u-boot 源码解析

u-boot 介绍

Uboot 是德国 DENX 小组的开发用于多种嵌入式 CPU 的 bootloader 程序, UBoot 不仅仅支持嵌入式 Linux 系统的引导，当前，它还支持 NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS 嵌入式操作系统。UBoot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、 x86、ARM、NIOS、XScale 等诸多常用系列的处理器。

board：和一些已有开发板有关的文件。每一个开发板都以一个子目录出现在当前目录中，子目录中存放与开发板相关的配置文件。它的每个子文件夹里都有如下文件：

makefile

config.mk

smdk2410.c 和板子相关的代码(以 smdk2410 为例)

flash.c Flash 操作代码

memsetup.s 初始化 SDRAM 代码

u-boot.lds 对应的连接文件

common：实现 uboot 命令行下支持的命令，每一条命令都对应一个文件。例如 bootm 命令对应就是 cmd_bootm.c。

cpu：与特定 CPU 架构相关目录，每一款 Uboot 下支持的 CPU 在该目录下对应一个子目录，比如有子目录 arm920t 等。cpu/ 它的每个子文件夹里都有如下文件：

makefile

config.mk

cpu.c 和处理器相关的代码

interrupts.c 中断处理代码

serial.c 串口初始化代码

start.s 全局开始启动代码

disk：对磁盘的支持。

doc：文档目录。Uboot 有非常完善的文档，推荐大家参考阅读。

drivers：Uboot 支持的设备驱动程序都放在该目录，比如各种网卡、支持 CFI 的 Flash、串口和 USB 等。

fs: 支持的文件系统，Uboot 现在支持 cramfs、fat、fdos、jffs2 和 registerfs。

include：Uboot 使用的头文件，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。该目录下 configs 目录有与开发板相关的配置头文件，如 smdk2410.h。该目录下的 asm 目录有与 CPU 体系结构相关的头文件，asm 对应的是 asmarm.。

lib_xxxx: 与体系结构相关的库文件。如与 ARM 相关的库放在 lib_arm 中。

net：与网络协议栈相关的代码，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现。

tools：生成 Uboot 的工具，如：mkimage, crc 等等。

uboot 的启动过程及工作原理

启动模式介绍

大多数 Boot Loader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别仅对于开发人员才有意义。但从最终用户的角度看，Boot Loader 的作用就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载（Boot loading）模式：这种模式也称为"自主"（Autonomous）模式。也即 Boot Loader 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。这种模式是 BootLoader 的正常工作模式，因此在嵌入式产品发布的时侯，Boot Loader 显然必须工作在这种模式下。
下载（Downloading）模式：在这种模式下，目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如：下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 BootLoader 保存到目标机的 RAM 中，然后再被 BootLoader 写到目标机上的 FLASH 类固态存储设备中。BootLoader 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 BootLoader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。
UBoot 这样功能强大的 Boot Loader 同时支持这两种工作模式，而且允许用户在这两种工作模式之间进行切换。
大多数 bootloader 都分为阶段 1(stage1)和阶段 2(stage2)两大部分，uboot 也不例外。依赖于 CPU 体系结构的代码（如 CPU 初始化代码等）通常都放在阶段 1 中且通常用汇编语言实现，而阶段 2 则通常用 C 语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

stage1 (start.s 代码结构)

u-boot 的 stage1 代码通常放在 start.s 文件中，它用汇编语言写成，其主要代码部分如下：

(1) 定义入口由于一个可执行的 Image 必须有一个入口点，并且只能有一个全局入口，通常这个入口放在 ROM(Flash) 的 0x0 地址，因此，必须通知编译器以使其知道这个入口，该工作可通过修改连接器脚本来完成。

(2) 设置异常向量(Exception Vector) 。

(3) 设置 CPU 的速度、时钟频率及中断控制寄存器。

(4) 初始化内存控制器

(5) 将 ROM 中的程序复制到 RAM 中。

(6) 初始化堆栈

(7) 转到 RAM 中执行，该工作可使用指令 ldr pc 来完成。

stage2 C 语言代码部分

lib_arm/board.c 中的 start_armboot 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(armboot) 的主函数，该函数主要完成如下操作：

(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) 进入命令循环(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。

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。

Makefile 中定义了源码及生成的目标文件存放的目录,目标文件存放目录 BUILD__DIR 可以通过 make O=dir 指定。如果没有指定，则设定为源码顶层目录。一般编译的时候不指定输出目录，则 BUILD__DIR 为空。其它目录变量定义如下：

```
#OBJTREE 和 LNDIR 为存放生成文件的目录，TOPDIR 与 SRCTREE 为源码所在目录
    OBJTREE  := $(if $(BUILD_DIR),$(BUILD_DIR),$(CURDIR))
    SRCTREE  := $(CURDIR)
    TOPDIR  := $(SRCTREE)
    LNDIR  := $(OBJTREE)
    export TOPDIR SRCTREE OBJTREE
```

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
```

定义变量 MKCONFIG：这个变量指向一个脚本，即顶层目录的 mkconfig。

```
    MKCONFIG := $(SRCTREE)/mkconfig
export MKCONFIG
```

在编译 U－BOOT 之前，先要执行

```
    [root@Binnary ~ ]# make smdk2410_config
```

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/           : 库文件 board/smdk2410/libsmdk2410.a
    cpu/arm920t/              : 库文件 cpu/arm920t/libarm920t.a
    cpu/arm920t/s3c24x0/      : 库文件 cpu/arm920t/s3c24x0/libs3c24x0.a
    lib_arm/                  : 库文件 lib_arm/libarm.a
    include/asm-arm/          : 下面两个是头文件。
    include/configs/smdk2410.h
```

我们可以按照上面的类似定义来定义我们自己的开发板。如：

```
    binnary2410_config   :   unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t binnary2410
    binnary s3c24x0
```

在定义好自己的开发板后还需要来创建属于自己开发板的文件夹。

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          //顺序很重要，start.o 必须放第一位
…
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。

```
#最终生成的各种镜像文件
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 $< > $@
#这里生成的是 U-boot  的 ELF 文件镜像
$(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) \ @执行连接命令其实就是把 start.o 和各个子目录 makefile 生成的库文件按照 LDFLAGS 连接在一起，生成 ELF 文件 u-boot 和连接时内存分配图文件 u-boot.map

```
    --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
    -Map u-boot.map -o u-boot
```

@依赖目标$(OBJS)，即 cpu/start.o

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

@依赖目标$(LIBS)，这个目标太多，都是每个子目录的库文件*.a ，通过执行相应子目录下的 make 来完成：

```
$(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
```

@依赖目标 version：生成版本信息到版本文件 VERSION_FILE 中

```
version:
    @echo -n "#define U_BOOT_VERSION \"U-Boot " > $(VERSION_FILE); \
    echo -n "$(U_BOOT_VERSION)" >> $(VERSION_FILE); \
    echo -n $(shell $(CONFIG_SHELL) $(TOPDIR)/tools/setlocalversion \
         $(TOPDIR)) >> $(VERSION_FILE); \
    echo "\"" >> $(VERSION_FILE)
……
env:
    $(MAKE) -C tools/env all || exit 1
```

@依赖目标 depend :生成各个子目录的.depend 文件，.depend 列出每个目标文件的依赖文件。生成方法，调用每个子目录的 make _depend

```
depend dep:
    for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir _depend ; doneMakefile 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。
```

整个 makefile 剩下的内容全部是各种不同的开发板的*_config:目标的定义。

概括起来，工程的编译流程也就是通过执行执行一个 make *_config 传入 ARCH，CPU，BOARD，SOC 参数，mkconfig 根据参数将 include 头文件夹相应的头文件夹连接好，生成 config.h。然后执行 make 分别调用各子目录的 makefile 生成所有的 obj 文件和 obj 库文件*.a。最后连接所有目标文件，生成镜像。不同格式的镜像都是调用相应工具由 elf 镜像直接或者间接生成的。

2、先分析一下 u-boot 启动的两个阶段，分别对应 start.S 和 board.c 这两个文件。

先看 board/smdk2410/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。

对.lds 文件形式的完整描述：

```
SECTIONS {定义域中所包含的段
    …
    secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
    { contents } >region :phdr =fill
    …
 }
```

secname 和 contents 是必须的，其他的都是可选的。下面是几个常用的：

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：本段连接（运行）的地址，如果没有使用 AT（ldadr），本段存储的地址也是 start。GNU 网站上说 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。

好了下面我们来看一下 u-boot.lds 文件。

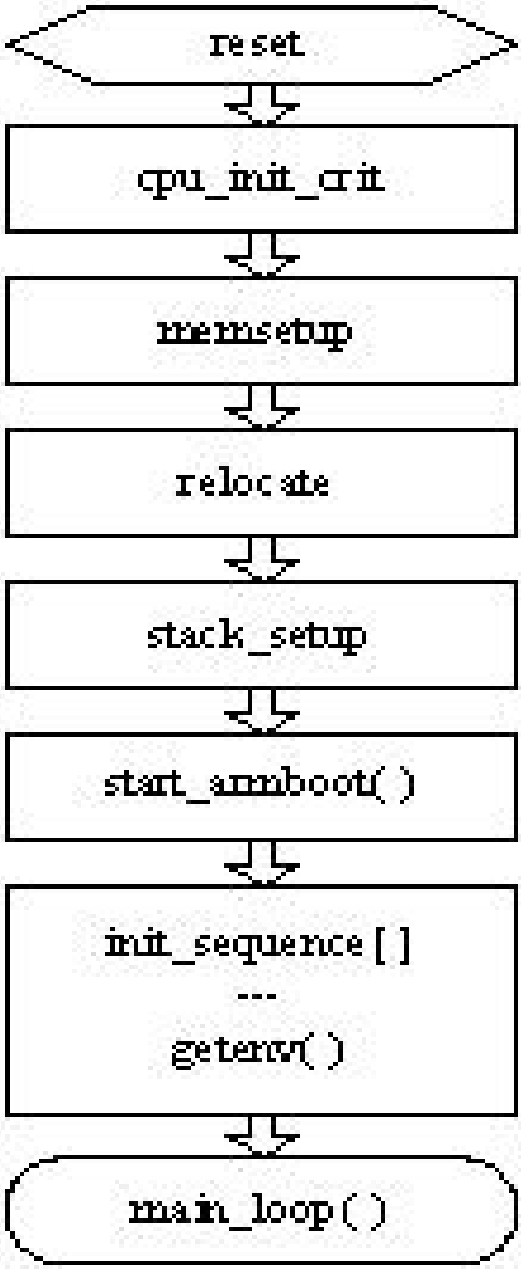
```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
指定输出可执行文件是 elf 格式,31 位 ARM 指令,小端
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
指定输出可执行文件的平台为 ARM
ENTRY(_start)
指定输出可执行文件的起始代码段为_start
SECTIONS
{
    . = 0x00000000;     从 0x0 位置开始
    . = ALIGN(4);       代码以 4 字节对齐
    .text       :     ;指定代码段,.text 的基地址由 LDFLAGS 中-Ttext $(TEXT_BASE)指定
    {                 ;smdk2410 指定的基地址为 0x33f80000
      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__u_boot_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
}
```

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。

Stage1：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。



U-Boot 启动代码流程图

下面我们来看一下 cpu/arm920t/start.S。

/*硬件环境初始化：
 进入 svc 模式;关闭 watch dog;屏蔽所有 IRQ 掩码;设置时钟频率 FCLK、HCLK、PCLK;清 I/D cache;禁止 MMU 和 CACHE;配置 memory control;
 重定位：
 如果当前代码不在连接指定的地址上（对 smdk2410 是 0x3f000000）则需要把 u-boot 从当前位置拷贝到 RAM 指定位置中；
 建立堆栈，堆栈是进入 C 函数前必须初始化的。
 清.bss 区。
 跳到 start_armboot 函数中执行。（lib_arm/board.c）
/*
.globl _start   ;定义一个全局符号,通常是为 ld 使用,系统复位位置
_start: b   reset   ;各个异常向量对应的跳转代码 复位 0x0
    ldr pc, _undefined_instruction   ;未定义的指命异常 0x4
    ldr pc, _software_interrupt   ;软件中断异常   0x8
    ldr pc, _prefetch_abort   ;内存操作异常   0xc
    ldr pc, _data_abort   ;数据异常   0x10
    ldr pc, _not_used   ;未使用异常   0x14
    ldr pc, _irq   ;慢速中断异常   0x18
    ldr pc, _fiq   ;快速中断异常   0x1c
; 当发生异常时，执行 cpu/arm920t/interrupts.c 中定义的中断处理函数。
_undefined_instruction: .word undefined_instruction ;.word 定义一个字,并为之分配空间
_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
_TEXT_BASE: ;   board\smdk2410\config.mk
    .word   TEXT_BASE   ;定义一个字并为之分配空间 4bytes

.globl _armboot_start
_armboot_start:
    .word _start
.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
/*
 * the actual reset code
 */
reset:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0,cpsr     ;读 cpsr 寄存器状态
    bic r0,r0,#0x1f ;位清除，清除 0x1f 对应的位
    orr r0,r0,#0xd3 ;设置 M=10011，superviser 模式
    msr cpsr,r0     ;写 cpsr 寄存器
/* turn off the watchdog */
#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
# define INTMSK     0x4A000008  /* Interupt-Controller base addresses */
# define INTSUBMSK  0x4A00001C
# define CLKDIVN    0x4C000014  /* clock divisor register */
#endif

#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)  ;关闭看门狗
    ldr     r0, =pWTCON
    mov     r1, #0x0    ;根据三星手册进行调置
    str     r1, [r0]
    /*
     * mask all IRQs by setting all bits in the INTMR - default
     */
    ;禁掉所有中断
    mov r1, #0xffffffff
    ldr r0, =INTMSK
    str r1, [r0]
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
```

```
    str r1, [r0]
# endif
    /* FCLK:HCLK:PCLK = 1:2:4 */
    ;默认频率为 FCLK:HCLK:PCLK = 1:2:4，默认 FCLK 的值为 120 MHz，该值为 S3C2410
手册的推荐值。
    /* default FCLK is 120 MHz ! */
    ;设置以 CPU 的频率
    ldr r0, =CLKDIVN
    mov r1, #3
    str r1, [r0]
#endif  /* CONFIG_S3C2400 || CONFIG_S3C2410 */

    /*
     * we do sys-critical inits only at reboot,
     * not when booting from ram!
     */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl  cpu_init_crit
#endif

#ifndef CONFIG_SKIP_RELOCATE_UBOOT
```

重定位（relocate）代码将 BootLoader 自身由 Flash 复制到 SDRAM，以便跳转到 SDRAM 执行。之所以需要进行重定位是因为在 Flash 中执行速度比较慢，而系统复位后总是从 0x00000000 地址取指。

```
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 如果执行到_start 时 PC 的值，放到 r0 中：
```

