回顾：

裸板软件编程的框架

系统上电

一系列硬件的初始化

```
while(1)
{
    ....
}
```

+

异常处理流程

ARM 异常处理流程：是按键为例

中断的触发

1）中断源

配置中断的触发方式：高/低/上/下

中断使能

2）中断控制器

配置中断源的中断优先级：值越小优先级越高

中断使能

特性：以 IRQ/FIQ 形式上报 报告给哪个 processor(核)...

3）ARM 核

中断使能 CPSR.I=0

按下按键触发 IRQ 异常 首先介入的硬件 硬件自动做 4 件事

1）备份 CPSR

2）修改 CPSR

3）保存返回地址到 LR

4）PC=vector_base + 0x18

软件要做的事：

1）vector_base + 0x18 放跳转指令 （异常向量表） 跳转到 asm_do_irq

2）asm_do_irq{ //汇编

保护现场

bl c_do_irq

恢复现场

}

3）c_do_irq{ //c 函数

判断哪个硬件触发的中断

调用对应的具体硬件中断处理函数 handle_hw_isr

清除中断源、中断控制器级的 pending 位

}

4）handle_hw_isr{ //实际驱动开发过程中，主要编码工作 （c 语言）

具体的硬件操作：访问特殊功能寄存器

}

FIQ 为什么会被称为 fast IRQ 呢? (面试题)

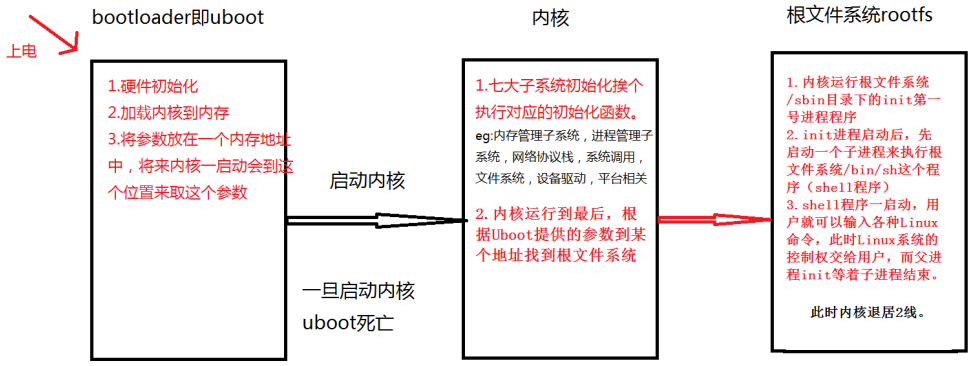
1）FIQ 比 IRQ 优先级高

2）做现场保护时可以少压栈几个寄存器：r8-r12

3) 在异常向量表中 FIQ 对应的位置可以直接放 FIQ 异常的处理代码 不需要跳转

---

linux 系统移植



DAY01

1、什么叫移植?

将已有的代码，根据目标硬件平台的差异，进行少量的代码修改
就能使得该代码在新的硬件平台上运行起来的过程叫做移植

2、移植的主要内容

移植的基础：

1）对代码很熟
2）对硬件的差异很熟

2.1 uboot 的移植

2.2 linux 内核的移植

2.3 根文件系统镜像的制作: busybox

rm cd ls touch vi ...

3、移植课程的目标：

掌握 uboot 的配置编译过程
熟悉 uboot 启动的两个阶段（代码）

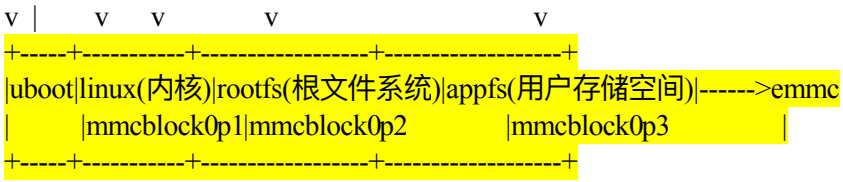
掌握 linux 的配置编译过程
了解 linux 内核的启动流程（代码）
掌握 linux 内核模块的操作方法

掌握根文件系统的制作与使用

4、开发板的烧写实验

bootcmd

```
  +--------+
0 |    1M  |        65M                  819M
```



4.0 设置 emmc 的分区

help fdisk

fdisk 2 3 0x100000:0x4000000 0x4100000:0x2f200000 0x33300000:0

1M = 1024 * 1024 再转 16 进制

重启开发板(reset)

fdisk 2(查看分区)

4.1 uboot 烧写

参考 arm 课程 day01 的笔记

4.2 linux 内核烧写

(1)cp /mnt/hgfs/esd2002/porting/env/uImage /tftpboot/

(2)在开发板上执行

tftp 48000000 uImage

(3)mmc write 48000000(ddram 上) 0x800 0x3000

0x800:要写入的扇区位置

一个扇区 512 字节

0x100000/512 = 0x800

0x3000: 要连续写入的扇区个数

>下载字节数/512

//加载内核：从 emmc 偏移为 0x800 扇区开始连续读取 0x3000 个扇区内容到内存 0x48000000 的位置

(4)mmc read 48000000 800 3000(下载的字节数/512)

//启动内核：bootm 会先准备好内核启动所需要的参数 然后跳转到 0x48000000 位置去执行

(5)bootm 48000000

实验结果：

Kernel panic - not syncing: VFS: Unable to mount root fs ...

Rebooting in 5 seconds..

开机自动启动 linux 内核

(6)setenv bootcmd mmc read 48000000 800 3000 \; bootm 48000000

(7)saveenv

4.3 烧写根文件系统镜像

(1)cp /mnt/hgfs/esd2002/porting/env/rootfs_ext4.img /tftpboot/

(2)在开发板上执行

tftp 48000000 rootfs_ext4.img

(3) mmc write 48000000 20800 32000

(4)setenv bootargs root=/dev/mmcblk0p2 rootfstype=ext4 init=/linuxrc console=ttySAC0 maxcpus=1 lcd=wy070ml tp=gslx680 loglevel=2

root, 指定根文件系统位置
rootfstype,指定根文件系统类型
init,指定用户空间的 1 号进程
console,指定控制台 ttySAC0 即 uart0
maxcpus=1, 单核启动
lcd
tp ,指定 LCD 触摸屏类型
loglevel=2,设置打印优先级阈值 (驱动课解释)

(5)saveenv
重启开发板
用户名：root
密码： 123456

容易出现的问题：
1）分区是否正常

1	2048	131072	00000000-01	83
2	133120（重点）	1544192	00000000-02	83
3	1677312	13592576	00000000-03	83

2）下载写入根文件系统时是否正常执行
tftp 48000000 rootfs_ext4.img
mmc write 48000000 20800 32000
3）正确告诉内核去哪找根文件系统
setenv bootargs root=/dev/mmcblk0p2 rootfstype=ext4 init=/linuxrc ...

