

Linux 系统移植

目 录

第一部分 前言.....	8
1 硬件环境.....	8
1.1 主机硬件环境.....	8
1.2 目标板硬件环境.....	8
1.3 工具介绍.....	8
2 软件环境.....	8
2.1 主机软件环境.....	8
2.1.1 Windows 操作系统.....	8
2.1.2 Linux 操作系统	8
2.1.3 目标板最后运行的环境.....	9
2.2 Linux 下工作用户及环境.....	9
2.2.1 交叉工具的安装.....	9
2.2.2 u-boot 移植工作目录.....	9
2.2.3 内核及应用程序移植工作.....	9
2.3 配置系统服务.....	10
2.3.1 tftp 服务器的配置.....	10
2.4 工具使用.....	12
2.4.1 minicom 的使用.....	12
3 作者介绍.....	13
3.1 策划, 组织, 指导, 发布者.....	13
3.2 ADS bootloader 部分.....	13
3.3 交叉工具部分.....	13
3.4 uboot 部分.....	13
3.5 内核部分.....	13
3.6 应用程序部分.....	13
3.7 网卡驱动部分.....	13
3.8 Nand Flash 驱动部分.....	13
第二部分 系统启动 bootloader 的编写(ADS).....	14
1 工具介绍.....	14
1.1 ADS 命令行命令介绍.....	14
1.1.1 armasm.....	14
1.1.2 armcc, armcpc.....	14
1.1.3 armlink.....	14
2 基本原理.....	15
2.1 可执行文件组成及内存映射.....	15
2.1.1 可执行文件的组成.....	15
2.1.2 装载过程.....	16

2.1.3 启动过程的汇编部分.....	17
2.1.4 启动过程的 C 部分.....	17
3 AXD 的使用以及源代码说明.....	18
3.1 源代码说明.....	18
3.1.1 汇编源代码说明.....	18
3.1.2 C 语言源代码说明.....	23
3.1.3 源代码下载.....	23
3.2 AXD 的使用.....	23
3.2.1 配置仿真器.....	23
3.2.2 启动 AXD 配置开发板.....	23
第三部分 GNU 交叉工具链.....	25
1 设置环境变量，准备源码及相关补丁.....	25
1.1 设置环境变量.....	25
1.2 准备源码包.....	25
1.2.1 binutils.....	25
1.2.2 gcc.....	25
1.2.3 glibc.....	25
1.2.4 linux kernel.....	26
1.3 准备补丁.....	26
1.3.1 ioperm.c.diff.....	26
1.3.2 flow.c.diff.....	26
1.3.3 t-linux.diff.....	26
1.4 编译 GNU binutils.....	26
1.5 准备内核头文件.....	26
1.5.1 使用当前平台的 gcc 编译内核头文件.....	26
1.5.2 复制内核头文件.....	27
1.6 译编 glibc 头文件.....	27
1.7 编译 gcc 第一阶段.....	27
1.8 编译完整的 glibc.....	27
1.9 编译完整的 gcc.....	28
2 GNU 交叉工具链的下载.....	28
2.1 ARM 官方网站.....	28
2.2 本文档提供的下载.....	28
3 GNU 交叉工具链的介绍与使用.....	29
3.1 常用工具介绍.....	29
3.2.1 arm-linux-gcc 的使用.....	29
3.2.2 arm-linux-ar 和 arm-linux-ranlib 的使用.....	30
3.2.3 arm-linux-objdump 的使用.....	30
3.2.4 arm-linux-readelf 的使用.....	31
3.2.6 arm-linux-copydump 的使用.....	32
4 ARM GNU 常用汇编语言介绍.....	32
4.1 ARM GNU 常用汇编伪指令介绍.....	32

4.2 ARM GNU 专有符号.....	33
4.3 操作码.....	33
5 可执行生成说明.....	33
5.1 lds 文件说明.....	33
5.1.1 主要符号说明.....	33
5.1.2 段定义说明.....	34
第四部分 u-boot 的移植.....	35
1 u-boot 的介绍及系统结构.....	35
1.1 u-boot 介绍.....	35
1.2 获取 u-boot.....	35
1.3 u-boot 体系结构.....	35
1.3.1 u-boot 目录结构.....	35
2 uboot 的启动过程及工作原理.....	36
2.1 启动模式介绍.....	36
2.2 阶段 1 介绍.....	36
2.2.1 定义入口.....	36
2.2.2 设置异常向量.....	37
2.2.3 设置 CPU 的模式为 SVC 模式.....	37
2.2.4 关闭看门狗.....	37
2.2.5 禁掉所有中断.....	37
2.2.6 设置以 CPU 的频率.....	37
2.2.7 设置 CP15.....	37
2.2.8 配置内存区控制寄存器.....	38
2.2.9 安装 U-BOOT 使用的栈空间.....	38
2.2.10 BSS 段清 0.....	38
2.2.11 搬移 Nand Flash 代码.....	39
2.2.12 进入 C 代码部分.....	39
2.3 阶段 2 的 C 语言代码部分	39
2.3.1 调用一系列的初始化函数.....	39
2.3.2 初始化网络设备.....	41
2.3.3 进入主 UBOOT 命令行.....	41
2.4 代码搬运.....	41
3 uboot 的移植过程.....	42
3.1 环境.....	42
3.2 步骤.....	42
3.2.1 修改 Makefile.....	42
3.2.2 在 board 子目录中建立 crane2410.....	42
3.2.3 在 include/configs/中建立配置头文件.....	42
3.2.4 指定交叉编译工具的路径.....	42
3.2.5 测试编译能否成功.....	42
3.2.6 修改 lowlevel_init.S 文件.....	43

3.2.9 UBOOT 的 Nand Flash 移植.....	45
3.2.8 重新编译 u-boot.....	45
3.2.9 把 u-boot 烧入 flash.....	45
4 U-BOOT 命令的使用.....	46
4.1 U-BOOT 命令的介绍.....	46
4.1.1 获得帮助信息.....	46
4.2 常用命令使用说明.....	47
4.2.1 askenv(F).....	47
4.2.2 autoscr.....	47
4.2.3 base	47
4.2.4 bdfinfo.....	47
4.2.5 bootp.....	47
4.2.8 tftp(tftpboot).....	48
4.2.9 bootm.....	48
4.2.10 go.....	48
4.2.11 cmp	48
4.2.12 coninfo	48
4.2.13 cp.....	48
4.2.14 date.....	49
4.2.15 erase(F).....	49
4.2.16 flinfo(F).....	49
4.2.17 iminfo.....	49
4.2.18 loadb.....	49
4.2.19 md.....	49
4.2.20 mm	50
4.2.21 mtest	50
4.2.22 mw.....	50
4.2.23 nm	50
4.2.24 printenv.....	50
4.2.25 ping	51
4.2.26 reset.....	51
4.2.27 run	51
4.2.28 saveenv(F).....	51
4.2.29 setenv.....	51
4.2.30 sleep.....	51
4.2.31 version.....	51
4.2.32 nand info.....	51
4.2.33 nand device <n>.....	51
4.2.34 nand bad.....	51
4.2.35 nand read.....	52
4.2.36 nand erase.....	52
4.2.37 nand write.....	52
4.3 命令简写说明.....	52

4.4 把文件写入 NandFlash.....	53
4.5 下载提供.....	53
5 参考资料.....	53
第五部分 linux 2.6 内核的移植.....	53
1 内核移植过程.....	53
1.1 下载 linux 内核.....	53
1.2 修改 Makefile.....	53
1.3 设置 flash 分区.....	54
1.3.1 指明分区信息.....	54
1.3.2 指定启动时初始化.....	56
1.3.3 禁止 Flash ECC 校验	56
1.4 配置内核.....	56
1.4.1 支持启动时挂载 devfs.....	56
1.4.2 配置内核产生.config 文件.....	57
1.4.3 编译内核.....	58
1.4.4 下载 zImage 到开发板.....	58
2 创建 uImage.....	61
2.1 相关技术背景介绍.....	61
2.2 在内核中创建 uImage 的方法.....	61
2.2.1 获取 mkimage 工具.....	61
2.2.2 修改内核的 Makefile 文件.....	61
3 追加实验记录.....	62
3.1 移植 linux-2.6.15.7.....	62
3.2 移植 linux-2.6.16.21.....	62
3.3 移植 linux-2.6.17.....	62
4 参考资料.....	62
第六部分 应用程序的移植.....	63
1 构造目标板的根目录及文件系统.....	63
1.1 建立一个目标板的空根目录.....	63
1.2 在 my_rootfs 中建立 Linux 目录树.....	63
1.3 创建 linuxrc 文件.....	63
2 移植 Busybox.....	64
2.1 下载 busybox.....	64
2.3 编译并安装 Busybox.....	65
3 移植 TinyLogin.....	66
3.1 下载.....	66
3.2 修改 tinyLogin 的 Makefile.....	66
3.3 编译并安装.....	66
4 相关配置文件的创建.....	66
4.1 创建帐号及密码文件.....	66
4.2 创建 profile 文件.....	67

