# 实验一 交叉工具链的安装

**【实验目的】**

了解交叉工具链的编译过程及其使用。

说明：在实验中**命令行提示符 为“$”表示在主机上运行，“#”表示在目标板上运行**

**【实验环境】**

1. ubuntu 12.04发行版
2. FS4412平台

**【实验步骤】**

1. 如果要自己编译工具链，从以下链接下载源码

crosstools-ng下载地址

<http://ymorin.is-a-geek.org/download/crosstool-ng/>

同时对每一个版本都有相应的补丁我们尽量把这些补丁打上，这些补丁的下载地址是

<http://ymorin.is-a-geek.org/download/crosstool-ng/01-fixes/>

1. 解压工具链压缩包

$ cd ~

$ mkdir toolchain

$ cd toolchain

将第一天/工具/gcc-4.6.4.tar.xz拷贝到toolchain目录下并解压

$ tar xvf gcc-4.6.4.tar.xz

1. 环境变量的添加

修改文件/etc/bash.bashrc添加如下内容

export PATH=$PATH:/home/linux/toolchain/gcc-4.6.4/bin

重启配置文件

$ source /etc/bash.bashrc

1. 工具链的测试

$ arm-none-linux-gnueabi-gcc –v

Using built-in specs.

COLLECT\_GCC=arm-none-linux-gnueabi-gcc

COLLECT\_LTO\_WRAPPER=/home/david/Exynos4412/toolchain/gcc-4.6.4/bin/../libexec/gcc/arm-arm1176jzfssf-linux-gnueabi/4.6.4/lto-wrapper

Target: arm-arm1176jzfssf-linux-gnueabi

Configured with: /work/builddir/src/gcc-4.6.4/configure --build=i686-build\_pc-linux-gnu --host=i686-build\_pc-linux-gnu --target=arm-arm1176jzfssf-linux-gnueabi --prefix=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4 --with-sysroot=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/sysroot --enable-languages=c,c++ --with-arch=armv6zk --with-cpu=arm1176jzf-s --with-tune=arm1176jzf-s --with-fpu=vfp --with-float=softfp --with-pkgversion='crosstool-NG hg+default-2685dfa9de14 - tc0002' --disable-sjlj-exceptions --enable-\_\_cxa\_atexit --disable-libmudflap --disable-libgomp --disable-libssp --disable-libquadmath --disable-libquadmath-support --with-gmp=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-mpfr=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-mpc=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-ppl=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-cloog=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-libelf=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --enable-threads=posix --enable-target-optspace --without-long-double-128 --disable-nls --disable-multilib --with-local-prefix=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/sysroot --enable-c99 --enable-long-long

Thread model: posix

gcc version 4.6.4 (crosstool-NG hg+default-2685dfa9de14 - tc0002)

这样我们的交叉工具链就安装好了

# 实验二 u-boot的烧写及使用

**【实验目的】**

了解u-boot的常用命令和linux内核的引导。

**【实验环境】**

1. ubuntu 12.04发行版
2. u-boot-2010.03
3. FS4412平台
4. 交叉编译器 arm-none-linux-gnueabi-gcc

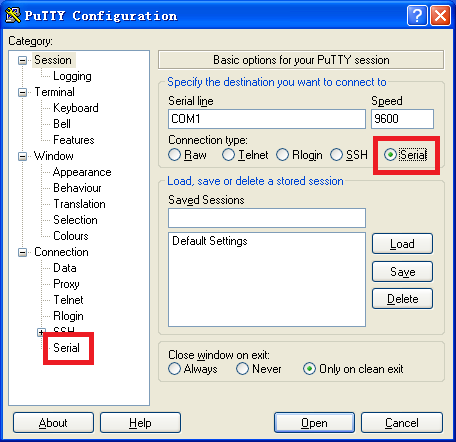
**【实验步骤】**

1. SD启动盘制作
   1. 将第一天/工具/ sd\_fusing拷贝到Linux下
   2. 将SD卡插入电脑并识别
   3. 进入sdfuse\_d执行如下操作

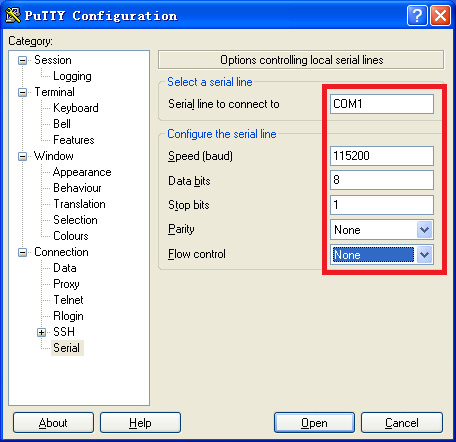
$ sudo ./mkuboot.sh /dev/sdb

* 1. 在SD卡中创建目录sdupdate并把第一天/镜像中的u-boot-fs4412.bin拷贝到这个目录下

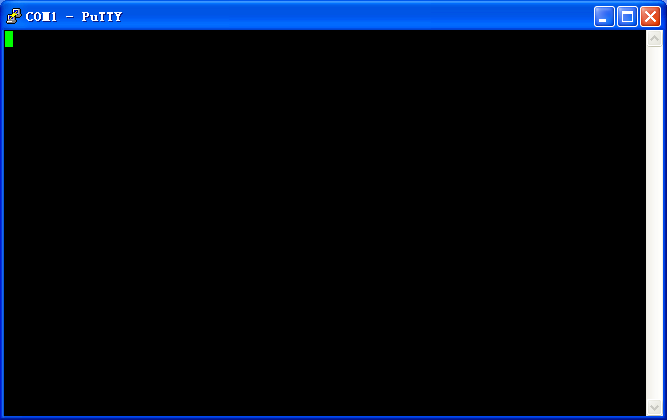
1. u-boot的烧写
   1. 连接串口和板子，运行串口通信程序(putty第一天工具中)



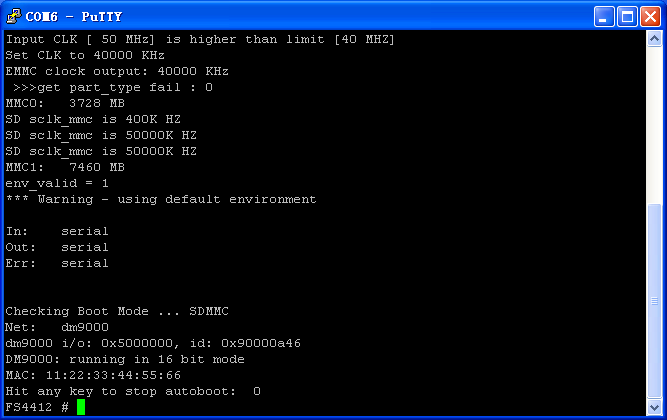
选择右上角的”Serial”，然后点击左下角的”Serial”



按照自己的主机的情况选择COM口其他必须一直，然后点击open打开串口



* 1. 关闭开发板电源，将拨码开关SW1调至(1000)(SD启动模式)后打开电源
  2. 将刚才做好的SD启动盘插入SD卡插槽
  3. 重新打开开发板能够看到如下界面



**在倒计时时按任意键**

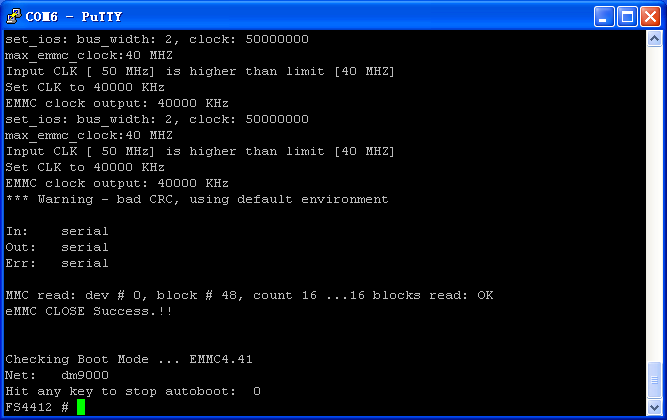
* 1. 烧写

在终端上执行

sdfuse flashall

等待终端无输出是表示烧写结束

* 1. 关闭开发板电源，将拨码开关SW1调至0110(EMMC启动模式)后打开电源可以看到如下界面表示烧写成功



**在倒计时时按任意键**

1. 通过网络加载内核和文件系统
   1. 将 第一天/镜像文件/uImage拷贝到ubuntu的 /tftpboot下
   2. 将 第一天/镜像文件/rootfs.tar.xz拷贝到ubuntu的 /source下并解压
   3. 将 第一天/镜像文件/exynos4412-fs4412.dtb拷贝到ubuntu的 /tftpboot下
   4. 修改虚拟机nfs配置文件/etc/exports，添加如下内容并重启nfs服务