5、UC 都可以放到开发板上来运行了
cd~
mkdir porting
cd porting/
vi hello.c
gcc hello.c -o hello_x86 //生成 pc 上的可执行
file hello_x86
arm-cortex_a9-linux-gnueabi-gcc hello.c -o hello_arm
file hello_arm//查看文件使用架构
cp hello_arm /tftpboot/

启动开发板进入 linux 系统
cd /
ifconfig eth0 192.168.1.6//在 linux 系统上设置网卡
ping 192.168.1.8
tftp -g -r hello_arm 192.168.1.8(下载 hello_arm 文件)
-g: get
-r: remote 远程

chmod +x hello_arm
./hello_arm

尝试将 uc 课练的代码搬到板子上执行一下， 体会嵌入式 linux 系统带来的好处

which tftp:tftp 在哪

6、通过 nfs 方式挂载根文件系统

nfs 是一种通信协议

nfs: net file system

6.1 安装 nfs server 软件

联网： sudo apt-get install nfs-kernel-server

未联网：

cd /home/tarena/Downloads/nfs/

sudo dpkg -i *.deb

可以通过 dpkg -l | grep "nfs" 检查是否安装成功

6.2 准备根文件系统中的文件

cd /opt/

cp /mnt/hgfs/esd2002/porting/env/rootfs_qt.tar.bz2 ./

tar xf rootfs_qt.tar.bz2(包含软连接文件。windows 不支持软连接文件，不能在 windows 下解压)

6.3 配置 nfs server(防止 nfs 客户端访问 pc 机上的目录)

sudo vi /etc/exports

/opt/rootfs *(rw,sync,no_root_squash)

/opt/rootfs: 允许 nfs 客户端访问的目录

*, 不设 IP 限制

192.168.1.* :只允许以“192.168.1”开头的主机访问该目录

rw:有读写权限

sync:同步

no_root_squash: 用户角色

6.4 重启 nfs server 使得新配置生效

sudo /etc/init.d/nfs-kernel-server restart

验证：sudo exportfs

/opt/rootfs <world>

注意：如果你使用了 ubuntu 18.04 系统

sudo vi /etc/default/nfs-kernel-server 末尾加一句

RPCNFSDOPTS="--nfs-version 2,3,4 --debug --syslog"

sudo /etc/init.d/nfs-kernel-server restart

6.5 客户端的配置

进入 uboot 命令行

printenv bootargs:

bootargs=root=/dev/mmcblk0p2 rootfstype=ext4 init=/linuxrc console=ttySAC0 maxcpus=1 lcd=wy070ml tp=gslx680 loglevel=2

setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 init=/linuxrc console=ttySAC0 maxcpus=1 lcd=wy070ml tp=gslx680 loglevel=2

saveenv

重启开发板

6.6 验证

cp /home/tarena/porting/hello_arm /opt/rootfs/

在板子上执行 ls /

./hello_arm

7、uboot

bootloader 的作用：

1）为操作系统的启动初始化硬件环境

2）加载启动操作系统

uboot 属于 bootloader 的一种,属于开源程序

uboot 号称通用的 bootloaer,

1)支持多种 CPU 架构

powerpc mips arm x86 ...

2)支持加载启动多种操作系统

linux vxworks, qnx ...

uboot 源码的获取方式：

1）http://www.denx.de/wiki/U-Boot/WebHome

2）从上游厂家使用的 uboot 源码开始改起（强烈建议） env/uboot.tar.bz2

如何配置编译 uboot 源码：

1）cd /home/tarena/porting

cp /mnt/hgfs/esd2002/porting/env/uboot.tar.bz2 ./

2) tar xvf uboot.tar.bz2

3) cd uboot/

4) find ./ -type f | wc -l //查看文件个数

5）make x6818_config//配置

6）选做：板子上 LCD 花屏的同学 ，给 uboot 打补丁

cp /mnt/hgfs/esd2002/porting/env/6818_7inchMIPIScreen_uboot.patch ./

查看补丁文件 6818_7inchMIPIScreen_uboot.patch

/linnux/bootloader/u-boot-2014.07/board/s5p6818/x6818/x6818-lcds.c

所以 p 后面跟着 4

shell:patch -p4 <6818_7inchMIPIScreen_uboot.patch//(p4-->跳过 4 级目录,数"/")

7）make

最终生成了 ubootpak.bin

练习：尝试将该 ubootpak.bin 烧写到开发板 看是否能够正常启动

DAY02

回顾：

开发板的烧写实验，让 linux 系统在开发板上运行起来

1）uboot

它是 bootloader 的一种

bootloader 是用于为操作系统的启动初始化硬件环境，加载启动操作系统

2）linux

功能：

1）内存管理功能

2）进程管理

3）进程间通信功能

4）虚拟文件子系统

5）网络子系统

3）根文件系统

bootcmd: uboot 启动后自动执行的命令

setenv bootcmd mmc read 48000000 800 3000 \;bootm 48000000 //从 emmc 中加载内核,启动内核

setenv bootcmd tftp 48000000 uImage \;bootm 48000000 //从网络服务器上加载内核到内存,启动内核

//先 ping 保证 tftp 下载时网卡可用

setenv bootcmd ping 192.168.1.8 \; ping 192.168.1.8 \; tftp 48000000 uImage \;bootm 48000000

saveenv

bootargs:当内核启动成功后，告诉内核去哪找根文件系统

//去 emmc 对应分区中找根文件系统

setenv bootargs root=/dev/mmcblk0p2 rootfstype=ext4 ....

//去 192.168.1.8 主机的/opt/rootfs 目录找根文件系统

setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ....

1、uboot 简介

属于 bootloader 的一种

开源程序

号称通用的 bootloader

1)支持多种 CPU 架构 <-------+

2）支持启动多种操作系统 |

2、uboot 的配置与编译过程 |

make x6818_config----------+

make

make clean //清除编译过程中产生的文件

make distclean //清除配置阶段和编译阶段产生的文件,配置文件也清除

3、为什么要先配置后编译

配置完成了选出特定硬件相关代码 ， 使其参与后续编译过程

4、阅读 uboot 的代码

4.1 寻找入口点文件

rm u-boot

make V=1 //显示编译链接的详细过程

arm-cortex_a9-linux-gnueabi-ld.bfd -pie --gc-sections -Bstatic -Ttext 0x43C00000 -o u-boot -T u-boot.lds arch/arm/cpu/slsiap/start.o --start-group arch/arm/cpu/built-in.o ...

vi u-boot.lds

arch/arm/cpu/slsiap/s5p6818/start.o (.text*)

shift+8------->查找字符串(如同输入模式的:/字符串)

入口点文件是 arch/arm/cpu/slsiap/s5p6818/start.S

4.2 uboot 启动流程

ctags -R *//分析文件

reset:

设置为 SVC 模式

禁止看门狗

使 L1 缓存无效

禁止 MMU

清空 BSS

bl board_init_f //board_f.c

```
{
    initcall_run_list(init_sequence_f) //完成了一系列硬件的初始化
}
```

ldr pc, =board_init_r //board_r.c

```
{
    initcall_run_list(init_sequence_r)//进一步完成硬件的初始化 最后 run_main_loop
}
```

run_main_loop：

```
main_loop{
    s = bootdelay_process(){
        s = getenv("bootdelay");
        //完成字符串到整型的转换
        bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
        ...
        s = getenv("bootcmd");
        stored_bootdelay = bootdelay;
        return s;
    }
    autoboot_command(s)
    {
        //倒数读秒计时 判断是否被打断
        if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay)){
            //顺序执行 bootcmd 中对应的命令
```

```
                run_command_list(s, -1, 0);
            }
        }
        cli_loop(){
            cli_simple_loop(){
                for (;;){
                    //输出命令提示符 接收用户输入的命令到 console_buffer
                    len = cli_readline(CONFIG_SYS_PROMPT);

                    strcpy(lastcommand, console_buffer);
                    //执行命令
                    run_command_repeatable(lastcommand, flag)
                }
            }
        }
    }
```

sourceinsight:编辑 阅读 修改代码的利器 (不能编译)
ubuntu + sourceinsight 系统使用 wine 的模拟器

阅读代码的经验：
1）有文档先看文档
2）看代码时关注框架 关注流程 放弃细节

5、添加一个启动 LOGO
5.1 LOGO 的作用
1）商业行为
2）更好的用户体验

5.2 LOGO 显示的原理
5.2.1 实现 LCD 屏的驱动程序 board/s5p6818/x6818/x6818-lcds.c
uboot 中已经实现了 LCD 的驱动程序，主要完成了

1）LCD 控制器的时序配置
2）申请一片连续的内存作为显存使用(从 ddram0x46000000 开始)
3）将显存的起始地址通知 LCD 控制器

效果：LCD 控制器会自动地、周期性地将显存中的数据刷新到 LCD 屏上去

应用程序如果要在 LCD 屏上显示图像的其实就是将图像相关的数据写入显存即可
5.2.2 图像显示的基本原理
图像是由像素点组成的
每个像素点的颜色值由 RGB 三原色构成，通常 RGB 的取值各占一个字节

32bit 真彩色: 透明度 R G B

开发板使用的 LCD 分辨率：1024*600

描述点(x0,y0)像素点的颜色公式：

p0=BASE_ADDRESS+（y0*1024+x0）*4

5.3 显示图片到 LCD 屏

1）将图片转换为 RGB 数据：Image2Lcd.rar

2）cd /home/tarena/porting/uboot

cp /mnt/hgfs/esd2002/porting/env/tarena_logo.c common/

vi common/draw_logo.c

```
#define LCD_BASE     (0x46000000)
#define LCD_WIDTH    (1024)
#define LCD_HEIGHT   (600)
#define PIC_WIDTH    (320)
#define PIC_HEIGH    (200)
#define X0           (352)
#define Y0           (200)
extern const unsigned char gImage_tarena_logo[];
/*描点*/
void draw_pixel(int row,int col,int color)
{
    int *p = (int *)LCD_BASE;
    p = p + (row*LCD_WIDTH + col);//计算像素点对应的显示地址
    *p = color;
}
/*画图*/
void draw_picture(void)
{
    int i = 0;
    int j = 0;
    int red = 0;
    int green = 0;
    int blue = 0;
    int color = 0;
    const unsigned char *p = gImage_tarena_logo;
    for(i = Y0;i<(Y0+PIC_HEIGH);i++)//描一行
    {
        for(j=X0;j<(X0+PIC_WIDTH);j++)
        {
            blue  = *p++;
            green = *p++;
            red   = *p++;
            color = red << 16|green<<8|blue;
            draw_pixel(i,j,color);
        }
    }
}
```

3) vi ~/porting/uboot/arch/arm/cpu/slsiap/common/cmd_draw_logo.c

```
408 extern void draw_picture(void);
409 #if (1)
410 static void fill_lcd(U32 FrameBase, int XResol, int YResol, U32 PixelByte)
411 {
412         draw_picture();
413 }
```

4) vi common/Makefile

```
7 obj-y += tarena_logo.o
8 obj-y += draw_logo.o
```

5)make

6)将新生成 ubootpak.bin 烧写到 emmc

cp ubootpak.bin /tftpboot/

在开发板上执行

tftp 48000000 ubootpak.bin

update_mmc 2 2ndboot 48000000 200 下载字节数

重启开发板观察实验现象

练习：男神女神 LOGO 显示

实现画直线 矩形 圆

DAY03

回顾：

uboot 的移植

1）配置 编译了 uboot 源码

2）阅读 uboot 源码

设置为 SVC 模式

禁止看门狗

清空缓存

禁止 MMU

清空 BSS 段

设置栈指针

一系列硬件的初始化

调用 run_main_loop

获取环境变量 bootdelay bootcmd 的值

倒数读秒计时 无打断 自动执行 bootcmd 中记录的命令

有打断 输出命令提示符 接收用户输入命令 执行命令

移植工作的重点：

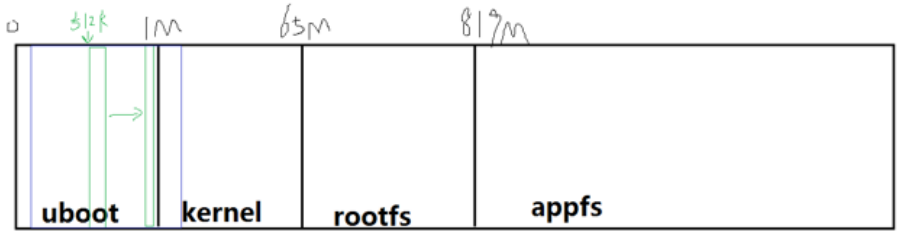
a)硬件初始化

b)x6818.h include/configs/boardname.h

3) 增加了一个新的启动 LOGO

图像由像素点组成

像素点的颜色取决于 RGB 三原色
应用程序在 LCD 屏画图，实则就是将图像画到显存中去
saveenv(代码位置):



----------linux 内核的移植

1、linux 内核
Linus 开发的 开源程序

官网： www.kernel.org

板子上使用的内核版本：3.4
shell:uname -a 查看 linux 版本

linux 宏内核：
内存管理
进程管理
进程间通信
虚拟文件子系统
网络协议子系统

板子上使用的内核源码：env/kernel.tar.bz2
2、快速配置编译内核
cd /home/tarena/porting
cp /mnt/hgfs/esd2002/porting/env/kernel.tar.bz2 ./
tar xf kernel.tar.bz2
cd kernel/
cp /home/tarena/porting/kernel/arch/arm/configs/x6818_defconfig .config
vi Makefile
195 ARCH ?= arm
196 CROSS_COMPILE ?= arm-cortex_a9-linux-gnueabi-
make uImage

出现 "mkimage" command not found 解决方式：
sudo cp /home/tarena/porting/uboot/tools/mkimage /bin
make uImage

