

U-Boot 的启动流程

课程名称：嵌入式硬件设计与实践

学号：1170400423

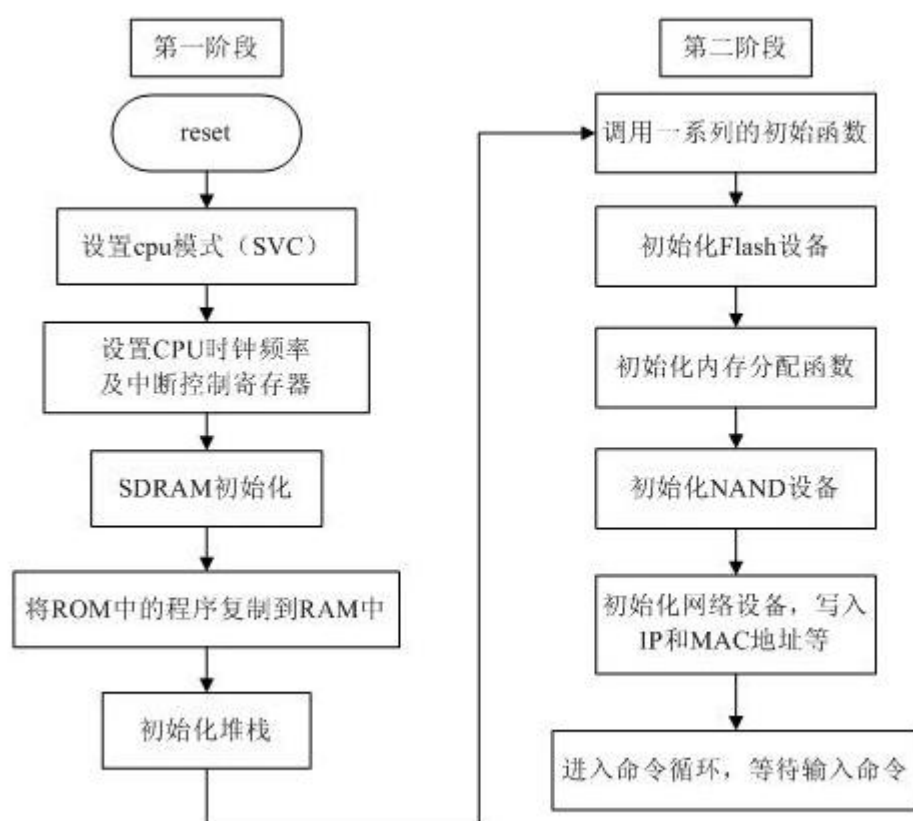
姓名：尉前进

日期：2020.6.22

U-boot 启动流程介绍

一、启动流程概述

U-Boot 启动内核的过程可以分为两个阶段，阶段一由汇编语言编写，阶段二由 C 语言编写，启动流程如下



二、实现代码

1. 第一阶段汇编代码

- 1) 硬件设备初始化将 CPU 的工作模式设为管理模式(svc),关闭 MMU、CACHE,设置外设寄存器地址。
- 2)为第二阶段代码准备 RAM 空间所谓准备 RAM 空间,初始化内存芯片,使它可用。通过在 start.S 中调用 lowlevel init 函数来设置存储控制器,使得外接的 SDRAM 可用,并且关闭 WATCHDOG,设置 FCLK、HCLK、PCLK 的比例(即设置 CLKDIVN 寄存器)。

1. 中断向量表的设置:

.globl _start /*声明一个符号可以被其他文件引用，相当于声明了一个全局变量，.globl 与.global 相同*/

_start:

```

b start_code                /* 复位 */
ldr pc, _undefined_instruction /* 未定义指令向量 */
ldr pc, _software_interrupt  /* 软件中断向量 */
ldr pc, _prefetch_abort      /* 预取指令异常向量 */
ldr pc, _data_abort          /* 数据操作异常向量 */
ldr pc, _not_used            /* 未使用 */
ldr pc, _irq                 /* irq 中断向量 */
ldr pc, _fiq                 /* fiq 中断向量 */
                             /* 中断向量表入口地址 */