4.4 创建 fstab 文件.....	67
4.5 创建 inetd.conf 配置文件.....	67
5 移植 inetd.....	67
5.1 inetd 的选择及获取.....	67
5.1.1 获取 inetd.....	67
5.2 编译 inetd.....	67
5.2.1 修改 configure 文件.....	67
5.2.2 编译	68
5.3 配置 inetd.....	68
5.3.1 拷贝 inetd 到根文件系统的 usr/sbin 目录中.....	68
6 移植 tftpd Web 服务器.....	69
6.1 下载.....	69
6.2 编译 tftpd.....	69
6.3 配置.....	69
6.3.1 拷贝 tftpd 二进制可执行文件到根文件系统/usr/sbin/ 目录中.....	69
6.3.2 修改 tftpd 配置文件.....	69
6.3.3 转移到根文件系统目录，创建相应的文件.....	69
7 建立根目录文件系统包.....	70
7.1 建立 CRAMFS 包.....	70
7.1.1 下载 cramfs 工具.....	70
7.1.2 制作 cramfs 包.....	70
7.1.3 写 cramfs 包到 Nand Flash.....	70
8 参考资料.....	70
第七部分 Nand flash 驱动的编写与移植.....	71
1 Nand flash 工作原理.....	71
1.1 Nand flash 芯片工作原理.....	71
1.1.1 芯片内部存储布局及存储操作特点.....	71
1.1.2 重要芯片引脚功能.....	71
1.1.3 寻址方式.....	71
1.1.4 Nand flash 主要内设命令详细介绍.....	72
1.2 Nand Flash 控制器工作原理.....	72
1.2.1 Nand Flash 控制器特性.....	72
1.2.2 Nand Flash 控制器工作原理.....	72
1.3 Nand flash 控制器中特殊功能寄存器详细介绍	72
1.4 Nand Flash 控制器中的硬件 ECC 介绍.....	73
1.4.1 ECC 产生方法.....	73
1.4.2 ECC 生成器工作过程.....	74
1.4.3 ECC 的运用.....	74
2 在 ADS 下 flash 烧写程序.....	74
2.1 ADS 下 flash 烧写程序原理及结构.....	74
2.2 第三层实现说明.....	74

2.1.1 特殊功能寄存器定义.....	74
2.1.2 操作的函数实现.....	74
2.3 第二层实现说明.....	75
2.3.1 Nand Flash 初始化.....	75
2.3.3 获取 Nand flash ID.....	75
2.3.4 Nand flash 写入.....	76
2.3.5 Nand flash 读取.....	77
2.3.6 Nand flash 标记坏块.....	78
2.3.7 Nand Flash 检查坏块.....	79
2.3.8 擦除指定块中数据.....	79
2.4 第一层的实现.....	80
3 在 U-BOOT 对 Nand Flash 的支持.....	82
3.1 U-BOOT 对从 Nand Flash 启动的支持.....	82
3.1.1 从 Nand Flash 启动 U-BOOT 的基本原理.....	82
3.1.2 支持 Nand Flash 启动代码说明.....	82
3.2 U-BOOT 对 Nand Flash 命令的支持.....	84
3.2.1 主要数据结构介绍.....	84
3.2.2 支持的命令函数说明.....	85
4 在 Linux 对 Nand Flash 的支持.....	87
4.1 Linux 下 Nand Flash 调用关系.....	87
4.1.1 Nand Flash 设备添加时数据结构包含关系.....	87
4.1.2 Nand Flash 设备注册时数据结构包含关系.....	87
4.2 Linux 下 Nand Flash 驱动主要数据结构说明.....	88
4.2.1 s3c2410 专有数据结构.....	88
4.2.2 Linux 通用数据结构说明.....	89
4.3.1 注册 driver_register.....	94
4.3.2 探测设备 probe.....	94
4.3.3 初始化 Nand Flash 控制器.....	94
4.3.4 移除设备.....	94
4.3.5 Nand Flash 芯片初始化.....	94
4.3.6 读 Nand Flash.....	95
4.3.7 写 Nand Flash.....	95
第八部分 Cs8900a 网卡驱动的编写与移植.....	95
1 Cs8900a 工作原理.....	95
2 在 ADS 下 cs8900a 的实现.....	95
2.1 在 cs8900a 下实现的 ping 工具.....	95
3 在 u-boot 下 cs8900a 的支持.....	96
3.1 u-boot 下 cs8900a 的驱动介绍.....	96
3.2 u-boot 下 cs8900a 的移植说明.....	96
4 在 linux 下 cs8900a 驱动的编写与移植.....	96
4.1 Linux 下 cs8900a 的驱动说明.....	96

4.2 Linux 下 cs8900a 的移植说明.....	96
4.2.1 为 cs8900a 建立编译菜单.....	96
4.2.2 修改 S3C2410 相关信息.....	97

序

该文档的目的是总结我们在工作中的一些经验，并把它们分享给喜欢 ARM 和 Linux 的朋友，如有错误之处，请大家多多指点。同样，我们也希望更多人能把自己的工作经验和体会加入该文档，让大家共同进步。

该文档是一份交流性文档，只供个人学习与交流，不允许公司和企业用于商业行为。

第一部分 前言

1 硬件环境

1.1 主机硬件环境

开发机: Pentium-4 CPU

内存: 512MB

硬盘: 60GB

1.2 目标板硬件环境

CPU: S3C2410

SDRAM: HY57V561620

Nand flash: K9F1208U0B (64MB)

以太网芯片: CS8900A (10M/100MB)

1.3 工具介绍

仿真器: Dragon-ICE

电缆: 串口线, 并口线

2 软件环境

2.1 主机软件环境

2.1.1 Windows 操作系统

ADS 编译工具: ADS1.2

仿真器软件: Dragon-ICE daemon 程序

2.1.2 Linux 操作系统

GNU 交叉编译工具:

2.95.3:

作用: 编译 u-boot

3.3.2, 3.4.4:

作用: 编译内核和应用程序

其它工作:

tree 工具:

作用: 查看文件目录树

下载: 从 <ftp://mama.indstate.edu/linux/tree/> 下载编译

2.1.3 目标板最后运行的环境

启动程序:

u-boot-1.1.4

内核:

linux-2.6.14.1

应用程序:

1. busybox-1.1.3

2. TinyLogin-1.4

3. Thttpd-2.25

2.2 Linux 下工作用户及环境

2.2.1 交叉工具的安装

工具链的编译过程请参考第三部分。

1. 下载交叉工具

2.95.3 下载地址: <ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-2.95.3.tar.bz2>

3.3.4 下载地址:

2. 编译交叉工具

```
[root@localhost ~]mkdir /usr/local/arm
```

```
[root@localhost ~]cd /usr/local/arm
```

把 cross-2.95.2.tar.bz2, cross-3.4.4.tar.gz 拷贝到 /usr/local/arm 目录中。解压这两个包。

```
[root@localhost ~]tar -xjvf cross-2.95.2.tar.bz2
```

```
[root@localhost ~]tar -xzvf cross-3.4.4.tar.gz
```

2.2.2 u-boot 移植工作目录

1. 添加工作用户

```
[root@localhost ~]#useradd -G root -g root -d/home/uboot uboot
```

2. 建立工作目录

```
[uboot@localhost ~]$mkdir dev_home
```

```
[uboot@localhost ~]$cd dev_home
```

```
[uboot@localhost dev_home]$mkdir doc mybootloader uboot
```

```
.
```

```
|-- doc
```

```
|-- mybootloader
```

```
`-- uboot
```

3. 建立环境变量

```
[uboot@localhost ~]vi ~/.bashrc
```

```
export PATH=/usr/local/arm/2.95.3/bin:$PATH
```

2.2.3 内核及应用程序移植工作

1. 添加工作用户

```
[root@localhost ~]#useradd -G root -g root -d/home/arm arm
```

2. 建立工作目录

```
[arm@localhost arm]$mkdir dev_home
[arm@localhost arm]$cd dev_home
[arm@localhost arm]$mkdir bootldr btools debug doc images kernel localapps \
rootfs sysapps tmp tools
[arm@localhost arm]$tree -L 1
.
|-- bootldr
|-- btools
|-- debug
|-- doc
|-- images
|-- kernel
|-- localapps
|-- rootfs
|-- sysapps
|-- tmp
\-- tools
```

可以看到如上树形结构。

注: **tree** 命令

3. 建立环境变量设置脚本

```
[arm@localhost arm]$vi env_sh
#!/bin/bash
PRJROOT=~ /dev_home
KERNEL=$PRJROOT/kernel
ROOTFS=$PRJROOT/rootfs
LAPP=$PRJROOT/localapps
DOC=$PRJROOT/doc
TMP=$PRJROOT/tmp

export PRJROOT KERNEL LAPP ROOTFS
export PATH=/usr/local/arm/3.4.4/bin:$PATH
```

4. 登陆时启动环境变量

```
[arm@localhost arm]$vi ~/.bashrc
. ~/dev_home/env_sh
```

重新登陆 **arm** 用户, 环境变量生效

```
[arm@localhost arm]$su arm
```

2.3 配置系统服务

2.3.1 tftp 服务器的配置

如果用下面一条命令能够看到服务已经启动, 则不用安装, 否则需要按 1 或 2 点安装 **tftp-server** 服务器。

```
[arm@localhost arm]#netstat -a | grep tftp
udp          0          0 *:tftp      *:*
```