/source/rootfs \*(rw,sync,no\_subtree\_check,no\_root\_squash)

* 1. 重新驱动nfs服务

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

* 1. 设置启动参数

# setenv serverip 192.168.9.120

# setenv ipaddr 192.168.9.233

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;bootm 41000000 - 42000000

#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.9.233

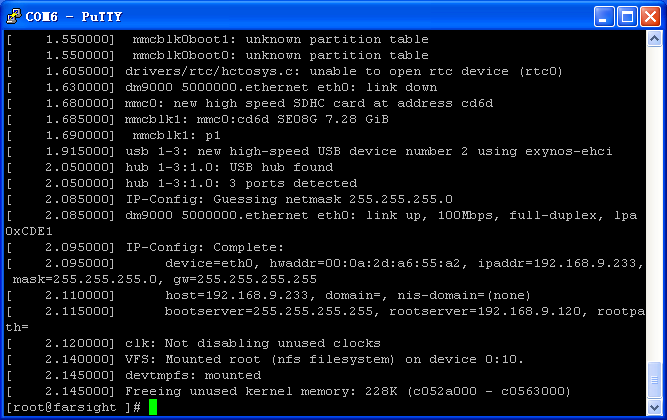
# saveenv

注意：192.168.9.120 对应Ubuntu的ip

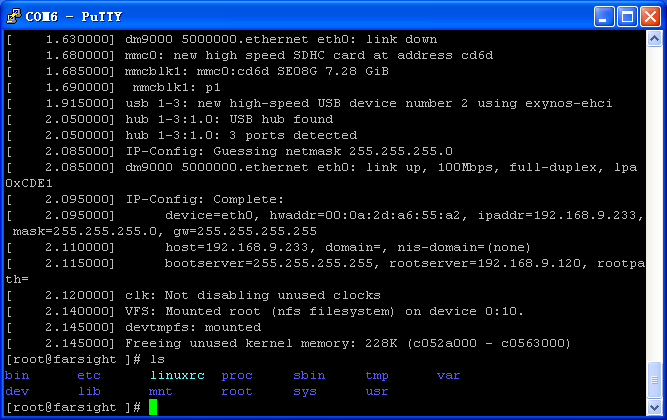
192.168.9.233 对应板子的ip

这两个ip应该根据自己的实际情况适当修改

* 1. 启动开发板看到如下现象表示成功通过网络挂载：



**这是可以输入一些Linux的命令测试**



1. 从EMMC加载内核和文件系统
   1. 拷贝第一天/镜像文件/ramdisk.img拷贝到虚拟机/tftpboot目录下
   2. 烧写内核镜像到EMMC上

# tftp 41000000 uImage

# movi write kernel 41000000

* 1. 烧写设备树文件到EMMC上

# tftp 41000000 exynos4412-fs4412.dtb

# movi write dtb 41000000

* 1. 烧写文件系统镜像到EMMC上

# tftp 41000000 ramdisk.img

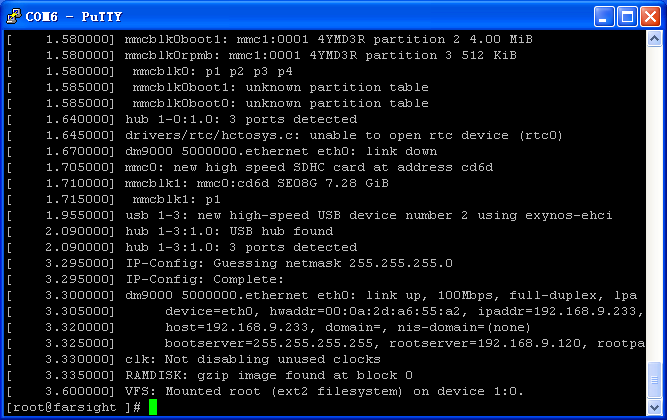
# movi write rootfs 41000000 300000

* 1. 设置启动参数

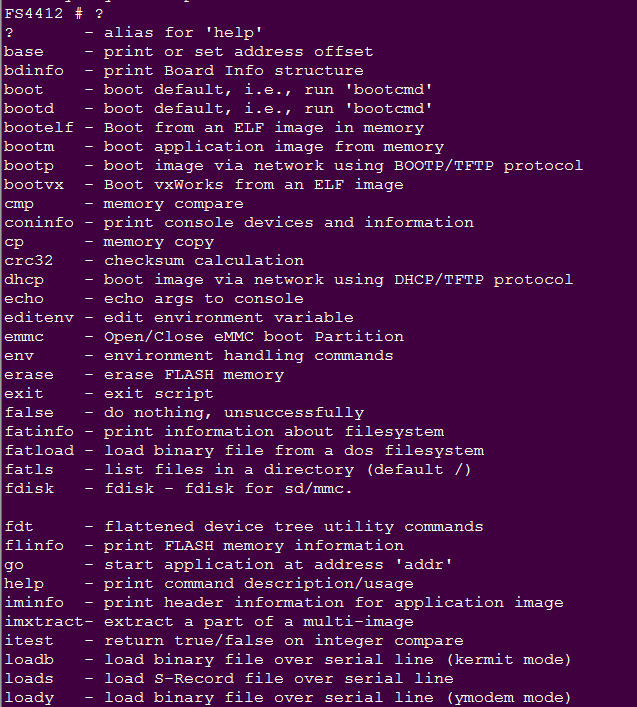
# setenv bootcmd movi read kernel 41000000\;movi read dtb 42000000\;movi read rootfs 43000000 300000\;bootm 41000000 43000000 42000000

# saveenv

* 1. 重新启动开发板，u-boot自动加载、执行内核



1. 其他命令练习



1. 交叉编译和交叉调试
   1. 将u-boot启动参数修改为网络启动

# setenv serverip 192.168.9.120

# setenv ipaddr 192.168.9.233

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;bootm 41000000 - 42000000

#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.9.233

# saveenv

注意：192.168.9.120 对应Ubuntu的ip

192.168.9.233 对应板子的ip

这两个ip应该根据自己的实际情况适当修改

* 1. 编辑程序源码myapp.c(自己写一个简单的c程序)
  2. 交叉编译后复制到/source/rootfs (编译时添加选项-g)

$ arm-none-linux-gnueabi-gcc myapp.c –o myapp –g

$ cp myapp /source/rootfs

* 1. 复制gdbserver到/source/rootfs/bin (gdbserver在交叉工具链中找，路径为：/home/linux/toolchain/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/debug-root/bin)
  2. 在开发板上如下运行

# gdbserver 192.168.9.233:12345 myapp &

* 1. 在主机上运行交叉调试器

$ arm-none-linux-gnueabi-gdb myapp

* 1. 在交叉调试器下和开发板gdbserver建立连接

(gdb) target remote 192.168.9.233:12345

设置断点，输入命令c开始调试程序(注意观察串口输出)

# 实验三 u-boot-2013.01的移植

**【实验目的】**

了解u-boot 的代码结构及移植方法。

**【实验环境】**

1. ubuntu 10.10发行版
2. u-boot-2013.01
3. FS\_4412平台 （EXYNOS 4412）
4. 交叉编译器 arm-none-linux-gnueabi-gcc

**【实验步骤】**

**一、建立自己的平台**

1. **下载源码**

我们可以在下面这个网站上下载最新的和以前任一版本的uboot

[ftp://ftp.denx.de/pub/u-boot/](http://www.icdev.com.cn/batch.viewlink.php?itemid=1694)

1. **解压uboot源码并进入目录**

$ tar xvf u-boot-2013.01.tar.bz2

$ cd u-boot-2013.01R

1. **指定交叉编译工具链**

$ vim Makefile

把

ifeq ($(HOSTARCH),$(ARCH))