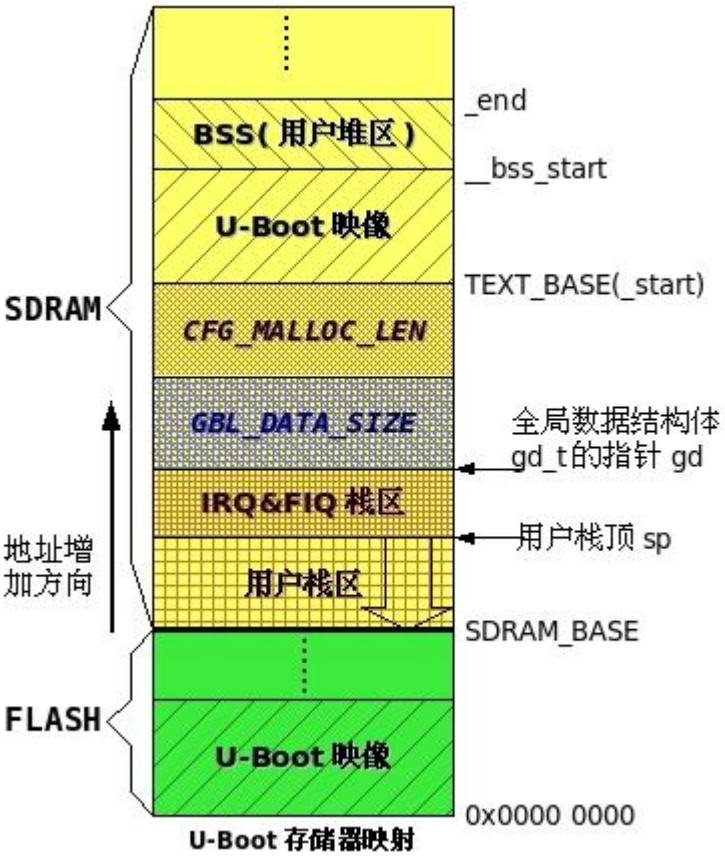
当此段在 flash 中执行时 r0 = _start = 0；当此段在 RAM 中执行时_start = _TEXT_BASE(在 board/smdk2410/config.mk 中指定的值为 0x33F80000，即 u-boot 在把代码拷贝到 RAM 中去执行的代码段的开始)

```
    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，因为此值是又编译器指定的
    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 */ ;;以上确定了复位启动代码是在 flash 中执行的(是系
统重启，而不是软复位)，就需要把代码拷贝到 RAM 中去执行，以下为计算即将拷贝的
代码的长度
    ldr r2, _armboot_start ;;前面定义了，就是_start
    ldr r3, _bss_start ;;所谓 bss 段，就是未被初始化的静态变量存放的地方，这
个地址是如何的出来的？根据 board/smsk2410/u-boot.lds 内容
    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 */ ;;开始循环拷贝启动的代码到 RAM 中
    ldmia {r3-r10}    /*从源地址[r0]复制 */ ;;r0 指向_start(=0)
```

```
    stmia {r3-r10}    /* 复制到目的地址[r1] */ ;;r1 指向
_TEXT_BASE(=0x33F80000)
    cmp r0, r2      /* 复制数据块直到源数据末尾地址[r2] */
    ble copy_loop#endif  /* CONFIG_SKIP_RELOCATE_UBOOT */
```

以上 relocate 代码段首先判断是否需要进行重定位，如果需要的话首先确定复制的源基址、源大小和目标基址，然后以 r3 ~ r13 为媒介，将 BootLoader 复制到 SDRAM 中。

分析



U-Boot 存储器映射

下面这段代码只对不是从 Nand Flash 启动的代码段有意义，对从 Nand Flash 启动的代码，没有意义。因为从 Nand Flash 中把 UBOOT 执行代码搬移到 RAM,

```
    /*初始化堆栈等                          */
stack_setup:
    ldr r0, _TEXT_BASE          /* 代码段的起始地址  */
    sub r0, r0, #CFG_MALLOC_LEN     /* 分配的动态内存区 */
    sub r0, r0, #CFG_GBL_DATA_SIZE  /* UBOOT 开发板全局数据存放 */
#ifdef CONFIG_USE_IRQ
/* 分配 IRQ 和 FIQ 栈空间 */
    sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif  //这些宏定义在/include/configs/smdk2410.h 中
    sub sp, r0, #12     /* 留下 3 个字为 Abort*/
```

//用 0x33F8000 - 0xC0 - 0x80 得到_TEXT_BASE 向下(低地址)的堆栈指针 sp 的起点地址。

```
clear_bss:
    ldr r0, _bss_start      /* BSS 段的起始地址*/
    ldr r1, _bss_end        /* BSS 段的结束地址*/
    mov     r2, #0x00000000     /* BSS 段置 0*/

clbss_l:str r2, [r0]        /* 循环清除 BSS 段 */
    add r0, r0, #4
    cmp r0, r1
    ble clbss_l

#if 0
    /* try doing this stuff after the relocation */
    ldr     r0, =pWTCON
    mov     r1, #0x0
    str     r1, [r0]

    /*
     * mask all IRQs by setting all bits in the INTMR - default
     */
    mov r1, #0xffffffff
    ldr r0, =INTMR
    str r1, [r0]

    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    mov r1, #3
    str r1, [r0]
    /* END stuff after relocation */
#endif
/* 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

/*
 ************************************************************************
 *
 * CPU_init_critical registers
 *
 * setup important registers
```

```
 * setup memory timing
 *
 *************************************************************************
 */

/ * cpu 初始化关键寄存器
 * 设置重要寄存器
 * 设置内存时钟
* /
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
    /*
     * flush v4 I/D caches
     */
    ;设置 CP15, 失效指令(I)Cache 和数据(D)Cache 后, 禁止 MMU 与 Cache。
    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   @先把 c1 和 c0 寄存器的各位置 0(r0 = 0)
    bic r0, r0, #0x00002300     @ clear bits 13, 9:8 (--V- --RS)
    bic r0, r0, #0x00000087     @分开设置。因为 arm 汇编要求的立即数格式所决定的
    orr r0, r0, #0x00000002     @上一条已经设置 bit1 为 0, 这一条又设置为 1
    orr r0, r0, #0x00001000     @ set bit 12 (I) I-Cache
    mcr p15, 0, r0, c1, c0, 0   @用上面(见下面)设定的 r0 的值设置 c1,(cache 类型寄
存器)和 c0(control 字寄存器), 以下为 c0 的位定义
                    ;;bit8: 0 = Disable System protection
                    ;;bit9: 0 = Disable ROM protection
                    ;;bit0: 0 = MMU disabled
                    ;;bit1: 0 = Fault checking disabled 禁止纠错
                    ;;bit2: 0 = Data cache disabled
                    ;;bit7: 0 = Little-endian operation
                    ;;bit12: 1 = Instruction cache enabled
    /*
     * 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.
     */
@ 配置内存区控制寄存器,寄存器的具体值通常由开发板厂商或硬件工程师提供. 如果您对
总线周期及外围
@ 芯片非常熟悉, 也可以自己确定, 在 UBOOT 中的设置文件是
board/crane2410/lowlevel_init.S, 该文件包含 lowleve_init 程序段.
```

```
    mov ip, lr
    bl  lowlevel_init @位于board/smdk2410/lowlevel_init.S：用于完成芯片存储器的初始化，执行完成后返回
    mov lr, ip
    mov pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */
/*
 *************************************************************************
 *
 * Interrupt handling
 *
 *************************************************************************
 */

@
@ IRQ stack frame.
@
#define S_FRAME_SIZE    72

#define S_OLD_R0    68
#define S_PSR       64
#define S_PC        60
#define S_LR        56
#define S_SP        52

#define S_IP        48
#define S_FP        44
#define S_R10       40
#define S_R9        36
#define S_R8        32
#define S_R7        28
#define S_R6        24
#define S_R5        20
#define S_R4        16
#define S_R3        12
#define S_R2        8
#define S_R1        4
#define S_R0        0

#define MODE_SVC 0x13
#define I_BIT    0x80

/*
 * use bad_save_user_regs for abort/prefetch/undef/swi ...
```

```
* use irq_save_user_regs / irq_restore_user_regs for IRQ/FIQ handling
*/

   .macro  bad_save_user_regs
   sub sp, sp, #S_FRAME_SIZE
   stmia   sp, {r0 - r12}          @ Calling r0-r12
   ldr r2, _armboot_start
   sub r2, r2, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
   sub r2, r2, #(CFG_GBL_DATA_SIZE+8)  @ set base 2 words into abort stack
   ldmia   r2, {r2 - r3}           @ get pc, cpsr
   add r0, sp, #S_FRAME_SIZE       @ restore sp_SVC

   add r5, sp, #S_SP
   mov r1, lr
   stmia   r5, {r0 - r3}           @ save sp_SVC, lr_SVC, pc, cpsr
   mov r0, sp
   .endm

   .macro  irq_save_user_regs
   sub sp, sp, #S_FRAME_SIZE
   stmia   sp, {r0 - r12}          @ Calling r0-r12
   add     r8, sp, #S_PC
   stmdb   r8, {sp, lr}^                   @ Calling SP, LR
   str     lr, [r8, #0]                    @ Save calling PC
   mrs     r6, spsr
   str     r6, [r8, #4]                    @ Save CPSR
   str     r0, [r8, #8]                    @ Save OLD_R0
   mov r0, sp
   .endm

   .macro  irq_restore_user_regs
   ldmia   sp, {r0 - lr}^          @ Calling r0 - lr
   mov r0, r0
   ldr lr, [sp, #S_PC]         @ Get PC
   add sp, sp, #S_FRAME_SIZE
   subs    pc, lr, #4          @ return & move spsr_svc into cpsr
   .endm

   .macro get_bad_stack
   ldr r13, _armboot_start     @ setup our mode stack
   sub r13, r13, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
   sub r13, r13, #(CFG_GBL_DATA_SIZE+8) @ reserved a couple spots in abort stack

   str lr, [r13]           @ save caller lr / spsr
```

```
    mrs lr, spsr
    str     lr, [r13, #4]

    mov r13, #MODE_SVC          @ prepare SVC-Mode
    @ msr   spsr_c, r13
    msr spsr, r13
    mov lr, pc
    movs    pc, lr
    .endm

    .macro get_irq_stack            @ setup IRQ stack
    ldr sp, IRQ_STACK_START
    .endm

    .macro get_fiq_stack            @ setup FIQ stack
    ldr sp, FIQ_STACK_START
    .endm

/*
 * exception handlers
 */
 /***************** 异常处理程序 ********************/
    .align  5
undefined_instruction:      //未定义指令
    get_bad_stack
    bad_save_user_regs
    bl  do_undefined_instruction

    .align  5
software_interrupt:     //软件中断
    get_bad_stack
    bad_save_user_regs
    bl  do_software_interrupt

    .align  5
prefetch_abort:         //预取异常中止
    get_bad_stack
    bad_save_user_regs
    bl  do_prefetch_abort

    .align  5
data_abort:         //数据异常中止
    get_bad_stack
    bad_save_user_regs
```

```
    bl  do_data_abort

    .align  5
not_used:           //未利用
    get_bad_stack
    bad_save_user_regs
    bl  do_not_used

#ifdef CONFIG_USE_IRQ

    .align  5
irq:                //中断请求
    get_irq_stack
    irq_save_user_regs
    bl  do_irq
    irq_restore_user_regs

    .align  5
fiq:                //快速中断请求
    get_fiq_stack
    /* someone ought to write a more effiction fiq_save_user_regs */
    irq_save_user_regs
    bl  do_fiq
    irq_restore_user_regs

#else

    .align  5
irq:
    get_bad_stack
    bad_save_user_regs
    bl  do_irq

    .align  5
fiq:
    get_bad_stack
    bad_save_user_regs
    bl  do_fiq

#endif
```

下面我们再看一下 start.S 中提到的 lowlevel_init.S 文件。

```
//主要执行内存相关的初始化
/* some parameters for the board */
```

```
/*
* Taken from linux/arch/arm/boot/compressed/head-s3c2410.S
*/
#define BWSCON  0x48000000  //这个是SFR区中控制SDRAM的寄存器的基址
/* BWSCON */
#define DW8             (0x0)
#define DW16            (0x1)
#define DW32            (0x2)
#define WAIT            (0x1<<2)
#define UBLB            (0x1<<3)

#define B1_BWSCON       (DW32)      /* 定义每个bank的数据总线宽度 */
#define B2_BWSCON       (DW16)
#define B3_BWSCON       (DW16 + WAIT + UBLB)
#define B4_BWSCON       (DW16)
#define B5_BWSCON       (DW16)
#define B6_BWSCON       (DW32)
#define B7_BWSCON       (DW32)

/* BANK0CON */
#define B0_Tacs         0x0 /*  0clk */
#define B0_Tcos         0x0 /*  0clk */
#define B0_Tacc         0x7 /* 14clk */
#define B0_Tcoh         0x0 /*  0clk */
#define B0_Tah          0x0 /*  0clk */
#define B0_Tacp         0x0
#define B0_PMC          0x0 /* normal */

/* BANK1CON */
/* 之间省略了多个bank区的配置，类似上面的BANK0CON。具体配置参数见源文件 */
/* BANK7CON */

/* BANK1CON */
#define B1_Tacs         0x0 /*  0clk */
#define B1_Tcos         0x0 /*  0clk */
#define B1_Tacc         0x7 /* 14clk */
#define B1_Tcoh         0x0 /*  0clk */
#define B1_Tah          0x0 /*  0clk */
#define B1_Tacp         0x0
#define B1_PMC          0x0

#define B2_Tacs         0x0
#define B2_Tcos         0x0
#define B2_Tacc         0x7
```

```
#define B2_Tcoh         0x0
#define B2_Tah          0x0
#define B2_Tacp         0x0
#define B2_PMC          0x0

#define B3_Tacs         0x0 /*  0clk */
#define B3_Tcos         0x3 /*  4clk */
#define B3_Tacc         0x7 /* 14clk */
#define B3_Tcoh         0x1 /*  1clk */
#define B3_Tah          0x0 /*  0clk */
#define B3_Tacp         0x3     /*  6clk */
#define B3_PMC          0x0 /* normal */

#define B4_Tacs         0x0 /*  0clk */
#define B4_Tcos         0x0 /*  0clk */
#define B4_Tacc         0x7 /* 14clk */
#define B4_Tcoh         0x0 /*  0clk */
#define B4_Tah          0x0 /*  0clk */
#define B4_Tacp         0x0
#define B4_PMC          0x0 /* normal */

#define B5_Tacs         0x0 /*  0clk */
#define B5_Tcos         0x0 /*  0clk */
#define B5_Tacc         0x7 /* 14clk */
#define B5_Tcoh         0x0 /*  0clk */
#define B5_Tah          0x0 /*  0clk */
#define B5_Tacp         0x0
#define B5_PMC          0x0 /* normal */

#define B6_MT           0x3 /* SDRAM */
#define B6_Trcd         0x1
#define B6_SCAN         0x1 /* 9bit */

#define B7_MT           0x3 /* SDRAM */
#define B7_Trcd         0x1 /* 3clk */
#define B7_SCAN         0x1 /* 9bit */

/* REFRESH parameter */
#define REFEN           0x1 /* Refresh enable */
#define TREFMD          0x0 /* CBR(CAS before RAS)/Auto refresh */
#define Trp         0x0 /* 2clk */
#define Trc         0x3 /* 7clk */
#define Tchr            0x2 /* 3clk */
#define REFCNT          1113    /*period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */
```

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

_TEXT_BASE:
    .word   TEXT_BASE

//把链接寄存器 LR(即 R14)的值转存到寄存器 R10 中,以便 lowlevel_init 完成后恢复执行
.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    /* SMRDATA 见下面，BWSCON 寄存器的后面紧接着就是 0-7 个 bank 的控制寄存器的地址,这里整个数据是 8 个 bank 的配置参数和另外 4 个寄存器的配置参数，具体按照 s3c2410 的 datasheet 进行配置 */
    ldr r1, _TEXT_BASE
    sub r0, r0, r1  /* 把当前 PC 寄存器的值与 0xa0000000 逻辑与*/
    ldr r1, =BWSCON /* Bus Width Status Controller 从 BWSCON 寄存器的地址开始写 13*4 这么长的数据，也就是从 SMRDATA 标号开始的那些配置好的数据，一并写入 */
    add     r2, r0, #13*4
0:
    ldr     r3, [r0], #4
    str     r3, [r1], #4
    cmp     r2, r0      /* 如果结果等于 0xa0000000，说明 uboot 是从 RAM 启动的 */
    bne     0b

