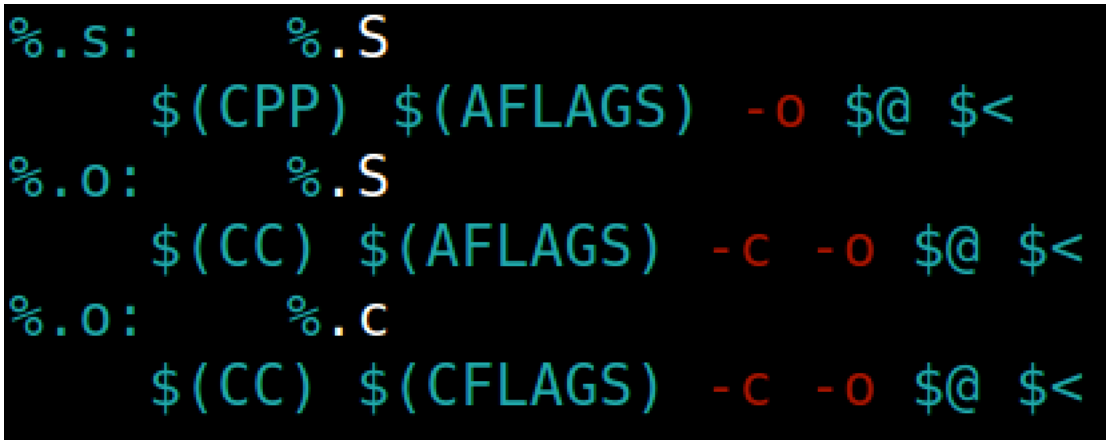
mkconfig脚本根据传递的几个参数, 进行下面的操作:

1. 确定开发板名称 BOARD_NAME;
   上面的命令将使得BOARD_NAME = smdk2400;
2. 创建到平台/开发板相关的头文件链接(如include/asm/, include/asm/arch/ 文件夹)
   cd ./include
   rm -f asm
   ln -s asm-arm asm
   cd asm
   ln -s arch-s3c24x0 asm-arm/arch
   ln -s proc-armv asm-arm/proc
   具体操作详见mkconfig脚本.
3. 创建顶层Makefile所要包含的文件include/config.mk
   对于上面的命令, config.mk的内容是
   ARCH = arm
   CPU = arm920t
   BOARD = smdk2400
   SOC = s3c24x0
4. 创建开发板相关头文件include/config.h
   对于上面的命令, 内容为
   #include <configs/smdk2400.h>

U-Boot Makefile分析

U-Boot Makefile看似有2000 + 行, 其实关键代码只有区区200+行(其中还有大量注释)。剩下的内容均是类似于make <board_name>_config 这样的make 命令, U-boot为了支持很多平台, 所以使得这种命令特别多。同样, 看似U-boot源代码很多, 其实对于某一平台, 都只是一个子目录的代码而已。

1. 开始一堆VERSION变量表示版本号。
2. HOSTARCH得到当前主机的架构。
3. 然后根据调用Makefile的Make命令确定下面几个变量。
   OBJTREE(目标代码生成根目录)
   SRCTREE(U-Boot源代码根目录)
   TOPDIR(目标根目录, =OBJTREE)
   LNDIR(源代码根目录, =SRCTREE)
   其中如果在类似make O=/tmp/build all 这样声明编译结果输出目录,
   则编译的中间文件及最终文件将存放指定目录, 不会影响U-boot源代码。
   如果指定, 默认OBJTREE=SRCTREE.
4. 添加mkconfig生成的include/config.mk文件, 得到ARCH,CPU,BOARD,SOC等。
5. 根据ARCH确定交叉编译器CROSS_COMPILE.
6. 读取源代码根目录下的config.mk.
   源代码根目录下的config.mk会根据CROSS_COMPILE确定不同的编译选项。
   同时声明其他工具。如:
   AS=$(CROSS_COMPILE)-as
   LD=$(CROSS_COMPILE)-ld
   这里确定的编译选项, 如CPPFLAGS, LDFLAGS, CFLAGS,AFLAGS等等。
   由这些编译选项, 确定了c,s文件的编译方式, 如:



7. 接下来就是U-boot的构成。

OBJS为U-Boot要编译的内容, 按照链接顺序依次添加。(可以看出, 特定CPU架构目录下的start.o将是U-boot的开始。)