CROSS\_COMPILE ?=

#endif

下添加

ifeq (arm,$(ARCH))

CROSS\_COMPILE ?= arm-none-linux-gnueabi-

#endif

1. **指定产品CPU**

我们产品用的CPU是 exynos 4412

查看u-boot源码该CPU 是否已支持

U-boot已支持，见 arch/arm/cpu/armv7/exynos/

1. **指定产品BOARD**

找一个最类似的board配置修改, 这里我们参考的是board/samsung/origen/

$ cp -rf board/samsung/origen/ board/samsung/fs4412

$ mv board/samsung/fs4412/origen.c board/samsung/fs4412/fs4412.c

$ vim board/samsung/fs4412/Makefile

修改 origen.o 为 fs4412.o

$ cp include/configs/origen.h include/configs/fs4412.h

$ vim include/configs/fs4412.h

修改

#define CONFIG\_SYS\_PROMPT "ORIGEN #"

为

#define CONFIG\_SYS\_PROMPT "fs4412 #"

修改

#define CONFIG\_IDENT\_STRING for ORIGEN

为

#define CONFIG\_IDENT\_STRING for fs4412

#vim boards.cfg

参考

origen arm armv7 origen samsung exynos

并在后面新增

fs4412 arm armv7 fs4412 samsung exynos

1. **编译u-boot**

$ make distclean

$ make fs4412\_config

$ make

编译完成后生成的u-boot.bin就是可执行的镜像文件。

但是该文件还不能在我们板子上运行，我们需要对u-boot源代码进行相应的修改。

1. **实现能看到串口终端信息**
2. **确认第一条指令有运行到 （点灯法）**

* 在arch/arm/cpu/armv7/start.S 134 行后添加点灯程序

#if 1

ldr r0, =0x11000c40 @GPK2\_7 led2

ldr r1, [r0]

bic r1, r1, #0xf0000000

orr r1, r1, #0x10000000

str r1, [r0]

ldr r0, =0x11000c44

mov r1,#0xff

str r1, [r0]

#endif

* 添加三星加密方式

exynos 需要三星提供的初始引导加密后，我们的u-boot,才能被引导运行

$cp sdfuse\_q u-boot-2013.01 -rf

注：sdfuse\_q 三星提供的加密处理

$cp CodeSign4SecureBoot u-boot-2013.01 -rf

注：CodeSign4SecureBoot 三星提供的安全启动方式

* 修改Makefile

$vim Makefile

修改实现sdfuse\_q的编译

在

$(obj)u-boot.bin: $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

$(BOARD\_SIZE\_CHECK)

下添加

@#./mkuboot

@split -b 14336 u-boot.bin bl2

@+make -C sdfuse\_q/

@#cp u-boot.bin u-boot-4212.bin

@#cp u-boot.bin u-boot-4412.bin

@#./sdfuse\_q/add\_sign

@./sdfuse\_q/chksum

@./sdfuse\_q/add\_padding

@rm bl2a\*

@echo

**注意是tab键缩进的，否则makefile编译报错**

**注意如果执行了make distclean 需重新拷贝CodeSign4SecureBoot**

* 拷贝编译脚本

$ cp build.sh u-boot-2013.01

$ chmod 777 u-boot-2013.01/ build.sh

$ ./buildsh

注：build.sh 脚本方式完成自动添加加密方式，

编译生成所需文件u-boot\_fs4412.bin

烧写新的u-boot\_fs4412.bin

复位，发现灯有点亮，说明 我们的u-boot有运行到

1. **实现串口输出**

修改lowlevel\_init.S文件

$vim board/samsung/fs4412/lowlevel\_init.S

* 添加临时栈

在

lowlevel\_init:

后添加

ldr sp,=0x02060000 @use iRom stack in bl2

* 添加关闭看门狗代码

在

beq wakeup\_reset

后添加

#if 1 /\*for close watchdog \*/

/\* PS-Hold high \*/

ldr r0, =0x1002330c

ldr r1, [r0]

orr r1, r1, #0x300

str r1, [r0]

ldr r0, =0x11000c08

ldr r1, =0x0

str r1, [r0]

/\* Clear MASK\_WDT\_RESET\_REQUEST \*/

ldr r0, =0x1002040c

ldr r1, =0x00

str r1, [r0]

#endif

* 添加串口初始化代码

在uart\_asm\_init: 的

str r1, [r0, #EXYNOS4\_GPIO\_A1\_CON\_OFFSET]

后添加

ldr r0, =0x10030000

ldr r1, =0x666666

ldr r2, =CLK\_SRC\_PERIL0\_OFFSET

str r1, [r0, r2]

ldr r1, =0x777777

ldr r2, =CLK\_DIV\_PERIL0\_OFFSET

str r1, [r0, r2]

注释掉trustzone初始化

注释掉

bl uart\_asm\_init

下的

bl tzpc\_init

重新编译u-boot

$ ./build.sh

烧写新的u-boot\_fs4412.bin

复位会看到串口信息



1. **网卡移植**
2. **添加网络初始化代码**

$ vim board/samsung/fs4412/fs4412.c

在struct exynos4\_gpio\_part2 \*gpio2; 后添加

#ifdef CONFIG\_DRIVER\_DM9000

#define EXYNOS4412\_SROMC\_BASE 0X12570000

#define DM9000\_Tacs (0x1)

#define DM9000\_Tcos (0x1)

#define DM9000\_Tacc (0x5)

#define DM9000\_Tcoh (0x1)

#define DM9000\_Tah (0xC)

#define DM9000\_Tacp (0x9)

#define DM9000\_PMC (0x1)

struct exynos\_sromc {

unsigned int bw;

unsigned int bc[6];

};

/\*

\* s5p\_config\_sromc() - select the proper SROMC Bank and configure the

\* band width control and bank control registers

\* srom\_bank - SROM

\* srom\_bw\_conf - SMC Band witdh reg configuration value

\* srom\_bc\_conf - SMC Bank Control reg configuration value

\*/

void exynos\_config\_sromc(u32 srom\_bank, u32 srom\_bw\_conf, u32 srom\_bc\_conf)

{

unsigned int tmp;

struct exynos\_sromc \*srom = (struct exynos\_sromc \*)(EXYNOS4412\_SROMC\_BASE);

/\* Configure SMC\_BW register to handle proper SROMC bank \*/

tmp = srom->bw;

tmp &= ~(0xF << (srom\_bank \* 4));

tmp |= srom\_bw\_conf;

srom->bw = tmp;

/\* Configure SMC\_BC register \*/

srom->bc[srom\_bank] = srom\_bc\_conf;

}

static void dm9000aep\_pre\_init(void)

{

unsigned int tmp;

unsigned char smc\_bank\_num = 1;

unsigned int smc\_bw\_conf=0;

unsigned int smc\_bc\_conf=0;

/\* gpio configuration \*/

writel(0x00220020, 0x11000000 + 0x120);

writel(0x00002222, 0x11000000 + 0x140);

/\* 16 Bit bus width \*/

writel(0x22222222, 0x11000000 + 0x180);

writel(0x0000FFFF, 0x11000000 + 0x188);

writel(0x22222222, 0x11000000 + 0x1C0);

writel(0x0000FFFF, 0x11000000 + 0x1C8);

writel(0x22222222, 0x11000000 + 0x1E0);

writel(0x0000FFFF, 0x11000000 + 0x1E8);

smc\_bw\_conf &= ~(0xf<<4);

smc\_bw\_conf |= (1<<7) | (1<<6) | (1<<5) | (1<<4);

smc\_bc\_conf = ((DM9000\_Tacs << 28)

| (DM9000\_Tcos << 24)

| (DM9000\_Tacc << 16)

| (DM9000\_Tcoh << 12)

| (DM9000\_Tah << 8)

| (DM9000\_Tacp << 4)

| (DM9000\_PMC));

exynos\_config\_sromc(smc\_bank\_num,smc\_bw\_conf,smc\_bc\_conf);

}

#endif

在gd->bd->bi\_boot\_params = (PHYS\_SDRAM\_1 + 0x100UL); 后添加

#ifdef CONFIG\_DRIVER\_DM9000

dm9000aep\_pre\_init();