    /* 上面都配置好了，返回 start.S  */
    mov pc, lr

    .ltorg

/* 下面共有 13 个 word，设置了一系列连续的寄存器的值，给上面循环赋值用*/
SMRDATA:
    .word
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20
)+(B6_BWSCON<<24)+(B7_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<<16)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30
```

我们使用的是两片容量为 32MB、位宽 16bit 的 HY57V561620CT-H 芯片拼成容量为 64M、32bit 的 SDRAM 存储器。根据 2410datasheet，要使用 SDRAM 需配置 13 个寄存器，以下逐个来看：

1、 BWSCON：Bus width & wait status control register 总线位宽和等待状态控制寄存器。

此寄存器用于配置 BANK0 - BANK7 的位宽和状态控制，每个 BANK 用 4 位来配置，分别是：

ST（启动/禁止 SDRAM 的数据掩码引脚。对于 SDRAM，此位置 0；对于 SRAM，此位置 1）

WS（是否使用存储器的 WAIT 信号，通常置 0 为不使用）

DW（两位，设置位宽。此板子的 SDRAM 是 32 位，故将 DW6 设为 10）

特殊的是 bit[2:1]，即 DW0，设置 BANK0 的位宽，又板上的跳线决定，只读的。我这板子 BWSCON 可设置为 0x22111110。其实只需将 BANK6 对应的 4 位设为 0010 即可。

2、 BANKCON0 - BANKCON7

用来分别配置 8 个 BANK 的时序等参数。SDRAM 是映射到 BANK6 和 BANK7 上的（内存只能映射到这两个 BANK，具体映射多大的空间，可用 BANKSIZE 寄存器设置），所以只需参照 SDRAM 芯片的 datasheet 配置好 BANK6 和 BANK7，BANKCON0 - BANKCON5 使用默认值 0x00000700 即可。

对于 BANKCON6 和 BANKCON7 中的各个位的描述：

（1）MT（bit[16:15]）：设置本 BANK 映射的物理内存是 SRAM 还是 SDRAM，后面的低位就根据此 MT 的选择而分开设置。本板子应置 0b11，所以只需要再设置下面两个参数

（2）Trcd（bit[3:2]）：RAS to CAS delay（00=2 clocks,01=3 clocks,10=4 clocks），推 2410 手册上的荐值是 0b01。我们 PC 的 BIOS 里也可以调节的。

（3）SCAN（bit[1:0]）：Column address number（00 = 8-bit，01 = 9-bit，10= 10-bit），SDRAM 列地址位数。查阅 HY57V561620CT-H 芯片手册得知此值是 9，所以 SCAN=0b01。

综合以上各值，BANKCON6 - 7 设为 0x00018005。

3、 REFRESH：刷新控制寄存器。

此寄存器的 bit[23:11]可参考默认值，或自己根据经验修改，这里用 0x008e0000，关键是最后的 Refresh Counter（简称 R_CNT，bit[10:0]）的设置，2410 手册上给出了公式计算方法。SDRAM 手册上“8192 refresh cycles / 64ms”的描述，得到刷新周期为 64ms/8192=7.8125us，结合公式，R_CNT=2^11 + 1 - 12 * 7.8125 = 1955。所以可得 REFRESH=0x008e0000+1995=0x008e07a3。

4、 BANKSIZE：设置 SDRAM 的一些参数。其中 BK76MAP（bit[2:0]）配置 BANK6/7 映射的大小，可设置为 010 = 128MB/128MB 或 001 = 64MB/64MB，只要比实际 RAM 大都行，因为 bootloader 和 linux 内核都可以检测可用空间的。BANKSIZE=0x000000b2。

5、 MRSRB6、MRSRB7：Mode register set register bank6/7

可以修改的只有 CL[6:4]（CAS latency，000 = 1 clock, 010 = 2 clocks, 011=3 clocks），其他的全部是固定的（fixed），故值为 0x00000030。这个 CAS 在 BIOS 中应该也设置过吧，对 PC 的速度提升很明显。

至此，13 个寄存器全部配置好了，下面就可以把代码复制到 SDRAM 中执行了，同样的程序速度要比片内 SRAM 运行的慢不少。

Stage2：lib_arm/board.c

此文件是 u-boot Stage2 部分，入口为 Stage1 最后调用的 start_armboot 函数。注意上面最后 ldr 到 pc 的是_start_armboot 这个地址，而非 start_armboot 变量。

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。

/*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。这里只简要列出了主要执行的函数流程
*/

```
/* U-Boot code: 00F00000 -> 00F3C774  BSS: -> 00FC3274
 * IRQ Stack: 00ebff7c
 * FIQ Stack: 00ebef7c
 */
```

/* 在 include/asm-arm/global_data.h 中定义的一个全局寄存器变量的声明*/

DECLARE_GLOBAL_DATA_PTR; 此宏定义了一个 gd_t 类型的指针 *gd，并指名用 r8 寄存器来存储

```
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
void nand_init (void);
#endif

ulong monitor_flash_len;

#ifdef CONFIG_HAS_DATAFLASH
extern int  AT91F_DataflashInit(void);
extern void dataflash_print_info(void);
#endif

#ifndef CONFIG_IDENT_STRING
#define CONFIG_IDENT_STRING ""
#endif

const char version_string[] =
    U_BOOT_VERSION" (" __DATE__ " - " __TIME__ ")"CONFIG_IDENT_STRING;
```

```
#ifdef CONFIG_DRIVER_CS8900
extern void cs8900_get_enetaddr (uchar * addr);
#endif

#ifdef CONFIG_DRIVER_RTL8019
extern void rtl8019_get_enetaddr (uchar * addr);
#endif

/*
 * Begin and End of memory area for malloc(), and current "brk"
 */
static ulong mem_malloc_start = 0;
static ulong mem_malloc_end = 0;
static ulong mem_malloc_brk = 0;

static
void mem_malloc_init (ulong dest_addr)
{
    mem_malloc_start = dest_addr;
    mem_malloc_end = dest_addr + CFG_MALLOC_LEN;
    mem_malloc_brk = mem_malloc_start;

    memset ((void *) mem_malloc_start, 0,
            mem_malloc_end - mem_malloc_start);
}

//本函数用于扩展堆空间,用 increment 指定要增加的大小
void *sbrk (ptrdiff_t increment)
{
    ulong old = mem_malloc_brk;
    ulong new = old + increment;

    if ((new < mem_malloc_start) || (new > mem_malloc_end)) {
        return (NULL);
    }
    mem_malloc_brk = new;

    return ((void *) old);
}

/****************************************************************************
 * Init Utilities                                *
 ****************************************************************************
 * Some of this code should be moved into the core functions,
```

```
 * or dropped completely,
 * but let's get it working (again) first...
 */

//获取环境变量，把其中的波特率值给全局 gd，若获取为空值，则赋默认的 115200
static int init_baudrate (void)
{
    char tmp[64];   /* long enough for environment variables */
//将环境变量获取到 tmp 中，i 为得到的环境变量的长度
    int i = getenv_r ("baudrate", tmp, sizeof (tmp));

//如果确实获取到了 env，则给 gd，否则把宏定义的 115200 给 gd
    gd->bd->bi_baudrate = gd->baudrate = (i > 0)

        //此函数把字符串转换成 ulong 整型
            ? (int) simple_strtoul (tmp, NULL, 10)
            : CONFIG_BAUDRATE;  //此宏为 115200

    return (0);
}

static int display_banner (void)
{
    printf ("\n\n%s\n\n", version_string);
    debug ("U-Boot code: %08lX -> %08lX  BSS: -> %08lX\n",
           _armboot_start, _bss_start, _bss_end);
#ifdef CONFIG_MODEM_SUPPORT
    debug ("Modem Support enabled\n");
#endif
#ifdef CONFIG_USE_IRQ
    debug ("IRQ Stack: %08lx\n", IRQ_STACK_START);
    debug ("FIQ Stack: %08lx\n", FIQ_STACK_START);
#endif

    return (0);
}

/*
 * WARNING: this code looks "cleaner" than the PowerPC version, but
 * has the disadvantage that you either get nothing, or everything.
 * On PowerPC, you might see "DRAM: " before the system hangs - which
 * gives a simple yet clear indication which part of the
 * initialization if failing.
 */
```

```
static int display_dram_config (void)
{
    int i;

#ifdef DEBUG
    puts ("RAM Configuration:\n");

    for(i=0; i<CONFIG_NR_DRAM_BANKS; i++) {
        printf ("Bank #%d: %08lx ", i, gd->bd->bi_dram[i].start);
        print_size (gd->bd->bi_dram[i].size, "\n");
    }
#else
    ulong size = 0;

    for (i=0; i<CONFIG_NR_DRAM_BANKS; i++) {
        size += gd->bd->bi_dram[i].size;
    }
    puts("DRAM:  ");
    print_size(size, "\n");
#endif

    return (0);
}

#ifndef CFG_NO_FLASH
static void display_flash_config (ulong size)
{
    puts ("Flash: ");
    print_size (size, "\n");
}
#endif /* CFG_NO_FLASH */


/*
 * Breathe some life into the board...
 *
 * Initialize a serial port as console, and carry out some hardware
 * tests.
 *
 * The first part of initialization is running from Flash memory;
 * its main purpose is to initialize the RAM so that we
 * can relocate the monitor code to RAM.
 */
```

```
/*
 * All attempts to come up with a "common" initialization sequence
 * that works for all boards and architectures failed: some of the
 * requirements are just _too_ different. To get rid of the resulting
 * mess of board dependent #ifdef'ed code we now make the whole
 * initialization sequence configurable to the user.
 *
 * The requirements for any new initalization function is simple: it
 * receives a pointer to the "global data" structure as it's only
 * argument, and returns an integer return code, where 0 means
 * "continue" and != 0 means "fatal error, hang the system".
 */
typedef int (init_fnc_t) (void);

int print_cpuinfo (void); /* test-only */
```

#初始化函数序列 init_sequence[]

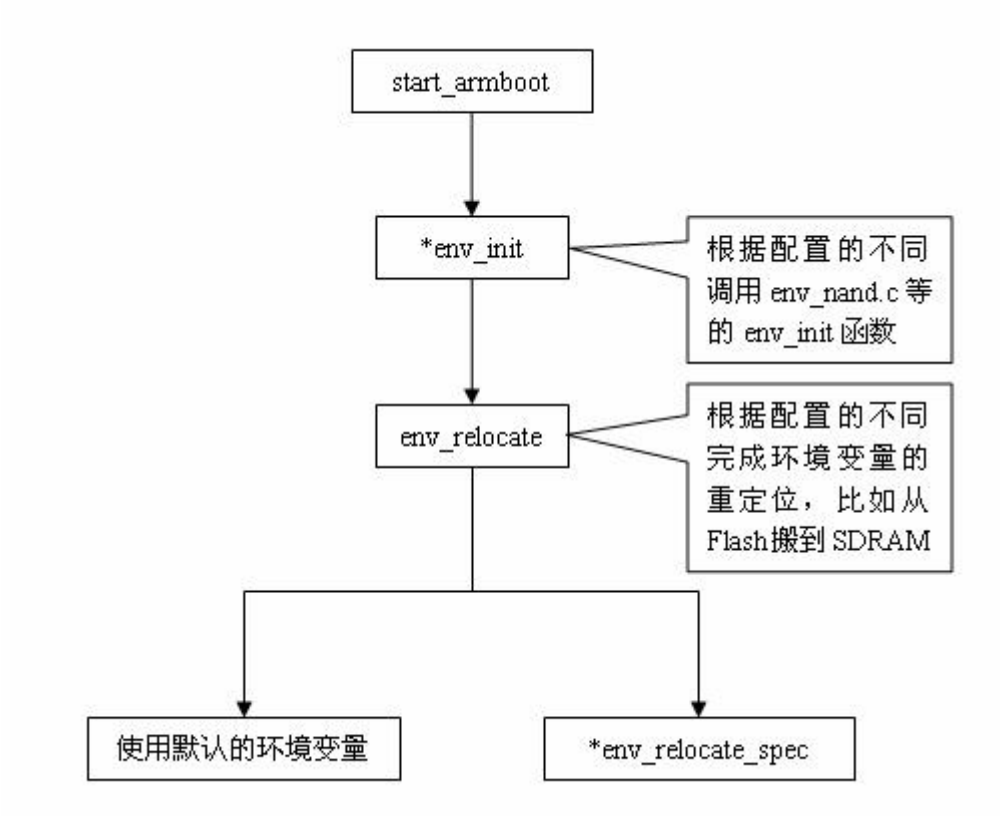
#init_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

```
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/env_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 */
#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 *//* 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,
};
//整个 u-boot 的执行就进入等待用户输入命令，解析并执行命令的死循环中。

void start_armboot (void)
{
    init_fnc_t **init_fnc_ptr;
```

```
    char *s;
#ifndef CFG_NO_FLASH
    ulong size;
#endif
#if defined(CONFIG_VFD) || defined(CONFIG_LCD)
    unsigned long addr;
#endif

    /* Pointer is writable since we allocated a register for it */
    /*初始化全局数据结构体指针 gd
此 C 语句引用的是 start.S 中的地址标号_armboot_start,但是得到的却是其中所指的变量_start 的值（在 RAM 中，_start = 0x33F80000）。    Ps：    _armboot_start:    .word _start，gd 是全局变量，位置在堆栈区以下(低地址)：*/
    gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__("": : :"memory");  /* 内联汇编语句__asm__(""::"memory")向 GCC 声明，在此内联汇编语句出现的位置内存内容可能改变了， 所以 GCC 在编译的时候，会将此因素考虑进去，而不会对代码进行优化及重新排序等操作,这样 GCC 会生成汇编代码 */

/* memset 在 lib_generic/string.c 中定义*/
    memset ((void*)gd, 0, sizeof (gd_t));/*用 0 填充全局数据表*gd */
    /* 给板子数据变量 gd->bd 得到 bd 的起点*/
    gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
    memset (gd->bd, 0, sizeof (bd_t));/*用 0 填充(初始化) *gd->bd */

    monitor_flash_len = _bss_start - _armboot_start;//取 u-boot 的长度，即 text 段和.data 段的总长

 /* 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 ();/* 打印错误信息并死锁 */
        }
    }

#ifndef CFG_NO_FLASH
/*配置可用的 Flash */
    size = flash_init ();//用于初始化 Nor flash 的函数，初始化了每个扇区的首地址 board/smdk2410/flash.c
    display_flash_config (size);    //打印到控制台：Flash: 512 kB
#endif /* CFG_NO_FLASH */

#ifdef CONFIG_VFD
#   ifndef PAGE_SIZE
```

```
#     define PAGE_SIZE 4096
#   endif
    /*
     * 为 VFD 显示预留内存(整个页面)
     */
    /* armboot_real_end 在 board-specific 链接脚本中定义 */
    addr = (_bss_end + (PAGE_SIZE - 1)) & ~(PAGE_SIZE - 1);
    size = vfd_setmem (addr);
    gd->fb_base = addr;
#endif /* CONFIG_VFD */

#ifdef CONFIG_LCD
#   ifndef PAGE_SIZE
#     define PAGE_SIZE 4096
#   endif
    /*
     * reserve memory for LCD display (always full pages)
     */
    /* bss_end is defined in the board-specific linker script */
    addr = (_bss_end + (PAGE_SIZE - 1)) & ~(PAGE_SIZE - 1);
    size = lcd_setmem (addr);
    gd->fb_base = addr;
#endif /* CONFIG_LCD */

    /* armboot_start is defined in the board-specific linker script */
     /* _armboot_start 在 u-boot.lds 链接脚本中定义*/
    mem_malloc_init (_armboot_start - CFG_MALLOC_LEN); //将 CFG_MALLOC_LEN 区域
用 memset 函数清零（直接往目的地址写 0）

//nand flash 初始化
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
    puts ("NAND:  ");
    nand_init();        /* 初始化 NAND */
#endif

#ifdef CONFIG_HAS_DATAFLASH
    AT91F_DataflashInit();
    dataflash_print_info();
#endif

    /* initialize environment */
    /* 重新定位环境变量，  */
```

env_relocate ();//刚才的初始化函数中有一个是 env_init()，根据 CRC 校验来初始化 gd->env_addr 变量(自己设定的还是初始值),此函数是作用是将环境变量值从某个 flash 和 RAM 之间的拷贝。下图描述了 ENV 的初始化过程：




```
#ifdef CONFIG_VFD
    /* must do this after the framebuffer is allocated */
    drv_vfd_init();
#endif /* CONFIG_VFD */

    /* IP Address */
    /* 从环境变量中获取 IP 地址,设置 IP 地址 */
    gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");

    /* MAC Address */
    /* 以太网接口 MAC 地址 */
    {
        int i;
        ulong reg;
        char *s, *e;
        char tmp[64];
        i = getenv_r ("ethaddr", tmp, sizeof (tmp));
        s = (i > 0) ? tmp : NULL;
        for (reg = 0; reg < 6; ++reg) {
            gd->bd->bi_enetaddr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
            if (s)
```