1. 从 RPM 包安装 **tftp-server**

从对应 Linux 操作系统版本的安装光盘上找到 **tftp-server** 的安装包。

下面 **tftp-server-0.32-4.i386.rpm** 包为例, 把 rpm 包拷贝到 **dev_home/btools/** 下。

```
[arm@localhost arm]#cp tftp-server-0.32-4.i386.rpm /home/arm/dev_home/btools/
[arm@localhost arm]#su root
[root@localhost arm]#rpm -q tftp-server
```

如果没有安装 **tftp-server**, 就要用下面命令安装, 否则, 直接进入第 2 步配置服务。

```
[root@localhost arm]#cd /home/arm/dev_home/btools/
[root@localhost btools]#rpm -ivh tftp-server-0.32-4.i386.rpm
```

建立 **tftp** 的主工作目录

```
[root@localhost btools]#mkdir /tftpboot
```

2. 修改配置文件并启动服务

备份配置文件

```
[root@localhost btools]#if [ -f /etc/xinetd.d/tftp ]
> then
> cp /etc/xinetd.d/tftp /etc/xinetd.d/tftp.old
> fi
```

修改配置文件

```
[root@localhost btools]#vi /etc/xinetd.d/tftp
service tftp
{
    disable = no
    socket_type          = dgram
    protocol             = udp
    wait                = yes
    user                 = root
    server               = /usr/sbin/in.tftpd
    server_args          = -s /tftpboot
    per_source           = 11
    cps                  = 100 2
    flags                = IPv4
}
```

检查 **tftp** 服务是否打开

```
[root@localhost btools]#chkconfig --list
```

如果 **tftp** 的服务没有打开,则用下面命令打开 **tftp** 服务开关

```
[root@localhost btools]#chkconfig tftp on
```

重启服务

```
#/etc/init.d/xinetd restart
#netstat -a | grep tftp
udp        0          0 *:tftp     *:*
```

2.3.2 NFS 服务器的配置

1. 安装 NFS 服务器

```
[root@localhost btools]#rpm -q nfs-utils
```

如果没有安装,从对应 Linux 操作系统版本的安装光盘上找到 **nfs-utils** 的安装包。**Fedora 5** 中的安装包名称为 **nfs-utils-1.0.8.rc2-4.FC5.2.i386.rpm**。下面以该安装包为例说明:

```
[root@localhost btools]#rpm -ivh nfs-utils-1.0.8.rc2-4.FC5.2.i386.rpm
```

2. 配置 NFS 服务器

```
[root@localhost btools]#vi /etc/exports
```

#加入要允许被另外计算机 mount 的目录:

#/home/arm/dev_home/tmp 为被另外计算机 mount 的目录

#192.168.1.134 允许另外计算机 mount 的 IP

#rw, sync, no_root_squash 表示访问限制,更详细说明见相关手册。

/home/arm/dev_home/tmp 192.168.1.134(rw, sync, no_root_squash)

3. 启动 NFS 服务器

第一启动 NFS 服务器时用下面命令。

```
[root@localhost btools]#/etc/init.d/nfs start
```

如果你已经启动了 NFS 服务器时,并且重新修改了 **/etc/exports** 文件,用如下命令使新加入的目录生效:

```
[root@localhost btools]#/etc/init.d/nfs reload
```

4. 测试 NFS 服务器

```
[root@localhost btools]#netstat -a | grep nfs
```

5. 显示被 export 出的目录列表

```
[root@localhost btools]#exportfs
```

2.4 工具使用

2.4.1 minicom 的使用

1. 切换到 root 用户.

```
[root@localhost btools]#su -
```

2. 查找有效的串设备.

```
[root@localhost ~]#cat /proc/devices
```

```
...
```

```
4 ttyS
```

```
...
```

```
188 ttyUSB
```

```
...
```

如果是普通串口设备, 设备名前缀为 **ttys**, 第一串口为 **ttys0**, 第二串口为 **ttys1**, 依次类推.

如果是 USB 转串口的设备, 设备名前缀为 **ttyUSB**, 第一串口为 **ttyUSB0**.

3. 配置 ttyUSB 设备

```
[root@localhost ~]#minicom -s ttyUSB0
```

会出现一个 configuration 窗口,

```
[configuration]
| Filenames and paths |
| File transfer protocols |
| Serial port setup   |
| Modem and dialing   |
| Screen and keyboard |
| Save setup as ttyUSB0 |
| Save setup as..     |
| Exit                 |
| Exit from Minicom   |
```

选择 **Serial port setup** 配置. 会出现如下窗口:

```
| A - Serial Device      : /dev/ttyUSB0 |
| B - Lockfile Location  : /var/lock    |
| C - Callin Program     :              |
| D - Callout Program    :              |
| E - Bps/Par/Bits       : 115200 8N1   |
| F - Hardware Flow Control : No        |
| G - Software Flow Control : No        |
|                          |
| Change which setting? |
```

我的设置如上所示, 设置完成后, **Change which setting?**项上按回车退出当前窗口, 回到第一个窗口. 按 **Save setup as ttyUSB0** 保存设置. 再按 **Exit from Minicom** 退出 Minicom.

4. 启动 minicom

```
[root@localhost ~]#minicom
```

3 作者介绍

3.1 策划, 组织, 指导, 发布者

刘勇

email: littlegenius2008@163.com

如果您有新的内容, 请发到这个电子邮件, 我们会把您的内容加入文档, 并在作者列表中加入您的名字.

3.2 ADS bootloader 部分

作者: 刘勇

email: littlegenius2008@163.com

3.3 交叉工具部分

作者: 孙贺

email: msunhe@gmail.com

3.4 uboot 部分

作者: 聂强

email: wolfwind9779@yahoo.com.cn

作者: 孙贺

email: msunhe@yahoo.com.cn

3.5 内核部分

作者: 聂大鹏

email: dozec@mail.csdn.net

作者: 牛须乐 (8900a 网卡移植部分)

email: clizniu@hotmail.com

3.6 应用程序部分

作者: 聂大鹏

email: dozec@mail.csdn.net

3.7 Nand Flash 驱动部分

作者: 孙磊, 刘勇

email: sunlei3448@yahoo.com.cn

4 支持企业

4.1 尚观科技

为我们提供统许多套远峰公司的 ARM 开发板, 才能让我们做出统一的文档.

第部分 系统启动 bootloader 的编写(ADS)

1 工具介绍

1.1 ADS 命令行命令介绍

1.1.1 armasm

1. 命令: **armasm** [选项] -o 目标文件 源文件

2. 选项说明

-Errors 错误文件名	;指定一个错误输出文件
-I 目录[,目录]	;指定源文件搜索目录
-PreDefine 预定义宏	;指定预定义的宏
-NOCache	;编译源代码时禁止使用 Cache 进行优化
-MaxCache <n>	;编译源代码时使用 Cache 进行优化
-NOWarn	;关闭所有的警告信息
-G	;输出调试表
-keep	;在目标文件中保存本地符号表
-LIttleend	;生成小端(Little-endian) ARM 代码
-BIgend	;生成大端(Big-endian) ARM 代码
-CPU <target-cpu>	;设立目标板 ARM 核类型,如: arm920t.
-16	;建立 16 位的 thumb 指令.
-32	;建立 32 位的 ARM 指令.

3. 编译一个汇编文件

```
c:\adsloader>armasm -LIttleend -cpu ARM920T -32 bdinit.s
```

把汇编语言编译成小端, 32 位, ARM920T CPU.

1.1.2 armcc, armcpp

1. 命令: **armcc** [选项] 源文件 1 源文件 2 ... 源文件 n

2. 选项说明

-c	;编译但是不连接
-D	;指定一个编译时使用的预定义宏常量
-E	;仅仅对 C 源文件做预处理
-g	;产生调试信息表
-I	;指头文件的搜索路径
-o<file>	;指定一个输出的目标文件
-O[0/1/2]	;指定源代码的优化级别
-S	;输出汇编代码来代替目标文件
-CPU <target-cpu>	;设立目标板 ARM 核类型,如: arm920t.

3. 编译一个 C 程序

```
c:\adsloader>armcc -c -O1 -cpu ARM920T bdisr.c
```

编译不连接, 二级优化, ARM920T CPU.

