嵌入式系统设计实验报告

实验四

编译 uboot 并分析 start.s 文件

学　　号：PE20060014

姓　　名：王晨

专　　业：计算机科学与技术

指导老师：张辉

2020 年 12 月 17 日

一、 实验要求

一、编译 uboot，在开发板上面启动操作系统

二、分析 uboot 的 start.s 文件，说明 start.s 进行了哪些初始化工作，为什么要进行这些初始化工作？

二、 实验条件

1、 硬件条件：Macbook pro

2、 软件条件：Mac OS Big Sur 11.01

VMware fusion pro 虚拟机

Ubuntu 20.04.1 64 位

三、 实验过程

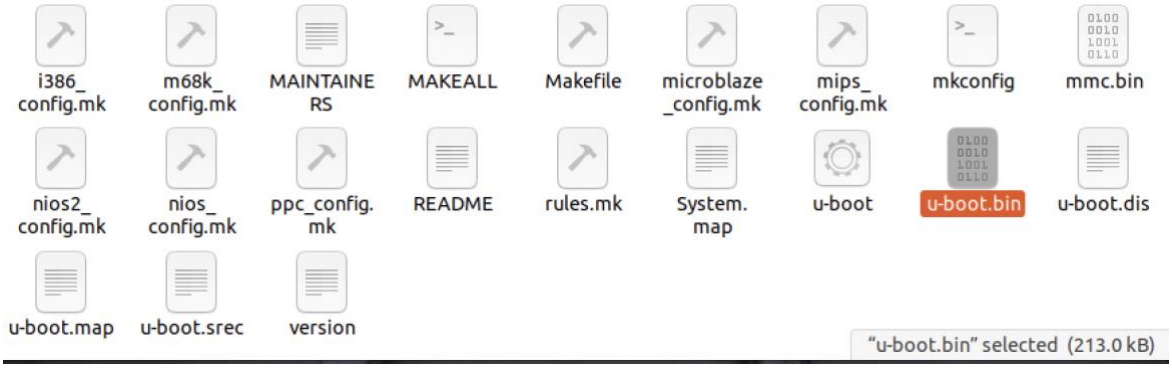
一、 编译 uboot

按照开发文档，下载 uboot1.1.6-V5.50-2014-09-19.tar 并解压。执行
make forlinx_nand_ram256_config （配置适用于 256M 内存平台的 config）
这里注意如果 make 失败报错：
/usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-gcc：命令未找到
则需要将 uboot1.1.6/Makefile 文件内该项改成自己 arm-linux-gcc 路径，例如：

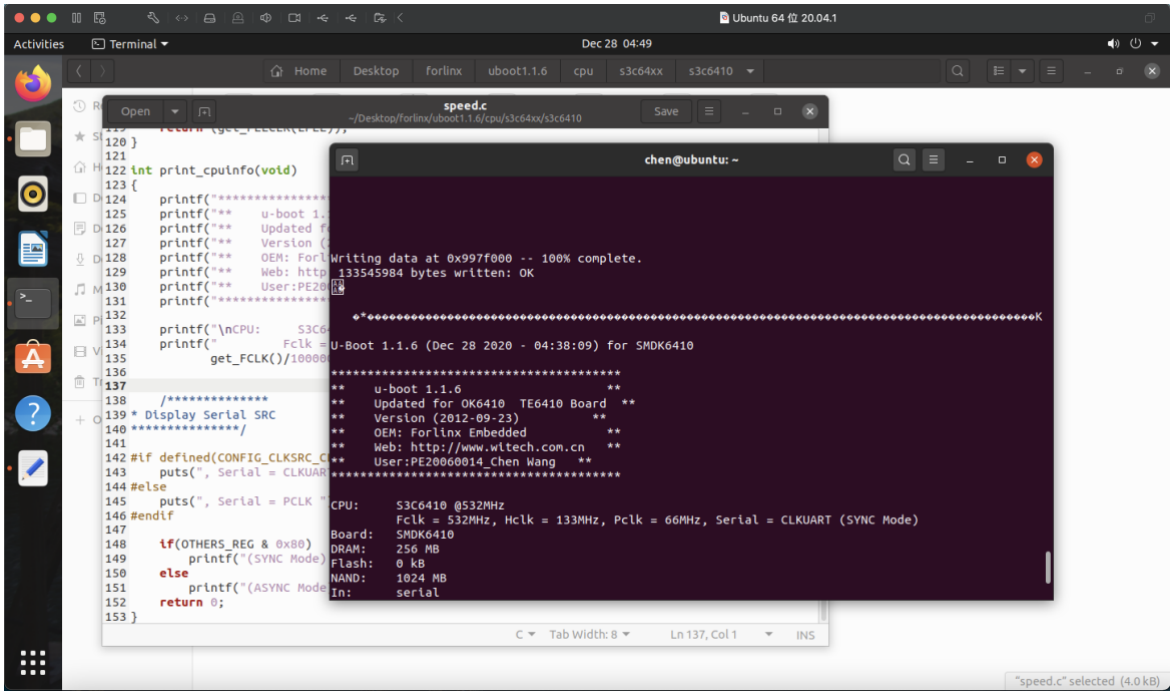
```
161 CROSS_COMPILE = /usr/local/arm/gcc-4.6.4/bin/arm-linux-
162 export  CROSS_COMPILE
```