```
                s = (*e) ? e + 1 : e;
        }

#ifdef CONFIG_HAS_ETH1
        i = getenv_r ("eth1addr", tmp, sizeof (tmp));
        s = (i > 0) ? tmp : NULL;

        for (reg = 0; reg < 6; ++reg) {
            gd->bd->bi_enet1addr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
            if (s)
                s = (*e) ? e + 1 : e;
        }
#endif
    }

    devices_init ();     /* 外围设备初始化 */

#ifdef CONFIG_CMC_PU2
    load_sernum_ethaddr ();
#endif /* CONFIG_CMC_PU2 */

    jumptable_init ();  //跳转表初始化

    console_init_r ();  /* 完整地初始化控制台设备 */

#if defined(CONFIG_MISC_INIT_R)
    /* 其他平台由初始化决定 */
    misc_init_r ();
#endif

    /* 启用异常处理 */
    enable_interrupts ();

    /* Perform network card initialisation if necessary */
#ifdef CONFIG_DRIVER_CS8900
    cs8900_get_enetaddr (gd->bd->bi_enetaddr);
#endif

#if defined(CONFIG_DRIVER_SMC91111) || defined (CONFIG_DRIVER_LAN91C96)
    if (getenv ("ethaddr")) {
        smc_set_mac_addr(gd->bd->bi_enetaddr);
    }
#endif /* CONFIG_DRIVER_SMC91111 || CONFIG_DRIVER_LAN91C96 */
```

```
    /* Initialize from environment */
    /* 通过环境变量初始化 */
    if ((s = getenv ("loadaddr")) != NULL) {
        load_addr = simple_strtoul (s, NULL, 16);
    }
#if (CONFIG_COMMANDS & CFG_CMD_NET)
    if ((s = getenv ("bootfile")) != NULL) {
        copy_filename (BootFile, s, sizeof (BootFile));
    }
#endif  /* CFG_CMD_NET */

#ifdef BOARD_LATE_INIT
    board_late_init ();
#endif
#if (CONFIG_COMMANDS & CFG_CMD_NET)
#if defined(CONFIG_NET_MULTI)
    puts ("Net:   ");
#endif
    eth_initialize(gd->bd);
#endif
    /* main_loop() can return to retry autoboot, if so just run it again. */
    /*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 */
}

void hang (void)
{
    puts ("### ERROR ### Please RESET the board ###\n");
    for (;;);
}

#ifdef CONFIG_MODEM_SUPPORT
static inline void mdm_readline(char *buf, int bufsiz);

/* called from main loop (common/main.c) */
extern void  dbg(const char *fmt, ...);
int mdm_init (void)
{
    char env_str[16];
    char *init_str;
```

```
    int i;
    extern char console_buffer[];
    extern void enable_putc(void);
    extern int hwflow_onoff(int);

    enable_putc(); /* enable serial_putc() */

#ifdef CONFIG_HWFLOW
    init_str = getenv("mdm_flow_control");
    if (init_str && (strcmp(init_str, "rts/cts") == 0))
        hwflow_onoff (1);
    else
        hwflow_onoff(-1);
#endif

    for (i = 1;;i++) {
        sprintf(env_str, "mdm_init%d", i);
        if ((init_str = getenv(env_str)) != NULL) {
            serial_puts(init_str);
            serial_puts("\n");
            for(;;) {
                mdm_readline(console_buffer, CFG_CBSIZE);
                dbg("ini%d: [%s]", i, console_buffer);

                if ((strcmp(console_buffer, "OK") == 0) ||
                    (strcmp(console_buffer, "ERROR") == 0)) {
                    dbg("ini%d: cmd done", i);
                    break;
                } else /* in case we are originating call ... */
                    if (strncmp(console_buffer, "CONNECT", 7) == 0) {
                        dbg("ini%d: connect", i);
                        return 0;
                    }
            }
        } else
            break; /* no init string - stop modem init */

        udelay(100000);
    }

    udelay(100000);

    /* final stage - wait for connect */
    for(;i > 1;) { /* if 'i' > 1 - wait for connection
```

```
                message from modem */
        mdm_readline(console_buffer, CFG_CBSIZE);
        dbg("ini_f: [%s]", console_buffer);
        if (strncmp(console_buffer, "CONNECT", 7) == 0) {
            dbg("ini_f: connected");
            return 0;
        }
    }

    return 0;
}

/* 'inline' - We have to do it fast */
static inline void mdm_readline(char *buf, int bufsiz)
{
    char c;
    char *p;
    int n;

    n = 0;
    p = buf;
    for(;;) {
        c = serial_getc();

        /*      dbg("(%c)", c); */

        switch(c) {
        case '\r':
            break;
        case '\n':
            *p = '\0';
            return;

        default:
            if(n++ > bufsiz) {
                *p = '\0';
                return; /* sanity check */
            }
            *p = c;
            p++;
            break;
        }
    }
}
```

```
#endif  /* CONFIG_MODEM_SUPPORT */
```
nand_read.c 用以添加对 nand 分区的读的函数。
NOR flash 采用位读写，因为它具有 sram 的接口，有足够的引脚来寻址，可以很容易的存取其内部的每一个字节。

NAND flash 使用复杂的 I/O 口来串行地存取数据。8 个引脚用来传送控制、地址和数据信息（复用）。NAND 的读和写单位为 512Byte 的页，擦写单位为 32 页的块。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小，相应的擦除电路更少。

在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。
再来看看 Nand flash 自身的特点（部分摘自 August0703 的文章）：

Nand Flash 的数据是以 bit 的方式保存在 memory cell 中，一般来说，一个 cell 中只能存储一个 bit。这些 cell 以 8 个或者 16 个为单位，连成 bit line，形成所谓的 byte(x8)/word(x16)，这就是 NAND Device 的位宽。

多个 line（多个位宽大小的数据）会再组成 Page。我使用的 Nand flash 是三星的 K9F1208U0M，从 datesheet 上得知，此 flash 每页 528Bytes（512byte 的 Main Area + 16byte 的 Spare Area)，每 32 个 page 形成一个 Block(32*528B)。具体一片 flash 上有多少个 Block 视需要所定。我所使用的 k9f1208U0M 具有 4096 个 block，故总容量为 4096*（32*528B）=66MB，但是其中的 2MB（Spaer Area）是用来保存 ECC 校验码等额外数据的，故实际中可使用的为 64MB。

Nand flash 以页（512Byte）为单位读写数据，而以块（16KB）为单位擦除数据。按照这样的组织方式可以形成所谓的三类地址：

- Column Address：列地址，地址的低 8 位
- Page Address ： 页地址
- Block Address ： 块地址

对于 NAND Flash 来讲，地址和命令只能在 I/O[7:0]上传递，数据宽度也是 8 位，这导致在读写指定地址的数据时，地址是分 4 次传递的（3 次右移），见后文。

s3c2410 这个处理器之所以可以直接从 Nand flash 启动，是因为 CPU 内置了 4KB 的片内 SRAM，手册上称作“Steppingstone”。板子上电复位之后，CPU 会自动将 Nand flash 的前 4KB 代码拷贝到片内 SRAM 中去执行（此过程是靠硬件实现的，见 datasheet 图 Figure 6-1. NAND Flash Controller Block Diagram），这也是导致从 Nor 和 Nand 启动后的内存映射不同的原因。所以，vivi 的 stage1 代码 head.S 必须要小于 4KB，其中实现基本的 CPU 初始化等工作，并且要实现把自身拷贝到 SDRAM 中，之后的 stage2 就实现复杂功能。

S3C2410从NAND Flash启动：NAND Flash的开始4k代码会被自动地复制到内部SRAM中。我们需要使用这4k代码来把更多的代码从NAND Flash中读到SDRAM中去。NAND Flash的操作通过NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT和NFECC六个寄存器

来完成。在开始下面内容前，请打开S3C2410数据手册和NAND Flash K9F1208的数据手册。
在S3C2410数据手册218页，我们可以看到读写NAND Flash的操作次序：
1. Set NAND flash configuration by NFCONF register.
2. Write NAND flash command onto NFCMD register.
3. Write NAND flash address onto NFADDR register.
4. Read/Write data while checking NAND flash status by NFSTAT register.
R/nB signal should be checked before read operation or after program operation.
1、NFCONF：设为0xf830——使能NAND Flash控制器、初始化ECC、NAND Flash片选信号nFCE=1(inactive，真正使用时再让它等于0)、设置TACLS、TWRPH0、TWRPH1。需要指出的是TACLS、TWRPH0和TWRPH1，请打开S3C2410数据手册218页，可以看到这三个参数控制的是NAND Flash信号线CLE/ALE与写控制信号nWE的时序关系。我们设的值为TACLS=0，TWRPH0=3，TWRPH1=0，其含义为：TACLS=1个HCLK时钟，TWRPH0=4个HCLK时钟，TWRPH1=1个HCLK时钟。请打开K9F1208数据手册第13页，在表“AC Timing Characteristics for Command / Address / Data Input”中可以看到：
CLE setup Time = 0 ns，CLE Hold Time = 10 ns，
ALE setup Time = 0 ns，ALE Hold Time = 10 ns，
WE Pulse Width = 25 ns
可以计算，即使在HCLK=100MHz的情况下，TACLS+TWRPH0+TWRPH1=6/100 uS=60 ns，也是可以满足NAND Flash K9F1208U0M的时序要求的。
2、NFCMD： 对于不同型号的Flash，操作命令一般不一样。对于本板使用的K9F1208，请打开其数据手册第8页“Table 1. Command Sets”，上面列得一清二楚。
3、NFADDR：无话可说
4、NFDATA：只用到低8位
5、NFSTAT：只用到位0，0-busy，1-ready
6、NFECC：待补
现在来看一下如何从NAND Flash中读出数据，请打开K9F1208数据手册第29页“PAGE READ”，跟上面的第2段是遥相呼应，提炼出来罗列如下(设读地址为addr)： 1、NFCONF = 0xf830 2、在第一次操作NAND Flash前，通常复位一下：NFCONF &= ~0x800 (使能NAND Flash) NFCMD = 0xff (reset命令) 循环查询NFSTAT位0，直到它等于1 3、NFCMD = 0 (读命令) 4、这步得稍微注意一下，请打开K9F1208U0M数据手册第7页，那个表格列出了在地址操作的4个步骤对应的地址线，A8没用到：
NFADDR = addr & 0xff
NFADDR = (addr>>9) & 0xff (注意了，左移9位，不是8位)
NFADDR = (addr>>17) & 0xff (左移17位，不是16位)
NFADDR = (addr>>25) & 0xff (左移25位，不是24位)
5、循环查询NFSTAT位0，直到它等于1
6、连续读NFDATA寄存器512次，得到一页数据(512字节)
7、NFCONF |= 0x800 (禁止NAND Flash)

nand_read.c

```
1 #include <config.h>
2 #define __REGb(x)(*(volatile unsigned char *)(x))
3 #define __REGi(x)(*(volatile unsigned int *)(x))
4 #define NF_BASE 0x4e000000
5 #define NFCONF    __REGi(NF_BASE + 0x0)
6 #define NFCMD     __REGb(NF_BASE + 0x4)
7 #define NFADDR    __REGb(NF_BASE + 0x8)
8 #define NFDATA    __REGb(NF_BASE + 0xc)
9 #define NFSTAT    __REGb(NF_BASE + 0x10)
10 #define BUSY 1
11 inline void wait_idle(void) {
12              int i;
    /* NFSTAT：只用到位 0，0-busy，1-ready */
13              while(!(NFSTAT & BUSY))
14              for(i=0; i<10; i++);//这段的作用是判断 NAND 的状态，NFSTAT=0 表明 NAND 忙，执行延时子程序。等待 NAND 空闲
15 }
16 /* 下面的读过程严格按照 2410 手册上的顺序 */
17 int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
18 {
19              int i, j;
20              if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
21                      return -1; /* invalid alignment */
22              }
23 /* 对应第一条：1. Set NAND flash configuration by NFCONF register. */
24              NFCONF &= ~0x800;   /* 现在真正使用 Nand flash，bit[11]要置 0，与初始化时相反 */
25              for(i=0; i<10; i++);
26              for(i=start_addr; i < (start_addr + size);) {
27/* 对应第二条：2. Write NAND flash command onto NFCMD register. */
28                      NFCMD = 0; //为什么写 0 参考芯片手册，读命令
29/* 对应第三条：3. Write NAND flash address onto NFADDR register. NFADDR 寄存器也只用到低八位来传输，所以需要分 4 次来写入一个完整的 32 位地址，需要注意后 3 次的移位操作*/
30                      NFADDR = i & 0xff;
31                      NFADDR = (i >> 9) & 0xff;
32                      NFADDR = (i >> 17) & 0xff;
33                      NFADDR = (i >> 25) & 0xff;
    /* 对应第四条：4. Read/Write data while checking NAND flash status by NFSTAT register. 一个地址对应 512 个字节数据。所以，由于 8bit 位宽的限制，每次读取 8 位（1 个字节），共读 512 次得到 1 页 512Byte 数据*/
34                      wait_idle();
```

这里定义了两个宏来设置 NAND 的控制寄存器地址，NFCONF 是指向整形的地址，而其他都是指向字符的地址,这与寄存器具体使用情况有关，因为 datesheet 中只有 NFCONF 使用了 16 位，其他都只使用了 8 位。

```
35                      for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
36                              *buf = (NFDATA & 0xff);
37                              buf++;//一次循环读 NFDATA 寄存器 512 次，得到 512 个字节
38                      }
39              }
40 /* 读写完毕需要禁止 Nand flash ，与开始相对应*/
41              NFCONF |= 0x800; /* chip disable */
42              return 0;
43 }
```

u-boot命令的添加

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。

U-BOOT 支持Nand Flash 命令移植说明
1、设置配置选项在CONFIG_COMMANDS中, 打开CFG_CMD_NAND选项.

```
#define CONFIG_COMMANDS \
(CONFIG_CMD_DFL | \
    CFG_CMD_CACHE | \
    CFG_CMD_NAND | \
    /*CFG_CMD_EEPROM |*/ \
    /*CFG_CMD_I2C |*/ \
    /*CFG_CMD_USB |*/ \
    CFG_CMD_PING | \
    CFG_CMD_REGINFO | \
    CFG_CMD_DATE | \
    CFG_CMD_ELF)
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
#define CFG_NAND_BASE 0x4E000000 /* Nand Flash控制器在SFR区中起始寄存器地址 */
#define CFG_MAX_NAND_DEVICE 1 /* 支持的最在Nand Flash数据 */
#define SECTORSIZE 512 /* 1页的大小 */
#define NAND_SECTOR_SIZE SECTORSIZE
#define NAND_BLOCK_MASK (NAND_SECTOR_SIZE - 1) /* 页掩码 */
#define ADDR_COLUMN 1 /* 一个字节的Column地址 */
#define ADDR_PAGE 3 /* 3字节的页块地址, A9A25*/
#define ADDR_COLUMN_PAGE 4 /* 总共4字节的页块地址 */
#define NAND_ChipID_UNKNOWN 0x00 /* 未知芯片的ID号 */
#define NAND_MAX_FLOORS 1
#define NAND_MAX_CHIPS 1
/* Nand Flash命令层底层接口函数 */
#define WRITE_NAND_COMMAND(d, adr) do {rNFCMD = d;} while(0)
#define WRITE_NAND_ADDRESS(d, adr) do {rNFADDR = d;} while(0)
#define WRITE_NAND(d, adr) do {rNFDATA = d;} while(0)
#define READ_NAND(adr) (rNFDATA)
#define NAND_WAIT_READY(nand) {while(!(rNFSTAT&(1<<0)));}
#define NAND_DISABLE_CE(nand) {rNFCONF |= (1<<11);}
#define NAND_ENABLE_CE(nand) {rNFCONF &= ~(1<<11);}
/* 下面一组操作对Nand Flash无效 */
#define NAND_CTL_CLRALE(nandptr)
#define NAND_CTL_SETALE(nandptr)
#define NAND_CTL_CLRCLE(nandptr)
```