#endif

在文件末尾添加

#ifdef CONFIG\_CMD\_NET

int board\_eth\_init(bd\_t \*bis)

{

int rc = 0;

#ifdef CONFIG\_DRIVER\_DM9000

rc = dm9000\_initialize(bis);

#endif

return rc;

}

#endif

1. **修改配置文件添加网络相关配置**

$ vim include/configs/fs4412.h

修改

#undef CONFIG\_CMD\_PING

为

#def ine CONFIG\_CMD\_PING

修改

#undef CONFIG\_CMD\_NET

为

#def ine CONFIG\_CMD\_NET

在文件末尾

#endif /\* \_\_CONFIG\_H \*/

前面添加

#ifdef CONFIG\_CMD\_NET

#define CONFIG\_NET\_MULTI

#define CONFIG\_DRIVER\_DM9000 1

#define CONFIG\_DM9000\_BASE 0x05000000

#define DM9000\_IO CONFIG\_DM9000\_BASE

#define DM9000\_DATA (CONFIG\_DM9000\_BASE + 4)

#define CONFIG\_DM9000\_USE\_16BIT

#define CONFIG\_DM9000\_NO\_SROM 1

#define CONFIG\_ETHADDR 11:22:33:44:55:66

#define CONFIG\_IPADDR 192.168.9.200

#define CONFIG\_SERVERIP 192.168.9.120

#define CONFIG\_GATEWAYIP 192.168.9.1

#define CONFIG\_NETMASK 255.255.255.0

#endif

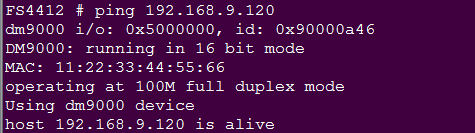
1. **重新编译u-boot**

$ ./build.sh

烧写新的u-boot\_fs4412.bin

复位后

# ping 192.168.9.120



1. **FLASH移植 （EMMC）**
2. **初始化EMMC**

$cp movi.c arch/arm/cpu/armv7/exynos/

$vim arch/arm/cpu/armv7/exynos/Makefile

在pinmux.o 后添加movi.o

修改板级文件

$vim board/samsung/fs4412/fs4412.c

在

#include <asm/arch/mmc.h>

后面添加

#include <asm/arch/clk.h>

#include "origen\_setup.h"

在

#ifdef CONFIG\_GENERIC\_MMC

后面添加

u32 sclk\_mmc4; /\*clock source for emmc controller\*/

#define \_\_REGMY(x) (\*((volatile u32 \*)(x)))

#define CLK\_SRC\_FSYS \_\_REGMY(EXYNOS4\_CLOCK\_BASE + CLK\_SRC\_FSYS\_OFFSET)

#define CLK\_DIV\_FSYS3 \_\_REGMY(EXYNOS4\_CLOCK\_BASE + CLK\_DIV\_FSYS3\_OFFSET)

int emmc\_init()

{

u32 tmp;

u32 clock;

u32 i;

/\* setup\_hsmmc\_clock \*/

/\* MMC4 clock src = SCLKMPLL \*/

tmp = CLK\_SRC\_FSYS & ~(0x000f0000);

CLK\_SRC\_FSYS = tmp | 0x00060000;

/\* MMC4 clock div \*/

tmp = CLK\_DIV\_FSYS3 & ~(0x0000ff0f);

clock = get\_pll\_clk(MPLL)/1000000;

for(i=0 ; i<=0xf; i++) {

sclk\_mmc4=(clock/(i+1));

if(sclk\_mmc4 <= 160) //200

{

CLK\_DIV\_FSYS3 = tmp | (i<<0);

break;

}

}

emmcdbg("[mjdbg] sclk\_mmc4:%d MHZ; mmc\_ratio: %d\n",sclk\_mmc4,i);

sclk\_mmc4 \*= 1000000;

/\*

\* MMC4 EMMC GPIO CONFIG

\*

\* GPK0[0] SD\_4\_CLK

\* GPK0[1] SD\_4\_CMD

\* GPK0[2] SD\_4\_CDn

\* GPK0[3:6] SD\_4\_DATA[0:3]

\*/

writel(readl(0x11000048)&~(0xf),0x11000048); //SD\_4\_CLK/SD\_4\_CMD pull-down enable

writel(readl(0x11000040)&~(0xff),0x11000040);//cdn set to be output

writel(readl(0x11000048)&~(3<<4),0x11000048); //cdn pull-down disable

writel(readl(0x11000044)&~(1<<2),0x11000044); //cdn output 0 to shutdown the emmc power

writel(readl(0x11000040)&~(0xf<<8)|(1<<8),0x11000040);//cdn set to be output

udelay(100\*1000);

writel(readl(0x11000044)|(1<<2),0x11000044); //cdn output 1

writel(0x03333133, 0x11000040);

writel(0x00003FF0, 0x11000048);

writel(0x00002AAA, 0x1100004C);

#ifdef CONFIG\_EMMC\_8Bit

writel(0x04444000, 0x11000060);

writel(0x00003FC0, 0x11000068);

writel(0x00002AAA, 0x1100006C);

#endif

#ifdef USE\_MMC4

smdk\_s5p\_mshc\_init();

#endif

}

将 int board\_mmc\_init(bd\_t \*bis)函数内容改写为

int board\_mmc\_init(bd\_t \*bis)

{

int i, err;

#ifdef CONFIG\_EMMC

err = emmc\_init();

#endif

return err;

}

在末尾添加

#ifdef CONFIG\_BOARD\_LATE\_INIT

#include <movi.h>

int chk\_bootdev(void)//mj for boot device check

{

char run\_cmd[100];

struct mmc \*mmc;

int boot\_dev = 0;

int cmp\_off = 0x10;

ulong start\_blk, blkcnt;

mmc = find\_mmc\_device(0);

if (mmc == NULL)

{

printf("There is no eMMC card, Booting device is SD card\n");

boot\_dev = 1;

return boot\_dev;

}

start\_blk = (24\*1024/MOVI\_BLKSIZE);

blkcnt = 0x10;

sprintf(run\_cmd,"emmc open 0");

run\_command(run\_cmd, 0);

sprintf(run\_cmd,"mmc read 0 %lx %lx %lx",CFG\_PHY\_KERNEL\_BASE,start\_blk,blkcnt);

run\_command(run\_cmd, 0);

/\* switch mmc to normal paritition \*/

sprintf(run\_cmd,"emmc close 0");

run\_command(run\_cmd, 0);

return 0;

}

int board\_late\_init (void)

{

int boot\_dev =0 ;

char boot\_cmd[100];

boot\_dev = chk\_bootdev();

if(!boot\_dev)

{

printf("\n\nChecking Boot Mode ... EMMC4.41\n");

}

return 0;

}

#endif

1. **添加相关命令**

$ cp cmd\_movi.c common/

$ cp cmd\_mmc.c common/

$ cp cmd\_mmc\_fdisk.c common/

修改Makefile

$ vim common/Makefile

在

COBJS-$(CONFIG\_CMD\_MMC) += cmd\_mmc.o

后添加

COBJS-$(CONFIG\_CMD\_MMC) += cmd\_mmc\_fdisk.o

COBJS-$(CONFIG\_CMD\_MOVINAND) += cmd\_movi.o

添加驱动

$ cp mmc.c drivers/mmc/

$ cp s5p\_mshc.c drivers/mmc/

$ cp mmc.h include/

$ cp movi.h include/

$ cp s5p\_mshc.h include/

修改Makefile

$vim drivers/mmc/Makefile

添加

COBJS-$(CONFIG\_S5P\_MSHC) += s5p\_mshc.o

1. **添加EMMC相关配置**

$vim include/configs/fs4412.h

添加

#define CONFIG\_EVT1 1 /\* EVT1 \*/

#ifdef CONFIG\_EVT1

#define CONFIG\_EMMC44\_CH4 //eMMC44\_CH4 (OMPIN[5:1] = 4)

#ifdef CONFIG\_SDMMC\_CH2

#define CONFIG\_S3C\_HSMMC

#undef DEBUG\_S3C\_HSMMC

#define USE\_MMC2

#endif

#ifdef CONFIG\_EMMC44\_CH4

#define CONFIG\_S5P\_MSHC

#define CONFIG\_EMMC 1

#define USE\_MMC4

/\* #define CONFIG\_EMMC\_8Bit \*/

#define CONFIG\_EMMC\_EMERGENCY

/\*#define emmcdbg(fmt,args...) printf(fmt ,##args) \*///for emmc debug

#define emmcdbg(fmt,args...)