1.1.3 armlink

1. 命令: **armlink** [选项] 输入文件

2. 选项说明

-partial	;合并目标文件
----------	---------

-Output 文件 ;指定输出文件名
 -scatter 文件 ;按照指定的文件为可执行文件建立内存映射
 -ro-base 地址值 ;只读代码段的起始地址
 -rw-base 地址值 ;RW/ZI 段的起始地址

3. 把多个目标文件合并成一个目标文件

```
c:\adsloader>armlink -partial bdmain.o bdport.o bdserial.o bdmmu.o bdisr.o -o bd.o
```

4. 把几个目标文件编译一个可执地文件

```
c:\adsloader>armlink bd.o bdinit.o -scatter bdscf.scf -o bd.axf
```

1.1.4 fromelf

1. 命令: **fromelf** [选项] 输入文件

2. 选项说明

-bin 二进制文件名 ;产生的二进制文件
 -elf elf 文件名 ;产生一个 elf 文件
 -text text 文件名 ;产生 text 文件

3. 产生一个可执行的二进制代码

```
c:\adsloader>fromelf bd.axf -bin -o bd.bin
```

2 基本原理

2.1 可执行文件组成及内存映射

2.1.1 可执行文件的组成

在 ADS 下,可执行文件有两种,一种是 .axf 文件,带有调试信息,可供 AXD 调试工具使用.另一种是 .bin 文件,可执行的二进制代码文件。我们重点是讲描 .bin 文件的组成。

我们把可执行文件分为两种情况: 分别为存放态和运行态。

1. 存放态

存放态是指可执行文件通过 **fromelf** 产生后,在存储介质(flash 或磁盘)上的分布. 此时可执行文件一般由两部分组成: 分别是代码段和数据段。代码段又分为可执行代码段(.text)和只读数据段(.rodata), 数据段又分为初始化数据段(.data)和未初始化数据段(.bss)。可执行文件的存放态如下:

```

+-----+
| .bss |
+-----+--- 数据段
| .data |
+-----+-----+
| .rodata |
|_____| 代码段
| .text |
+-----+

```

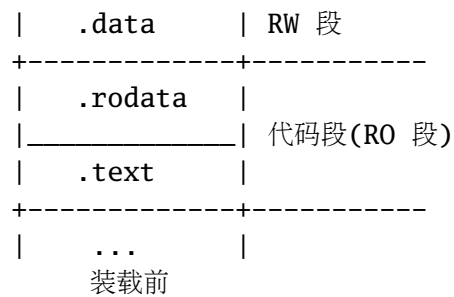
2. 运行态

可执行文件通过装载过程, 搬入到 RAM 中运行, 这时候可执行文件就变成运行态。在 ADS 下对可执行代码各段有另一个名称:

```

| ... |
+-----+
| .bss | ZI 段
+-----+--- 数据段

```



当可执行文件装载后，在 **RAM** 中的分布如下：



所以装载过程必须完成把执行文件的各个段从存储介质上搬到 **RAM** 指定的位置。而这个装载过程由谁来完成呢？由我们的启动程序来完成。

2.1.2 装载过程

在 **ADS** 中，可以通过两种方式来指定可执行代码各段在 **RAM** 中的位置，一个是用 **armlink** 来指定，一种是用 **scatter** 文件来指定。**RAM** 区的起始地址：**0x30000000**。

1. armlink 指定代码段地址

我们通常的代码，只用指定两个段开始地址，**RO** 段的起始地址和 **RW** 段的起始地址，**ZI** 段紧接在 **RW** 段之后。示例见该部分的 **1.1.3**。

2. scatter 指定代码段地址

我们也可以通过 **scatter** 文件指定可执行文件各段的详细地址。**Scatter** 文件如下：

```

MYLOADER 0x30000000
;MYLOADER: 为可执行文件的名称, 可自定义
;0x30000000: 起始地址
{
    RO 0x30000000
    ;RO 只读代码段的名称
    ;0x30000000: 只读代码段的起始地址
    {
        init.o (Init, +First)
        ; Init 代码段为可执行文件的第一部分.
        * (+RO)           ;所有其它的代码段和只读数据段放在该部分
    }
}

```

RW +0

;RW: RW 段的名称

;+0: 表示 RW 段紧接着 RO 段

```
{
    *(+RW)      ;所有 RW 段放在该部分
}
```

ZI +0

;ZI: ZI 段的名称

;+0: 表示 ZI 段紧接着 RW 段

```
{
    *(+ZI)      ;所有 ZI 段放在该部分
}
```

}

3. ADS 产生的各代码段宏

lImage\$\$RO\$\$Base /* RO 代码段起始地址 */

lImage\$\$RO\$\$Limit /* RO 代码段结束地址 */

lImage\$\$RW\$\$Base /* RW 代码段起始地址 */

lImage\$\$RW\$\$Limit /* RW 代码段结束地址 */

lImage\$\$ZI\$\$Base /* ZI 代码段起始地址 */

lImage\$\$ZI\$\$Limit /* ZI 代码段结束地址 */

注意:在两个\$\$之间的名称,与 scatter 中指定的段的名称相同.

4. 装载过程说明

当从 NorFlash 启动时,要把 flash 芯片的首地址映射到 0x00000000 位置,系统启动后,启动程序本身把自己从 flash 中搬到 RAM 中运行.搬移后的各段起始地址,由以上宏来确定.

当从 NandFlash 启动时,S3C2410 会自动把前 NandFlash 的前 4k 搬到 S3C2410 的内部 RAM 中,并把内部 RAM 的首地址设为 0x00000000, CPU 从 0x00000000 开始执行.所以,在 nandFlash 的前 4k 程序中,必须包含从 NandFlash 把 BootLoader 的其余部分装入 RAM 的程序.

2.1.3 启动过程的汇编部分

当系统启动时,ARM CPU 会跳到 0x00000000 去执行.一般 BootLoader 都包括如下几个部分:

1. 建立中断向量异常表
2. 显示的切换到 SVC 且 32 指令模式
3. 关闭 S3C2410 的内部看门狗
4. 禁止所有的中断
5. 配置系统时钟频率和总线频率
6. 设置内存区的控制寄存器
7. 初始化中断
8. 安装中断向量表
9. 把可执行文件的各个段搬到运行态的各个位置
10. 跳到 C 代码部分执行

2.1.4 启动过程的 C 部分

1. 初始化 MMU
2. 初始化外部端口
3. 中断处理程序表初始化
4. 串口初始化

5. 其它部分初始化(可选)
6. 主程序循环

3 AXD 的使用以及源代码说明

3.1 源代码说明

3.1.1 汇编源代码说明

```

;=====
; 引用头文件
;=====
        get bdinit.h

;=====
; 引用标准变量
;=====
        IMPORT |Image$$RO$$Base|    ; Base address of RO section
        IMPORT |Image$$RO$$Limit|   ; End address of RO section
        IMPORT |Image$$RW$$Base|    ; Base address of RW section
        IMPORT |Image$$RW$$Limit|   ; End address of RW section
        IMPORT |Image$$ZI$$Base|    ; Base address of ZI section
        IMPORT |Image$$ZI$$Limit|   ; End address of ZI section

        IMPORT bdmain      ; The entry function of C program

;=====
; 宏定义
;=====
; macro HANDLER
;     MACRO
$HandlerLabel HANDLER $HandleLabel

$HandlerLabel
        sub     sp,sp,#4          ;Decrement sp (to store jump address)
        stmfd   sp!,{r0}          ;PUSH the work register to stack
        ldr     r0,$HandleLabel;Load the address of HandleXXX to r0
        ldr     r0,[r0]           ;Load the contents(service routine start address) of HandleXXX
        str     r0,[sp,#4]        ;Store the contents(ISR) of HandleXXX to stack
        ldmfd   sp!,{r0,pc}       ;POP the work register and pc(jump to ISR)
        MEND

;=====
; 汇编语言的入口代码
;=====
        AREA    Init, CODE, READONLY
                CODE32

        ENTRY

;=====
; 建立中断向量表
;=====

```

```

b   reset_handler   ;0x00000000:   Reset (SVC)
b   undef_handler   ;0x00000004:   Undefined instruction (Undef)
b   swi_handler      ;0x00000008:   Software Interrupt (SVC)
b   iabr_handler     ;0x0000000C:   Instruction Abort (Abort)
b   dabr_handler     ;0x00000010:   Data Abort (Abort)
b   no_handler       ;0x00000014:
b   irq_handler      ;0x00000018:   IRQ (IRQ)
b   fiq_handler      ;0x0000001C:   FIQ (FIQ)

```

LTORG

```

undef_handler  HANDLER HandleUndef
swi_handler    HANDLER HandleSWI
iabr_handler   HANDLER HandlePabort
dabr_handler   HANDLER HandleDabort
no_handler     HANDLER HandleReserved
irq_handler    HANDLER HandleIRQ
fiq_handler    HANDLER HandleFIQ

```

```

;=====
; 复位时运行的主程序
;=====

```

reset_handler

```

;Set the cpu to SVC32 mode

```

```

mrs    r0,cpsr
bic    r0,r0,#0xf
orr    r0,r0,#0xd3
msr    cpsr_cxsf,r0

```

```

;Turn off watchdog

```

```

ldr    r0,=WTCON
ldr    r1,=0x0
str    r1,[r0]

```

```

;Disable all the first level interrupts

```

```

ldr    r0,=INTMSK
ldr    r1,=0xffffffff
str    r1,[r0]

```

```

;Disable all the second level interrupts

```

```

ldr    r0,=INTSUBMSK
ldr    r1,=0x7ff
str    r1,[r0]

```

```

;Configure MPLL

```

```

ldr    r0,=MPLLCON
ldr    r1,=((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV) ;Fin=12MHz,Fout=200MHz
str    r1,[r0]

```

```

;Set FCLK:HCLK:PCLK = 1:2:4

```

```

ldr    r0,=CLKDIVN
mov    r1,#3
str    r1,[r0]