```
#define NAND_CTL_SETCLE(nandptr)
/* 允许Nand Flash写校验 */
#define CONFIG_MTD_NAND_VERIFY_WRITE 1
#endif /* CONFIG_COMMANDS & CFG_CMD_NAND*/
```
2. 加入自己的Nand Flash芯片型号
在include/linux/mtd/nand_ids.h中的对如下结构体赋值进行修改:
```
static struct nand_flash_dev nand_flash_ids[] = {
......
{"Samsung K9F1208U0M", NAND_MFR_SAMSUNG, 0x76, 26, 0, 4, 0x4000, 0},
.......
}
```
这样对于该款Nand Flash芯片的操作才能正确执行。
3. 编写自己的Nand Flash初始化函数
在board/crane2410/binnary2410.c中加入nand_init()函数.
```
void nand_init(void)
{
/* 初始化Nand Flash控制器, 以及Nand Flash 芯片 */
nand_reset();
/* 调用nand_probe()来检测芯片类型 */
printf ("%4lu MB\n", nand_probe(CFG_NAND_BASE) >> 20);
}
```
该函数在启动时被start_armboot()调用.

U-BOOT命令的使用

U-BOOT命令的介绍
UBOOT常用命令
通常使用help(或者只使用问号？)，来查看所有的UBOOT命令。将会列出在当前配置下所有支持的命令。但是我们要注意，尽管UBOOT提供了很多配置选项，并不是所有选项都支持各种处理器和开发板,有些选项可能在你的配置中并没有被选上。
获得帮助信息通过help可以获得当前开发板的UBOOT中支持的命令.
```
binnary2410# help
```
常用命令使用说明

askenv(F)
在标准输入（stdin）获得环境变量
autoscr
从内存（Memory）运行脚本。（注意，从下载地址开始，例如我们的开发板是从0x30008000处开始运行).
```
binnary2410# autoscr 0x30008000
## Executing script at 30008000  base
```
打印或者设置当前指令与下载地址的地址偏移。

bdinfo

打印开发板信息

```
binnary2410# bdinfo
-arch_number = 0x000000C1 （CPU体系结构号）
-env_t = 0x00000000 （环境变量）
-boot_params = 0x30000100 （启动引导参数）
-DRAM bank = 0x00000000 （内存区）
--> start = 0x30000000 （SDRAM起始地址）
--> size = 0x04000000 （SDRAM大小）
-ethaddr = 01:23:45:67:89:AB （以太网地址）
-ip_addr = 192.168.1.5 （IP地址）
-baudrate = 115200 bps （波特率）
```

bootp

通过网络使用Bootp或者TFTP协议引导境像文件。

```
binnary2410# help bootp
bootp [loadAddress] [bootfilename]
```

bootelf

默认从0x30008000引导elf格式的文件(vmlinux)

```
binnary2410# help bootelf
bootelf [address] - load address of ELF image.
```

bootd(=boot)

引导的默认命令，即运行U-BOOT中在“include/configs/smdk2410.h” 中设置的“bootcmd” 中的命令。如下：

```
#define CONFIG_BOOTCOMMAND "tftp 0x30008000 uImage; bootm 0x30008000";
```

在命令下做如下试验:

```
binnary2410# set bootcmd printenv
binnary2410# boot
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
binnary2410# bootd
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
```

tftp(tftpboot)

即将内核镜像文件从PC中下载到SDRAM的指定地址，然后通过bootm来引导内核，前提是所用PC要安装设置tftp服务。

下载信息：

```
binnary2410# tftp 0x30008000 zImage
TFTP from server 10.0.0.1; our IP address is 10.0.0.110
Filename 'zImage'.
Load address: 0x30008000
Loading:
#################################################################
```

```
#################################################################
#################################################
done
Bytes transferred = 913880 (df1d8 hex)
```
bootm

内核的入口地址开始引导内核。

```
binnary2410# bootm 0x30008000
## Booting image at 30008000 ...
Starting kernel ...
Uncompressing
Linux.....................................................................
......
done, .
```
go

直接跳转到可执行文件的入口地址，执行可执行文件。

```
binnary2410# go 0x30008000
## Starting application at 0x30008000 ...
```
cmp

对输入的两段内存地址进行比较。

```
binnary2410# cmp 0x30008000 0x30008040 64
word at 0x30008000 (0xe321f0d3) != word at 0x30008040 (0xc022020c)
Total of 0 words were the same
binnary2410# cmp 0x30008000 0x30008000 64
Total of 100 words were the same
```
coninfo

打印所有控制设备和信息，例如

```
-List of available devices:
-serial 80000003 SIO stdin stdout stderr
```
cp

内存拷贝，cp 源地址 目的地址 拷贝大小（字节）

```
binnary2410# help cp
cp [.b, .w, .l] source target count
binnary2410# cp 0x30008000 0x3000f000 64
```
date

获得/设置/重设日期和时间

```
binnary2410# date
Date: 2008-6-6 (Tuesday) Time: 06:06:06
```
erase(F)

擦除FLASH MEMORY, 由于该ARM板没有Nor Flash, 所有不支持该命令.

```
binnary2410# help erase
erase start end
- erase FLASH from addr 'start' to addr 'end'
erase start +len
- erase FLASH from addr 'start' to the end of sect w/addr 'start'+'len'-1
```

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

flinfo(F)

打印Nor Flash信息, 由于该ARM板没有Nor Flash, 所有不支持该命令.

iminfo

打印和校验内核镜像头, 内核的起始地址由CFG_LOAD_ADDR指定:
#define CFG_LOAD_ADDR 0x30008000 /* default load address */
该宏在include/configs/crane2410.h中定义.

```
binnary2410# iminfo
## Checking Image at 30008000 ...
Image Name: Linux-2.6.18
Created: 2008-06-5 7:43:01 UTC
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1047080 Bytes = 1022.5 kB
Load Address: 30008000
Entry Point: 30008040
Verifying Checksum ... OK
```

loadb

从串口下载二进制文件

```
binnary2410# loadb
## Ready for binary (kermit) download to 0x30008000 at 115200 bps...
## Total Size = 0x00000000 = 0 Bytes
## Start Addr = 0x30008000
```

md

显示指定内存地址中的内容

```
binnary2410## md 0
00000000: ea000012 e59ff014 e59ff014 e59ff014 ................
00000010: e59ff014 e59ff014 e59ff014 e59ff014 ................
00000020: 33f80220 33f80280 33f802e0 33f80340 ..3...3...3@..3
00000030: 33f803a0 33f80400 33f80460 deadbeef ...3...3`..3....
00000040: 33f80000 33f80000 33f9c0b4 33fa019c ...3...3...3...3
00000050: e10f0000 e3c0001f e38000d3 e129f000 .............).
00000060: e3a00453 e3a01000 e5801000 e3e01000 S...............
00000070: e59f0444 e5801000 e59f1440 e59f0440 D.......@...@...
00000080: e5801000 e59f043c e3a01003 e5801000 ....<...........
00000090: eb000051 e24f009c e51f1060 e1500001 Q.....O.`.....P.
000000a0: 0a000007 e51f2068 e51f3068 e0432002 ....h ..h0... C.
000000b0: e0802002 e8b007f8 e8a107f8 e1500002 . .............P.
000000c0: daffffb e51f008c e2400803 e2400080 ..........@...@.
000000d0: e240d00c e51f0094 e51f1094 e3a02000 ..@.......... ..
```

```
000000e0: e5802000 e2800004 e1500001 daffffb . ........P.....
000000f0: eb000006 e59f13d0 e281f000 e1a00000 ................
```

mm

顺序显示指定地址往后的内存中的内容,可同时修改,地址自动递增。

```
binnary2410# mm 0x30008000
30008000: e1a00000 ? fffff
30008004: e1a00000 ? eeeeee
30008008: e1a00000 ? q
binnary2410# md 30008000
30008000: 000fffff 00eeeeee e1a00000 e1a00000 ................
30008010: e1a00000 e1a00000 e1a00000 e1a00000 ................
30008020: ea000002 016f2818 00000000 000df1d8 .....(o.........
30008030: e1a07001 e3a08000 e10f2000 e3120003 .p....... ......
```

mtest

简单的RAM检测

```
binnary2410# mtest
Pattern FFFFFFFD Writing... Reading...
```

mw

向内存地址写内容

```
binnary2410# md 30008000
30008000: ffffdffd ffffdffc ffffdffb ffffdffa ................
binnary2410# mw 30008000 0 4
binnary2410# md 30008000
30008000: 00000000 00000000 00000000 00000000 ................
```

nm

修改内存地址, 地址不递增

```
binnary2410# nm 30008000
30008000: de4c457f ? 00000000
30008000: 00000000 ? 11111111
30008000: 11111111 ?
```

printenv

打印环境变量

```
binnary2410# printenv
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
ipaddr=10.0.0.110
serverip=10.0.0.1
netmask=255.255.255.0
stdin=serial
stdout=serial
stderr=serial
Environment size: 153/65532 bytes
```

ping

ping主机
binnary2410# ping 10.0.0.1
host 10.0.0.1 is alive
reset
复位CPU
run
运行已经定义好的U-BOOT的命令
binnary2410# set myenv ping 10.0.0.1
binnary2410# run myenv
host 10.0.0.1 is alive
saveenv(F)
保存设定的环境变量
setenv
设置环境变量
binnary2410# setenv ipaddr 10.0.0.254
binnary2410# printenv
ipaddr=10.0.0.254
sleep
命令延时执行时间
binnary2410# sleep 1
version
打印U-BOOT版本信息
binnary2410# version
U-Boot 1.2.0 (Jul 4 2008 - 12:42:27)
nand info
打印nand flash信息
binnary2410# nand info
Device 0: Samsung K9F1208U0M at 0x4e000000 (64 MB, 16 kB sector)
nand device <n>
显示某个nand设备
binnary2410# nand device 0
Device 0: Samsung K9F1208U0M at 0x4e000000 (64 MB, 16 kB sector)
... is now current device
nand bad
binnary2410# nand bad
Device 0 bad blocks:
nand read
nand read InAddr FlAddr size
InAddr: 从nand flash中读到内存的起始地址。
FlAddr: nand flash 的起始地址。
size: 从nand flash中读取的数据的大小。
binnary2410# nand read 0x30008000 0 0x100000
NAND read: device 0 offset 0, size 1048576 ...
1048576 bytes read: OK

nand erease
nand erase FlAddr size
FlAddr: nand flash 的起始地址
size: 从nand flash中擦除数据块的大小
binnary2410# nand erase 0x100000 0x20000
NAND erase: device 0 offset 1048576, size 131072 ... OK
nand write
nand write InAddr FlAddr size
InAddr: 写到Nand Flash中的数据在内存的起始地址
FlAddr: Nand Flash的起始地址
size: 数据的大小
binnary2410# nand write 0x30f00000 0x100000 0x20000
NAND write: device 0 offset 1048576, size 131072 ...
131072 bytes written: OK

命令简写说明

所以命令都可以简写，只要命令前面的一部分不会跟其它命令相同,就可以不用写全整个命令.

save命令

binnary2410# sa
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...Erasing sector 10 ... Erased 1 sectors

把文件写入Nand Flash如果把一个传到内存中的文件写入到Nand Flash中, 如:新的uboot.bin, zImage(内核),rootfs等, 如何做呢？我们可以用Nand Flash命令来完成. 但是Nand Flash写时,必须先要把Nand Flash的写入区全部擦除后,才能写. 下面以把内存0x30008000起长度为0x20000的内容写到Nand Flash中的0x100000为例.

binnary2410# nand erase 0x100000 20000
NAND erase: device 0 offset 1048576, size 131072 ... OK
CRANE2410 # nand write 0x30008000 0x100000 0x20000
NAND write: device 0 offset 1048576, size 131072 ...
131072 bytes written: OK

附：vivi命令

reset 命令

reset 命令可以复位Normandy 系统。

cpu 命令

cpu 命令用于管理系统的CPU 时钟频率。Normandy 系统中的CPU －S3C2410 最高时钟频率可以达到203MHz。通过cpu help 命令，系统将提示cpu 命令使用方法。

cpu help 命令使用示例：

```
vivi> cpu help
Usage:
cpu info -- Display cpu informatin
cpu set <clock> <ratio> -- Change cpu clock and bus clock
```

从cpu help 的显示结果，我们可以看到cpu 命令，有两个子命令参数，一个为info，一个为set。下面分别讲述cpu info 和cpu set 这两个命令。

cpu info

cpu info 命令用于显示系统CPU 当前的各种时钟频率，包括处理器主时钟频率（FCLK），AHB 总线频率（HCLK）和APB 总线频率（PCLK）。关于FCLK，HCLK 和PCLK 详细论述请参考Samsung 公司S3C2410 的用户手册。

```
vivi> cpu info
Processor Information (Revision: 0x41129200)
-----------------------------------------
Processor clock: 200000000 Hz （处理器主时钟频率）
AHB bus clock : 100000000 Hz （AHB 总线频率）
APB bus clock : 50000000 Hz （APB 总线频率）
Register values （S3C2410 中相关寄存器的值）
MPLLCON: 0x0005c040 (MDIV: 0x005c, PDIV: 0x04, SDIV: 0x00)
CLKDIVN: 0x00000003
vivi>
```

cpu set

cpu set 命令用于设置系统 CPU 的主时钟频率。

命令的详细格式如下：

```
cpu set clock ratio
```

参数clock 是要设置的系统CPU 主时钟频率（即FCLK）的值，单位为MHz；参数ratio 是要设置的FCLK：HCLK：PCLK 的比率，取值范围：0 ～ 3；0表示FCLK：HCLK：PCLK 为1:1:1，1 表示FCLK：HCLK：PCLK 为2:2:1，2 表示FCLK：HCLK：PCLK 为2:1:1，3 表示FCLK ： HCLK ： PCLK 为4:2:1。其实设置ratio 的值就是设置S3C2410 的CLKDIVN 寄存器的值。Samsung公司S3C2410的用户手册中有如下描述：

“The S3C2410X supports selection of Dividing Ratio between FCLK, HLCK and PCLK. This ratio is determined byHDIVN and PDIVN of CLKDIVN control register.

HDIVN PDIVN FLCK HCLK PCLK Divide Ratio

0 0 FLCK FLCK FLCK 1:1:1

0 1 FLCK FLCK FLCK/2 1:1:2
1 0 FLCK FLCK/2 FLCK/2 1:2:2
1 1 FLCK FLCK/2 FLCK/4 1:2:4
”
系统默认的CPU 主时钟频率为200MHz，FCLK : HCLK : PCLK 为4:2:1。
例如，

```
vivi> cpu info
Processor Information (Revision: 0x41129200)
Processor clock: 200000000 Hz
AHB bus clock : 100000000 Hz
APB bus clock : 50000000 Hz
Register values
MPLLCON: 0x0005c040 (MDIV: 0x005c, PDIV: 0x04, SDIV: 0x00)
CLKDIVN: 0x00000003
vivi> cpu set 100 2
vivi> cpu info
Processor Information (Revision: 0x41129200)
-------------------------------------------
Processor clock: 100000000 Hz
AHB bus clock : 50000000 Hz
APB bus clock : 50000000 Hz
Register values
MPLLCON: 0x0005c041 (MDIV: 0x005c, PDIV: 0x04, SDIV: 0x01)
CLKDIVN: 0x00000002
```

sleep 命令

sleep 命令用于系统睡眠。系统睡眠指让系统CPU 停止工作一段时间。sleep 命令没有help 子命令（即没有sleep help 这个命令），因为其用法比较简单，所以直接通过help 命令的提示，就可以指示明白其使用的方法了。