之后执行：
make forlinx_nand_ram256_config
make clean
make
成功编译后将在/uboot1.1.6 目录下产生名为 u-boot.bin 的二进制文件。该文件即我们
需要烧写到 Nandflash 的 U-boot 映像文件，如下图所示：

之后烧写系统到开发板和实验 1 是一样的。



若要在编译后开机显示个人信息，可以在 cpu/s3c64xx/s3c6410/speed.c 下添加打印个人信息，之后重新编译 uboot 烧写系统即可：



二、 分析 start.s 文件

本实验看的是 cpu/s3c64xx/start.S，对应 6410 板 cpu，结合老师给的几个文档——uboot 启动分析和 start.s 源码详解，start.s 是作 uboot 启动内核的第一阶段工作，包括硬件初始化，复制第二阶段代码到 RAM 等等。

start.s 主要进行了以下这些初始化工作：

1. 设置 CPU 模式

以下代码将 CPU 的工作模式位设置为 SVC 管理模式，并将中断禁止位和快中断禁止位置 1，从而屏蔽了 IRQ 和 FIQ 中断。

为什么要设置成管理模式，是为了获取尽量多的权限，以方便操作硬件，初始化系统相关硬件资源；另外，在跳转到 kernel 之前，本身必须满足 CPU 处于 SVC 模式。因此，uboot 在最初的初始化阶段，就将 CPU 设置为 SVC 模式，也是最合适的。

```
136 reset:
137     /*
138      * set the cpu to SVC32 mode
139      */
140     mrs r0,cpsr
141     bic r0,r0,#0x1f
142     orr r0,r0,#0xd3
143     msr cpsr,r0
```

2. 关闭 MMU 和 cache

Uboot 阶段 CPU 还不能管理 Caches。上电的时候指令 Cache 可关闭，也可不关闭，但数据 Cache 一定要关闭，否则可能导致刚开始的代码里面，去取数据的时候，从 Cache 里面取，而这时候 RAM 中数据还没有 Cache 过来，导致数据预取异常。
Uboot 在硬件初始化访问的都是内存实际地址，MMU 在此阶段没有作用，关闭即可。第二阶段代码 copy 到 RAM 之后可以 enable MMU。

```
167     /*
168      * disable MMU stuff and caches
169      */
170     mrc p15, 0, r0, c1, c0, 0
171     bic r0, r0, #0x00002300   @ clear bits 13, 9:8 (--V- --RS)
172     bic r0, r0, #0x00000087   @ clear bits 7, 2:0 (B--- -CAM)
173     orr r0, r0, #0x00000002   @ set bit 2 (A) Align
174     orr r0, r0, #0x00001000   @ set bit 12 (I) I-Cache
175     mcr p15, 0, r0, c1, c0, 0
```