```

```

;Set memory control registers

```

```

ldr r0,=SMRDATA

```

```

        ldr    r1,=BWSCON
        add    r2, r0, #52                ;End address of SMRDATA
0
        ldr    r3, [r0], #4
        str    r3, [r1], #4
        cmp    r2, r0
        bne    %B0

;Initialize stacks
        bl     InitStacks

;Setup IRQ handler
        ldr    r0,=HandleIRQ              ;This routine is needed
        ldr    r1,=IsrIRQ
        str    r1,[r0]

;Copy RW/ZI section into RAM
        ldr    r0, =|Image$$RO$$Limit| ;Get pointer to ROM data
        ldr    r1, =|Image$$RW$$Base| ;and RAM copy
        ldr    r3, =|Image$$ZI$$Base|

        cmp    r0, r1                    ; Check that they are different
        beq    %F2
1
        cmp    r1, r3                    ; Copy init data
        ldrcc  r2, [r0], #4               ;--> LDRCC r2, [r0] + ADD r0, r0, #4
        strcc  r2, [r1], #4               ;--> STRCC r2, [r1] + ADD r1, r1, #4
        bcc    %B1
2
        ldr    r1, =|Image$$ZI$$Limit| ; Top of zero init segment
        mov    r2, #0
3
        cmp    r3, r1                    ; Zero init
        strcc  r2, [r3], #4
        bcc    %B3

        bl     bdmain                    ;Jump to the main function

;Dead loop
1
        nop
        b      %B1

;=====
; 初始中断处理程序
;=====

IsrIRQ
        sub    sp,sp,#4                  ;reserved for PC
        stmfd  sp!,{r8-r9}

        ldr    r9,=INTOFFSET
        ldr    r9,[r9]
        ldr    r8,=HandleEINT0
        add    r8,r8,r9,ls1 #2
        ldr    r8,[r8]

```

```

    str    r8,[sp,#8]
    ldmfd  sp!,{r8-r9,pc}

```

```

;=====
; 初始化各个模式下堆栈
;=====

```

InitStacks

```

    mrs    r0,cpsr
    bic    r0,r0,#MODEMASK
    orr    r1,r0,#UNDEFMODE|NOINT
    msr    cpsr_cxsf,r1          ;UndefMode
    ldr    sp,=UndefStack

    orr    r1,r0,#ABORTMODE|NOINT
    msr    cpsr_cxsf,r1          ;AbortMode
    ldr    sp,=AbortStack

    orr    r1,r0,#IRQMODE|NOINT
    msr    cpsr_cxsf,r1          ;IRQMode
    ldr    sp,=IRQStack

    orr    r1,r0,#FIQMODE|NOINT
    msr    cpsr_cxsf,r1          ;FIQMode
    ldr    sp,=FIQStack

    bic    r0,r0,#MODEMASK|NOINT
    orr    r1,r0,#SVCMODE
    msr    cpsr_cxsf,r1          ;SVCMode
    ldr    sp,=SVCStack

    mov    pc,lr                ;Return the call routine

```

LTORG

```

;=====
; 内存区控制寄存器值表; 你可根据需要修改 bdinit.h 文件, 下面代码不用做任何改动
;=====

```

SMRDATA DATA

```

    DCD
    ((0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(
    B7_BWSCON<<28)))

    DCD
    ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<<2)+(B0_PMC))    ;GCS0

    DCD
    ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp<<2)+(B1_PMC))    ;GCS1

    DCD
    ((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tacp<<2)+(B2_PMC))    ;GCS2

    DCD
    ((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tacp<<2)+(B3_PMC))    ;GCS3

    DCD
    ((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tacp<<2)+(B4_PMC))    ;GCS4

    DCD
    ((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tacp<<2)+(B5_PMC))    ;GCS5

    DCD ((B6_MT<<15)+(B6_Trld<<2)+(B6_SCAN))          ;GCS6
    DCD ((B7_MT<<15)+(B7_Trld<<2)+(B7_SCAN))          ;GCS7

```

```

DCD ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
DCD 0x32          ;SCLK power saving mode, BANKSIZE 128M/128M
DCD 0x30          ;MRSR6 CL=3c1k
DCD 0x30          ;MRSR7

```

ALIGN

```

;=====
; 异常及中断向量表空间; 安装异常或中断处理程序在 bdisr.c 中,isr_setup()来完成.
;=====

```

AREA RamData, DATA, READWRITE

^ _ISR_STARTADDRESS ;表示下面数据区从_ISR_STARTADDRESS 指定的位置开始

```

HandleReset      # 4
HandleUndef      # 4
HandleSWI        # 4
HandlePabort     # 4
HandleDabort     # 4
HandleReserved   # 4
HandleIRQ        # 4
HandleFIQ        # 4

```

```

;=====
; The Interrupt table
;=====

```

```

HandleEINT0      # 4
HandleEINT1      # 4
HandleEINT2      # 4
HandleEINT3      # 4
HandleEINT4_7    # 4
HandleEINT8_23   # 4
HandleRSV6       # 4
HandleBATFLT     # 4
HandleTICK       # 4
HandleWDT        # 4
HandleTIMER0     # 4
HandleTIMER1     # 4
HandleTIMER2     # 4
HandleTIMER3     # 4
HandleTIMER4     # 4
HandleUART2      # 4
HandleLCD        # 4
HandleDMA0       # 4
HandleDMA1       # 4
HandleDMA2       # 4
HandleDMA3       # 4
HandleMMC        # 4
HandleSPI0       # 4
HandleUART1      # 4
HandleRSV24      # 4
HandleUSBBD      # 4
HandleUSBH       # 4
HandleIIC        # 4
HandleUART0      # 4
HandleSPI1       # 4

```

```
HandleRTC      # 4
HandleADC      # 4
```

END

3.1.2 C 语言源代码说明

```
void bdmain(void)
{
    /* 禁止 Cache 和 MMU */
    cache_disable();
    mmu_disable();

    /* 端口初始化 */
    port_init();

    /* 中断处理程序 */
    isr_init();

    /* 串口初始化 */
    serial_init(0, 115200);

    /* 输出信息进行主循环 */
    serial_printf("is ok!\n");
    while(1) {

    }
}
```

通常基本 ADS 的测试程序都可以在这个架构上加入自己的代码。

3.2 AXD 的使用

3.2.1 配置仿真器

1. 为仿真器安装 Server

一般的仿真器都对应有一个 Server 程序，所以在进行在线仿真之前，必须先安装这个 Server 程序。我使用的是 Dragon-ICE 仿真器，所以先要安装 Dragon-ICE Server 程序。

2. 连接仿真器

把 dragon-ICE 仿真器的 JTAG 口连接上 ARM 板(注意：ARM 板要断电连接)，另一端通过并口连接到 PC 上，有的仿真器是通过 USB 口连接到 PC 上，这与仿真器的硬件相关。连接好后，打开 ARM 电源，启动 ARM 板。当 ARM 通电启动后，启动 Dragon-ICE Server 检测 ARM 板，详细步骤及设置参见对应的仿真器手册。我的 dragon-ICE Server 启动，按“自动检测”可以检测到 ARM920T。

3.2.2 启动 AXD 配置开发板

1. 启动 AXD

先启动 Dragon-ICE Server 程序。

按如下步骤启动 AXD：

开始->所有程序->ARM Developer Suite v1.2->AXD Debugger

2. 装载仿真器库文件

从 AXD 菜单的 Options--> Configure Target...启动“Choose Target”目标板配置窗口。

在“Choose Target”窗口中，点击“Add”按钮，选择仿真器的库文件。我的仿真器服务器程序安装在 c:\Dragon-ICE 下，所以选择项 c:\Dragon-ICE\dragon-ice.dll 文件。

3. 为 AXD 在线仿真配置仿真器

在"Target Environments"中选中 Dragon-ICE 中,点击右边的"Configure"按钮。

在" FJB Dragon-ICE Release v1.2"窗口点击"This computer..."按钮,再点击"OK"按钮。

回到" Choose Target"窗口,点击"OK"按钮。完成配置。

回到主界面,在右边的" Target"窗口会出现 ARM920T_0.这表明 AXD 已经进入 ARM 板的在线仿真状态。

点击菜单"System Views"-->"Controls Monitors".会出现"ARM920T-Register"窗口.此时,会显示当前 ARM 板上所有寄存器的状态。

4. 配置 ARM 板

如果 ARM 板通电后,没有程序运行并把内存区控制寄存器配置好的说,外部 RAM 是不能使用的. 所以必须通过仿真器来设置这些寄存器. 如果 ARM 板已经有启动程序并且已经配置好,这一步可以省略。

首先把 2410cfg.txt 拷贝到 c:\下。

回到 AXD 主界面,从菜单" System Views" --> "Command Line Interface"。会出现一个 Command Line Interface 的调试命令行窗口,并显示如下提示符:

Debug >

输入 obey c:\2410cfg.txt 装载所有配置命令。

Debug >obey c:\2410cfg.txt

5. 2410cfg.txt 文件说明

sreg psr, 0x00000013

;设置当前 CPSR 的值,把 CPU 的模式切换到 SVC 模式和 32 位指令集,关闭 IRQ 和 FIQ。

smem 0x53000000,0,32

;设置看门狗控制寄存器 WTCN

;禁止看门狗定时器

smem 0x4C000004,((0x74<<12)+(0x3<<4)+0x1),32

;设置主频率设置寄存器 MPLLCON

;目前 CPU 的工作频率 FCLK 是 124.00MHz

smem 0x4C000014,0x3,32

;设置时钟分频寄存器 CLKDIVN

;设置 FCLK/HCLK/PCLK 的频率比例 1:2:4

smem 0x48000000,((2<<28)+(2<<24)+(1<<20)+(1<<16)+(1<<12)+(1<<8)+(1<<4)+0),32

;设置内存总线控制 BWSCON

;SDRAM BANK 6&7 is 32 位

;其它 BANK is 16 位

smem 0x48000004,((3<<13)+(3<<11)+(7<<8)+(3<<6)+(3<<4)+(3<<2)+3),32

;设置寄存器区 0 控制寄存器: BANKCON0

smem 0x4800001c,((3<<15)+(1<<2)+1),32

;设置寄存器区 6 控制寄存器: BANKCON6(SDRAM)

;RAS to CAS 延时 3 时钟周期

;列地址是 9 位

smem 0x48000020,((3<<15)+(1<<2)+1),32

;设置寄存器区 7 控制寄存器: BANKCON7(SDRAM)

;RAS to CAS 延时 3 时钟周期

;列地址是 9 位

smem 0x48000024,((1<<23)+(3<<18)+(2<<16)+1113),32

;set 外部 RAM 刷新寄存器: REFRESH

;允许自刷新

;HCLK=FCLK/2, 60MHz,刷新计算器是 1113

smem 0x48000028,0x31,32

```
;设置寄存器的大小
;禁止 burst 操作
;允许 SDRAM power down 模式
;SCLK 在访问期间仍在活动状态
;SDRAM 模式寄存器设置
smem 0x4800002c,0x30,32
smem 0x48000030,0x30,32
```

3.2.3 使用 AXD 在线仿真调试程序

1. 装载可执行的文件

AXD 只支持 .axf 格式的可执行文件.

启动 AXD, 在菜单的 File 中,选择 Load Image..., 选择 c:\adsbloadter\prj\prj_Data\DebugRel\prj.axf 加载执行 image. 就可以执行并调试了. AXD 提供了非常方便的调试手段, 包括在线单步, 自由设置断点等.

第三部分 GNU 交叉工具链

1 设置环境变量, 准备源码及相关补丁

1.1 设置环境变量

```
[arm@localhost arm]#vi ~/.bashrc
export PREFIX=/usr/local/arm/3.4.4
export TARGET=arm-linux
export SYSROOT=${PREFIX}/sysroot
export ARCH=arm
export CROSS_COMPILE=${TARGET}-
export PATH=${PREFIX}/bin:$PATH
export SRC=/home/arm/dev_home/btools/tchain3.4.4
```

1.2 准备源码包

1.2.1 binutils

名称: binutils-2.16.tar.gz

下载地址: <http://ftp.gnu.org/gnu/binutils/binutils-2.16.tar.gz>

1.2.2 gcc

名称: gcc-3.4.4.tar.bz2

下载地址: <http://ftp.gnu.org/gnu/gcc/gcc-3.4.4/gcc-3.4.4.tar.bz2>

1.2.3 glibc

名称: glibc-2.3.5.tar.gz

glibc-linuxthreads-2.3.5.tar.gz

下载地址: <http://ftp.gnu.org/gnu/glibc/glibc-2.3.5.tar.gz>

<http://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.3.5.tar.gz>

1.2.4 linux kernel

名称: linux-2.6.14.1.tar.gz

下载地址: <http://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.1.tar.gz>

1.3 准备补丁

1.3.1 *ioperm.c.diff*

作用: 打修正 `ioperm()` 函数.

下载地址: <http://frank.harvard.edu/~coldwell/toolchain/ioperm.c.diff>

1.3.2 *flow.c.diff*

作用: 该补丁用于产生 `crti.o` 和 `crtm.o` 文件。

下载地址: http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/flow.c.diff?cvsroot=gcc&only_with_tag=cs1-arm-branch&r1=1.563.4.2&r2=1.563.4.3

1.3.3 *t-linux.diff*

作用: 修改 `gcc` 一处 `bug`

下载地址: <http://frank.harvard.edu/~coldwell/toolchain/t-linux.diff>

1.4 编译 GNU binutils

重新以 `arm` 用户登陆, 让新设置的环境变量起作用.

```
[arm@localhost arm]#su arm
[arm@localhost arm]#cd ${SRC}
[arm@localhost tchain3.4.4]#tar xzvf binutils-2.16.tar.gz
[arm@localhost tchain3.4.4]#mkdir -p BUILD/binutils-2.16
[arm@localhost binutils-2.16]#cd BUILD/binutils-2.16
[arm@localhost binutils-2.16]# ../binutils-2.16/configure --prefix=${PREFIX} --target=${TARGET} \
--with-sysroot=${SYSROOT}
[arm@localhost binutils-2.16]#make
[arm@localhost binutils-2.16]#su root
[root@localhost binutils-2.16]#make install
[root@localhost binutils-2.16]#exit
[arm@localhost binutils-2.16]#
```

1.5 准备内核头文件

1.5.1 使用当前平台的 *gcc* 编译内核头文件

```
[arm@localhost tchain3.4.4]#cd ${KERNEL}
[arm@localhost kernel]#tar xvfz linux-2.6.14.1.tar.gz
[arm@localhost kernel]#cd linux-2.6.14.1
[arm@localhost linux-2.6.14.1]#make ARCH=arm menuconfig
[arm@localhost linux-2.6.14.1]#make
```

1.5.2 复制内核头文件

```
[arm@localhost kernel]#su root
[root@localhost kernel]#mkdir -p ${SYSROOT}/usr/include
```

```
[root@localhost kernel]#cp -a include/linux ${SYSROOT}/usr/include/linux
[root@localhost kernel]#cp -a include/asm-i386 ${SYSROOT}/usr/include/asm
[root@localhost kernel]#cp -a include/asm-generic ${SYSROOT}/usr/include/asm-generic
[root@localhost kernel]#exit
[arm@localhost kernel]#
```

1.6 译编 glibc 头文件

```
[arm@localhost kernel]#cd ${SRC}
[arm@localhost chain3.4.4]#tar xvfz glibc-2.3.5.tar.gz
[arm@localhost chain3.4.4]#patch -d glibc-2.3.5 -p1 < ioperm.c.diff
[arm@localhost glibc-2.3.5]#cd glibc-2.3.5
[arm@localhost glibc-2.3.5]#tar xvfz ../glibc-linuxthreads-2.3.5.tar.gz
[arm@localhost chain3.4.4]#cd ..
[arm@localhost chain3.4.4]#mkdir BUILD/glibc-2.3.5-headers
[arm@localhost chain3.4.4]#cd BUILD/glibc-2.3.5-headers
[arm@localhost glibc-2.3.5-headers]#../glibc-2.3.5/configure --prefix=/usr --host=${TARGET} \
--enable-add-ons=linuxthreads --with-headers=${SYSROOT}/usr/include
[arm@localhost glibc-2.3.5-headers]#su root
[root@localhost glibc-2.3.5-headers]#make cross-compiling=yes install_root=${SYSROOT} install-headers
[root@localhost glibc-2.3.5-headers]#touch ${SYSROOT}/usr/include/gnu/stubs.h
[root@localhost glibc-2.3.5-headers]#touch ${SYSROOT}/usr/include/bits/stdio_lim.h
[root@localhost glibc-2.3.5-headers]#exit
[arm@localhost glibc-2.3.5-headers]#
```

注意: --prefix=/usr 是 gcc 寻找库的搜索路径。

1.7 编译 gcc 第一阶段

```
[arm@localhost glibc-2.3.5-headers]#cd ${SRC}
[arm@localhost chain3.4.4]#tar xjvf gcc-3.4.4.tar.bz2
[arm@localhost chain3.4.4]#patch -d gcc-3.4.4 -p1 < flow.c.diff
[arm@localhost chain3.4.4]#patch -d gcc-3.4.4 -p1 < t-linux.diff
[arm@localhost chain3.4.4]#mkdir -p BUILD/gcc-3.4.4-stage1
[arm@localhost chain3.4.4]#cd BUILD/gcc-3.4.4-stage1
[arm@localhost gcc-3.4.4-stage1]#../gcc-3.4.4/configure --prefix=${PREFIX} --target=${TARGET} \
--enable-languages=c --with-sysroot=${SYSROOT}
```

注意: 不能加上"--disable-shared"选项。

```
[arm@localhost gcc-3.4.4-stage1]#make all-gcc
[arm@localhost gcc-3.4.4-stage1]#su root
[root@localhost gcc-3.4.4-stage1]#make install-gcc
[root@localhost gcc-3.4.4-stage1]#exit
[arm@localhost gcc-3.4.4-stage1]#
```