```

注： Start.s 文件一开始，就定义了 `_start` 的全局变量。也即，在别的文件，照样能引用这个 `_start` 变量。这段代码验证了我们之前学过的 arm 体系的理论知识：中断向量表放在从 0x0 开始的地方。其中，每个异常中断的摆放次序，是事先规定的。比如第一个必须是 `reset` 异常，第二个必须是未定义的指令异常等等。

需要注意的是，在这里，我们也可以理解：为何系统一上电，会自动运行代码。因为系统上电后，会从 0x0 地方取指令，而 0x0 处放置的是 `reset` 标签，直接就跳去 `reset` 标签处去启动系统了。

另外，这里使用了 `ldr` 指令。而 `ldr` 指令中的 `label`，分别用一个 `.word` 伪操作来定义。比如：

```
_undefined_instruction: .word undefined_instruction
```

我们用 source insight 跟踪代码后，发现，`undefined_instruction` 在 `start.s` 的后面给出了具体的操作，如下：

```

undefined_instruction:
get_bad_stack
bad_save_user_regs
bl do_undefined_instruction

```

在跳转到中断服务子程序之前，先有两个宏代码，一个是对 `stack` 的操作，一个是用户 `regs` 的保存。然后才能跳转如中断服务子程序中执行。

```

reset:
/** set the cpu to SVC32 mode */      /* CPU 进入 SVC （管理）模式 */
mrs r0,cpsr
r0,r0,#0x1f
bic r0,r0,#01f
msr cpsr,r0
cpu_init_crit:
/* flush v4 I/D caches*/              /* CPU 数据初始化，数据缓冲*/
mov r0,#(
/*flush v3/v4 cache */
mcr p15,0,r0,c7,c7,0
/* flush v4 TLB*/
mcr p15,0,r0,c8,c7,0

```

注： 关闭数据预取功能；DSB：多核 CPU 对数据处理指令；ISB：流水线清空指令；关闭 MMU，使能 I-cache。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。

```
/** disable MMU stuff and caches */    /*通过 p15 关闭 mmu*/
```

```
mrC p15,0,r0,c1,c0,0
bic r0,r0,#0x00002300
bic r0,r0,#0x00000087
orr r0,r0,#0x00000002
orr r0,r0,#0x00001000
mcr p15,0,r0,c1,c0,0
/* Peri port setup*/                /* 外设内存空间*/
ldr r0,#0x70000000
orr r0,r0,#0x13
mcrp15,0,r0,c15,c2,4
@ 256M(0x7000 0000-0x7fff ffff)      /*指明地址*/
/* go setup pll,mux,memory */
bl lowlevel init
```

1) 上电后 CPU 为 SVC 模式

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

注：CPU 复位后，系统会立即被设置成 SVC 模式。记得之前有网友发帖咨询这个问题，问系统复位后，cpu 处于哪个处理器模式。这个代码，就回答了这个问题。从这个代码中，我们也可以得到一个对寄存器操作的经验：读—修改--写。这里先把 cpsr 的值读到 r0 中，清除掉我们想修改的 bit 位，然后用 orr 指令来保证其他 bit 位不被改动，并达到修改寄存器低 5 位值的目的。最后用 msr 指令把 r0 的值给 cpsr 寄存器达到我们的修改目的。

2) cpu_init_crit:

```
/* flush v4 I/D caches */
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
bic r0, r0, #0x00002300
bic r0, r0, #0x00000087
orr r0, r0, #0x00000002
orr r0, r0, #0x00001000
mcr p15, 0, r0, c1, c0, 0
/* 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. */
```

```

mov ip, lr
bl lowlevel_init
mov lr, ip
mov pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */

```

无效掉了指令 cache 和数据 cache, 并禁止 MMU 与 cache。为什么会有这一步呢？这里无效 cache 和 MMU 肯定的原因：在初始化阶段，可以认为我们只有一个任务在跑，没有必要，也不允许使用地址变换。因此最好应该无效掉 MMU。由于在 `cpu_init_cri` 子程序中又一次调用子程序 `lowlevel_init`，因此，需要事先保护好 `lr` 寄存器的内容。当返回时候，再恢复它。在进入 `lowlevel_init` 之前，有必要详细说一下 `mov ip, lr`，这个语句的 `ip`。为了使单独编译的 C 语言程序和汇编程序之间能相互调用，必须为子程序间的调用规定一定的规则。这就是 ATPCS 规则。它规定了一些子程序间调用的基本规则。在寄存器的使用规则里，寄存器 `R12` 作用子程序间的 `scratch` 寄存器，记做 `ip`。`mov ip, lr` 语句的 `ip` 由此而来。

```

.globl lowlevel_init
lowlevel_init: mov r12, lr
/* LED on only #8*/ /* LED 上电*/
ldrr0, =ELFIN_GPIO_BASE ldr r1, =0x55540000
str r1, [r0, #GPNCON_OFFSET]
/* Disable Watchdog*/
ldrr0, =0x7e000000
@0x7e004000 /* 关闭看门狗*/
orr r0, r0, #0x4000
mov r1, #0
str r1, [r0]
/* 看门狗控制器的最低位为 0 时，看门狗不输出复位信号 */

@ External interrupt pending clear
ldr r0,
=(ELFIN_GPIO_BASE+EINTPEND_OFFSET)
/*EINTPEND*/
r r1, [r0]
str r1, [r0] /* 关闭所有中断*/
@ Disable all interrupts(VIc0 and VIc1)
mov r3, #0x0
str r3, [r0, #oINTMSK]
str r3, [r1, #oINTMSK] /* 将所有中断设置为 IRQ*/
@ Set all interrupts as IRQ
mov r3, #0x0
str r3, [r0, #oINTMOD]
str r3, [r1, #oINTMOD]
@ Pending Interrupt clear /*清除所有中断*/

```

```

/* init system clock */
bl system_clock_init
/* for UART*/
bl uart_asm_init
bl mem_ctrl_asm_init
if 1
ldr r0
=(ELFIN CLOCK POWER BASE+RST STAT OFFSET) E1,[r0]
ldr
r1,r1,#0xffffffff7
bic
cmp r1,#0x8
beq wakeup_reset

/* 初始化内存*/
/* 初始化时钟 dll*/

/* 唤醒内存*/

ldr r0,=0xff000ff
/* r0 <- current base addr of code */
bic r1,pc,r0
/* r1<- original base addr in ram */
/* 判断代码是否位于 RAM*/
ldr r2,_TEXT_PHY_BASE
/* r0<- current base addr of code */
/*如果是则跳过拷贝*/
bicr2,r2,r0
cmpr1,r2 /*compare r0,r1 */
/*_start 等于_TEXT_BASE 说明是下
载到 RAM 中运行 */

/* r0 == r1 then skip flash copy
moveq r3,#0xf
bl load_bl2_irom
movinand:
Cmp r3, #0xf
beq after_copy
bl movi_bl2_copy
drer_copy

#ifdef CONFIG_ENABLE_MMU
enable mmu:
/*开启 MMU*/
/* enable domain access */mcrp15,0,r5,c3,c0,0
@ load domain access register
/* Set the TTB register */mcrp15,0,r1,c2,c0,0
/* Enable the MMU */
mmu_on:
mrc p15,0,r0,c1,c0,0
orr r0,r0,#1
/* 如果定义了宏 CONFIG_ENABLE_MMU,
开启 MMU*/

/* Set CR M to enable MMU*/
mcr p15,0,r0,c1,c0,0

```

注：设置存储控制器，使得外接的 SDRAM 可用，并且关闭 WATCHDOG，设置 FCLK、HCLK、PCLK 的比例（即设置 CLKDIVN 寄存器）。

```

/*设置堆栈 */
/*开始转入 C 程序*/
stack_setup:
ldr r0, _TEXT_BASE
/* upper 128 KiB: relocated uboot */
sub r0, r0, #CONFIG_SYS_MALLOC_LEN /* malloc area */
sub r0, r0, #CONFIG_SYS_GBL_DATA_SIZE /* 跳过全局数据区*/
#ifdef CONFIG_USE_IRQ
sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
sub sp, r0, #12

BSS 段的清零
clear_bss: /* 清除 bss 段*/
ldr r0, _bss_start /*BSS 段开始地址，在 u-boot.lds 中指定*/
ldr r1, _bss_end /* BSS 段结束地址，在 u-boot.lds 中指定*/
mov r2, #0x00000000
clbss_1:str
r2, [r0] /* 将 bss 段清零*/
add r0, r0, #4
cmp r0, r1
ble clbss_1

ldr pc, _start_armboot
_start_armboot: .word start_armboot /* 跳转到第二阶段代码入口
start_armboot 处*/
```

