【1】Uboot 启动流程：

分析版本：uboot-2013

入口查看 u-boot.lds

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x00000000;
 . = ALIGN(4);
 .text :
 {
  __image_copy_start = .;
  arch/arm/cpu/armv7/start.o (.text*)
  *(.text*)
 }
```

通过链接脚本可知入口为_start，位于 arch/arm/cpu/armv7/start.o。

第一阶段开始：

进入 arch/arm/cpu/armv7/start.S

```
.globl _start
_start: b    reset
ldr pc, _undefined_instruction
ldr pc, _software_interrupt
ldr pc, _prefetch_abort
ldr pc, _data_abort
ldr pc, _not_used
ldr pc, _irq
ldr pc, _fiq
```

异常向量表设置

```
reset:
bl  save_boot_params
/*
 * set the cpu to SVC32 mode
 */
mrs r0, cpsr
bic r0, r0, #0x1f
orr r0, r0, #0xd3
msr cpsr,r0
```

设置cpu处于svc工作模式

```
/* Set vector address in CP15 VBAR register */
ldr r0, =_start
mcr p15, 0, r0, c12, c0, 0  @Set VBAR
#endif
```

协处理器p15的c12寄存器来重新定位

Bl cpu_init_cp15（使分支预测无效，数据）

```
ENTRY(cpu_init_cp15)
    /*
     * Invalidate L1 I/D
     */
    mov r0, #0          @ set up for MCR
    mcr p15, 0, r0, c8, c7, 0   @ invalidate TLBs
    mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
    mcr p15, 0, r0, c7, c5, 6   @ invalidate BP array
    mcr     p15, 0, r0, c7, c10, 4  @ DSB
    mcr     p15, 0, r0, c7, c5, 4   @ ISB
```

关闭数据预取功能

DSB:多核cpu对数据处理指令

流水线清空指令

```
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
    orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
    orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
    bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
#else
    orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
#endif
    mcr p15, 0, r0, c1, c0, 0
    mov pc, lr          @ back to my caller
ENDPROC(cpu_init_cp15)
```

关闭MMU，使能I-cache

NOTE:

分支预测：在流水线里，会将后面的代码优先加载到处理器中，由于是循环，会使后面加载的代码无效，故出现了分支预测技术。(统计跳的次数来选择装载循环的代码还是下面的代码)。

Bl cpu_init_crit

```
ENTRY(cpu_init_crit)
    /*
     * Jump to board specific initialization...
     * The Mask ROM will have already initialized
     * basic memory. Go here to bump up clock rate and handle
     * wake up conditions.
     */
    b   lowlevel_init       @ go setup pll,mux,memory
ENDPROC(cpu_init_crit)
```

跳到 Low_level_init,位于 board/samsung/fs4412/lowlevel_init.S

```
    ldr r0, =0x1002040c
    ldr r1, =0x00
    str r1, [r0]
```

关闭看门狗

```
ldr r0, =0x0ffffff      /* r0 <- Mask Bits*/
bic r1, pc, r0      /* pc <- current addr of code */
/* r1 <- unmasked bits of pc */
ldr r2, _TEXT_BASE      /* r2 <- original base addr in ram */
bic r2, r2, r0      /* r2 <- unmasked bits of r2*/
cmp r1, r2          /* compare r1, r2 */
beq 1f          /* r0 == r1 then skip sdram init */

/* init system clock */
bl system_clock_init

/* Memory initialize */
bl mem_ctrl_asm_init

1:
/* for UART */
bl uart_asm_init
/*bl tzpc_init*/
pop {pc}
```

比较当前pc指针与TEXT_BASE的高8位是否一样来判断，当前代码是否在内存中

对系统时钟初始化

对内存初始化

对串口初始化

返回start.S

第一阶段结束，总结如下：

1 前面总结过的部分，初始化异常向量表，设置 svc 模式

2 配置 cp15，初始化 mmu cache tlb

3 板级初始化，clk,memory,uart 初始化

第二阶段开始：

Bl _main ,跳转到 arch/arm/lib/crt0.S

初始 c 运行环境

```
#if defined(CONFIG_NAND_SPL)
	/* deprecated, use instead CONFIG_SPL_BUILD */
	ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
#elif defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
	ldr sp, =(CONFIG_SPL_STACK)
#else
	ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
	bic sp, sp, #7	/* 8-byte alignment for ABI compliance */
	sub sp, #GD_SIZE	/* allocate one GD above SP */
	bic sp, sp, #7	/* 8-byte alignment for ABI compliance */
	mov r8, sp	/* GD is above SP */
	mov r0, #0
	bl board_init_f
```

初始化sp,为支行c语言做准备

8位对齐

保留128B放GD结构体，存放全局信息，GD的地址存放在r8中

跳转到 arch/arm/lib/board.c