1.8 编译完整的 glibc

```
[arm@localhost gcc-3.4.4-stage1] #cd ${SRC}
[arm@localhost tchain3.4.4]#mkdir BUILD/glibc-2.3.5
[arm@localhost tchain3.4.4]#cd BUILD/glibc-2.3.5
[arm@localhost glibc-2.3.5]#BUILD_CC=gcc CC=${CROSS_COMPILE} gcc AR=${CROSS_COMPILE} ar \
RANLIB=${CROSS_COMPILE} ranlib AS=${CROSS_COMPILE} as LD=${CROSS_COMPILE} ld \
../glibc-2.3.5/configure --prefix=/usr --build=i386-redhat-linux --host=arm-unknown-linux-gnu \
--target=arm-unknown-linux-gnu --without-__thread --enable-add-ons=linuxthreads \
--with-headers=${SYSROOT}/usr/include
```

说明:

- prefix: 指定安装路径。
- target: 指定目标平台。
- host: 指定当前平台。
- build: 指定编译平台。
- with-sysroot: 用于指定编译所需要的头文件，及链接库。
- enable-add-ons: 加入其它的库，如线程库等。
- enable-languages: 指定 gcc 所支持的语言。

```
[arm@localhost glibc-2.3.5]#make
[arm@localhost glibc-2.3.5]#su root
[root@localhost glibc-2.3.5]#make install_root=${SYSROOT} install
[root@localhost glibc-2.3.5]#exit
[arm@localhost glibc-2.3.5]#
```

1.9 编译完整的 gcc

```
[arm@localhost glibc-2.3.5]#cd ${SRC}
[arm@localhost tchain3.4.4]#mkdir BUILD/gcc-3.4.4
[arm@localhost tchain3.4.4]#cd BUILD/gcc-3.4.4
[arm@localhost gcc-3.4.4]#./../gcc-3.4.4/configure --prefix=${PREFIX} --target=${TARGET} \
--enable-languages=c --with-sysroot=${SYSROOT}
[arm@localhost gcc-3.4.4]#make
[arm@localhost gcc-3.4.4]#su root
[root@localhost gcc-3.4.4]#make install
[root@localhost gcc-3.4.4]#exit
[arm@localhost gcc-3.4.4]#
```

2 GNU 交叉工具链的下载

2.1 ARM 官方网站

工具链的官方下载地址:

<http://www.arm.linux.org.uk>

可以从该站点下载 2.95.3, 3.0 以及 3.2 工具链

<ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-2.95.3.tar.bz2>

<ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-3.0.tar.bz2>

<ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/cross-3.2.tar.bz2>

3 GNU 交叉工具链的介绍与使用

3.1 常用工具介绍

名称	归属	作用
arm-linux-as	binutils	编译 ARM 汇编程序
arm-linux-ar	binutils	把多个.o 合并成一个.o 或静态库(.a)
arm-linux-ranlib	binutils	为库文件建立索引，相当于 arm-linux-ar -s
arm-linux-ld	binutils	连接器(Linker), 把多个.o 或库文件连接成一个可执行文件

名称	归属	作用
arm-linux-objdump	binutils	查看目标文件(.o)和库(.a)的信息
arm-linux-objcopy	binutils	转换可执行文件的格式
arm-linux-strip	binutils	去掉 elf 可执行文件的信息. 使可执行文件变小
arm-linux-readelf	binutils	读 elf 可执行文件的信息
arm-linux-gcc	gcc	编译.c 或.S 开头的 C 程序或汇编程序
arm-linux-g++	gcc	编译 c++ 程序

3.2 主要工具的使用

3.2.1 arm-linux-gcc 的使用

1. 编译 C 文件, 生成 elf 可执行文件

h1.c 源文件

```
#include <stdio.h>
void hellofirst(void)
{
    printf("The first hello! \n");
}
```

h2.c 源文件

```
#include <stdio.h>
void hellosecond(void)
{
    printf("The second hello! \n");
}
```

hello.c 源文件

```
#include <stdio.h>
void hellosecond(void);
void hellofirst(void);

int main(int argc, char *argv[])
{
    hellofirst();
    hellosecond();
    return(0);
}
```

编译以上 3 个文件, 有如下几种方法:

方法 1:

```
[arm@localhost gcc]#arm-linux-gcc -c h1.c
[arm@localhost gcc]#arm-linux-gcc -c h2.c
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c h1.o h2.o
```

方法 2:

```
[arm@localhost gcc]#arm-linux-gcc -c h1.c h2.c
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c h1.o h2.o
```

方法 3:

```
[arm@localhost gcc]#arm-linux-gcc -c -o h1.o h1.c
```

```
[arm@localhost gcc]#arm-linux-gcc -c -o h1.o h1.c
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c h1.o h2.o
```

方法 4:

```
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c h1.c h2.c
```

-c: 只编译不连接。

-o: 编译且连接。

2. 产生一个预处理文件

当要看一个宏在源文件中产生的结果时, 比较合适。

```
[arm@localhost gcc]#arm-linux-gcc -E h1.i h1.c
```

-E: 产生一个预处理文件。

3. 产生一个动态库

动态库是在运行时需要的库。

```
[arm@localhost gcc]#arm-linux-gcc -c -fpic h1.c h2.c
```

```
[arm@localhost gcc]#arm-linux-gcc -shared h1.o h2.o -o hello.so
```

```
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c hello.so
```

把 hello.so 拷贝到目标板的/lib 目录下,把可执行文件拷贝目标板的/tmp 目录下,在目标板上运行 hello.

```
#/tmp/hello
```

或把 hello.so 和 hello 一起拷贝到/tmp 目标下, 并设置 LD_LIBRARY_PATH 环境变量

```
#export LD_LIBRARY_PATH =/tmp:$LD_LIBRARY_PATH
```

```
#/tmp/hello
```

3.2.2 arm-linux-ar 和 arm-linux-ranlib 的使用

静态库是在编译时需要的库。

1. 建立一个静态库

```
[arm@localhost gcc]#arm-linux-ar -r libhello.a h1.o h2.o
```

2. 为静态库建立索引

```
[arm@localhost gcc]#arm-linux-ar -s libhello.a
```

```
[arm@localhost gcc]#arm-linux-ranlib libhello.a
```

3. 由静态库产生可执行文件

```
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c -lhello -L./
```

```
[arm@localhost gcc]#arm-linux-gcc -o hello hello.c libhello.a
```

hello 文件可以直接拷贝到/tmp 目录下运行, 不需 libhello.a.

3.2.3 arm-linux-objdump 的使用

1. 查看静态库或.o 文件的组成文件

```
[arm@localhost gcc]$ arm-linux-objdump -a libhello.a
```

2. 查看静态库或.o 文件的组成部分的头部分

```
[arm@localhost gcc]$ arm-linux-objdump -h libhello.a
```

3. 把目标文件代码反汇编

```
[arm@localhost gcc]$ arm-linux-objdump -d libhello.a
```

3.2.4 arm-linux-readelf 的使用

1. 读 elf 文件开始的文件头部

```
[arm@localhost gcc]$ arm-linux-readelf -h hello
```

ELF Header:

```
Magic:  7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00
Class:                               ELF32
Data:                               2's complement, little endian
Version:                            1 (current)
OS/ABI:                             ARM
ABI Version:                        0
Type:                               EXEC (Executable file)
Machine:                             ARM
Version:                             0x1
Entry point address:                0x82b4
Start of program headers:          52 (bytes into file)
Start of section headers:         10240 (bytes into file)
Flags:                              0x2, has entry point
Size of this header:                52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:          6
Size of section headers:           40 (bytes)
Number of section headers:         28
Section header string table index: 25
```

2. 读 elf 文件中所有 ELF 的头部:

```
[arm@localhost gcc]#arm-linux-readelf -e hello
```

.....

3. 显示整个文件的符号表

```
[arm@localhost gcc]#arm-linux-readelf -s hello
```

.....

4. 显示使用的动态库

```
[arm@localhost gcc]#arm-linux-readelf -d hello
```

.....

3.2.5 arm-linux-strip 的使用

1. 移除所有的符号信息

```
[arm@localhost gcc]#cp hello hello1
```

```
[arm@localhost gcc]#arm-linux-strip -strip-all hello
```

--strip-all: 是移除所有符号信息

```
[arm@localhost gcc]#ll
```

```
-rwxr-xr-x 1 arm root 2856 7月  3 15:14 hello
```

```
-rwxr-xr-x 1 arm root 13682 7月  3 15:13 hello1
```

被 strip 后的 hello 程序比原来的 hello1 程序要小很多。

2. 移除调试符号信息

```
[arm@localhost gcc]#arm-linux-strip -g hello
```

```
[arm@localhost gcc]#ll
```

```
-rwxr-xr-x 1 arm root 4501 7月  3 15:17 hello
```

```
-rwxr-xr-x 1 arm root 13682 7月  3 15:13 hello1
```

3.2.6 arm-linux-copydump 的使用

生成可以执行的 2 进制代码

```
[arm@localhost gcc]#arm-linux-copydump -O binary hello hello.bin
```

4 ARM GNU 常用汇编语言介绍

4.1 ARM GNU 常用汇编伪指令介绍

1. abort

.abort: 停止汇编

.align abs-expr1, abs-expr2: 以某种对齐方式,在未使用的存储区域填充值. 第一个值表示对齐方式,4, 8,16 或 32. 第二个表达式值表示填充的值.