```
vivi> help
Usage:
flash [{cmds}] -- Manage Flash memory
cpu [{cmds}] -- Manage cpu clocks
md5sum <addr> <size> | downfile <size> -- Compute MD5
test [{cmds}] -- Test functions
prompt <string> -- Change a prompt
sleep <vlaue>{s|m|u} -- system sleep command. unit: s - second,
m - mili-second, u - micro-second
bon [{cmds}] -- Manage the bon file system
reset -- Reset the system
param [set|show|save|reset] -- set/get parameter
part [add|del|show|reset] -- Manage MTD partitions
mem {<param> | <subcmds>} -- Manage Memory
load {...} -- Load a file to RAM/Flash
go <addr> <a0> <a1> <a2> <a3> -- jump to <addr>
```

dump <addr> <length> -- Display (hex dump) a range of memory.
call <addr> <a0> <a1> <a2> <a3> -- jump_with_return to <addr>
boot [{cmds}] -- Booting linux kernel
help [{cmds}] -- Help about help?
sleep 命令详细格式如下：
sleep value{s|m|u}
参数vlaue 表示要让系统睡眠的时间值。参数vlaue 后面需要跟一个时间单位，有三种时间单位可以选择：s 表示 second，秒； m 表示mili-second，毫秒； u 表示micro-second，微秒。
例如，下面的命令将让系统睡眠5 秒钟。
vivi> sleep 5s
sleep 5 seconds
vivi>
例如，下面的命令将让系统睡眠15 毫秒。
vivi> sleep 15m
sleep 15 mili-seconds
vivi>
例如，下面的命令将让系统睡眠500 微秒。
vivi> sleep 500u
sleep 500 micro-seconds
vivi>
prompt 命令
prompt 命令用于改变bootloader 系统提示符，系统默认的提示符为“vivi”。prompt 命令没有help 子命令（即没有prompt help 这个命令，如果敲入prompt help，将不会得到系统的帮助，只会将系统提示符设置成“help”），因为其用法比较简单，所以直接通过help 命令的提示，就可以指示明白其使用的方法了。
vivi> help
Usage:
flash [{cmds}] -- Manage Flash memory
cpu [{cmds}] -- Manage cpu clocks
md5sum <addr> <size> | downfile <size> -- Compute MD5
test [{cmds}] -- Test functions
prompt <string> -- Change a prompt
sleep <vlaue>{s|m|u} -- system sleep command. unit: s - second,
m - mili-second, u - micro-second
bon [{cmds}] -- Manage the bon file system
reset -- Reset the system
param [set|show|save|reset] -- set/get parameter
part [add|del|show|reset] -- Manage MTD partitions
mem {<param> | <subcmds>} -- Manage Memory
load {...} -- Load a file to RAM/Flash
go <addr> <a0> <a1> <a2> <a3> -- jump to <addr>
dump <addr> <length> -- Display (hex dump) a range of memory.
call <addr> <a0> <a1> <a2> <a3> -- jump_with_return to <addr>

boot [{cmds}] -- Booting linux kernel
help [{cmds}] -- Help about help?
例如，将系统提示符设置成bootloader。
vivi> prompt bootloader
Prompt is chagned to "bootloader"
bootloader>
例如，将系统提示符设置成system。
vivi> prompt system
Prompt is chagned to "system"
system>

mem 命令

mem 系列命令用于对系统的内存进行操作。通过mem help 可以显示系统对mem 命令的帮助提示。

```
vivi> mem help
'mem' command usage:
----------command parameter list-----------
mem size -- probe dram size
mem read <addr> -- read a word(4bytes) from special dram address
mem write <addr> <vlaue> -- write a word(4bytes) into special dram
address
mem test <start_addr> <size> [<quiet>] -- memory test
-----------sub command list-------------
mem cmp <dst_addr> <src_addr> <length> -- compare
mem copy <dst_addr> <src_addr> <length> -- copy memory from<src_addr> to
<dst_addr>
mem info -- display memory infomation
mem reset -- reset memory control register
mem search <start_addr> <end_addr> <value> -- search memory address that
contain value in the special memory address range
vivi>
```

mem 系列命令中，涉及到写操作的命令只能适用于SRAM，DRAM 和SDRAM 之类的RAM 类存储器，以及S3C2410 的各种寄存器（CPU 的寄存器已经映射到内存地址空间了），而不适用于NOR Flash 这类ROM 类存储器。mem 系列命令中，只涉及到读操作的命令则既可以用于SRAM，DRAM 和SDRAM 之类的RAM 类存储器，以及S3C2410 的各种寄存器，也可以用于NOR Flash 这类ROM 类存储器。

mem 命令有很多子命令参数，下面我们分别的讲述。

mem size

mem size 命令用于侦测系统配置的RAM 类型存储器（如SRAM，DRAM或SDRAM）的大小。这条命令只侦测系统配置的RAM 类型存储器，不侦测ROM 类型的存储器。例如ARMer 配置64M SDRAM，那么侦测结果如下：

```
vivi> mem size
Detected memory size = 0x04000000, 64M (67108864 bytes)
vivi>
```

mem read

mem read 命令用于读取RAM 类型存储器中的一个字（ARM 系统中，4 个字节称为一个字，即32 位为一个字。16 位为半字，8 位为字节。）。
命令的详细格式如下：
mem read addr
参数addr 要读取的字数据的内存地址。mem read 命令也能够用于读取NOR Flash 这类ROM 存储器中的内容。
例如，Normandy系统中配置了32M NOR Flash 和64M SDRAM，NOR Flash用S3C2410 的片选nGCS0，SDRAM 用S3C2410 的片选nGCS6；那么NOR Flash 映射到地址空间0x0000000 ～ 0x01FFFFFF 之间，SDRAM 映射到地址空间0x30000000 ～ 0x33FFFFFF 之间。将Bootloader（即VIVI）通过Normandy 系统附带的JTAG 烧写程序sjf2410w.exe ， 烧写到0x00000000 处（即NOR Flash 开头部分）。按系统板上的复位键，复位系统，通过超级终端程序连接到Normandy 系统中，按PC 机空格键，当进入系统的bootloader 命令行交互界面后，执行下面的mem read 命令：

```
vivi> mem read 0
addr: 0x00000000, value: 0x00000000
vivi> mem read 0x00000004
addr: 0x00000004, value: 0xea0000cb
vivi>
```

由上面的命令执行结果可见，mem read 命令可以用于读取NOR Flash 的内容。

```
vivi> mem read 0x33f00000
addr: 0x33f00000, value: 0x00000000
vivi> mem read 0x33f00004
addr: 0x33f00004, value: 0xea0000cb
vivi> mem read 0x33f00010
addr: 0x33f00010, value: 0xea0000da
vivi> mem read 0x33f00100
addr: 0x33f00100, value: 0xeb00003b
vivi> mem read 0x33f01000
addr: 0x33f01000, value: 0xe8bd83f0
```

由上面的命令执行结果可见，mem read 命令用于读取RAM 的内容。

mem write

mem write 命令用于读取RAM 类型存储器中的一个字（ARM 系统中，4 个字节称为一个字，即32 位为一个字。16 位为半字，8 位为字节。）。
命令的详细格式如下：
mem write addr vlaue
参数addr 是要写入数据的内存地址；
参数vlaue 是要写入的数据，一个字（4 个字节）。
mem write 命令不能够用于读取NOR Flash 这类ROM 存储器中的内容，因为NOR Flash 的写、擦除等操作要通过向NOR Flash 内部的写状态机输入专门的操作命令，按照步骤一步一步地进行操作，而不是象RAM 一样直接操作即可。
例如，Normandy系统中配置了32M NOR Flash 和64M SDRAM，NOR Flash用S3C2410 的片选nGCS0，SDRAM 用S3C2410 的片选nGCS6；那么NOR Flash 映射到地址空间0x0000000 ～ 0x01FFFFFF 之间，SDRAM 映射到地址空间0x30000000 ～

0x33FFFFFF 之间。将Bootloader（即VIVI）通过Normandy 系统附带的JTAG 烧写程序sjf2410w.exe ， 烧写到0x00000000 处（即NOR Flash 开头部分）。按系统板上的复位键，复位系统，通过超级终端程序连接到Normandy 系统中，按PC 机空格键，当进入系统的bootloader 命令行交互界面后，执行下面的mem write 命令：
例如：向地址0x01000000（NOR Flash 区域，Normandy 有32M NOR Flash，地址范围为0x0000000 ～ 0x02000000）写入一个字，写入不成功！
写入前，读取地址0x01000000 的内容，发现为0xFFFFFFFF；
vivi> mem read 0x01000000
addr: 0x01000000, value: 0xffffffff
写操作，向地址0x01000000 处写入0x111111FF；
vivi> mem write 0x01000000 0x111111ff
addr: 0x01000000, value: 0x111111ff
写入后，再读取地址0x01000000 的内容，发现数据0x111111FF 根本就没有写进去。
vivi> mem read 0x01000000
addr: 0x01000000, value: 0xffffffff
vivi>
例如：向地址0x30000000（SDRAM 区域，Normandy 有64M NOR Flash，地址范围为0x30000000 ～ 0x33FFFFFF）写入一个字0x87654321，可以写入成功。写入前，读取地址0x30000000 的内容，发现为0x12345678；
vivi> mem read 0x30000000
addr: 0x30000000, value: 0x12345678
写操作，向地址0x30000000 处写入0x87654321；
vivi> mem write 0x30000000 0x87654321
addr: 0x30000000, value: 0x87654321
写入后，再读取地址0x30000000 的内容，发现0x30000000 地址中的字数据变为了0x87654321，写入成功。
vivi> mem read 0x30000000
addr: 0x30000000, value: 0x87654321
vivi>
mem test
mem test 命令用于系统内存测试。测试某断内存是否可以正常使用。命令的详细格式如下：
mem test start_addr size
参数start_addr 是内存测试的起始地址；
参数size 是内存测试的长度，从起始地址开始要测试多长字节，就由这个参数决定。mem test 命令不能够用于NOR Flash 这类ROM 存储器的测试，因为测试的过程中要进行写操作。
例如：从地址0x0100000（属于NOR Flash 区域，Normandy 有32M NOR Flash，地址范围为0x0000000 ～ 0x02000000)开始测试，测试0x100000个字节(即1M 字节），可以发现mem test 是不能用于测试Flash 的。
vivi> mem test 0x0100000 0x100000
Memory Test: addr = 0x100000 size = 0x100000

```
data test 1...
writing block : 0001
checking block : 0001
data test 2...
checking block : 0001
address line test...
writing block : 0001
checking block : 0001
2nd checking block: 0001
address line test (swapped)...
writing block : 0001
checking block : 0001
2nd checking block: 0001
random data test...
writing block : 0001
checking block : 0001
Memory Test Result
Stage 1: data test 1... FAIL
Stage 2: data test 2... FAIL
Stage 3: address line test... FAIL
Stage 4: address line test (swapped)... FAIL
Stage 5: random data test... FAIL
```

从地址0x3000000（属于NOR Flash 区域，Normandy 有64M NOR Flash，地址范围为0x30000000 ～ 0x33FFFFFF)开始测试,测试0x100000 个字节(即1M 字节)，可见mem test 是只能用于RAM 类型的存储器的测试。

```
vivi> mem test 0x30000000 0x1000000
Memory Test: addr = 0x30000000 size = 0x1000000
data test 1...
writing block : 001F
checking block : 001F
data test 2...
writing block : 001F
checking block : 001F
address line test...
writing block : 001F
checking block : 001F
2nd checking block: 001F
address line test (swapped)...
writing block : 001F
checking block : 001F
2nd checking block: 001F
random data test...
writing block : 001F
checking block : 001F
```

```
Memory Test Result
Stage 1: data test 1... OK
Stage 2: data test 2... OK
Stage 3: address line test... OK
Stage 4: address line test (swapped)... OK
Stage 5: random data test... OK
```

1.5.5 mem cmp

mem cmp 命令用于两段内存的内容比较。命令中将指定通过起始地址来指定两内存区域，从起始地址开始，对这两段内存区域进行逐个字（4 个字节）逐个字地比较，发现在两段内存区域中相同偏移处如果有不同的字数据，则报告给用户。命令的详细格式如下：

mem cmp dst_addr src_addr length

参数dst_addr 是目的内存段的起始地址；

参数src_addr 是源内存段的起始地址；

参数length 是两个内存段的要进行比较的字节数。

mem cmp 命令也能够用于NOR Flash 这类ROM 存储器中的内容比较。

例如：目的段内存区域从0x00000000 开始，源段内存区域从0x00001000开始，内存长度为0x1000；命令“mem cmp 0x0 0x1000 0x1000”也就是要比较0x00001000～0x00001FFF 和0x0000000～0x00000FFF 这两段内存区域中的数据是否完全一致。

```
vivi> mem cmp 0x0 0x1000 0x1000
||(0%)(进度条指示,这里为0%,表示在开始就发现了不匹配的字)Not matched.
offset = 0x00000000(第一个不匹配的字数据在这两段内存区域中的偏移)value:
src = 0xe8bd83f0, dst = 0x00000000（两段内存区域中第一个不匹配的字数据的值）OK.
```

例如：两段完全重叠的内存区域比较，当然完全匹配。

```
vivi> mem cmp 0x0 0x0 0x1000
|====================|(100%)
OK.
vivi>
```

mem copy

mem copy 命令用于内存拷贝操作，即将一段内存的内容拷贝到另外一段内存中。命令的详细格式如下：

mem copy dst_addr src_addr length

参数dst_addr 是目的内存段的起始地址，该参数所指定的地址不能在NOR Flash 区域中，只能在RAM 类型存储器区域中；参数src_addr 是源内存段的起始地址，该参数所指定的地址既可以是在RAM 类型存储器区域中的，也可以是在NOR Flash 区域中的；参数length 是拷贝的字节数。拷贝前，目的内存段0x30010000 开始处的256 个字节内容如下：

```
vivi> dump 0x30010000 0x100
30010000: 1C 06 81 B9 F1 5C 3F 23-78 8E 9F 91 C2 18 30 37
| .....\?#x.....07.....
30010010: 9F D3 67 64 CD C9 33 B2-E4 C4 19 D9 8E 3D 73 13
```

拷贝前，源内存段0x30001000 开始处的256 个字节内容如下：

```
vivi> dump 0x30001000 0x100
| ..>...`..^0...g>
30001080: B4 17 CC 60 35 F4 99 4F-0A FA CC A7 EB 82 19 2C
| ...`5..O.......,
30001090: 64 C1 0C 96 5F 9F F9 34-BD CF 7C 9A CD 67 3E CD |
d..._..4..|..g>.
300010A0: F5 33 9F E6 EE 99 4F F3-1C 33 58 06 64 E6 D3 7C
| .3....O..3X.d..|.....................
```

将地址0x30001000 开始的256 个字节拷贝到地址0x30010000 处。

```
vivi> mem copy 0x30001000 0x30000000 0x100
Copy from 0x30000000, to 030001000. length is 0x00000100
Copied 256 (0x00000100) bytes
```

检查一下拷贝后，目的内存段0x30010000 开始处的内容是否和源内存段0x30001000 开始处的256 个字节相同，结果是相同的。

```
vivi> dump 0x30010000 0x100
30010000: 21 43 65 87 C3 D4 E5 76-61 EA 72 BB 31 75 B9 DD
| !Ce....va.r.1u..
30010010: 99 BA DC EE 4E 5D 6E F7-5A D1 48 04 51 97 DB 7D
| ....N]n.Z.H.Q..}
30010020: AB CB ED BE D1 E5 76 DF-EC 72 BB EF 8C 46 22 08
| ......v..r...F".
30010030: BC DC EE 7B A7 91 08 42-D5 48 04 A1 6D 24 82 D0
| ...{...B.H..m$..
30010040: 31 12 41 E8 10 89 20 F4-7F BB EF 05 B6 DD F7 82 |
```

当然更简单的检查方法是利用前面讲过的mem cmp 命令，来比较拷贝前后的两个内存段既可。

```
vivi> mem cmp 0x30010000 0x30001000 0x100
|====================|(100%)
OK.
```

mem info

mem info 命令用于显示Normandy 系统的存储器映射信息以及S3C2410的内存控制寄存器的值。

例如：

```
vivi> mem info
RAM Information:
Default ram size: 64M
Real ram size : 64M
Free memory : 61M
RAM mapped to : 0x30000000 - 0x34000000 （SDRAM 映射的地址范围）
Flash memory mapped to : 0x10000000 - 0x12000000 （Flash 映射的地址范围）
Available memory region : 0x30000000 - 0x33de8000 （用户可以使用的有效的内存区域地址范围）
Stack base address : 0x33defffc （栈的基地址）
```

Current stack pointer : 0x33defc7c （当前栈指针的值）
Memory control register vlaues （S3C2410 的内存控制寄存器的当前值）
BWSCON = 0x22111114
BANKCON0 = 0x00001ff0
BANKCON1 = 0x00000700
BANKCON2 = 0x00000700
BANKCON3 = 0x00000700
BANKCON4 = 0x00000700
BANKCON5 = 0x00000700
BANKCON6 = 0x00018005
BANKCON7 = 0x00018005
REFRESH = 0x008e0459
BANKSIZE = 0x000000b2
MRSRB6 = 0x00000030
MRSRB7 = 0x00000030

mem reset

mem reset 命令用于复位S3C2410 中的内存控制寄存器的内容。该命令执行的空指令，暂时没有做任何事，留待以后扩展。

mem search

mem search 命令用于在一段指定的内存区域中（从参数start_addr 开始到参数end_addr 结束的内存区域）搜索是否有和参数vlaue 的值相同的字，将这个字的地址显示给用户。
命令详细格式如下：
mem search start_addr end_addr value
参数dst_addr 是目的内存段的起始地址；
参数src_addr 是源内存段的起始地址；
参数length 是两个内存段的要进行比较的字节数。
例如：在0x30000000～0x30001000 地址范围内搜索字数据0xddb97531。