将该 uImage 在板子上运行一下
1）cp arch/arm/boot/uImage /tftpboot/

2）开发板上执行

setenv ipaddr 192.168.1.6

setenv serverip 192.168.1.8

setenv bootcmd ping 192.168.1.8\;ping 192.168.1.8\;tftp 48000000 uImage \;bootm 48000000

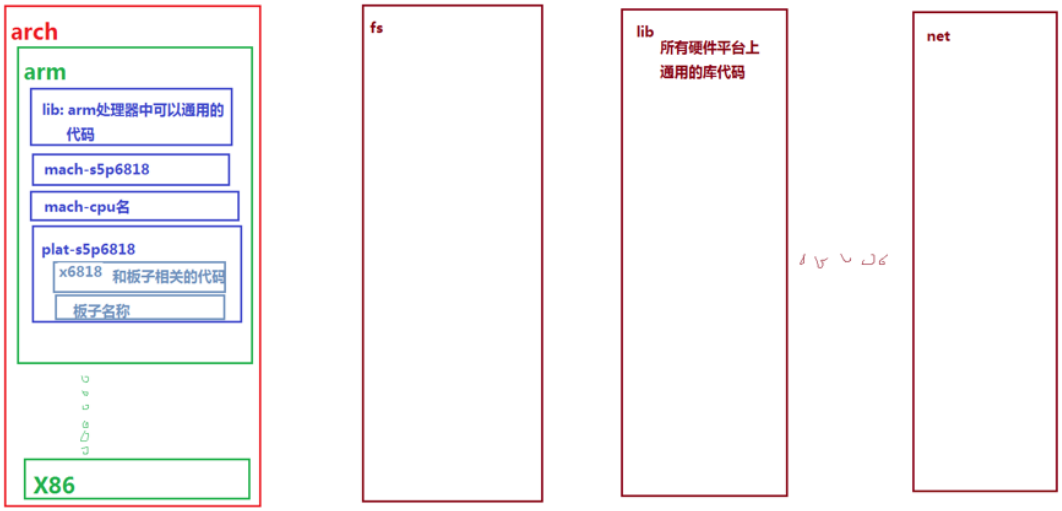
saveenv

setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 init=/linuxrc console=ttySAC0 maxcpus=1 lcd=wy070ml tp=gslx680 loglevel=2

saveenv

3、内核的配置

3.1 为什么要配置内核



1）linux 内核也支持多种 CPU 架构: x86 arm mips ...(在 kernel/arch 中可查看)

通过配置把特定硬件相关代码选出来 参与后续编译过程

2）linux 作为操作系统包含了大量的功能

而某款嵌入式平台上极有可能只需要其中部分功能

这时候可以把不需要的功能通过配置裁剪掉

3.2 如何配置内核

拿相近的配置

cp /home/tarena/porting/kernel/arch/arm/configs/x6818_defconfig .config

来改

cd /home/tarena/porting/kernel

make menuconfig

General setup ---> //通用配置

(X6818) Default hostname

[*] System V IPC

(5) Default panic timeout

System Type ---> //系统类型

ARM system type (SLsiAP S5P6818) --->

Boot options ---> //启动选项

Kernel command line type (Use bootloader kernel arguments if available) ---> //从 uboot 传入的 bootargs 获取启动参数

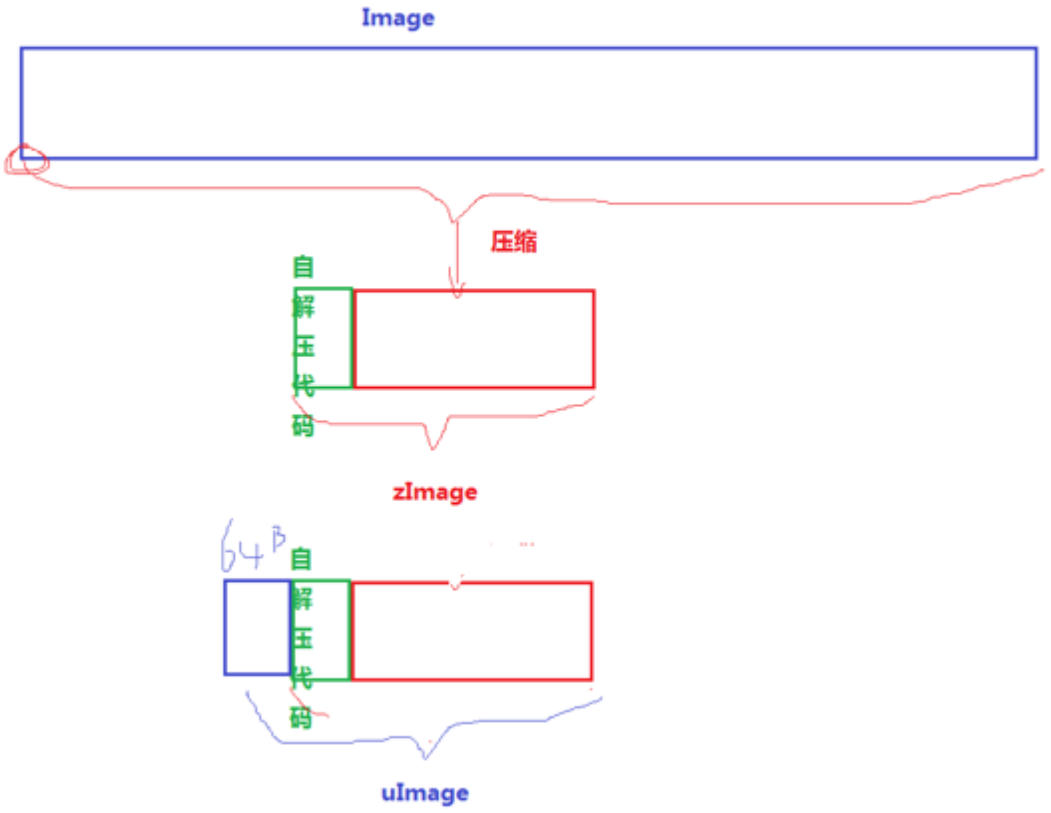
Device Drivers ---> //配置要将哪些硬件驱动编译到 uImage

File systems ---> //配置内核支持的文件系统
[*] Network File Systems ---> //支持 nfs 网络文件系统

Save an Alternate Configuration File //配置结果保存到哪个文件中去

4、编译内核

make uImage
make clean //清除编译过程中产生的文件
make distclean //清除配置和编译过程中产生的文件



xxx.o yyy.o ....------>vmlinux(ELF)------>Image(BIN)------>zImage(自解压代码+压缩之后的Image)

------>uImage(64B+ zImage)

uImage: 专门为 uboot 准备的

64B 中放了 uboot 启动内核所必须的信息

5、linux 启动代码的阅读

5.1 寻找入口点文件

rm vmlinux
make uImage V=1

arm-cortex_a9-linux-gnueabi-ld -EL -p --no-undefined -X --build-id -o vmlinux -T arch/arm/kernel/vmlinux.lds arch/arm/kernel/head.o（先出现的，先看.s 文件是否有.head.text,如果有，则 head.S 位入口点文件） arch/arm/kernel/init_task.o init/built-in.o --start-group usr/built-in.o arch/arm/vfp/built-in.o arch/arm/kernel/built-in.o arch/arm/mm/built-in.o arch/arm/common/built-in.o arch/arm/net/built-in.o arch/arm/mach-s5p6818/built-in.o ...