#endif

#endif /\*end CONFIG\_EVT1\*/

#define CONFIG\_CMD\_MOVINAND

#define CONFIG\_CLK\_1000\_400\_200

#define CFG\_PHY\_UBOOT\_BASE CONFIG\_SYS\_SDRAM\_BASE + 0x3e00000

#define CFG\_PHY\_KERNEL\_BASE CONFIG\_SYS\_SDRAM\_BASE + 0x8000

#define BOOT\_MMCSD 0x3

#define BOOT\_EMMC43 0x6

#define BOOT\_EMMC441 0x7

#define CONFIG\_BOARD\_LATE\_INIT

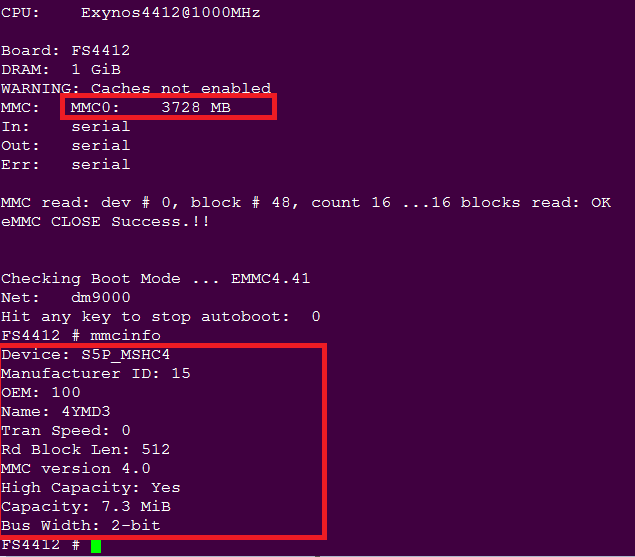
1. **重新编译u-boot**

$ ./build.sh

烧写新的u-boot\_fs4412.bin

复位后

# mmcinfo



# 实验四 内核的配置和编译

**【实验目的】**

了解内核的编译过程及配置选项的内容

说明：在本系统移植课程实验中命令行提示符 “$”表示是在主机上执行，“#”表示在目标板执行

**【实验环境】**

* 主机：ubuntu 12.04发行版
* 目标机：FS4412平台
* 交叉编译工具：arm-none-linux-gnueabi-gcc

**【实验步骤】**

* 解压内核

将linux-3.14.tar.xz拷贝到/home/linux下并解压

$tar xvf linux-3.14.tar.xz

$ cd linux-3.14

* 修改内核顶层目录下的Makefile

$ vim Makefile

修改：

ARCH ?= $(SUBARCH)

CROSS\_COMPILE ?= $(CONFIG\_CROSS\_COMPILE:"%"=%)

为：

ARCH ?= arm

CROSS\_COMPILE ?= arm-none-linux-gnueabi-

* 导入默认配置

$ make exynos\_defconfig

* 配置内核

$ make menuconfig

System Type --->

(2) S3C UART to use for low-level messages

该命令执行时会弹出一个菜单，我们可以对内核进行详细的配置。这里我们先查看一下，内核都提供了那些功能！

* 编译内核

$ make uImage

通过上述操作我们能够在arch/arm/boot目录下生成一个uImage文件，这就是经过压缩的内核镜像。

如果编译过程中提示缺少mkimage工具，需将第二天编译的uboot源码中的tools/mkimage拷贝到ubuntu的/usr/bin目录下

$ cp u-boot-2013.01/tools/mkimage /usr/bin

修改设备树文件

生成设备树文件，以参考板origen的设备数文件为参考。

$ cp arch/arm/boot/dts/exynos4412-origen.dts arch/arm/boot/dts/exynos4412-fs4412.dts

添加新文件需修改Makefile才能编译

$ vim arch/arm/boot/dts/Makefile

在

exynos4412-origen.dtb \

下添加如下内容

exynos4412-fs4412.dtb \

* 编译设备树文件

$ make dtbs

* 拷贝内核和设备树文件到/tftpboot目录下

$ cp arm/arm/boot/uImage /tftpboot

$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/

* 修改uboot启动参数

重启板子在系统倒计时是按任意键结束启动，输入如下内容修改uboot环境变量：

# setenv serverip 192.168.9.120

# setenv ipaddr 192.168.9.233

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;bootm 41000000 – 42000000

#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.9.233

# saveenv

注意：192.168.9.120 对应Ubuntu的ip

192.168.9.233 对应板子的ip

这两个ip应该根据自己的实际情况适当修改

重启开发板查看现象

# 实验五 网卡驱动的移植

**【实验目的】**

通过上面的实验我们能够获得一个内核，但是这个内核只是一个最基本的配置，很多的功能并没有包含。

网卡是嵌入式产品最常用的设备，这里我们需要完成网卡驱动的移植。FS4412使用的是DM9000网卡，我们通过这个实验能够了解如何在内核中添加网卡驱动及网络功能的基本配置。

说明：在本系统移植课程实验中命令行提示符 “$”表示是在主机上执行，“#”表示在目标板执行

**【实验环境】**

* 主机：ubuntu 12.04发行版
* 目标机：FS4412平台
* 交叉编译工具：arm-none-linux-gnueabi-gcc

**【实验步骤】**

* 设备树文件修改:

$ vim arch/arm/boot/dts/exynos4412-fs4412.dts

添加如下内容：

srom-cs1@5000000 {

compatible = "simple-bus";

#address-cells = <1>;

#size-cells = <1>;

reg = <0x5000000 0x1000000>;

ranges;

ethernet@5000000 {

compatible = "davicom,dm9000";

reg = <0x5000000 0x2 0x5000004 0x2>;

interrupt-parent = <&gpx0>;

interrupts = <6 4>;

davicom,no-eeprom;

mac-address = [00 0a 2d a6 55 a2];

};

};

* 修改文件driver/clk/clk.c

修改

static bool clk\_ignore\_unused;

为

static bool clk\_ignore\_unused = true;

* 配置内核：

make menuconfig

[\*] Networking support --->

Networking options --->

<\*> Packet socket

<\*> Unix domain sockets

[\*] TCP/IP networking

[\*] IP: kernel level autoconfiguration

Device Drivers --->

[\*] Network device support --->

[\*] Ethernet driver support (NEW) --->

<\*> DM9000 support

File systems --->

[\*] Network File Systems (NEW) --->

<\*> NFS client support

[\*] NFS client support for NFS version 3

[\*] NFS client support for the NFSv3 ACL protocol extension

[\*] Root file system on NFS

* 编译内核和设备树

$ make uImage

$ make dtbs

* 测试：

拷贝内核和设备树文件到/tftpboot目录下

$ cp arm/arm/boot/uImage /tftpboot

$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/

启动开发板，修改内核启动参数，通过NFS方式挂载根文件系统

# 实验六 LED驱动的移植

**【实验目的】**

1. 驱动程序的编译
2. 应用程序如何打开/操作设备

**【实验环境】**

* 主机：ubuntu 12.04发行版
* 目标机：FS4412平台
* 交叉编译工具：arm-none-linux-gnueabi-gcc

**【实验步骤】**

1. 添加驱动文件

将实验代码Led\_test/fs4412\_led\_drv.c拷贝到drivers/char下

1. 修改drivers/char/Kconfig

在menu "Character devices"下面

添加如下内容：

config FS4412\_LED

tristate "FS4412 LED Device Support"