```
vivi> mem search 0x30000000 0x30001000 0xddb97531
serach 0xddb97531 value from 0x30000000 to 0x30001000
address = 0x3000000c
vivi>
```

flash 命令

flash 系列命令用于对系统的Flash（NOR Flash 或NAND Flash）进行操作。通过flash help 可以显示系统对flash 系列命令的帮助提示。

```
vivi> flash help
Usage:
flash help
flash erase [<partition>] or [<start_addr> <length>]
flash lock <start_addr> <length>
flash unlock <start_addr> <length>
flash info
vivi>
```

flash 命令主要用来对NOR Flash 或NAND Flash 进行操作。不同定制的Normandy

系统中，可能只有NOR Flash，也可能只有NAND Flash，也可能既有NOR Flash ，又有NAND Flash。flash 命令是针对NOR Flash 操作还是针对NAND Flash 操作，这取决于bootloader 编译的过程中，所进行的配置，这就要看配置的时候将MTD设备配置成NOR Flash 还是NANDFlash。
Flash 系列命令中不是所有的命令都适用于NAND Flash 的，但是所有命令都适用于NOR Flash。

flash erase

flash erase 命令用于擦除flash 设备。这条命令既适用于NOR Flash，也适用于NAND Flash。
命令的详细格式如下：
flash erase { partition | start_addr length }
参数partition 是MTD 分区表中的分区名称，参见part 命令，这样将擦除该MTD partition。
参数start_addr 是要擦除的区域的起始地址，对NAND Flash（Normandy系统中配置的NAND Flash 为一片K9S1208U0M，根据具体系统以及应用，可选择配置）来说，一个页面大小512 bytes；对于NOR Flash（Normandy系统中配置的NOR Flash为两片E28F128J3，根据具体系统以及应用，也可选择配置成E28F320J3 或E28F640J3）来说，一个块的大小为128Kbytes。建议将该参数设置在页或者块的边界上，这样明了一些。参数length 是要擦除的区域的大小，单位字节。该参数值要设置成页大小（对于NAND Flash）或者块大小（对于NOR Flash）的整数倍。例如，如果Normandy 系统配置为2 片E28F128J3，一共32M NOR Flash，那么因为E28F128J3 一个擦除块的大小为128K（0x20000），那么了两片E28F128J3并联，对于这个系统来说，一个最小的擦除单位是128K＊2=256K（0x40000，即每片E28F128J3 中擦除一块），所以在执行flash erase 命令的时候，参数length应该设置成0x40000 的整数倍。参数partition 和参数start_addr length 二者选择其一。
例如，Normandy 系统配置了一片K9S1208U0M NAND Flash，没有配置NOR Flash，现在擦除从Nand Flash 的0x02000000 开始的0x100000 个字节的区域。

```
vivi> flash erase 0x02000000 0x100000
Erasing block from 0x02000000 to 0x02100000
Erasing block from 0x02000000 to 0x02100000... ... done
vivi>
```

flash lock

flash lock 命令用于锁定NOR Flash 的某些块。这条命令只适用于NOR Flash，不能用于NAND Flash。
flash lock { partition | start_addr length }
参数start_addr 是要锁定的区域的起始地址，建议将该参数设置在页或者块的边界上，这样明了一些；
参数length 是要锁定的区域的大小，单位字节，该参数值要设置成页大小（对于NAND Flash）或者块大小（对于NOR Flash）的整数倍。

flash unlock

flash unlock 命令用于解除NOR Flash 中的某些块锁定位。这条命令只适用于NOR Flash，不能用于NAND Flash。
flashunlock { partition | start_addr length }

参数start_addr 是要解除锁定的区域的起始地址，建议将该参数设置在页或者块的边界上，这样明了一些；
参数length 是要解除锁定的区域的大小，单位字节，该参数值要设置成页大小（对于NAND Flash）或者块大小（对于NOR Flash）的整数倍。

flash info

flash info 命令用于显示系统的NOR Flash 配置信息。这条命令只适用于NOR Flash，不能用于NAND Flash。

load 命令

load 命令下载程序到存储器中（Flash 或者RAM 中）。通过load help 可以显示系统对load 系列命令的帮助提示。

```
vivi> load help
Usage:
load <flash|ram> [ <partname> | <addr> <size> ] <x|y|z>
vivi>
```

命令的详细格式如下：

```
load { flash | ram } { partname | addr size } { x | y | z }
```

关键字参数flash 和ram 用于选择目标介质是Flash 还是RAM。其实下载到Flash 中还是先要下载到RAM 中（临时下载到SDRAM 的起始地址处0x30000000 保存一下，然后再转写入FLASH），然后再通过Flash 驱动程序提供的写操作，将数据写入到Flash 中。如果选择了flash 参数，那名然到底是对NOR Flash 操作还是对NAND Flash 操作，这取决于bootloader编译的过程中，所进行的配置，这就要看配置的时候将MTD设备配置成NORFlash 还是NAND Flash。参数partname 和addr size 二者选其一，partname 是vivi 的MTD 分区表中的分区名称，表示下载的目标地址是指定的MTD 分区的起始地址;addr 和size 是让用户自己选择下载的目标存储区域，而不是使用vivi 的MTD 分区，addr 表示下载的目标地址，size 表示下载的文件大小，单位字节，size 参数不一定非要指定得和待下载的文件大小一样大，但是一定要大于等于待下载的文件的字节数。关键字参数x、y 和z 分别表示从PC 主机上下载文件到Normandy 系统中，采用哪种串行文件传送协议，x 表示采用XModem协议，y 表示采用Ymodem协议，z 表示采用ZModem 协议。请注意。目前该bootloader — vivi 还没有实现ZModem 协议，所以该参数只能选择x 和y。

例如：通过Xmodem 文件传送协议将PC 主机上的某个文件下载到
Normandy 系统的SDRAM 中（地址0x30100000 开始）。

```
vivi> load ram 0x30100000 0x30000 x
Ready for downloading using xmodem...
Waiting...
download file at 0x30100000, size = 172032 bytes
```

param 命令

param 系列命令用于对bootloader 的参数进行操作。通过param help 可以显示系统对param 系列命令的帮助提示。

```
vivi> param help
Usage:
param help -- Help aout 'param' command
param reset -- Reset parameter table to default table
```

```
param save -- Save parameter table to flash memeory
param set <name> <value> -- Reset value of parameter
param set linux_cmd_line "..." -- set boot parameter
param show -- Display parameter table
vivi>
```

在系统复位，bootloader 启动的过程中，其中有如下的提示，这就是bootloader的一系列参数。

```
vivi> reset
vivi>
@00000100
MTST OK
STKP
33DEFFFC
VIVI version 0.1.4 (root@localhost.localdomain) (gcc version 2.95.2
20000516 (release) [Rebel.com]) #0.1.4 五 8 月 6 23:23:12 HKT 2004
Evacuating 1MB of Flash to DRAM at 0x33F00000
MMU table base address = 0x33DFC000
Map flash virtual section to DRAM at 0x33F00000
Found default vivi parameters
Number of parameters: 9
name : hex integer
-------------------------------------------------------------
mach_type : 000000c1 193
media_type : 00000002 2
boot_mem_base : 30000000 805306368
baudrate : 0001c200 115200
xmodem_one_nak : 00000000 0
xmodem_initial_timeout : 000493e0 300000
xmodem_timeout : 000f4240 1000000
ymodem_initial_timeout : 0016e360 1500000
boot_delay : 01000000 16777216
Linux command line: noinitrd root=/dev/mtdblock3 init=/linuxrc
console=ttySAC0
…… …… …… …… …… …… …… …… ……
```

下面对各个bootloader 的参数做一个说明。

(1) mach_type

机器类型，193 表示S3C2410 的开发系统。

(2) media_type

媒介类型，即指示了bootloader 从哪个媒介启动起来的。

(3) boot_mem_base

引导linux 内核启动的基地址。内核映像将被从Flash 中拷贝到boot_mem_base ＋ 0x8000 的地址处， 内核参数将被建立在boot_mem_base＋0x100 的地址处。

(4) baudrate

bootloader 启动时，默认设置的串口波特率。

(5) xmodem_one_nak
(6) xmodem_initial_timeout
(7) xmodem_timeout
以上三个参数和Xmodem 文件传送协议相关。
xmodem_one_nak 表示接收端（即Normandy 系统这端）发起第一个NAK信号给发送端（即PC 主机这端）到启动；
xmodem_initial_timeout 表示接收端（即Normandy 系统这端）启动XModem协议后的初始超时时间，第一次接收超时按照这个参数的值来设置，但是超时一次后，后面的超时时间就不再是这个参数的值了，而是xmodem_timeout的值；
xmodem_timeout 表示在接收端（即Normandy 系统这端）等待接受发送端（即PC 主机这端）送来的数据字节过程中，如果发生了一次超时，那么后面的超时时间就设置成参数xmodem_timeout 的值了。
这三个参数不需要修改，系统默认的值就可以了，不建议用户去修改这几个参数值。
(8) ymodem_initial_timeout
ymodem_initial_timeout 表示接收端（即Normandy 系统这端）在启动了YModem 协议后的初始超时时间。
这个参数不需要修改，系统默认的值就可以了，不建议用户去修改这几个参数值。
(9) boot_delay
boot_delay 是bootloader 自动引导linux kernel 功能的延时时间。前面已经说过（reset 命令的解释中），当bootloader 启动的时候，在启动信息的最后面有下列提示：
“Press Return to start the LINUX now, any other key for vivi type "help" for help.（敲回车键将引导linux 内核，敲其他任意键将进入bootloader 命令交互界面，help 命令提供了命令行使用的帮助。）
这个时候系统将等待用户输入，如果用户在boot_delay 参数设置的时间内，还没有按下除了回车键以外的任意键，那么系统将自动引导linux kernel，如果指定位置处没有linux kernel，bootloader 将运行飞掉。
(10) Linux command line
Linux command line 不是bootloader 的参数，而是kernel 启动的时候，kernel 不能自动检测到的必要的参数。这些参数需要bootloader 传递给linux kernel。设置Linux command line 就是设置linux kernel 启动时，需要手工传给kernel 的参数。
例如
param show
param show 命令用于显示bootloader 的参数的当前值。
vivi> param show
Number of parameters: 9
name : hex integer
---
mach_type : 000000c1 193
media_type : 00000002 2
boot_mem_base : 30000000 805306368
baudrate : 0001c200 115200

xmodem_one_nak : 00000000 0
xmodem_initial_timeout : 000493e0 300000
xmodem_timeout : 000f4240 1000000
ymodem_initial_timeout : 0016e360 1500000
boot_delay : 01000000 16777216
Linux command line: noinitrd root=/dev/mtdblock3 init=/linuxrc
console=ttySAC0
vivi>
param reset
param reset 命令用于将bootloader 的参数值复位成系统默认的值。
Normandy 系统默认的bootloader 的参数值为如下：
Number of parameters: 9
name : hex integer
---
mach_type : 000000c1 193
media_type : 00000002 2
boot_mem_base : 30000000 805306368
baudrate : 0001c200 115200
xmodem_one_nak : 00000000 0
xmodem_initial_timeout : 000493e0 300000
xmodem_timeout : 000f4240 1000000
ymodem_initial_timeout : 0016e360 1500000
boot_delay : 01000000 16777216
Linux command line: noinitrd root=/dev/mtdblock3 init=/linuxrc
console=ttySAC0其实以上系统默认的值已经完全满足要求了，可以不用修改，唯一可能需要修改的是Linux command line，以便来适应Normandy 系统上不同的linux系统配置。
例如，复位bootloader 的参数值
vivi> param reset
param set
param set 命令用于修改bootloader 的参数值。该命令不能用来增加新的bootloader 参数。新的bootloader 参数也没有任何意义，因为目前的参数足够用了。
命令的详细格式如下：
param set name value
参数name 是bootloader 参数的名字。只能取上面9 个bootloader 参数的名字。
参数value 是bootloader 参数的修改后的值。
例如，修改bootloader 参数boot_mem_base 的值为0x31000000
vivi> param set boot_mem_base 0x31000000
Change 'boot_mem_base' value. 0x30000000(805306368) to 0x31000000(822083584)
vivi>param show
Number of parameters: 9
name : hex integer

```
------------------------------------------------------------
mach_type : 000000c1 193
media_type : 00000002 2
boot_mem_base : 31000000 822083584
baudrate : 0001c200 115200
xmodem_one_nak : 00000000 0
xmodem_initial_timeout : 000493e0 300000
xmodem_timeout : 000f4240 1000000
ymodem_initial_timeout : 0016e360 1500000
boot_delay : 01000000 16777216
Linux command line: noinitrd root=/dev/bon/2 init=/linuxrc console=ttyS0
vivi>
```

param set linux_cmd_line

param set linux_cmd_line 命令用于修改bootloader 传递给linux kernel的内核参数。这条命令非常有用。例如，你可能针在Normandy 系统中Porting多种文件系统，这时，这条命令帮助你设置传递给linux kernel 的不同的内核参数。命令的详细格式如下：

param set linux_cmd_line “linux_command_line”

参数linux_command_line 表示要设置的linux kernel 命令行参数。

例如，欲通过/dev/mtdblock/3 来加载JFFS2 文件系统。

```
vivi> param set linux_cmd_line "noinitrd root=/dev/mtdblock/3
init=linuxrcconsole=ttyS0"
Change linux command line to "noinitrd root=/dev/mtdblock/3 init=linuxrc
console=ttyS0"
```

例如，配置kernel 启动的时候使用Normandy 系统的UART1。

```
vivi> param set linux_cmd_line "noinitrd root=/dev/bon/3 init=linuxrc
console=ttyS1"
Change linux command line to "noinitrd root=/dev/bon/3 init=linuxrc
console=ttyS1"
```

param save

param save 命令用于将bootloader 的参数当前值保存到Flash 中。如果Normandy 系统配置的MTD 设备是NAND Flash，那么则将保存到NANDFlash 中；如果如果Normandy 系统配置的MTD 设备是NOR Flash，那么则将保存到NOR Flash 中。编译bootloader 的过程中，如果make menuconfig 配置的时候，定义了宏CONFIG_USE_PARAM_BLK，那么执行param save 命令将把bootloader的参数保存到MTD 分区的param 分区中（param 分区的偏移为0x20000，长度为0x10000）。这里还有两种情况，如果Normandy 系统配置的MTD设备是NOR Flash，那么因为bootloader 对系统做了地址映射，所以此时param 分区偏移0x20000 的实际物理地址为0x33f20000（因为0x20000 为虚地址，通过MMU 做了映射），这就不是Flash 了，而是SDRAM，因此系统断电后，bootloader 的参数修改情况并没有得到有效保存；如果Normandy系统配置的MTD设备是NAND Flash，那么就保存到NAND Flash的0x2000 地址处（因为NAND Flash 没有地址线，所以它是不占用系统地址空间的，它的地址空间是单独的）。

编译bootloader 的过程中，如果make menuconfig 配置的时候，没有定义宏

CONFIG_USE_PARAM_BLK ， 那么执行param save 命令将把bootloader 的参数保存到VIVI_PRIV_ROM_BASE（即0x01FC0000）地址处。这里也有两种情况，如果Normandy 系统配置的MTD设备是NOR Flash，那么虽然bootloader 对系统做了地址映射，但是映射的结果是虚拟地址0x01FC0000 的实际物理地址也为0x01FC0000，所以这个地址仍然在Flash中；如果Normandy 系统配置的MTD 设备是NAND Flash，那么就保存到NAND Flash 的0x01FC0000 地址处（因为NAND Flash没有地址线，所以它是不占用系统地址空间的，它的地址空间是单独的）。这两种情况，bootloader 的参数修改情况都得到了有效的保存。
编译bootloader 的过程中，如果make menuconfig 配置的时候，没有定义宏CONFIG_PARSE_PRIV_DATA，那么系统每次启动的时候将读取上次用户保存的bootloader 参数，而不是系统默认的bootloader 参数。