```
void board_init_f(ulong bootflag)
{
	bd_t *bd;
	init_fnc_t **init_fnc_ptr;
	gd_t *id;
	ulong addr, addr_sp;
#ifdef CONFIG_PRAM
	ulong reg;
#endif
	void *new_fdt = NULL;
	size_t fdt_size = 0;

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

		for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
#if			if ((*init_fnc_ptr)() != 0) {
				hang ();
			}
#el		}
	/* FDT is at end of image */
	gd->fdt_blob = (void *)(_end_ofs + _TEXT_BASE);
#endif
	/* Allow the early environment to override the fdt address */
	gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
						(uintptr_t)gd->fdt_blob);
```

对全局信息GD结构体进行填充，mon_len通过链接脚本可以知道放的是uboot代码大小。

循环执行init_fnc_t数组中的函数，作硬件初始化

结构如下：

```
init_fnc_t *init_sequence[] = {
	arch_cpu_init,		/* basic arch cpu dependent setup */
	mark_bootstage,
#ifdef CONFIG_OF_CONTROL
	fdtdec_check_fdt,
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
	board_early_init_f,
#endif
	timer_init,		/* initialize timer */
#ifdef CONFIG_BOARD_POSTCLK_INIT
	board_postclk_init,
#endif
#ifdef CONFIG_FSL_ESDHC
	get_clocks,
#endif
	env_init,		/* initialize environment */
	init_baudrate,		/* initialze baudrate settings */
	serial_init,		/* serial communications setup */
	console_init_f,		/* stage 1 init of console */
	display_banner,		/* say that we are here */
#if defined(CONFIG_DISPLAY_CPUINFO)
	print_cpuinfo,		/* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
	checkboard,		/* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
	init_func_i2c,
#endif
	dram_init,		/* configure available RAM banks */
	NULL,
};
```

初始化各硬件

Dram_init 初始化成功之后，剩余代码将会对 sdram 空间进行规划。

```
	addr = CONFIG_SYS_SDRAM_BASE + gd->ram_size;
```

addr 的值由 CONFIG_SYS_SDRAM_BASE 加上 ram_size。也就是到了可用 sdram 的顶端。

```
#if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF))
	/* reserve TLB table */
	gd->tlb_size = 4096 * 4;
	addr -= gd->tlb_size;

	/* round down to next 64 kB limit */
	addr &= ~(0x10000 - 1);

	gd->tlb_addr = addr;
	debug("TLB table from %08lx to %08lx\n", addr, addr + gd->tlb_size);
#endif

	/* round down to next 4 kB limit */
	addr &= ~(4096 - 1);
	debug("Top of RAM usable for U-Boot at: %08lx\n", addr);
```

如果icache与dcache是打开的，就留出64K的空间作为tlb空间，最后addr就是tlb的地址，4K对齐

继续对 gd 结构体填充

```
	memcpy(id, (void *)gd, sizeof(gd_t));
```

填充完成将信息拷贝到内存指定位置。

继续回到 main

```
ldr sp, [r8, #GD_START_ADDR_SP] /* r8 = gd->start_addr_sp */
bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
ldr r8, [r8, #GD_BD]        /* r8 = gd->bd */
sub r8, r8, #GD_SIZE        /* new GD is below bd */
```

将r8指向新的gd地址

```
adr lr, here
ldr r0, [r8, #GD_RELOC_OFF]     /* lr = gd->start_addr_sp */
add lr, lr, r0
ldr r0, [r8, #GD_START_ADDR_SP] /* r0 = gd->start_addr_sp */
mov r1, r8              /* r1 = gd */
ldr r2, [r8, #GD_RELOCADDR]     /* r2 = gd->relocaddr */
b   relocate_code
ere:
```

代码重定位

对 lr 的操作为了让返回时，返回的是重定位的 here 处。

代码自搬移，防止与内核冲突，代码位于 arch/arm/cpu/armv7/start.S