vi arch/arm/kernel/vmlinux.lds

```
498    .head.text : {
499      _text = .;
500      *(.head.text)
501    }
```

入口点文件： arch/arm/kernel/head.S

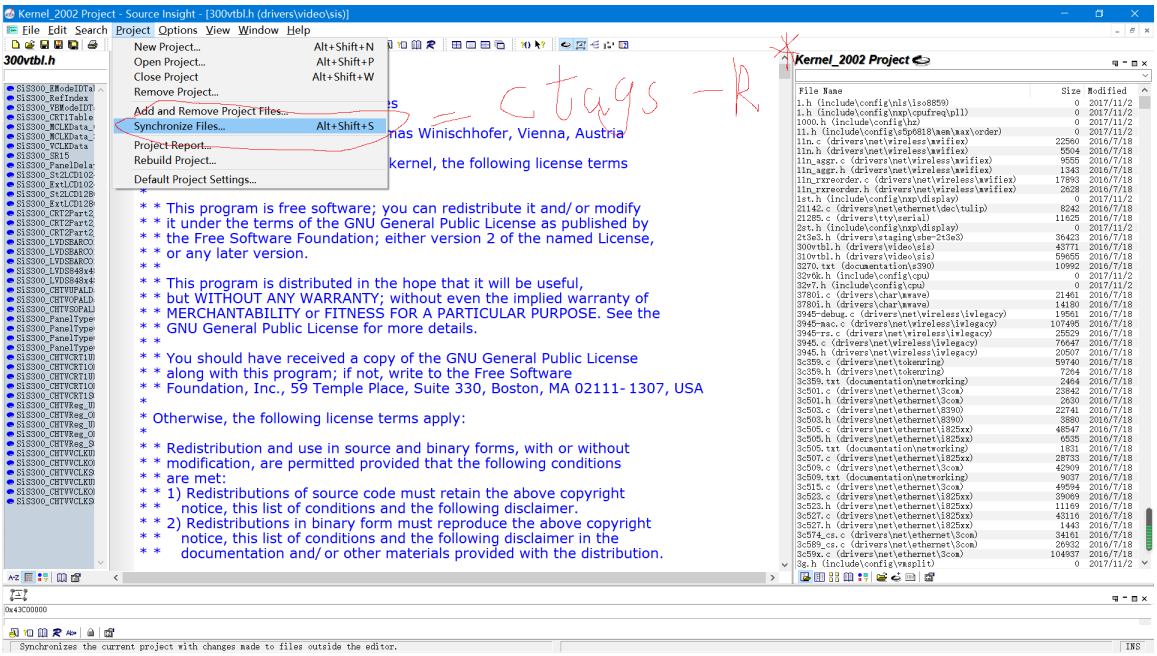
5.2 内核启动代码分析

ctags -R *

vi arch/arm/kernel/head.S

```
//判断内核是否支持该 cpu
__lookup_processor_type
//创建页表
bl          __create_page_tables//地址映射表的创建

ldr         r13, =__mmap_switched

b           __enable_mmu

__enable_mmu: //开启 MMU，后面的地址都是虚拟地址了
    ...
    b           __turn_mmu_on

ENTRY(__turn_mmu_on)
    ...
    mov         r3, r13
    mov         pc, r3    //跳转到__mmap_switched 去执行

__mmap_switched:
    ...

    b           start_kernel
```

启动 sourecInsight---->kernel.tar




```
start_kernel
    rest_init(){
        //创建内核线程
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND)
    }
kernel_init(){
    prepare_namespace(){
        //挂载根文件系统
        mount_root();
    }
    init_post(){
        /*setenv bootargs ... init=/linuxrc .... <-----------+
          根据用户指定的信息去启动用户空间的 1 号进程                |
                                                              |
          */                                                  |
        run_init_process(execute_command);//execute_command---+

        /*尝试使用默认的程序作为用户空间的 1 号进程*/
        run_init_process("/sbin/init");

        ...
        //内核恐慌
        panic("No init found.  Try passing init= option to kernel. "

    }
}
```

系统上电---->设置为 SVC 模式----->禁止看门狗--->禁止 MMU----->清空 BSS 段------->一系列硬件的初始化

----->读取环境变量 bootcmd---->倒数读秒计时------>加载内核----->启动内核------>检查内核是否支持该 CPU

------>创建页表---->启用 MMU (至此以后用到的地址都是虚拟地址) ----> start_kernel-------->

创建一个新的内核线程---->该线程中去挂载根文件系统----->启动用户空间的 1 号进程------>启动后续进程

------>启动一个 shell------>用户就可以输入命令了

6、配置过程是如何影响编译过程的

make menuconfig

Device Drivers --->

-*- LED Support --->

<*> LED Support for GPIO connected LEDs

配置的结果保存在了.config 文件中

每个配置项都对应一个不同的 CONFIG_开头的变量

配置结果以不同的 CONFIG_开头的变量对应不同值的形式保存在了.config 中

内核源码几乎每个子目录下都有一个 Makefile

该 Makefile 决定所在目录下的文件(夹)是否参与 uImage 文件的生成

语法规则:

obj-y += xxx.o //xxx.c 会被编译到 uImage

obj-m += xxx.o //xxx.c 不会被编译到 uImage 会被编译为独立的模块文件

obj- += xxx.o //xxx.c 不会被编译

配置项	.config	drivers/leds/Makefile
LED Support for GPIO...|	|obj-$(CONFIG_LEDS_GPIO) += leds-gpio.o
<*>	CONFIG_LEDS_GPIO=y	obj-y += leds-gpio.o
<M>	CONFIG_LEDS_GPIO=m	obj-m += leds-gpio.o
< >	# CONFIG_LEDS_GPIO is not set	obj- += leds-gpio.o

shell:grep "字符串" * -nRw 搜索当前目录下所有文件中的字符串

7、将已有.c 文件编译到 uImage

env/led_drv.c

实验步骤:

1) cp /mnt/hgfs/esd2002/porting/env/led_drv.c drivers/char/
2) vi drivers/char/Makefile
   obj-y += led_drv.o
3) make uImage
4) cp arch/arm/boot/uImage /tftpboot/
5) setenv bootcmd ping 192.168.1.8\;ping 192.168.1.8\;tftp 48000000 uImage \;bootm 48000000
   saveenv
6) 重启开发板

7）dmesg：将内核启动过程中的打印信息重新输出
dmesg >1.txt
或者
dmesg | grep "call"