depends on ARCH\_EXYNOS4

help

support led device on FS4412 develop board

1. 修改drivers/char/Makefile

在文件最后添加如下代码

obj-$(CONFIG\_FS4412\_LED) += fs4412\_led\_drv.o

1. 将s5pv210\_led\_app.c拷贝到Linux任意目录下并交叉编译测试程序

$ arm-none-linux-gnueabi-gcc fs4412\_led\_app.c –o fs4412\_led\_app

sudo cp fs4412\_led\_app /source/rootfs

1. **编译LED驱动到内核中**

* 配置内核时按“空格”选择，配置完成后保存退出

$ make menuconfig

Device Drivers --->

Character devices --->

<\*> FS4412 LED Device Support

* 保存退出，重新编译后把uImage拷贝到tftpboot下

$ make uImage

$ cp arch/arm/boot/uImage /tftpboot

重新启动开发板，加载内核并运行。在终端下执行下面操作

* 创建设备节点

# mknod /dev/led c 500 0

* 运行测试程序并观察现象

# ./fs4412\_led\_app

1. **编译LED驱动为模块**

* 配置内核时按“空格”选择，配置完成后保存退出

$ make menuconfig

Device Drivers --->

Character devices --->

<M> FS4412 LED Device Support

* 保存退出，重新编译后把zImage拷贝到tftpboot下，把驱动模块拷贝到/source/rootfs下

$ make uImage modules

$ cp arch/arm/boot/uImage /tftpboot/

$ cp drivers/char/fs4412\_led\_drv.ko /source/rootfs/

重新启动开发板，linux运行起来后在终端下操作

* 创建设备节点

# mknod /dev/led c 500 0

* 加载LED驱动模块

# insmod fs4412\_led\_drv.ko

* 运行测试程序并观察现象

# ./fs4412\_led\_app

# 实验七 SD卡驱动移植

**【实验目的】**

SD卡是嵌入式系统最常用的外部扩展存储设备，这里介绍SD驱动移植的过程。

说明：在系统移植课程实验中命令行提示符 “$”表示是在主机上执行，“#”表示在目标板执行

**【实验环境】**

* 主机：ubuntu 12.04发行版
* 目标机：FS4412平台
* 交叉编译工具：arm-none-linux-gnueabi-gcc

**【实验步骤】**

1. 修改设备树文件

$ vim arch/arm/boot/dts/exynos4412-fs4412.dts

修改

sdhci@12530000 {

bus-width = <4>;

pinctrl-0 = <&sd2\_clk &sd2\_cmd &sd2\_bus4 &sd2\_cd>;

pinctrl-names = "default";

vmmc-supply = <&mmc\_reg>;

status = "okay";

};

为:

sdhci@12530000 {

bus-width = <4>;

pinctrl-0 = <&sd2\_clk &sd2\_cmd &sd2\_bus4>;

cd-gpios = <&gpx0 7 0>;

cd-inverted = <0>;

pinctrl-names = "default";

/\*vmmc-supply = <&mmc\_reg>;\*/

status = "okay";

};

1. 配置内核

$ make menuconfig

Device Drivers --->

<\*> MMC/SD/SDIO card support --->

<\*> Secure Digital Host Controller Interface support

<\*> SDHCI support on Samsung S3C SoC

File systems --->

DOS/FAT/NT Filesystems --->

<\*> MSDOS fs support

<\*> VFAT (Windows-95) fs support

(437) Default codepage for FAT

(iso8859-1) Default iocharset for FAT

-\*- Native language support --->

<\*> Codepage 437 (United States, Canada)

<\*> Simplified Chinese charset (CP936, GB2312)

<\*> ASCII (United States)

<\*> NLS ISO 8859-1 (Latin 1; Western European Languages)

<\*> NLS UTF-8

1. 编译内核和设备树

$ make uImage

$ make dtbs

1. 测试：

拷贝内核和设备树文件到/tftpboot目录下

$ cp arm/arm/boot/uImage /tftpboot

$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/

启动开发板会有如下内容显示：

[ 1.620000] mmc0: new high speed SDHC card at address cd6d

[ 1.625000] mmcblk1: mmc0:cd6d SE08G 7.28 GiB

[ 1.630000] **mmcblk1: p1(mmcblk1 为设备名 p1 为分区名)**

挂载，**注意不要挂在EMMC的分区**

# mount -t vfat /dev/mmcblk1p1 /mnt

查看/mnt/目录即可看到sd卡中内容

# 实验八 USB驱动的移植

**【实验目的】**

USB接口是现在计算机系统中最通用的一种接口，

说明：在本系统移植课程实验中命令行提示符 “$”表示是在主机上执行，“#”表示在目标板执行

**【实验环境】**

* 主机：ubuntu 12.04发行版
* 目标机：FS4412平台
* 交叉编译工具：arm-none-linux-gnueabi-gcc

**【实验步骤】**

1. 修改设备树文件

$ vim arch/arm/boot/dts/exynos4412-fs4412.dts

添加如下内容：

usbphy: usbphy@125B0000 {

#address-cells = <1>;

#size-cells = <1>;

compatible = "samsung,exynos4x12-usb2phy";

reg = <0x125B0000 0x100>;

ranges;

clocks = <&clock 2>, <&clock 305>;

clock-names = "xusbxti", "otg";

usbphy-sys {

reg = <0x10020704 0x8 0x1001021c 0x4>;

};

};

ehci@12580000 {

status = "okay";

usbphy = <&usbphy>;

};

usb3503@08 {

compatible = "smsc,usb3503";

reg = <0x08 0x4>;

connect-gpios = <&gpm3 3 1>;

intn-gpios = <&gpx2 3 1>;

reset-gpios = <&gpm2 4 1>;

initial-mode = <1>;

};

1. 配置内核

make menuconfig

Device Drivers --->

[\*] USB support --->

<\*> EHCI HCD (USB 2.0) support

<\*> EHCI support for Samsung S5P/EXYNOS SoC Series

<\*> USB Mass Storage support

<\*> USB3503 HSIC to USB20 Driver

USB Physical Layer drivers --->

<\*> Samsung USB 2.0 PHY controller Driver

SCSI device support --->

<\*> SCSI device support

<\*> SCSI disk support

<\*> SCSI generic support

1. 编译内核和设备树

$ make uImage

$ make dtbs

1. 测试：

拷贝内核和设备树文件到/tftpboot目录下

$ cp arm/arm/boot/uImage /tftpboot

$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/

启动目标板并在目标板上完成如下操作：

插入U盘显示如下

[ 72.695000] usb 1-3.2: USB disconnect, device number 3

[ 74.435000] usb 1-3.2: new high-speed USB device number 4 using exynos-ehci

[ 74.555000] usb-storage 1-3.2:1.0: USB Mass Storage device detected

[ 74.560000] scsi1 : usb-storage 1-3.2:1.0

[ 75.645000] scsi 1:0:0:0: Direct-Access Kingston DataTraveler 160 PMAP PQ: 0 ANSI: 4

[ 75.660000] sd 1:0:0:0: Attached scsi generic sg0 type 0

[ 76.695000] sd 1:0:0:0: [sda] 15556608 512-byte logical blocks: (7.96 GB/7.41 GiB)

[ 76.700000] sd 1:0:0:0: [sda] Write Protect is off

[ 76.705000] sd 1:0:0:0: [sda] No Caching mode page found

[ 76.710000] sd 1:0:0:0: [sda] Assuming drive cache: write through

[ 76.725000] sd 1:0:0:0: [sda] No Caching mode page found

[ 76.730000] sd 1:0:0:0: [sda] Assuming drive cache: write through

[ 76.760000] sda: sda1 (sda是设备名 sda1是分区名)

[ 76.770000] sd 1:0:0:0: [sda] No Caching mode page found

[ 76.770000] sd 1:0:0:0: [sda] Assuming drive cache: write through

[ 76.780000] sd 1:0:0:0: [sda] Attached SCSI removable disk

在终端上执行挂载的设备与上边显示相关

# mount -t vfat /dev/sda1 /mnt

# ls

可以查看到U盘内容，即完成实验。

# 实验九 内存调试

**【实验目的】**

段错误和内存错误是C语言编程经常遇到的问题，使用memwatch是由johan lindh编写，是一个开发源代码C语言内存错误检测工具。能检测双重释放，错误释放，没有释放内存，溢出等等情况。

**【实验环境】**

1. ubuntu 12.04发行版
2. FS4412平台
3. 交叉编译器 arm-none-linux-gnueabi-gcc

**【实验步骤】**

1. 解压memwatch.-2.7.1.tar.gz，在解开的目录下添加代码memtest.c

#include <stdlib.h>

#include <stdio.h>

#include "memwatch.h"

int main(int argc,char \*\*argv)