```
vivi> param save
Found block size = 0x0000c000
Erasing... ... done
Writing... ... done
Written 49152 bytes
Saved vivi private data
vivi>
```

part 命令

part 命令用于对MTD 分区进行操作。通过part help 可以显示系统对part系列命令的帮助提示。

```
vivi> part help
Usage:
part help
part add <name> <offset> <size> <flag> -- Add a mtd partition entry
part del <name> -- Delete a mtd partition entry
part reset -- Reset mtd parition table
part save -- Save mtd partition table
part show -- Display mtd partition table
vivi>
```

MTD 分区是针对Flash(NOR Flash 或者NAND Flash)的分区,以便于对bootloader对Flash 进行管理。

part add

part add 命令用于添加一个MTD 分区。
命令的详细格式如下：

```
part add name offset size flag
```

参数name 是要添加的分区的分区名称；
参数offset 是要添加的分区的偏移（相对于整个MTD 设备的起始地址的偏移，在Normandy 系统中不论配置的是NOR Flash，还是NAND Flash，都只注册了一个mtd_info 结构，也就是说逻辑上只有一个MTD 设备，这个MTD 设备的起始地址为0x00000000）；
参数size 是要添加的分区的大小，单位为字节；
参数flag 是要添加的分区的标志，参数flag 的取值只能为以下字符串（请注意必须为大写）或者通过连接符“|”将以下字符串组合起来的组合字符串。这个

标志表示了这个分区的用途
“BONFS” —— 作为BONFS 文件系统的分区；
“JFFS2” —— 作为JFFS2 文件系统的分区；
“LOCK” —— 该分区被锁定了；
“RAM” —— 该分区作为RAM 使用。
例如，添加新的MTD 分区mypart。

```
vivi> part add mypart 0x500000 0x100000 JFFS2
mypart: offset = 0x00500000, size = 0x00100000, flag = 8
vivi> part show
mtdpart info. (5 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
mypart : 0x00500000 0x00100000 8 1M
vivi> part add mypart2 0x2000000 0x100000 BONFS|RAM
mypart2: offset = 0x02000000, size = 0x00100000, flag = 20
vivi> part show
mtdpart info. (6 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
mypart : 0x00500000 0x00100000 8 1M
mypart2 : 0x02000000 0x00100000 20 1M
```

part del
part del 命令用于删除一个MTD 分区。
命令的详细格式如下：

```
part del name
```

参数name 是要删除的MTD 分区的分区名称。
例如，删除刚刚添加的分区mypart2

```
vivi> part show
mtdpart info. (6 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
mypart : 0x00500000 0x00100000 8 1M
```

```
mypart2 : 0x02000000 0x00100000 20 1M
vivi> part del mypart2
deleted 'mypart2' partition
vivi> part show
mtdpart info. (5 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
mypart : 0x00500000 0x00100000 8 1M
```

part reset

part reset 命令用于将MTD 分区表复位成系统默认的MTD 分区表。

Normandy 系统bootloader 默认的MTD 分区表为：

```
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
```

part reset 命令用法举例如下

```
vivi> part show
mtdpart info. (5 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
mypart : 0x00500000 0x00100000 8 1M
vivi> part reset
```

显示表示，分区表已经复位成系统默认的MTD 分区表。

```
vivi> part show
mtdpart info. (4 partitions)
name offset size flag
-----------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
```

part save

part save 命令用于将MTD 分区表保存到Flash（NOR Flash 或者NAND Flash）中。

part show

part show 命令用于显示系统当前的MTD 分区表信息。

```
vivi> part show
mtdpart info. (4 partitions)
name offset size flag
-------------------------------------------
vivi : 0x00000000 0x00020000 0 128k
param : 0x00020000 0x00010000 0 64k
kernel : 0x00030000 0x00170000 0 1M+448k
root : 0x00180000 0x02000000 4 32M
```

bon 命令

bon 命令用于对BON 分区进行操作。通过bon help 可以显示系统对bon系列命令的帮助提示。

```
vivi> bon help
Usage:
bon part info ---display bon part infomation
bon part <offsets> [<offsets>] [<offsets>] ..... ---Build whole the bo
n part
Note:
1. digit number is decimal
2. character beside the digit numbers:
'k' or 'K' - Kilo bytes
'm' or 'M' - Mega bytes
3. character beside the ':' :
'm' or 'M' - MTD PART
4. the ':' can be omitted. default mean BON PART
E.g.
vivi> bon part 0k 192k 1m 3m:m
Function: Build a bon filesystem include 4 parts, bon/3 is a MTD PART.
```

BON 分区是只针对NAND Flash 设备的一种简单的分区管理方式。Bootloader支持BON分区，同时Samsung提供的针对S3C2410移植的linux版本中也支持了BON 分区，这样就可以利用BON 分区来加载linux 的root根文件系统了。请注意区分MTD 分区和BON 分区，当Normandy 系统配置了NAND Flash 作为MTD 设备，那么MTD 分区和BON 分区都在同一片NAND Flash 上。不过它们相互没有什么影响。

bon part info

bon part info 命令用于显示系统中BON 分区的信息。

```
vivi> bon part info
BON info. (4 partitions)
Note about 'flags': 0 - BON PART , 1 - MTD PART
No: offset size flags number_of_badblock
-------------------------------------------------------------------
----------
0: 0x00000000 0x00030000 00000000 0 192k
1: 0x00030000 0x0014c000 00000000 1 1M+304k
```

2: 0x00180000 0x01e80000 00000000 0 30M+512k
3: 0x02000000 0x01ff8000 00000001 1 31M+992k

bon part

bon part 命令用于建立系统的BON 分区表。BON 分区表被保存到NAND Flash 的最后0x4000 个字节中，即在NAND Flash 的0x03FFC000 ～0x33FFFFFF 范围内，分区表起始于0x03FFC000（注意： BON 分区是只针对NAND Flash 设备的一种简单的分区管理方式）。

命令的详细格式如下：

bon part offsets1[flag] offsets2[flag] offsets3[flag] ……

参数offsetsN 是每个BON 分区的起始地址;flag 是跟每个BON 分区的起始地址后面的标识符这个标识的作用是前面数值的单位，‘k’’或‘K’表示kilo，千;‘m’或‘M’表示mega，兆。如果再跟上‘:’，后面再跟上‘m’或‘M’，表示该分区被标记为MTD 分区，如果没有‘:’以及后面的字母‘m’ 或‘M’，则表示该分区被标记为BON 分区。

bon part 命令在建立系统的BON 分区表，会检测每个分区是否有坏块（Samsung的NAND Flash 芯片K9S1208U0M，一个块含32 个页，一个页有512 个字节，一个块有16K 字节，即0x4000），如果发现坏块将标记出来，并且在分区表中体现，分区的大小将减去坏快的容量，得到实际可用的分区容量。bon part info 命令执行后显示的信息中， number_of_badblock所指示的就是分区中的坏块数目。

例如，用bon part 命令建立下列分区结构：

0x00000000～0x00030000 为BON 分区;

0x00030000～0x00180000 为BON 分区;

0x00180000～0x02000000 为BON 分区;

0x02000000～整个NAND Flash 结束位置处为MTD 分区。执行命令如下，建立0～192k(即0x00030000),192k～1536K(即0x00180000),1536K～32M(即0x02000000)和32M～整个NAND Flash结束位置处这4 个分区，：

vivi> bon part 0 192k 1536k 32m:m

建立好BON 分区后，查看一下，建立成功，但是bon/1 分区中有1 个坏块，bon/3 分区中有1 个坏块。

```
vivi> bon part info
BON info. (4 partitions)
Note about 'flags': 0 - BON PART , 1 - MTD PART
No: offset size flags number_of_badblock
------------------------------------------------------------
0: 0x00000000 0x00030000 00000000 0 192k
1: 0x00030000 0x0014c000 00000000 1 1M+304k
2: 0x00180000 0x01e80000 00000000 0 30M+512k
3: 0x02000000 0x01ff8000 00000001 1 31M+992k
vivi>
```

boot 命令

boot 命令用于引导linux kernel 启动。通过boot help 可以显示系统对boot 命令的帮助提示。这条命令的系统帮助写的很明白了。

```
vivi> boot help
Usage:
```

boot {ram|nor|smc} -- booting kernel
boot {ram|nor|smc} <mtd_part> -- boot from specific mtd partition
boot {ram|nor|smc} <addr> <size> -- boot from specific address
boot help -- help about 'boot' command
value of media_type (location of kernel image)
keyword type vlaue
ram RAM 1
nor NOR Flash Memory 2
smc SMC (On S3C2410) 3
命令的详细格式如下：
boot { ram | nor | smc } { mtd_part | addr size }
boot 关键字后面必须指定媒介类型,因为boot 命令对不同媒介的处理方式是不同的，例如如果kernel 在SDRAM 中，那么boot 执行的过程中就可以跳过拷贝kernel 映像到SDRAM 中这一步骤了。
Boot 命令识别的媒介类型有以下三种：
ram 表示从RAM（在Normandy 系统中即为SDRAM）中启动linux kernel，linux kernel 必须要放在RAM 中。
nor 表示从NOR Flash 中启动linux kernel，linux kernel 必须已经被烧写到了NOR Flash 中。
smc 表示从NAND Flash 中启动linux kernel，linux kernel 必须已经被烧写到了NAND Flash 中。
取值参数意义如下：
参数mtd_part 是MTD 分区的名字，选取这个参数表示从MTD 设备的一个分区中启动linux kernel，kernel 映像必须被放到这个分区中；addr 和size 分别表示linux kernel 起始地址和kernel 的大小。为什么要指定kernel 大小呢？因为kernel 首先要被copy 到boot_mem_base + 0x8000 的地方，然后在boot_mem_base + 0x100 开始的地方设置内核启动参数，要拷贝kernel，当然需要知道kernel 的大小啦，这个大小不一定非要和kernel实际大小一样，但是必须许大于等于kernel 的大小，单位字节。
例如，Normandy 系统配置了64M 的NAND Flash K9S1208U0M，在K9S1208U0M 上建立BON 分区，然后从NAND Flash 中引导linux。
(1) 建立BON 分区表如下。
0x00000000～0x00030000 为BON 分区；
0x00030000～0x00180000 为BON 分区；
0x00180000～0x02000000 为BON 分区；
0x02000000～整个NAND Flash 结束位置处为MTD 分区。
vivi> bon part 0 192k 1536k 32m:m
(2) 将kernel 映像文件zImage 通过load flash 命令烧写到0x30000 地址处。
vivi> load flash 0x30000 0x100000 x
(3) 将root.cramfs（一个cramfs 文件系统），通过load flash 命令烧写到0x180000 地址处。
vivi> load flash 0x180000 0x200000 x
(4) 完成了以上几步之后，开始通过boot 命令开始引导linux kernel 启动。
vivi> boot smc kernel

```
Copy linux kernel from 0x00030000 to 0x30008000, size = 0x00170000 ...
done
zImage magic = 0x016f2818
Setup linux parameters at 0x30000100
linux command line is: "noinitrd root=/dev/bon/2 init=/linuxrc
console=ttyS0"
MACH_TYPE = 193
NOW, Booting Linux......
Uncompressing Linux.....................................
done, booting
the kernel.
Linux version 2.4.18-rmk7-pxa1 (root@localhost.localdomain) (gcc
version2.95.2 20000516 (release) [Rebel.com]) #2 五 8 月 6 23:53:20 HKT
2004CPU: ARM/CIRRUS Arm920Tsid(wb) revision 0
```

go 命令

go 命令用于跳转到指定地址处执行该地址处的代码。如果指定的地址处不是有效代码，那么系统将在那里运行得“飞掉”。

命令得详细格式如下：

go addr a0 a1 a2 a3

参数addr 为跳转的目的地址；

a0 、a1 、a2 和 a3 分别为传递给指定地址处的程序的参数，通过CPU 的通用寄存器a0、a1、a2 和a3 来传递给指定地址处的程序。

go 命令和下面将要讲述的 call 命令的不同点是：call 命令是调用子程序的命令，不仅仅是跳到指定地址处执行当处的程序，而且考虑了从指定地址处的程序返回的功能；而go 命令只是直接跳到指定地址处执行程序，没有考虑到返回的问题

call 命令

call 命令用于跳转到指定地址处执行该地址处的代码，执行完成了还要返回。如果指定的地址处不是有效代码，那么系统将在那里运行得“飞掉”。

命令得详细格式如下：

call addr a0 a1 a2 a3

参数addr 为跳转的目的地址；

a0 、a1 、a2 和 a3 分别为传递给指定地址处的程序的参数，通过CPU 的通用寄存器a0、a1、a2 和a3 来传递给指定地址处的程序。

call 命令和上面已经讲述过的的go 命令的不同点是：call 命令是调用子程序的命令，不仅仅是跳到指定地址处执行当处的程序，而且考虑了从指定地址处的程序返回的功能；而go 命令只是直接跳到指定地址处执行程序，没有考虑到返回的问题。

dump 命令

dump 命令用于显示系统内存的内容，可以查看到NOR Flash 和SDRAM中的内容，但不能是NAND Flash 中的内容。

命令的详细格式如下：

dump addr length

参数addr 为要显示的内存起始地址；

参数length 为要显示的内存字节数。
例如，显示从0x0 地址处（NOR Flash 的开头）开始的256 个字节的数据（这些就是bootloader 开头的256 字节）。

```
vivi> dump 0x0 0x100
00000000: 00 00 00 00 CB 00 00 EA-D0 00 00 EA D5 00 00 EA | ................
00000010: DA 00 00 EA EB 00 00 EA-DE 00 00 EA E3 00 00 EA
| ................
00000020: 00 00 00 00 00 00 00 00-00 00 00 00 C1 00 06 01 | ................
00000030: 00 00 00 00 49 00 00 EA-58 00 00 EA 53 14 A0 E3 | ....I...X...S...
00000040: 00 20 A0 E3 00 20 81 E5-4A 14 A0 E3 00 20 E0 E3 | . ...
```

例如，显示从0x30000000 地址处（SDRAM 的开头）开始的512 个字节的数据。

```
vivi> dump 0x30000000 0x200
30000000: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 | ................
30000010: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 | ................
30000020: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 | ................
```

test 命令

test 命令是Normandy 系统简单的测试命令。通过test help 可以显示系统对test 系列命令的帮助提示。

```
vivi> test help
Usage:
test sleep -- Test sleep mode. (pwbt is eint0)
test int -- Test external interrupt 0
test led -- Test LEDs
test btn -- Test btn
```

1.15.1 test sleep

test sleep 命令用于系统sleep 模式的测试。

test led

test led 命令用于利用GPIO 口驱动LED 的测试。

```
vivi> test led
```

运行test led 命令后，板子上两个灯会交互地闪动。敲入回车后，测试结束，灯结束闪动。

md5sum 命令

md5sum 命令用于对指定地址处的指定长度的字节数据进行MD5 算法校验，计算出16 个字节的校验和。
命令的详细格式如下：

```
md5sum addr size | downfile size
```

参数addr 表示要进行MD5 校验和计算的原始数据的起始地址；关键字参数downfile 表示SDRAM 在整个系统地址空间中的起始地址（即0x30000000）；参数size 表示进行MD5 校验和计算的原始数据的长度，单位字节；例如，计算从0x0 地址处开始的4096 个字节数据的MD5 校验和。

```
vivi> md5sum 0x0 0x1000
address = 0x0, size = 4096 bytes
MD5SUM: d1d6a020dab84c8b43f960d8473a920a
```

例如，计算从0x30000000 地址处开始的4096 个字节数据的MD5 校验和。

```
vivi> md5sum downfile 0x100000
address = 0x30000000, size = 1048576 bytes
```