DAY04

回顾：

linux 内核的移植

1）linux 开源程序 宏内核 包含核心功能：
内存管理
进程管理
进程间通信
虚拟文件子系统
网络协议子系统

2）配置内核
选择硬件相关代码
裁剪不需要的功能

make menuconfig

3) 编译内核
make uImage
make clean
make distclean

一系列的.o ------> vmlinux(ELF)-----> Image (BIN) -----> zImage(自解压代码+压缩之后的 Image)
------>uImage(64B 头信息+zImage)

4) 阅读内核源码
上电----》设置为 SVC 模式-----》禁止看门狗----》禁止 MMU----》清空 BSS 段---》设置 SP
------>一系列硬件的初始化-----》获取环境变量 bootcmd---->倒数读秒计时-------》
执行 bootcmd 加载启动内核-----》检查内核是否支持该 CPU------》创建页表-----》使能 MMU
-------》start_kernel------>开启内核线程-----》挂载根文件系统------》启动 1 号进程
------》开启后续进程-----》启动 shell ------>用户输入命令

5）linux 内核移植的关键文件
arch/arm/plat-s5p6818/x6818/： 板级支持包

arch/arm/mach-s5p6818/cpu.c：调用了板级支持包中的大部分函数
重点函数 ： cpu_init_machine

```
#define MACHINE_START(_type,_name)            \
static const struct machine_desc __mach_desc_##_type   \
 __used                                \
 __attribute__((__section__(".arch.info.init"))) = {  \
     .nr      = MACH_TYPE_##_type,      \
     .name      = _name,
MACHINE_START(S5P6818, "s5p6818")
static const struct machine_desc  __mach_desc_S5P6818 ={
     .nr = MACH_TYPE_S5P6818,
     .name = "s5p6818",
     .fixup          =  cpu_fixup,
     .map_io           =  cpu_map_io,
     .init_irq       =  nxp_cpu_irq_init,
     .handle_irq     =  gic_handle_irq,
     .timer          = &nxp_cpu_sys_timer,
     .init_machine =  cpu_init_machine,
MACHINE_END
};
```

6）配置是如何影响编译的

make menuconfig 配置结果是以不同的 CONFIG_XXX 取不同值的形式保存在.config 文件中的

linux 内核源码几乎每个子目录下都有一个 Makefile ，
它决定了其所在目录下的文件（夹）是否参与内核的编译过程

obj-y += xxx.o//编译到内核中去
obj-m += xxx.o//编译为独立的模块文件
obj-   += xxx.o//不编译
obj-y += 子目录/ 进该子目录进行编译

obj-$(CONFIG_XXX) += xxx.o

1、内核模块

cp led_drv.c drivers/char/
vi drivers/char/Makefile
    obj-m   += led_drv.o
make uImage
make modules//编译生成独立的模块

前提：板子内核使用 nfs 方式挂载的/opt/rootfs 目录的根文件系统
    cp drivers/char/led_drv.ko   /opt/rootfs/

在开发板上执行
    cd /
    insmod led_drv.ko   //安装内核模块
    lsmod   //显示系统中已经动态安装的内核模块信息

```
rmmod led_drv //卸载内核模块
lsmod
```

2、通过 make menuconfig 来决定 led_drv.c 是否编译

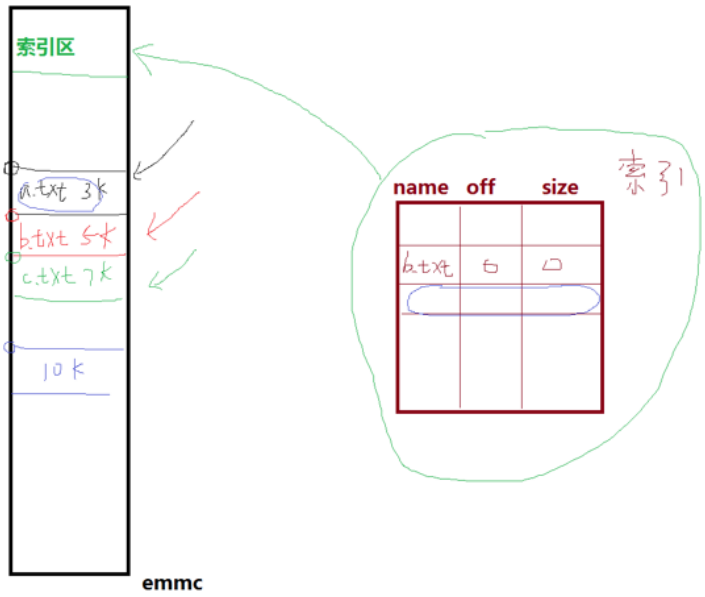
make menuconfig 时看到的菜单信息来源于 Kconfig 文件

linux 源码几乎每个子目录下都有一个 Kconfig 文件

关于 Kconfig 文件的语法规则：Documentation\kbuild\kconfig-language.txt

```
cp led_drv.c drivers/char/
vi drivers/char/Kconfig
    config MY_LEDS   #对应一个 CONFIG_MY_LEDS 变量
        tristate "My first led driver" #配置条目的内容就是“My first led driver"
                                        #tristate, 三态 该配置条目可以配置为<*/M/ >
                                        #bool ,     两种选择 [*/  ]
        default y   #该配置项的默认取值
        help #帮助信息，注意缩进对齐
         this is my frist driver
         fasdfasdf
         aaaaaaaaaaaaaaaaaaa
         bbbbbbbbbbbbbbb

make menuconfig
    Device Drivers  --->
      Character devices  --->
        <*> My first led driver
vi .config   观察变量的取值
    CONFIG_MY_LEDS = y
vi drivers/char/Makefile
    obj-$(CONFIG_MY_LEDS)              += led_drv.o
make uImage
```

3、根文件系统



文件系统 = 具体的数据 + 索引信息 + 操作索引的代码(kernel 中的 fs)

linux 支持多种类型的文件系统：ext4 nfs fat ntfs yaffs ....

根文件系统，就是 linux 内核加载使用的第一个文件系统

浏览 ubuntu 系统根目录下各子目录中存放的内容

3.1 准备根文件系统中必备的文件

1）准备板子必备的命令 ls rm cd ....