本段代码先设置了 BSS 段的起始地址与结束地址，然后循环清楚所有的 BSS 段。至此，所有的 cpu 初始化工作（stage1 阶段）已经全部结束了。后面的代码，将通过 ldr pc, _start_armboot，进入 C 代码执行。这个 C 入口的函数，是在 u-boot-1.1.6/lib_arm/board.c 文件中。它标志着后续将全面启动 C 语言程序，同时它也是整个 u-boot 的主函数。

2. 第二阶段 C 语言代码

uboot 的第二阶段就是要初始化剩下的还没被初始化的硬件，主要是 SOC 外部硬件（譬如 inand、网卡芯片）、uboot 本身的一些东西（uboot 的命令、环境变量等），然后最终初始化完必要的东西后进入 uboot 的命令行准备接受命令。start_armboot 函数不仅标志着后续将全面启动 C 语言程序，同时它也是整个 u-boot 的主函数。

uboot 启动后自动运行打印出很多信息，这些信息就是 uboot 第一和第二阶段不断进行初始化时，打印出来的信息，然后 uboot 进入了 bootdelay 然后执行 bootcmd

对应的启动命令，如果这时候用户不干涉，会执行 bootcmd 进入自动启动内核的流程了。（uboot 的生命周期就结束了）；所以 uboot 完结于命令行下，读取命令，解析命令，执行命令。命令行死循环是 uboot 的最终归宿。

U-Boot 使用一个数组 init_sequence 来存储对于大多数开发板都要执行的初始化函数的函数指针。

```
typedef int (init_fnc_t) (void);
init_fnc_t *init_sequence[] = {
    board_init,                /*开发板相关的配置 */
    timer_init,                /* 时钟初始化-- cpu/arm920t/s3c24x0/timer.c */
    env_init,                  /*初始化环境变量*/
    init_baudrate,             /*初始化波特率 */
    serial_init,               /* 串口初始化 */
    console_init_f,            /* 控制通讯台初始化阶段 */
    display_banner, /*打印 U-Boot 版本、编译的时间- gedit lib_arm/board.c */
    dram_init,                 /*配置可用的 RAM */
    display_dram_config, /* 显示 RAM 大小-- lib_arm/board.c */
    NULL,
};
```

start_armboot 函数

```
void start_armboot (void)
{
    init_fnc_t **init_fnc_ptr;
    char *s;
    /* 计算全局数据结构的地址 gd */
    gd = (gd_t*)(_armboot_start - CONFIG_SYS_MALLOC_LEN - sizeof(gd_t));
    "",
    memset ((void*)gd, 0, sizeof (gd_t));
    gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
    memset (gd->bd, 0, sizeof (bd_t));
    gd->flags |= GD_FLG_RELOC;
    monitor_flash_len = _bss_start - _armboot_start;
    /* 逐个调用 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 ();
        }
    }
    /* armboot_start 在 cpu/arm920t/start.S 中被初始化为 u-boot.lds 连接脚本中的
    _start */
    mem_malloc_init (_armboot_start - CONFIG_SYS_MALLOC_LEN,
```



```

CONFIG_SYS_MALLOC_LEN);          /* NOR Flash 初始化 */
#ifdef CONFIG_SYS_NO_FLASH
display_flash_config (flash_init ());
#endif                          /* NAND Flash 初始化*/
#ifdef CONFIG_CMD_NAND
puts ("NAND: ");
nand_init();
#endif                          /*配置环境变量，重新定位 */
env_relocate ();                /* 从环境变量中获取 IP 地址 */
gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
stdio_init ();
/* get the devices list going. */
jumpable_init ();

""
console_init_r ();              /* 作为设备完全初始化控制台 */
""                               /* 启用异常 */
enable_interrupts ();
#ifdef CONFIG_USB_DEVICE
usb_init_slave();
#endif                          /* 环境初始化*/
if ((s = getenv ("loadaddr")) != NULL) {
load_addr = simple_strtoul (s, NULL, 16);}
#ifdef CONFIG_CMD_NET
if ((s = getenv ("bootfile")) != NULL) {
copy_filename (BootFile, s, sizeof (BootFile));
}
#endif                          /* 网卡初始化 */
#ifdef CONFIG_CMD_NET
#ifdef CONFIG_NET_MULTI
puts ("Net: ");
#endif
eth_initialize(gd->bd);

""
#endif
/* main_loop() 可以返回重试自动启动，如果是,重新运行. */
for (;;) {
main_loop ();
}