```
ENTRY(relocate_code)
    mov r4, r0  /* save addr_sp */
    mov r5, r1  /* save addr of gd */
    mov r6, r2  /* save addr of destination */

    adr r0, _start  /* ldr r0, =_start*/
    cmp r0, r6
    moveq   r9, #0      /* no relocation. relocation offset(r9) = 0 */
    beq relocate_done       /* skip relocation */
    mov r1, r6          /* r1 <- scratch for copy_loop */
    ldr r3, _image_copy_end_ofs
    add r2, r0, r3      /* r2 <- source end address     */

    copy_loop:
    ldmia   r0!, {r9-r10}       /* copy from source address [r0]    */
    stmia   r1!, {r9-r10}       /* copy to   target address [r1]    */
    cmp r0, r2          /* until source end address [r2]    */
    blo copy_loop
```

循环将代码搬移到指定高地址

这里只是将链接脚本中_image_copy_end 到_start 中的代码，其它段还没有操作。

在这里我们有疑惑就是将代码重定位到高地址，那运行的地址不就和链接地址不一样了，那运行可能不正常？这个疑惑就是.rel.dyn 帮我们解决了，主要还是编译器帮我们做的工作，在链接中有如下：【参考：http://blog.csdn.net/skyflying2012/article/details/37660265】

```
ifndef CONFIG_NAND_SPL
LDFLAGS_u-boot += -pie
endif
```

重

定位到高地址之后，再次回到 main

```
here:

/* Set up final (full) environment */

    bl  c_runtime_cpu_setup /* we still call old routine here */

    ldr r0, =__bss_start    /* this is auto-relocated! */
    ldr r1, =__bss_end__    /* this is auto-relocated! */

    mov r2, #0x00000000     /* prepare zero to clear BSS */

clbss_l:cmp r0, r1          /* while not at end of BSS */
    strlo   r2, [r0]        /* clear 32-bit BSS word */
    addlo   r0, r0, #4      /* move to next */
    blo clbss_l
```

关icache,保证数据从sdram中更新，更新异常向量表，因为代码被重定位了

清BSs

```
    /* call board_init_r(gd_t *id, ulong dest_addr) */
    mov r0, r8              /* gd_t */
    ldr r1, [r8, #GD_RELOCADDR] /* dest_addr */
    /* call board_init_r */
    ldr pc, =board_init_r   /* this is auto-relocated! */
```

R0=gd R1=RELOCADDR

调用board_init_r

调用 board_init_r 主要是对外设的初始化。

```
    for (;;) {
        main_loop();
    }
```

进入超循环

Main_loop 函数主要功能是处理环境变量，解析命令

install_auto_complete(); //安装自动补全的函数，分析如下 。

getenv(bootcmd)

bootdelay(自启动)

```
    if (bootdelay != -1 && s && !abortboot(bootdelay)) {
```

如果延时大于等于零，并且没有在延时过程中接收到按键，则引导内核。abortboot 函数的分析见下面