通过移植开源程序 busy box 来实现，busybox 被称作嵌入式领域中的瑞士军刀

```
cd /home/tarena/porting
cp /home/tarena/workdir/rootfs/busybox-1.23.2.tar.bz2 ./
tar xf busybox-1.23.2.tar.bz2
cd busybox-1.23.2/
make menuconfig
      Busybox Settings  --->
          Build Options  --->
              [*] Build shared libbusybox #共享库相关内容
              [*] Build with Large File Support (for accessing files > 2 GB) (NEW)
          Installation Options ("make install" behavior)--->//配置 install 行为
              What kind of applet links to install (as soft-links)  ---> #软链接
              (./_install) BusyBox installation prefix (NEW)<----+
vi Makefile                                                      |
  190 ARCH ?= arm                                                  |
  164 CROSS_COMPILE ?= arm-cortex_a9-linux-gnueabi-                |
make                                                               |
make install//把生成的命令放在指定的目录下------------------------+
cd _install/
```

2) 准备必要的库文件

mkdir lib

cp /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/sysroot/lib/libc.so.6 lib/ -d

cp /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/sysroot/lib/libc-2.18-2013.10.so lib/

cp /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/sysroot/lib/libm.so.6 lib/ -a

cp /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/sysroot/lib/libm-2.18-2013.10.so lib/

cp /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/sysroot/lib/ld-* lib/ -a

3) 准备必要的配置和脚本文件

mkdir etc

vi etc/inittab

```
::sysinit:/etc/init.d/rcS
::respawn:-/bin/sh
::shutdown:/bin/umount -a -r //关机执行
```

mkdir etc/init.d

vi etc/init.d/rcS//开机系统自动执行的命令，脚本文件

```
#!/bin/sh

#自动挂载/etc/fstab 文件中指定的分区设备
mount -a

#配置热插拔事件产生后要执行的动作
echo /sbin/mdev >/proc/sys/kernel/hotplug
#创建设备文件
mdev -s

echo "hello esd2002!"
```

注意：手工给该脚本文件添加可执行属性 chmod +x etc/init.d/rcS

vi etc/fstab

```
proc    /proc    proc    defaults    0    0
tmpfs   /tmp     tmpfs   defaults    0    0
sysfs   /sys     sysfs   defaults    0    0
tmpfs   /dev     tmpfs   defaults    0    0
```

4) 准备必要的目录

mkdir proc sys tmp dev

mkdir var mnt opt home

5) 准备必要的设备文件

sudo mknod dev/console c 5 1

sudo mknod dev/null c 1 3

6) 通过 nfs 方式验证以上根文件系统文件的有效性

sudo vi /etc/exports

/home/tarena/porting/busybox-1.23.2/_install *(rw,sync,no_root_squash)
sudo /etc/init.d/nfs-kernel-server restart

开发板上执行
setenv bootargs root=/dev/nfs
nfsroot=192.168.1.8:/home/tarena/porting/busybox-1.23.2/_install
ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 init=/linuxrc console=ttySAC0 maxcpus=1
lcd=wy070ml tp=gslx680
saveenv

7）linux 系统启动过程中需要两个设备文件
console：控制台设备
null ：黑洞设备
创建设备：ls /dev/console -l//查看 pc 机上的控制台设备号
sudo mknod dev/xxx c major minor//创建

4、部分实验原理
1）在开发板中 2 执行
rm a2.txt 等价于 busybox rm a2.txt
ls 等价于 busybox ls
2) readelf -d bin/busybox 分析得到它需要共享库文件,所有的软链接文件的指向
file bin/busybox

libm.so.6 数学库
libc.so.6 C 库
ld-linux.so: 加载器库（隐含）

以上三个库文件需要的是 ARM 版本
ARM 版本的库文件去哪找？去交叉编译工具包中找交叉编译工具包安装在哪个目录下,如何确定?

which arm-cortex_a9-linux-gnueabi-gcc

如何确定 libc.so.6 存在于交叉编译工具包的哪个子目录下呢?

find /opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/ -name "libc.so.6"
3）关于 cp 命令
cp -d, 保持原文件的软链接属性，一般用于单个文件拷贝
cp -a, 等价于-dR 保持拷贝的文件夹中所有文件的软连接属性 一般用于拷贝文件夹

4）一般指定的用户空间 1 号进程 linuxr 对应的代码 位于 busybox-1.23.2/init/init.c
```
int init_main(int argc UNUSED_PARAM, char **argv){
    parse_inittab(){
        parser_t *parser = config_open2("/etc/inittab", fopen_for_read)
    }
}
```
5）inittab 文件有着固定的格式
<id>:<runlevels>:<action>:<process>

id: 嵌入式环境通常忽略
runlevels: 运行级别 嵌入式环境中通用忽略
action: 何时执行 process
process: 要执行的程序或者脚本

6)fstab 文件的格式
#device mount-point type options dump fsck-order

device, 要挂载的分区设备
moount-point,挂载点
type, 要挂载的分区设备文件系统类型
options, 挂载时给定的参数
rw, 对该分区可以执行读写操作
defaults, 通常给默认参数
dump, 是否对该分区做备份
0, 不备份
fsck-order, 是否对分区做磁盘格式检查
0, 不检查

7) linux 系统启动过程中需要两个设备文件
console：控制台设备
null ：黑洞设备

mknod dev/xxx c major minor
xxx,设备文件名称
c, 创建的是字符设备
major, 主设备号
minor, 次设备号
ls -l xxx----->查看主设备号和次设备号

DAY05

回顾：
文件系统 = 数据 + 索引 + 代码（fs/）
根文件系统，linux 加载的第一个真实文件系统

准备根文件系统中必备的文件：
1) 通过移植 busybox 准备必要的可执行命令
make meunconfig
vi Makefile
ARCH
CROSS_COMPILE
make && make install
2) 准备了必要的库文件
需要的库文件系统去交叉编译工具包中找

which
find
cp -d/-a

3) 准备了必要的配置和脚本文件
vi etc/inittab
vi etc/init.d/rcS
vi etc/profile

4) 准备了必要的目录文件
proc sys dev tmp
var mnt opt home

5) 准备了必要的设备文件
mknod dev/console c xxx yyy
mknod dev/null c xxx yyy

6) 通过 nfs 方式挂载验证

1、hello 程序在开发板上的运行

vi hello.c
arm-cortex_a9-linux-gnueabi-gcc hello.c -o hello_arm
cp hello_arm busybox-1.23.2/_install/

在板子上执行
./hello_arm
出现的问题：libgcc_s.so.1: cannot open shared object file
解决方式：
cp
/opt/arm-cortex_a9-eabi-4.7-eglibc-2.18/arm-cortex_a9-linux-gnueabi/lib/arm-linux-gnueabi/libgcc_s.so.1 busybox-1.23.2/_install/lib/ -d

2、卸载模块模式的问题
cp /opt/rootfs/led_drv.ko busybox-1.23.2/_install/

insmod led_drv.ko
lsmod
rmmod led_drv
提示安装模块失败
解决方式：
mkdir /lib/modules/3.4.39-tarena -p
或者
mkdir /lib/modules/`uname -r ` -p //uname -a ---->查看 linux 版本
` `-->内容嵌套
uname -r ---->只输出版本号 3.4.39-tarena

3、通过 telnet 远程登录开发板(网线)
3.1 保证上位机中安装了 telnet 客户端软件
which telnet
3.2 保证下位机中安装了 telnetd 服务器软件
在板子上执行
which telnetd 查看是否存在服务器软件//服务器程序在~/porting/busybox-1.23.2 中,

用 make menuconfig 修改

3.3 保证开发板开机自启动 telnetd

开发板上执行

vi /etc/init.d/rcS 最后加入

telnetd

3.4 挂载 devpts 设备

开发板上执行

vi /etc/init.d/rcS

...

mdev -s

mkdir /dev/pts

mount -t devpts devpts /dev/pts

telnetd

3.5 用户名与密码的验证

ubuntu:

vi /etc/passwd //保存用户信息

用户名:密码:用户 ID:组 ID:用户描述信息:用户的家目录:会话的 shell

tarena: x :1000 :1000:tarena,,, :/home/tarena:/bin/bash

sudo vi /etc/shadow //密码字段

开发板：vi /etc/passwd

root::0:0:super user ....:/:/bin/sh

3.6 远程登录开发板

telnet 192.168.1.6

exit---->退出

4、根文件系统镜像的制作与使用

根文件系统镜像 = 准备的文件 + 存储时的索引信息整合到一个文件中去

linux 系统支持多种类型的文件系统(在 fs 中)：fat ext4 yaffs jffs ubifs ...