```

gd_t 结构体

U-Boot 使用了一个结构体 `gd_t` 来存储全局数据区的数据，这个结构体在 `include/asm-arm/global_data.h` 中定义如下：

```

typedef
struct global_data {

```

```

bd_t
*bd;
unsigned long flags;
unsigned long baudrate;
unsigned long have_console;
unsigned long env_addr;           /* 结构地址 */
unsigned long env_valid;
unsigned long fb_base;
void
} gd_t;

```

U-Boot 使用了一个存储在寄存器中的指针 `gd` 来记录全局数据区的地址：

```

#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm
("r8")

```

`DECLARE_GLOBAL_DATA_PTR` 定义一个 `gd_t` 全局数据结构的指针，这个指针存

放在指定的寄存器 `r8` 中。这个声明也避免编译器把 `r8` 分配给其它的变量。任何 想要访问全局数据区的代码，只要代码开头加入

“`DECLARE_GLOBAL_DATA_PTR`”

一行代码，然后就可以使用 `gd` 指针来访问全局数据区了。

根据 U-Boot 内存使用图中可以计算 `gd` 的值：

```

gd = TEXT_BASE - CONFIG_SYS_MALLOC_LEN - sizeof(gd_t)

```

bd_t 结构体

`bd_t` 在 `include/asm-arm.u/u-boot.h` 中定义如下：

```

typedef struct bd_info {
int
bi_baudrate;           /* 串口通讯波特率 */
unsigned long bi_ip_addr; /* IP 地址*/
struct environment_s *bi_env; /* 环境变量开始地址 */
ulong bi_arch_number; /* 开发板的机器码 */
ulong bi_boot_params; /* 内核参数的开始地址 */
    struct /* RAM 配置信息 */
    {
ulong start;
ulong size;
} bi_dram[CONFIG_NR_DRAM_BANKS];
} bd_t;

```

U-Boot 启动内核时要给内核传递参数，这时就要使用 `gd_t`, `bd_t` 结构体中的信息来设置标记列表

加载 linux 内核

```

int run_command (const char *cmd, int flag)
{

```

```

    cmd_tbl_t *cmdtp;
...           /* 在命令表中查找命令 */
    if ((cmdtp = find_cmd(argv[0])) == NULL) {
...           /* 调用函数执行命令*/
    if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {
...
    }

void main_loop(void)
{
...
#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
    debug ("### main_loop entered: bootdelay=%d\n\n", bootdelay);

s = getenv ("bootcmd");
# ifndef CFG_HUSH_PARSER
    run_command (s, 0);
...
}

U_BOOT_CMD(
    bootm, CFG_MAXARGS, 1, do_bootm,
    "bootm    - boot application image from memory\n",
    "[addr [arg ...]]\n    - boot application image stored in memory\n"
    "\tpassing arguments 'arg ...'; when booting a Linux kernel,\n"
    "\t'targ' can be the address of an initrd image\n"
);

int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
...
    image_header_t *hdr = &header; //uimage 是内核加了一个头部

if (argc < 2) {
    addr = load_addr; //如果 bootm 的参数小于 2 则使用默认的连接地址
} else {
    addr = simple_strtoul(argv[1], NULL, 16); }
...
data = addr + sizeof(image_header_t);
...
switch (hdr->ih_comp) {
    case IH_COMP_NONE:

```

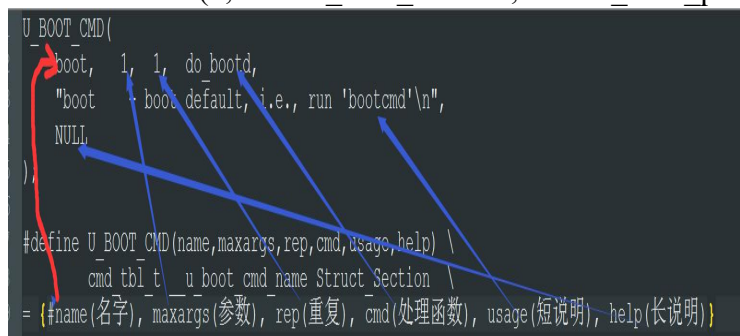
```

        if(ntohl(hdr->ih_load) == addr) {
            printf("XIP %s ... ", name);
        } else {
            memmove((void *) ntohl(hdr->ih_load), (uchar *)data, len);
        }
    }
    .....
after_header_check:
do_bootm_linux (cmdtp, flag, argc, argv, addr, len_ptr, verify);
}

void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
                    ulong addr, ulong *len_ptr, int verify)
{
    /* 启动内核的函数指针 */
    void (*theKernel)(int zero, int arch, uint params);
    image_header_t *hdr = &header; bd_t *bd = gd->bd;
    /* 获得命令行参数 */
#ifdef CONFIG_CMDLINE_TAG
    char *commandline = getenv ("bootargs");
#endif
    theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep)

    setup_start_tag (bd);                // 标记必须以它起始
    setup_commandline_tag (bd, commandline);
    setup_end_tag (bd);                  /* 标记必须以它结尾 */
    printf("\nStarting kernel ...\n\n");
    cleanup_before_linux ();
    /* 启动内核 第1个参数 0, 第二个参数 机器ID 第三个参数 tag 地址 */
    theKernel (0, bd->bi_arch_number, bd->bi_boot_params);

```



```

U_BOOT_CMD(
boot, 1, 1, do bootd,
"boot - boot default, i.e., run 'bootcmd'\n",
NULL,
)

#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t _u_boot_cmd_name Struct Section \
= {#name(名字), maxargs(参数), rep(重复), cmd(处理函数), usage(短说明), help(长说明)}

```

```

void main_loop(void)
{
    ...
#ifdef CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
    debug ("### main_loop entered: bootdelay=%d\n\n", bootdelay);
    s = getenv ("bootcmd");

```

```
# ifndef CFG_HUSH_PARSER
    run_command(s, 0);
...
}
```

run_main_loop 是 board_r 中函数运行列表 init_fnc_t init_sequence_r[] 最后一个函数，它又调用了 main_loop，且 run_main_loop 永不返回。

三、常用命令

1.nand 命令

功能：nand 读写，从 Nand 的 off 偏移地址处读取 size 字节的数据到 SDRAM 的 addr 地址

语法：nand cmd param

uboot 下 NAND 操作指令

指令	Flash 内地址	擦除长度	
nand erase	0x100000	0x200000	
指令	内存中地址	Flash 内地址	写入长度
nand write	0x20000000	0x100000	0x200000
nand read	0x20000000	0x100000	0x200000

3. tftp 命令

4. 功能：ftp 传输。tftp 命令用在本机和 tftp 服务器之间使用 TFTP 协议传输文件。TFTP 是用来下载远程文件的最简单网络协议，它其于 UDP 协议而实现。嵌入式 linux 的 tftp 开发环境包括两个方面：一是 linux 服务器端的 tftp-server 支持，二是嵌入式目标系统的 tftp-client 支持。

语法：tftp addr file
tftp(选项)(参数)

3.md, mw 命令

功能：读写内存

语法：md addr count

使用实例：

修改:mw [内存地址] [值] [长度]

例如:mw 0x02000000 0 128

表示修改地址为 0x02000000~0x02000000+128 的内存值为 0.

显示:md [内存地址] [长度]

例如:md 0x02000000 128

表示显示 0x02000000 的内存数据,长度为 128 个 32bit.

四、总结与感悟

首先感谢老师的线上指导,这是一门实践性较强的课,而且由于本人忙于准备考研相关事宜,因此报告中借鉴了网络上的许多知识,而且内容也有许多不足和漏洞,日后若进行嵌入式方面的研究或者使用 Linux 系统部署项目,一定在本课程所学的基础上进一步深入研究。