```
        rc = run_command(lastcommand, flag);
```

处理用户输入命令

uboot 启动流程分析如下：

第一阶段：

设置 cpu 工作模式为 SVC 模式

关闭中断，mmu,cache

关看门狗

初始化内存，串口

设置栈

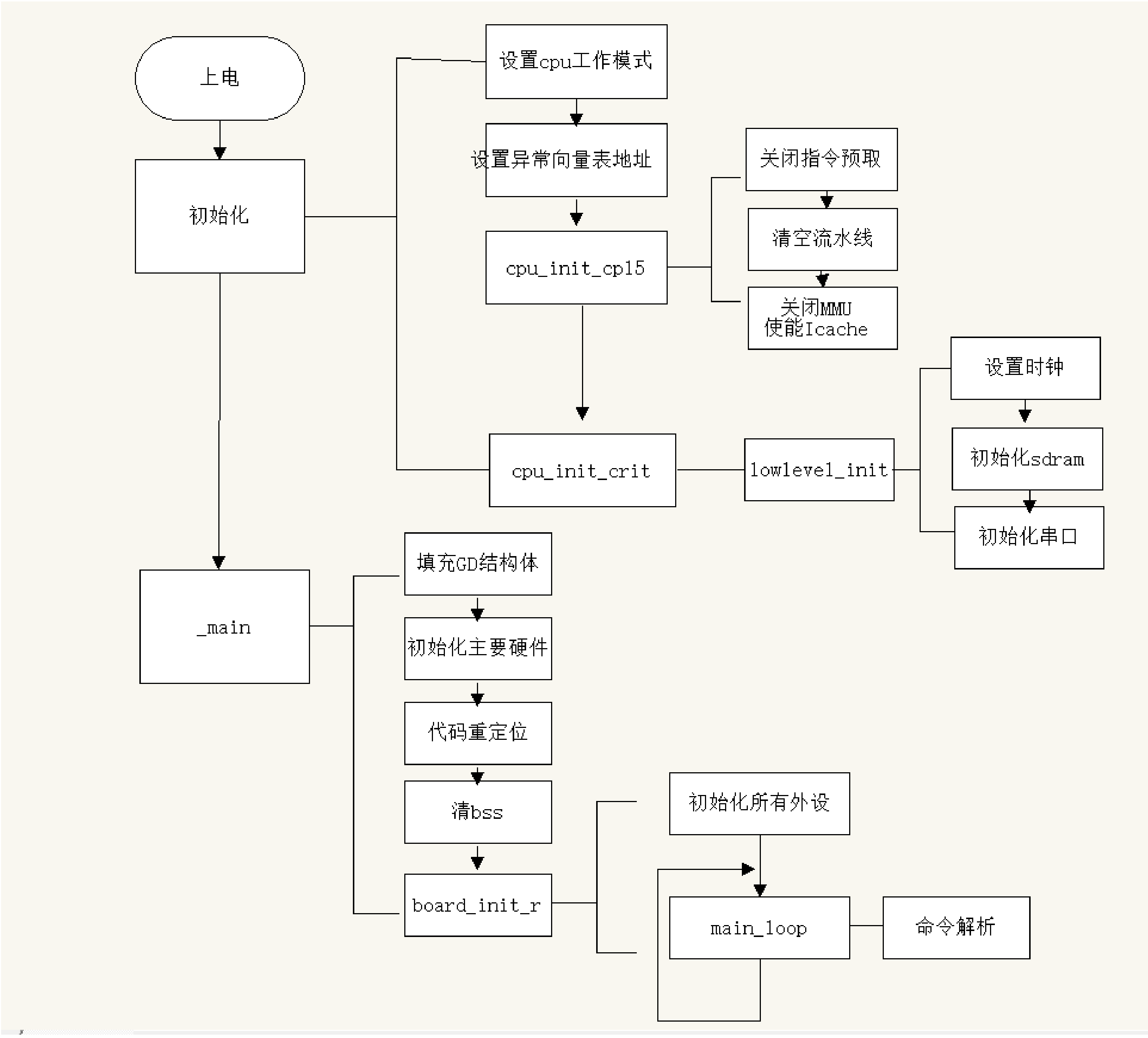
代码自搬移

清 bss

跳 c

第二阶段
初始化外设，进入超循环
超循环处理用户命令

针对 uboot2013 启动流程图如下：




下面是最精简的自己实现 uboot 的代码形式【代码见附件】

```
.
├── arch
│   └── arm
│       └── arm_cortexa8
│           ├── map.lds
│           ├── start.o
│           └── start.S
├── board
│   └── samsung
│       ├── low_levelinit.o
│       ├── low_levelinit.S
│       ├── mem_setup.o
│       ├── mem_setup.S
│       ├── nand.c
│       ├── nand.o
│       └── test.c
├── common
│   ├── do_go.c
│   ├── do_go.o
│   ├── main.c
│   └── main.o
├── driver
│   ├── string.c
│   ├── string.o
│   ├── uart.c
│   ├── uart.h
│   └── uart.o
├── include
│   ├── s5pc100.h
│   └── setup.h
├── lib_arm
│   ├── board.c
│   └── board.o
├── Makefile
├── mini_uboot.bin
├── mini_uboot.dis
├── mini_uboot.elf
```

针对a8开发板自己编写uboot的代码结构

【2】命令解析过程 run_command：

```
int run_command(const char *cmd, int flag)
{
#ifndef CONFIG_SYS_HUSH_PARSER
    /*
     * builtin_run_command can return 0 or 1 for success, so clean up
     * its result.
     */
    if (builtin_run_command(cmd, flag) == -1)
        return 1;

    return 0;
#else
    return parse_string_outer(cmd,
            FLAG_PARSE_SEMICOLON | FLAG_EXIT_FROM_LOOP);
#endif
}
```

进入 builtin_run_command,分析 cmd_process

```
if (cmd_process(flag, argc, argv, &repeatable))
```

```
cmdtp = find_cmd(argv[0]);
if (cmdtp == NULL) {
    printf("Unknown command '%s' - try 'help'\n", argv[0]);   查找命令
    return 1;
}
```