4.1 基于掉电不丢失设备的文件系统镜像

4.1.1 jffs2 文件系统

专门针对 norflash 芯片开发的文件系统

4.1.2 yaffs2 文件系统

专门针对 nandflash 芯片开发的文件系统

4.1.3 cramfs

它是由 linus 开发的

是一种只读的压缩文件系统

其特点：

1）镜像文件压缩存储

2）其管理的分区只能执行读操作 不能执行写和删除操作

实验步骤：操作索引的代码 数据+索引=镜像文件

1）配置内核 让内核支持 cramfs (将 fs/cramfs 代码编译到 uImage)

cd /home/tarena/porting/kernel/

make menuconfig

File systems --->

[*] Miscellaneous filesystems --->
<*> Compressed ROM file system support (cramfs)

make uImage

让开发板使用新内核

cp arch/arm/boot/uImage /tftpboot/

setenv bootcmd ping 192.168.1.8\;ping 192.168.1.8\;tftp 48000000 uImage \;bootm 48000000

2) 将根文件系统数据和索引制作成镜像文件

cd /home/tarena/porting/busybox-1.23.2/

sudo mkfs.cramfs _install/ rootfs.cramfs

3) 验证镜像的有效性

cp rootfs.cramfs /tftpboot/

tftp 48000000 rootfs.cramfs

mmc write 48000000 20800 0x1200

setenv bootargs root=/dev/mmcblk0p2 rootfstype=cramfs ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 init=/linuxrc console=ttySAC0 maxcpus=1 lcd=wy070ml tp=gslx680

saveenv

4) 验证 cramfs 的特点

只读文件系统： 在板子上执行 touch 1.txt

压缩：cd /home/tarena/porting/busybox-1.23.2

du -h _install/

du -h rootfs.cramfs

4.1.4 ext4

ext4 的文件系统常用于手机 目前广泛使用

在 linux 系统如何使用 U 盘

1) 分区

2) 格式化分区

3) 挂载 U 盘

mount -t vfat /dev/sda1 /mnt

4) 使用 U 盘

touch /mnt/1.txt

5) 卸载 U 盘

umount /mnt

注意：使用 ubuntu 18.04 系统的同学 需要修改 mke2fs.conf

cp ubuntu 12.04 mke2fs.conf /etc/

然后开始试验步骤

实验步骤：

1) 配置内核 支持 ext4 类型文件系统 (fs/ext4)

cd porting/kernel/

make menuconfig
File systems --->
<*> The Extended 4 (ext4) filesystem
make uImage
让开发板使用新内核

2) ext4 类型镜像的制作

cd /home/tarena/porting/busybox-1.23.2/

a)创建一个 8M 字节全 0 文件(类似 u 盘)

dd if=/dev/zero of=rootfs.ext4 bs=1024 count=8192

if=/dev/zero：input file 源数据,/dev/zero 虚拟设备,读到就是 0
of：output file 目标文件
bs=1k：每次从 zero 设备中读取 1024 字节的 0 值 写入到 rootfs.ext4
count=8192：连续读写 8192 字

du -h rootfs.ext4

b)mkfs.ext4 rootfs.ext4
格式化该 8M 分区为 ext4 类型文件系统

c)创建挂载点
mkdir esd1911

d)挂载分区设备
sudo mount -t ext4 rootfs.ext4 esd1911/

e)向该分区写入数据
sudo cp _install/* esd1911/ -a

f)卸载分区设备
sudo umount esd1911

3) 验证镜像的有效性

cp rootfs.ext4 /tftpboot/

在开发板上执行

tftp 48000000 rootfs.ext4
mmc write 48000000 20800 0x4000

setenv bootargs root=/dev/mmcblk0p2 rootfstype=ext4 ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 console=ttySAC0 init=/linuxrc maxcpus=1 lcd=wy070ml tp=gslx680

saveenv

4.2 基于 ram 文件系统镜像

4.2.1 ramdisk

将内存的一部分当磁盘使用
系统启动时将根文件系统中的文件加载到内存中去
内核启动后使用内存中的根文件系统文件
文件系统 I/O 速度更快
但是也消耗更多的内存空间

试验步骤：

1）配置内核 让内核支持 ramdisk

cd /home/tarena/porting/kernel/

make menuconfig

Device Drivers --->

[*] Block devices --->

<*> RAM block device support

(8192) Default RAM disk size (kbytes)

make uImage

cp arch/arm/boot/uImage /tftpboot

2) 生成镜像文件

与 ext4 基本相似，只有格式化不一样

格式化：mkfs.ext2 rootfs.ramdisk

3）验证有效性

将生成的镜像烧写到 emmc 中去

这里由于要使用 ext4 镜像在上个试验中已经烧到 emmc 中去了 烧写步骤可以跳过

加载操作系统时也需要将根文件系统加载到内存中去

setenv bootcmd ping 192.168.1.8 \; ping 192.168.1.8 \;tftp 48000000 uImage \;mmc read 49000000 20800 4000\;bootm 48000000

告诉内核具体去内存哪个地址上找根文件系统数据

setenv bootargs root=/dev/ram rw initrd=0x49000000,8M ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0 console=ttySAC0 init=/linuxrc maxcpus=1 lcd=wy070ml tp=gslx680

saveenv

如果想把 mmcblk0p3 分区使用起来

在开发板上：

ls /dev/mmcblk0p*------>查看所有分区设备

1）第一次使用时要格式化该分区（格式化一次就 OK 后续不需要再格式化）

cat /proc/partitions

mkfs.vfat /dev/mmcblk0p3

2) 挂载该分区

mount /dev/mmcblk0p3 /mnt/

df------>查看分区使用情况

3）使用该分区

touch /mnt/1.txt

echo fadsfasfa >/mnt/1.txt

sync //保证数据写入完毕

重启开发板

重新挂载 mount /dev/mmcblk0p3 /mnt/ （自动挂载：rcS---->/etc/init.d/rcS）

rcS:

加入：mount -t vfat /dev/mmcblk0p3 /mnt

4.3 网络文件系统

nfs 方式可以挂载:

挂载根文件系统

setenv bootargs root=/dev/nfs nfsroot=serverip:路径 .....

挂载应用文件系统：在开发板上执行

mount -o nolock -t nfs 192.168.1.8:路径 /mnt

-o nolock:不使用文件锁