{

char \*ptr1;

char \*ptr2;

ptr1 = malloc(512);

ptr2 = malloc(512);

ptr1[512]= 'A';

ptr2 = ptr1;

free(ptr2);

free(ptr1);

return 0;

}

1. 修改 Makefile

Makefile文件为：

memtest:

$(CC) -DMEMWATCH -DMW\_STDIO memtest.c memwatch.c

5、运行make并在主机上执行a.out，执行后会生成一个记录文件memwatch.log，内容如下：

============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =========

Started at Thu Jan 1 00:08:33 1970

Modes: \_\_STDC\_\_ 32-bit mwDWORD==(unsigned long)

mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32

overflow: <3> memtest.c(12), 512 bytes alloc'd at <1> memtest.c(8)

double-free: <4> memtest.c(13), 0x1a1b4 was freed from memtest.c(12)

Stopped at Thu Jan 1 00:08:33 1970

unfreed: <2> test.c(9), 512 bytes at 0x1a3e4 {FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE ................}

Memory usage statistics (global):

N)umber of allocations made: 2

L)argest memory usage : 1024

T)otal of all alloc() calls: 1024

U)nfreed bytes totals : 512

//overflow: <3> memtest.c(12)缓冲区溢出，当程序执行到第12行free(ptr2)才检测到的；

512 bytes alloc'd at <1> memtest.c(8)

//表示出错缓冲区的大小为512字节，是在memtest.c的第8行分配的。很容易发现代码的ptr1[512]= 'A'出现错误。

double-free: <4> memtest.c(13), 0x1a7f4 was freed from memtest.c(12)

//double-free: <4> memtest.c(13)是一个双重释放的错误，表示程序执行到13行的时候才检测到。

0x1a7f4 was freed from memtest.c(12)

//表示首地址为 0x1a7f4的内存在12行已经被释放。

Stopped at Wed Dec 31 19:00:38 1969

unfreed: <2> test.c(9), 512 bytes at 0x1a3e4 {FE FE FE FE FE FE FE FE

//表示一块内存没有释放，表示这块内存是在memtest.c的第9行分配，大小为512字节，首地址为0x1a3e4。

Memory usage statistics (global):

N)umber of allocations made: 2

L)argest memory usage : 1024

//程序结束时能够是使用的最大动态内存

T)otal of all alloc() calls: 1024 //总共分配的动态内存

U)nfreed bytes totals : 512

//表示未释放的内存

# 实验十 内核调试

**【实验目的】**

由于在驱动开发中经常看到内核崩溃的问题，最常见的就是OOPS错误，本实验要求学员掌握这种调试方法。

**【实验环境】**

1. ubuntu 12.04发行版
2. FS4412平台
3. 交叉编译器 arm-none-linux-gnueabi-gcc

**【实验步骤】**

1、通过OOPS信息中PC寄存器的值可以知道出错指令的地址，通过栈回朔信息可以知道出错时的函数调用的关系，根据这两点可以很快定位错误。

2、 修改drivers/net/ethernet/davicom/dm9000.c，在dm9000\_probe函数中u32 id\_val;下增加下面语句：

int \*ptr = NULL;

\*ptr = 0xff;

3、编译内核下载到开发板上，内核启动会出现如类似下信息：

Unable to handle kernel NULL pointer dereference at virtual address 00000000

pgd = c0004000

[00000000] \*pgd=00000000

Internal error: Oops: 805 [#1] PREEMPT SMP ARM

Modules linked in:

CPU: 1 PID: 1 Comm: swapper/0 Not tainted 3.14.0 #11

task: ee8a0000 ti: ee8a4000 task.ti: ee8a4000

PC is at dm9000\_probe+0x1c/0x8f0

LR is at platform\_drv\_probe+0x18/0x48

pc : [<c0277cc8>] lr : [<c0247f7c>] psr: 60000153

sp : ee8a5e48 ip : 00000000 fp : 00000000

r10: c052a4fc r9 : 00000000 r8 : c0591e98

r7 : 00000000 r6 : ee97c810 r5 : ee97c800 r4 : 00000000

r3 : 000000ff r2 : 00000000 r1 : ee8a5de8 r0 : ee97c800

Flags: nZCv IRQs on FIQs off Mode SVC\_32 ISA ARM Segment kernel

Control: 10c5387d Table: 4000404a DAC: 00000015

Process swapper/0 (pid: 1, stack limit = 0xee8a4240)

Stack: (0xee8a5e48 to 0xee8a6000)

5e40: ee975cf0 00000000 ee1503a8 00000001 c0561afc ee150438

5e60: 00000000 ee97c810 c0591e98 ee97c810 00000000 c0591e98 c0561afc c052a4fc

5e80: 00000000 c0247f7c c0247f64 c05d931c c0591e98 c0246668 ee97c810 c0591e98

5ea0: ee97c844 00000000 c054332c c0246804 c0591e98 c0246778 00000000 c0244fbc

5ec0: ee805478 ee9771c0 c0591e98 eeb73a00 c0590028 c0245e28 c04c3128 c0591e98

5ee0: 00000000 c0591e98 00000000 c054e2ac c059f280 c0246e1c 00000000 ee8a4000

5f00: 00000000 c00087b4 ee903b00 c05c3d50 60000153 c0571c00 60000100 c0571c00

5f20: 00000000 00000000 c0571bfc 00000000 c0505bc8 ef7fc918 00000089 c0034c6c

5f40: c04ca680 c0505338 00000006 00000006 00000000 c054e2c8 c054e2cc 00000006

5f60: c054e2ac c059f280 00000089 c052a4fc 00000000 c052ac4c 00000006 00000006

5f80: c052a4fc c003e0dc 00000000 c03b46ec 00000000 00000000 00000000 00000000

5fa0: 00000000 c03b46f4 00000000 c000e4b8 00000000 00000000 00000000 00000000

5fc0: 0000000000000000 00000000 00000000 00000000 00000000 00000000 00000000

5fe0: 00000000 00000000 00000000 00000000 00000013 00000000 ffffffff ffffffff

[<c0277cc8>] (dm9000\_probe) from [<c0247f7c>] (platform\_drv\_probe+0x18/0x48)

[<c0247f7c>] (platform\_drv\_probe) from [<c0246668>] (driver\_probe\_device+0x100/0x210)

[<c0246668>] (driver\_probe\_device) from [<c0246804>] (\_\_driver\_attach+0x8c/0x90)

[<c0246804>] (\_\_driver\_attach) from [<c0244fbc>] (bus\_for\_each\_dev+0x58/0x88)

[<c0244fbc>] (bus\_for\_each\_dev) from [<c0245e28>] (bus\_add\_driver+0xd8/0x1cc)

[<c0245e28>] (bus\_add\_driver) from [<c0246e1c>] (driver\_register+0x78/0xf4)

[<c0246e1c>] (driver\_register) from [<c00087b4>] (do\_one\_initcall+0x30/0x144)

[<c00087b4>] (do\_one\_initcall) from [<c052ac4c>] (kernel\_init\_freeable+0xfc/0x1c8)

[<c052ac4c>] (kernel\_init\_freeable) from [<c03b46f4>] (kernel\_init+0x8/0xe4)

错误分析：

1、第一行

Unable to handle kernel NULL pointer dereference at virtual address 00000000

说明是空指针造成的错误

2、寄存器信息主要是PC的值

PC is at dm9000\_probe+0x1c/0x8f0

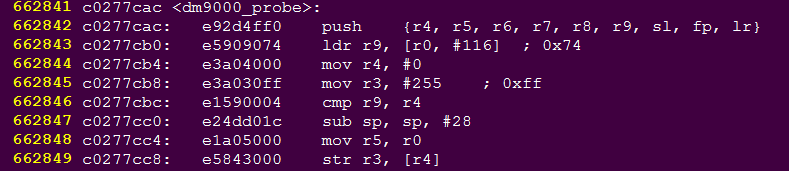
和

pc : [<**c0277cc8**>]