```
/* If OK so far, then do the command */
if (!rc) {
    rc = cmd_call(cmdtp, flag, argc, argv);  真正执行命令
    *repeatable &= cmdtp->repeatable;
}
```

【3】Uboot 添加自定义命令：uboot 中的命令使用 U_BOOT_CMD 这个宏声明来注册进系统，链接脚本会把所有的 cmd_tbl_t 结构体放在相邻的地方。

```
U_BOOT_CMD(
    help,   CONFIG_SYS_MAXARGS, 1,  do_help,    这个宏将信息添加到
    "print command description/usage",          .u_boot_list.cmd段
    "\n"
    "   - print brief description of all commands\n"
    "help command ...\n"
    "   - print detailed usage of 'command'"
);
```

uboot 命令结构体如下：

```
struct cmd_tbl_s {
    char        *name;      /* Command Name         */
    int     maxargs;    /* maximum number of arguments  */
    int     repeatable; /* autorepeat allowed?      */
                    /* Implementation function  */
    int     (*cmd)(struct cmd_tbl_s *, int, int, char * const []);
    char        *usage;     /* Usage message    (short) */
#ifdef  CONFIG_SYS_LONGHELP
    char        *help;      /* Help  message    (long)  */
#endif
#ifdef CONFIG_AUTO_COMPLETE
    /* do auto completion on the arguments */
    int     (*complete)(int argc, char * const argv[], char last_char, int maxv, char *cmdv[]);
#endif
};
```

通过链接脚本查看如下段

```
_u_boot_list_cmd__start = .;
*(SORT(.u_boot_list.cmd.*));
_u_boot_list_cmd__end = .;
```

```
#define U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,      \
                _usage, _help, _comp)                      \
        { #_name, _maxargs, _rep, _cmd, _usage,             \
            _CMD_HELP(_help) _CMD_COMPLETE(_comp) }

#define U_BOOT_CMD_MKENT(_name, _maxargs, _rep, _cmd, _usage, _help)    U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep

#define U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, _comp) \
    ll_entry_declare(cmd_tbl_t, _name, cmd, cmd) =              \
        U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,  \
                _usage, _help, _comp);

#define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help)      \
    U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, NULL)
```

再次跟踪

```
#define ll_entry_declare(_type, _name, _section_u, _section_d)      \
    _type _u_boot_list_##_section_u##_##_name __attribute__((    \
            unused, aligned(4),                    \
            section(".u_boot_list."#_section_d"."#_name)))
```

指定相关代码指定段

【4】bootm 分析：

```
U_BOOT_CMD(
    bootm,  CONFIG_SYS_MAXARGS, 1,  do_bootm,
    "boot application image from memory", bootm_help_text
);
```

位于函数 do_bootm

```
if (bootm_start(cmdtp, flag, argc, argv))
    return 1;
```

执行bootm

位于函数 bootm_start

```
/* get kernel image header, start address and length */
os_hdr = boot_get_kernel(cmdtp, flag, argc, argv,
        &images, &images.os.image_start, &images.os.image_len);
if (images.os.image_len == 0) {
    puts("ERROR: can't get kernel image!\n");
    return 1;
}

/* get image parameters */
switch (genimg_get_format(os_hdr)) {
case IMAGE_FORMAT_LEGACY:
    images.os.type = image_get_type(os_hdr);
    images.os.comp = image_get_comp(os_hdr);
    images.os.os = image_get_os(os_hdr);
```

填充内核相关信息

回到 do_bootm

```
static boot_os_fn *boot_os[] = {
#ifdef CONFIG_BOOTM_LINUX
	[IH_OS_LINUX] = do_bootm_linux,
#endif
#ifdef CONFIG_BOOTM_NETBSD
	[IH_OS_NETBSD] = do_bootm_netbsd,
#endif
#ifdef CONFIG_LYNXKDI
	[IH_OS_LYNXOS] = do_bootm_lynxkdi,
#endif
#ifdef CONFIG_BOOTM_RTEMS
	[IH_OS_RTEMS] = do_bootm_rtems,
#endif
#if defined(CONFIG_BOOTM_OSE)
	[IH_OS_OSE] = do_bootm_ose,
#endif
```

boot_os 函数指针数组，从中可以知道，上面的执行是do_boo

位于 do_bootm_linux