3. 关闭看门狗

以下代码向看门狗控制寄存器写入 0，关闭看门狗。否则在 U-Boot 启动过程中，CPU 将不断重启。硬件初始化阶段不需要看门狗功能，因此把它关了。

```
188
189 #if defined(CONFIG_S3C6410) || defined(CONFIG_S3C6430)
190     orr r0, r0, #300     @ disable watchdog
191     mov r1, #1
192     str r1, [r0]
193
```

4. 屏蔽中断

Uboot 主要完成硬件初始化，环境参数设置，代码搬运等工作，用不到中断。屏蔽中断是为了避免因为意外中断使得 boot 失败，毕竟很多外设还没有初始化，对应中断代码也都没有准备好。

5. 设置堆栈 sp 指针

这是在初始化内存的用户栈区，sp 指向一段没有被使用的内存，作为用户栈顶，至于堆栈区设置多少，要看具体需求和使用场景：

```
397     /* Set up the stack                             */
398 stack_setup:
399 #ifdef CONFIG_MEMORY_UPPER_CODE
400     ldr sp, =(CFG_UBOOT_BASE + CFG_UBOOT_SIZE - 0xc)
401 #else
402     ldr r0, _TEXT_BASE      /* upper 128 KiB: relocated uboot   */
403     sub r0, r0, #CFG_MALLOC_LEN    /* malloc area                    */
404     sub r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                        */
405 #ifdef CONFIG_USE_IRQ
406     sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
407 #endif
408     sub sp, r0, #12      /* leave 3 words for abort-stack    */
409
410 #endif
```

6. 清除 bss 段

BSS 段是用来存放程序中未初始化的或者初始化为 0 的全局变量和静态变量的一块内存区域。这些变量的初始值应赋为 0，否则这些变量的初始值将是一个随机的值，若有些程序直接使用这些没有初始化的变量将引起未知的后果，比如同一段程序不同时间运行可能会产生不同结果。

```
412 clear_bss:
413     ldr r0, _bss_start      /* find start of bss segment        */
414     ldr r1, _bss_end        /* stop here                        */
415     mov     r2, #0x00000000       /* clear                          */
416
417 clbss_l:
418     str r2, [r0]        /* clear loop...                    */
419     add r0, r0, #4
420     cmp r0, r1
421     ble clbss_l
```

7. 异常中断处理

异常中断处理，就是实现对应的常见的那些处理中断的中断函数。uboot 在初始化的时候，主要目的是为了初始化系统，以及引导系统，因此中断处理部分的代码，往往相对比较简单，不是很复杂。

以下代码是初始化异常向量表和不同异常下对应的代码入口地址，例如“软件中断”，“预取指错误”，“数据错误”，“未定义”，“普通中断”，“快速中断”：

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

_undefined_instruction:
    .word undefined_instruction
_software_interrupt:
    .word software_interrupt
_prefetch_abort:
    .word prefetch_abort
_data_abort:
    .word data_abort
_not_used:
    .word not_used
_irq:
    .word irq
_fiq:
    .word fiq
```

当一个异常产生时，CPU 根据异常号在异常向量表中找到对应的异常向量，然后执行异常向量处的跳转指令，CPU 就跳转到对应的异常处理程序执行。

```
/*
 * exception handlers
 */
    .align  5
undefined_instruction:
    get_bad_stack
    bad_save_user_regs
    bl  do_undefined_instruction

    .align  5
software_interrupt:
    get_bad_stack_swi
    bad_save_user_regs
    bl  do_software_interrupt

    .align  5
prefetch_abort:
    get_bad_stack
    bad_save_user_regs
    bl  do_prefetch_abort

    .align  5
data_abort:
    get_bad_stack
    bad_save_user_regs
    bl  do_data_abort
```

当有异常出现 ARM 会自动执行以下步骤：

1 将下一条指令的地址存放在连接寄存器 LR(通常是 R14).---保存位置

2 将相应的 CPSR(当前程序状态寄存器)复制到 SPSR(备份的程序状态寄存器)中
3 根据异常类型，强制设置 CPSR 运行模式位
4 强制 PC 从相应异常向量地址取出下一条指令执行，从而跳转到异常处理函数中执行