错误定位：

$ arm-none-linux-gnueabi-objdump -D vmlinux > vmlinux.dis

文件vmlinux.dis非常大打开需要一定时间



对于大多数情况，从反汇编代码定位到C代码并不会如此容易，需要有较强的阅读汇编代码的能力。

另外一种方法是通过addr2line去定位

$ arm-none-linux-gnueabi-addr2line **0xc0277cc8** -e vmlinux -f

**重要 ： 该实验完成后不要忘记恢复被修改的代码**

# 实验十一 文件系统的移植

**【实验目的】**

熟悉Linux文件系统目录结构，创建自己的文件系统，通过NFS方式测试；用文件系统工具生成ramdisk文件系统映象文件。

**【实验环境】**

1. ubuntu 12.04发行版
2. FS4412平台
3. 交叉编译器 arm-none-linux-gnueabi-gcc

**【实验步骤】**

**一、根文件系统制作**

1. 源码下载

我们选择的版本是busybox-1.22.1.tar.bz2下载路径为：

<http://busybox.net/downloads/>

1. 解压源码

$ tar xvf busybox-1.22.1.tar.bz2

1. 进入源码目录

$ cd busybox-1.22.1

1. 配置源码

$ make menuconfig

Busybox Settings --->

Build Options --->

[\*] Build BusyBox as a static binary (no shared libs)

[ ] Force NOMMU build

[ ] Build with Large File Support (for accessing files > 2 GB)

(arm-cortex\_a8-linux-gnueabi-) Cross Compiler prefix

() Additional CFLAGS

1. 编译

$ make

1. 安装

busybox默认安装路径为源码目录下的\_install

$ make install

1. 进入安装目录下

$ cd \_install

$ ls

bin linuxrc sbin usr

1. 创建其他需要的目录

$ mkdir dev etc mnt proc var tmp sys root

1. 添加库

* 将工具链中的库拷贝到\_install目录下

$ cp /home/linux/toolchain/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/lib/ . -a

* 删除静态库和共享库文件中的符号表

$ rm lib/\*.a

$ arm-none-linux-gnueabi-strip lib/\*

* 删除不需要的库，确保所有库大小不超过8M

$ du -mh lib/

1. 添加系统启动文件

在etc下添加文件inittab，文件内容如下：

#this is run first except when booting in single-user mode.

::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys

# start an "askfirst" shell on the console (whatever that may be)

::askfirst:-/bin/sh

# stuff to do when restarting the init process

::restart:/sbin/init

# stuff to do before rebooting

::ctrlaltdel:/sbin/reboot

在etc下添加文件fstab，文件内容如下：

#device mount-point type options dump fsck order

proc /proc proc defaults 0 0

tmpfs /tmp tmpfs defaults 0 0

sysfs /sys sysfs defaults 0 0

tmpfs /dev tmpfs defaults 0 0

这里我们挂载的文件系统有三个proc、sysfs和tmpfs。在内核中proc和sysfs默认都支持，而tmpfs是没有支持的，我们需要添加tmpfs的支持

修改内核配置：

$ make menuconfig

File systems --->

Pseudo filesystems --->

[\*] Virtual memory file system support (former shm fs)

[\*] Tmpfs POSIX Access Control Lists

重新编译内核

在etc下创建init.d目录，并在init.d下创建rcS文件，rcS文件内容为：

#!/bin/sh

# This is the first script called by init process

/bin/mount -a

echo /sbin/mdev > /proc/sys/kernel/hotplug

/sbin/mdev -s

为rcS添加可执行权限：

$ chmod +x init.d/rcS

在etc下添加profile文件，文件内容为：

#!/bin/sh

export HOSTNAME=farsight

export USER=root

export HOME=root

export PS1="[$USER@$HOSTNAME \W]\# "

PATH=/bin:/sbin:/usr/bin:/usr/sbin

LD\_LIBRARY\_PATH=/lib:/usr/lib:$LD\_LIBRARY\_PATH

export PATH LD\_LIBRARY\_PATH

**重要：新制作的文件系统尺寸若超出8M，删除不需要的库文件**

**二、NFS测试**

1、删除原先的/source/rootfs

$ sudo rm -rf /source/rootfs

2、将我们新建的根文件系统拷贝到/source/rootfs目录下

$sudo mkdir /source/rootfs

$ sudo cp \_install/\* /source/rootfs –a

3、设置uboot环境变量

# setenv serverip 192.168.9.120

# setenv ipaddr 192.168.9.233

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;bootm 41000000 – 42000000

#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200 init=/linuxrc ip=192.168.9.233

# saveenv

重新启动开发板，查看是否能够正常挂载，功能是否正常



**三、制作ramdisk文件系统**

通过NFS测试以后，就可以制作ramdisk文件系统了，具体如下：

1、制作一个大小为8M的镜像文件

$ cd ~

$ dd if=/dev/zero of=ramdisk bs=1k count=8192 （ramdisk为8M）

2、格式化这个镜像文件为ext2

$ mkfs.ext2 -F ramdisk

3、在mount下面创建initrd目录作为挂载点

$ sudo mkdir /mnt/initrd

4、将这个磁盘镜像文件挂载到/mnt/initrd下

注意这里的ramdisk不能存放在rootfs目录中

$ sudo mount -t ext2 ramdisk /mnt/initrd

5、将我们的文件系统复制到ramdisk中

将测试好的文件系统里的内容全部拷贝到 /mnt/initrd目录下面

$ sudo cp /source/rootfs/\* /mnt/initrd –a

6、卸载initrd

$ sudo umount /mnt/initrd

7、压缩ramdisk为ramdisk.gz并拷贝到/tftpboot下

$ gzip --best -c ramdisk > ramdisk.gz

8、格式化为uboot识别的格式

$ mkimage -n "ramdisk" -A arm -O linux -T ramdisk -C gzip -d ramdisk.gz ramdisk.img

$ cp ramdisk.img /tftpboot

9、配置内核支持RAMDISK

制作完 ramdisk.img后，需要配置内核支持RAMDISK作为启动文件系统

make menuconfig

File systems --->

<\*> Second extended fs support

Device Drivers

SCSI device support --->

<\*> SCSI disk support

Block devices --->

<\*>RAM block device support

(16)Default number of RAM disks

(8192) Default RAM disk size (kbytes) (修改为8M)

General setup --->

[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support

重新编译内核，复制到/tftpboot

10、在U-BOOT命令行重新设置启动参数：

# setenv bootcmd tftp 41000000 uImage\;tftp 42000000 exynos4412-fs4412.dtb\;tftp 43000000 ramdisk.img\;bootm 41000000 43000000 42000000

# saveenv

重新启动开发板查看能否正常启动

**四、ext4文件系统制作**

1、格式化工具制作

拷贝e2fsprogs-1.42.5.tar.xz到Linux下

解压

$ tar xvf e2fsprogs-1.42.5.tar.xz

进入工程

$ cd e2fsprogs-1.42.5

创建脚本build.sh并添加如下内容：

$ vim build.sh

#!/bin/sh

CC=arm-none-linux-gnueabi-gcc

./configure --enable-elf-shlibs --host=arm-none-linux-gnueabi \

--prefix=/home/linux/tools

make

make install

执行脚本

./build.sh

在编译到最后可能会有个错误如下：

make[1]: [libext2fs.dvi] Error 1 (ignored)

这个不要紧其实我们需要的文件都编译好了

拷贝可执行文件和库到文件系统中

$ cp /home/linux/tools/sbin/mkfs.ext3 /source/rootfs/sbin

$ cp /home/linux/tools/lib/\* /source/rootfs/lib

2、修改u-boot启动参数

setenv bootargs root=/dev/nfs rw nfsroot=192.168.9.120:/source/rootfs init=/linuxrc console=ttySAC2,115200 ip=192.168.9.233

注意：192.168.9.120 对应Ubuntu的ip

192.168.9.233 对应板子的ip

这两个ip应该根据自己的实际情况适当修改

3、分区

启动开发板在倒计时期间按任意键结束启动，执行如下命令

# fdisk -c 0

4、格式化

重新启动开发板进入系统后执行

# mkfs.ext3 -F /dev/mmcblk0p2

5、修改我们/source/rootfs/etc/fstab

在最后添加如下内容

/dev/mmcblk0p2 /mnt ext3 defaults 0 0

重新启动系统系统在启动最后会挂载mmcblk0p2 作为用户文件系统