```
	boot_jump_linux(images);
```

位于 boot_jump_linux

```
/* Subcommand: GO */
static void boot_jump_linux(bootm_headers_t *images)
{
	unsigned long machid = gd->bd->bi_arch_number;
	char *s;
	void (*kernel_entry)(int zero, int arch, uint params);
	unsigned long r2;

	kernel_entry = (void (*)(int, int, uint))images->ep;

	s = getenv("machid");
	if (s) {
		strict_strtoul(s, 16, &machid);
		printf("Using machid 0x%lx from environment\n", machid);
	}

	debug("## Transferring control to Linux (at address %08lx)" \
		"...\n", (ulong) kernel_entry);
#endif
	bootstage_mark(BOOTSTAGE_ID_RUN_OS);
	announce_and_cleanup();

#ifdef CONFIG_OF_LIBFDT
```

【5】u-boot 编译流程分析说下：(也可以从 build.sh 分析)

Make fs4412_config

```
%_config::  unconfig
    @$(MKCONFIG) -A $(@:_config=)
```

替换目标名fs4412_config 为 fs

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

分析 mkconfig 脚本

解析 boards.cfg fs4412 相关数据

```
fs4412                arm       armv7         fs4412        samsung     exynos
```

作链接

```
ln -s ../arch/${arch}/include/asm asm
ln -s ${LNPREFIX}arch-${cpu} asm/arch
ln -s ${LNPREFIX}proc-armv asm/proc
```

针对平台作了一系列链接

导出信息到 config.mk

```
( echo "ARCH   = ${arch}"
    if [ ! -z "$spl_cpu" ] ; then
    echo 'ifeq ($(CONFIG_SPL_BUILD),y)'
    echo "CPU     = ${spl_cpu}"
    echo "else"
    echo "CPU     = ${cpu}"
    echo "endif"
    else
    echo "CPU     = ${cpu}"
    fi
    echo "BOARD   = ${board}"

    [ "${vendor}" ] && echo "VENDOR = ${vendor}"
    [ "${soc}"    ] && echo "SOC    = ${soc}"
    exit 0 ) > config.mk
```

Include/config.mk 内容如下：

```
ARCH    = arm
CPU     = armv7
BOARD   = fs4412
VENDOR = samsung
SOC     = exynos
```

导出平台数据到 config.h

```
echo "#define CONFIG_SYS_ARCH  \"${arch}\""  >> config.h
echo "#define CONFIG_SYS_CPU   \"${cpu}\""   >> config.h
echo "#define CONFIG_SYS_BOARD \"${board}\"" >> config.h

[ "${vendor}" ] && echo "#define CONFIG_SYS_VENDOR \"${vendor}\"" >> config.h
```

Include/config.h 导出结果如下：

```
/* Automatically generated - do not edit */
#define CONFIG_SYS_ARCH  "arm"
#define CONFIG_SYS_CPU   "armv7"
#define CONFIG_SYS_BOARD "fs4412"
#define CONFIG_SYS_VENDOR "samsung"
#define CONFIG_SYS_SOC    "exynos"
#define CONFIG_BOARDDIR board/samsung/fs4412
#include <config_cmd_defaults.h>
#include <config_defaults.h>
#include <configs/fs4412.h>
#include <asm/config.h>
```

Make -jxx 执行过程如下：

找第一个目标 all:

```
all:        $(ALL-y) $(SUBDIR_EXAMPLES)
```

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

```
$(obj)u-boot.bin:   $(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
        $(BOARD_SIZE_CHECK)
        @+make -C sdfuse_q/
        @./sdfuse_q/add_padding
```

上面代码是对 u-boot 进行格式转换，变成二进制 bin 格式之后，再加一些校验与 4412 开如平台加密信息。

依赖 u-boot:（573 行左右）

```
$(obj)u-boot:   depend \
        $(SUBDIR_TOOLS) $(OBJS) $(LIBBOARD) $(LIBS) $(LDSCRIPT) $(obj)u-boot.lds
        $(GEN_UBOOT)
ifeq ($(CONFIG_KALLSYMS),y)
        smap=`$(call SYSTEM_MAP,$(obj)u-boot) | \
            awk '$$2 ~ /[tTwW]/ {printf $$1 $$3 "\\\\000"}'` ; \
        $(CC) $(CFLAGS) -DSYSTEM_MAP="\"$${smap}\"" \
            -c common/system_map.c -o $(obj)common/system_map.o
```

编译相关文件

【6】扩展题目练习

给学生留一个课后扩展练习题，如何通过现有 u-boot 将代码加载并烧录到外存，如何调整当前 emmc 分区大小，应该看哪块代码？