LIBS为U-Boot要编译的所需要的库文件, 按照顺序依次添加。

OBJS/LIBS所代表的*.o, *.a文件就是U-Boot的构成。

```
#########################################################################
# U-Boot objects....order is important (i.e. start must be first)

OBJS  = cpu/$(CPU)/start.o
ifeq ($(CPU),i386)
OBJS += cpu/$(CPU)/start16.o
OBJS += cpu/$(CPU)/reset.o
endif
ifeq ($(CPU),ppc4xx)
OBJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc83xx)
OBJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc85xx)
OBJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),mpc86xx)
OBJS += cpu/$(CPU)/resetvec.o
endif
ifeq ($(CPU),bf533)
OBJS += cpu/$(CPU)/start1.o cpu/$(CPU)/interrupt.o  cpu/$(CPU)/cache.o
OBJS += cpu/$(CPU)/cplbhdlr.o   cpu/$(CPU)/cplbmgr.o    cpu/$(CPU)/flush.o
endif

OBJS := $(addprefix $(obj),$(OBJS))

LIBS  = lib_generic/libgeneric.a
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
```

8. 真正的Make 目标开始, 以及其他实现的make 命令。

   make要生成的最终目标是:

   - u-boot.srec
   - u-boot.bin
   - System.map
   - [u-boot-nand.bin] (如果配置CONFIG_NAND_U_BOOT)

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

all:        $(ALL)

$(obj)u-boot.hex:   $(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O ihex $< $@

$(obj)u-boot.srec:  $(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@

$(obj)u-boot.bin:   $(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

$(obj)u-boot.img:   $(obj)u-boot.bin
        ./tools/mkimage -A $(ARCH) -T firmware -C none \
        -a $(TEXT_BASE) -e 0 \
        -n $(shell sed -n -e 's/.*U_BOOT_VERSION//p' $(VERSION_FILE) | \
            sed -e 's/"[     ]*$$/ for $(BOARD) board"/') \
        -d $< $@

$(obj)u-boot.dis:   $(obj)u-boot
        $(OBJDUMP) -d $< > $@

$(obj)u-boot:       depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
        UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
        cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJS) \
            --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
            -Map u-boot.map -o u-boot
```

u-boot.hex, u-boot.srec, u-boot.bin, 都是u-boot的不同格式映像。
u-boot.dis是u-boot的反汇编文件。
u-boot是可执行文件。
编译u-boot就是编译OBJS, LIBS然后链接起来。

```
$(OBJS):
        $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))

$(LIBS):
        $(MAKE) -C $(dir $(subst $(obj),,$@))

$(SUBDIRS):
        $(MAKE) -C $@ all

$(NAND_SPL):    version
        $(MAKE) -C nand_spl/board/$(BOARDDIR) all

$(U_BOOT_NAND): $(NAND_SPL) $(obj)u-boot.bin
        cat $(obj)nand_spl/u-boot-spl-16k.bin $(obj)u-boot.bin > $(obj)u-boot-nand.bin
```

其中要得到ELF格式的U-Boot, LDFLAGS确定其链接方式。
其中的”-T board/smdk2400/U-Boot.lds -Ttext 0x33F80000”字样指定了程序的布局和链接地址。

特定开发板的U-Boot.lds如下:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
```

开始指定了输出为32位ELF格式, 小短模式，输出架构为arm结构, u-boot的入口函数是_start函数(在start.S中指定).

```
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text      :
    {
      cpu/arm920t/start.o   (.text)
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }

    . = ALIGN(4);
    .got : { *(.got) }

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

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) }
    _end = .;
}
```

关于lds链接文件的格式分析, 详见<iCloud/ARM/lds文件>。

U-Boot启动过程分析(以smdk2410为例)

第一阶段:

cpu/arm920t/start.S (平台相关)
board/smdk2410/lowlevel_init.S (开发板相关)

start.S分析:
入口地址: _start (在u-boot.lds中已定义)
前面的指令都是跳转指令。将各种异常向量地址存放在word字节中, CPU切换到不同模式, 自动配置PC到指定指定位置, 执行ldr pc, _xxx指令, 即从内存中读取异常向量地址, 然后跳转到对应的服务。

```
.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
```

其中最后的0xdeadbeef为魔数。
.balignl 16, 0xdeadbeef分析:
.balignl是.balign的变体，.balign是意思是，在以当前地址开始，地址计数器必须是以第一个参数为整数倍的地址为尾，在前面记录一个字节长度的信息，信息内容为第二个参数。.balign 8, 0xde的意思就是在以当前地址开始，在地址为8的倍数的位置的前面填入一个字节内容为0xde的内容。如果当前地址正好是8的倍数，则没有东西被写入到内存。
这里的0xdeadbeef作为魔数, 表示异常向量地址存放的内存段终止。
balignl指令之后, 还有一段指令，仍为存放地址。
分别为
.word TEXT_BASE
.word _armboot_start
.word __bss_start
.word __bss_end
.word IRQ_STACK_START
.word FIQ_STACK_START

然后, 就是reset代码段。
cpu上电后, 自动跳转到这里。

1. 设置CPU工作模式为SVC32模式
2. 关闭看门狗, 屏蔽中断
3. 配置CPU时钟频率
4. 跳转到cpu_init_crit段执行
   a. 刷新数据/命令Cache
   b. 禁止MMU和Cache
   c. 调用lowlevel_init函数, 进行板载内存的初始配置.
   其中lowlevel_init代码在board/smdk2410/lowlevel_init.S
5. 判断U-boot现在执行的代码是否在RAM中, 不在则进行拷贝(为第二阶段做准备)
   拷贝U-boot至RAM的TEXT_BASE(这里为0x33F80000)
6. 设置栈, 清空bss段(堆栈划分如下图所示)
   从TEXT_BASE开始, 向下分配栈空间。
   分配CFG_MALLOC_LEN
   分配CFG_GBL_DATA_SIZE
   分配CONFIG_STACKSIZE_IRQ / CONFIG_STACKSIZE_FIQ
   分配几个字节隔离
   配置栈指针SP
7. 跳转到第二阶段start_armboot

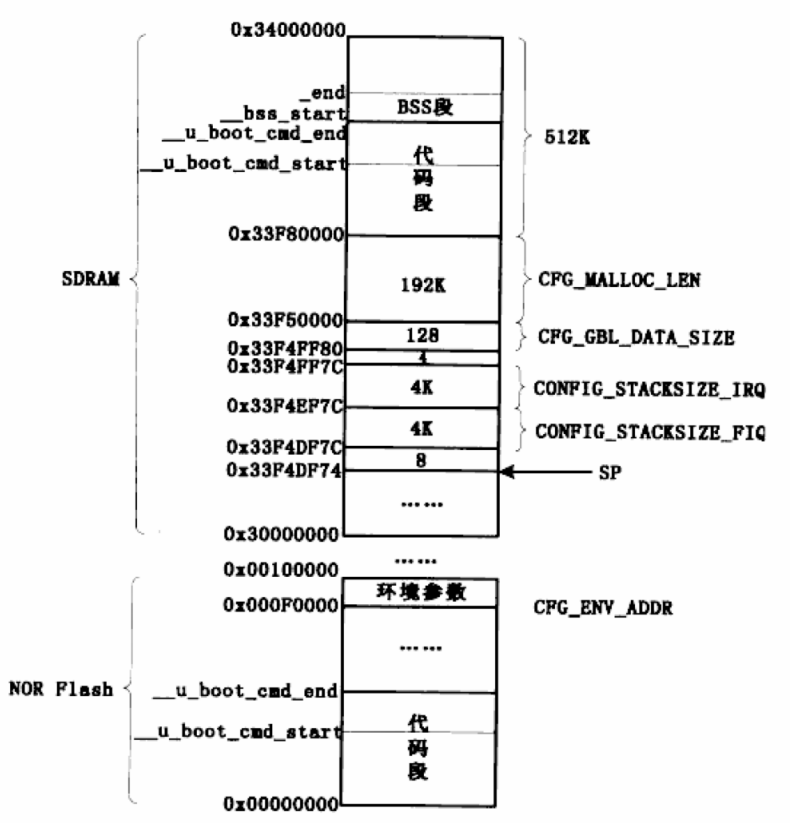


图 15.3　U-Boot 内存使用情况

第二阶段:

调用lib_arm/board.c中的start_armboot函数。

start_armboot

1. 在前面分配的CFG_MALLOC_LEN中, 划分一块内存给全局变量数据结构gd_t.
2. 再划分一块内存给bd_t结构体。
3. for循环调用init_sequence数组中预先定义的初始化函数
   - cpu_init（初始化IRQ/FIQ模式的栈）
   - board_init （设置系统时钟等）
   - interrupt_init （初始化定时器）
   - env_init （检查Flash上的环境参数等）
   - init_baudrate （初始化波特率）
   - serial_init （初始化串口设置）
   - console_init_f （初始化控制台）
   - dram_init （检测系统内存映射
4. start_armboot函数后续调用的初始化函数
   - flash_init （初始化NOR 或 Nand Flash）
   - env_relocate （读取环境参数入内存）
   - dm9000_get_enetaddr （初始化网络设备）
5. 调用main_loop

   A. 设置了环境参数, bootdelay秒内无相应, 自动启动

   B. 等待命令行操作。

常用的修改

1. board/smdk2410/smdk2410.c的board_init函数, 配置MPLL/系统时钟/机器ID等。
2. smdk2410.c文件的dram_init函数, 修改Memory map.
3. U-Boot命令修改. include/command.h, commond/

具体U-Boot的移植参见《mini2440_U-boot使用移植手册》

参考教材:

《嵌入式Linux应用开发完全手册》 – 韦东山