2. if...else...endif

.if

.else

.endif: 支持条件预编译

3. include

.include "file": 包含指定的头文件, 可以把一个汇编常量定义放在头文件中.

4. comm

.comm symbol, length: 在 bss 段申请一段命名空间,该段空间的名称叫 symbol, 长度为 length. Ld 连接器在连接会为其留出空间.

5. data

.data subsection: 说明接下来的定义归属于 subsection 数据段.

6. equ

.equ symbol, expression: 把某一个符号(symbol)定义成某一个值(expression).该指令并不分配空间.

7. global

.global symbol: 定义一个全局符号, 通常是为 ld 使用.

8. ascii

.ascii "string": 定义一个字符串并为之分配空间.

9. byte

.byte expressions: 定义一个字节, 并为之分配空间.

10. short

.short expressions: 定义一个短整型, 并为之分配空间.

11. int

.int expressions: 定义一个整型,并为之分配空间.

12 long

.long expressions: 定义一个长整型, 并为之分配空间.

13 word

.word expressions: 定义一个字,并为之分配空间, 4bytes.

14. macro/endum

.macro: 定义一段宏代码, .macro 表示代码的开始, .endum 表示代码的结束.

15. req

name .req register name: 为寄存器定义一个别名.

16. code

.code [16|32]: 指定指令代码产生的长度, 16 表示 Thumb 指令, 32 表示 ARM 指令.

17. ltorg

.ltorg: 表示当前往下的定义在归于当前段,并为之分配空间.

4.2 ARM GNU 专有符号

1. @

表示注释从当前位置到行尾的字符.

2.

注释掉一整行.

3. ;

新行分隔符.

4.3 操作码

1. NOP

`nop`

空操作, 相当于 `MOV r0, r0`

2. LDR

`ldr <register> , = <expression>`

相当于 PC 寄存器或其它寄存器的长转移.

3. ADR

`adr <register> <label>`

相于 PC 寄存器或其它寄存器的小范围转移.

`ADRL`

`adrl <register> <label>`

相于 PC 寄存器或其寄存器的中范围转移.

5 可执行生成说明

5.1 lds 文件说明

5.1.1 主要符号说明

1. OUTPUT_FORMAT(bfdname)

指定输出可执行文件格式.

2. OUTPUT_ARCH(bfdname)

指定输出可执行文件所运行 CPU 平台

3. ENTRY(symbol)

指定可执行文件的入口段

5.1.2 段定义说明

1. 段定义格式

`SECTIONS { ...`

```
    段名 : {  
        内容  
    }
```

`...`

`}`

5.1.3 u-boot.lds 文件说明

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")

;指定输出可执行文件是 elf 格式,32 位 ARM 指令,小端

OUTPUT_ARCH(arm)

;指定输出可执行文件的平台为 ARM

ENTRY(_start)

;指定输出可执行文件的起始代码段为 _start.

SECTIONS

```
{
    . = 0x00000000 ; 从 0x0 位置开始

    . = ALIGN(4) ; 代码以 4 字节对齐
    .text : ; 指定代码段
    {
        cpu/arm920t/start.o (.text) ; 代码的第一个代码部分
        *(.text) ; 其它代码部分
    }

    . = ALIGN(4)
    .rodata : { *(.rodata) } ; 指定只读数据段

    . = ALIGN(4);
    .data : { *(.data) } ; 指定读/写数据段

    . = ALIGN(4);
    .got : { *(.got) } ; 指定 got 段, got 段式是 uboot 自定义的一个段, 非标准段

    __u_boot_cmd_start = . ; 把 __u_boot_cmd_start 赋值为当前位置, 即起始位置
    .u_boot_cmd : { *(.u_boot_cmd) } ; 指定 u_boot_cmd 段, uboot 把所有的 uboot 命令放在该段.
    __u_boot_cmd_end = . ; 把 __u_boot_cmd_end 赋值为当前位置, 即结束位置

    . = ALIGN(4);
    __bss_start = . ; 把 __bss_start 赋值为当前位置, 即 bss 段的开始位置
    .bss : { *(.bss) } ; 指定 bss 段
    _end = . ; 把 _end 赋值为当前位置, 即 bss 段的结束位置
}
```

第四部分 u-boot 的移植

1 u-boot 的介绍及系统结构

1.1 u-boot 介绍

U-boot 是德国 DENX 小组的开发用于多种嵌入式 CPU 的 bootloader 程序, U-Boot 不仅仅支持嵌入式 Linux 系统的引导, 当前, 它还支持 NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS 嵌入式操作系统。U-Boot 除了支持 PowerPC 系列的处理器外, 还能支持 MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。

1.2 获取 u-boot

以 uboot 用户登陆。

```
[uboot@localhost ~]#mkdir -p dev_home/uboot
[uboot@localhost ~]#cd dev_home/uboot
从下面地址下载 u-boot 的源代码。
http://sourceforge.net/projects/u-boot
[uboot@localhost uboot]#tar -xjvf u-boot-1.1.4.tar.bz2
[uboot@localhost uboot]#cd u-boot-1.1.4
```

1.3 u-boot 体系结构

1.3.1 u-boot 目录结构

1. 目录树

```
[uboot@localhost u-boot-1.1.4]#tree -L 1 -d
```

```
.
|-- board
|-- common
|-- cpu
|-- disk
|-- doc
|-- drivers
|-- dtb
|-- examples
|-- fs
|-- include
|-- lib_arm
|-- lib_generic
|-- lib_i386
|-- lib_m68k
|-- lib_microblaze
|-- lib_mips
|-- lib_nios
|-- lib_nios2
|-- lib_ppc
|-- net
|-- post
|-- rtc
`-- tools
```

2. board: 和一些已有开发板有关的文件. 每一个开发板都以一个子目录出现在当前目录中, 比如说:SMDK2410, 子目录中存放与开发板相关的配置文件.

3. common: 实现 u-boot 命令行下支持的命令, 每一条命令都对应一个文件。例如 bootm 命令对应就是 cmd_bootm.c。

4. cpu: 与特定 CPU 架构相关目录, 每一款 U-boot 下支持的 CPU 在该目录下对应一个子目录, 比如有子目录 arm920t 等。

5. disk: 对磁盘的支持。

6. doc: 文档目录。U-boot 有非常完善的文档, 推荐大家参考阅读。

7. drivers: U-boot 支持的设备驱动程序都放在该目录, 比如各种网卡、支持 CFI 的 Flash、串口和 USB 等。

8. fs: 支持的文件系统, U-boot 现在支持 cramfs、fat、fdos、jffs2 和 registerfs。

9. include: U-boot 使用的头文件, 还有对各种硬件平台支持的汇编文件, 系统的配置文件和对文件系统支持的文件。该目录下 configs 目录有与开发板相关的配置头文件, 如 smdk2410.h。该目录下的 asm 目录有与 CPU 体

系结构相关的头文件，asm 对应的是 asm-arm.

9. lib_XXXX: 与体系结构相关的库文件。如与 ARM 相关的库放在 lib_arm 中。

10. net: 与网络协议栈相关的代码，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现。

11. tools: 生成 U-boot 的工具，如: mkimage, crc 等等。

2 uboot 的启动过程及工作原理

2.1 启动模式介绍

大多数 Boot Loader 都包含两种不同的操作模式: "启动加载"模式和"下载"模式, 这种区别仅对于开发人员才有意义。但从最终用户的角度看, Boot Loader 的作用就是用来加载操作系统, 而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载 (Boot loading) 模式: 这种模式也称为"自主" (Autonomous) 模式。也即 Boot Loader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行, 整个过程并没有用户的介入。这种模式是 Boot Loader 的正常工作模式, 因此在嵌入式产品发布的时候, Boot Loader 显然必须工作在这种模式下。

下载 (Downloading) 模式: 在这种模式下, 目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机 (Host) 下载文件, 比如: 下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Boot Loader 保存到目标机的 RAM 中, 然后再被 BootLoader 写到目标机上的 FLASH 类固态存储设备中。Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用; 此外, 以后的系统更新也会使用 Boot Loader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。

U-Boot 这样功能强大的 Boot Loader 同时支持这两种工作模式, 而且允许用户在这两种工作模式之间进行切换。

大多数 bootloader 都分为阶段 1(stage1)和阶段 2(stage2)两大部分, u-boot 也不例外。依赖于 CPU 体系结构的代码 (如 CPU 初始化代码等) 通常都放在阶段 1 中且通常用汇编语言实现, 而阶段 2 则通常用 C 语言来实现, 这样可以实现复杂的功能, 而且有更好的可读性和移植性。

2.2 阶段 1 介绍

u-boot 的 stage1 代码通常放在 start.s 文件中, 它用汇编语言写成, 其主要代码部分如下:

2.2.1 定义入口

由于一个可执行的 Image 必须有一个入口点, 并且只能有一个全局入口, 通常这个入口放在 ROM(Flash)的 0x0 地址, 因此, 必须通知编译器以使其知道这个入口, 该工作可通过修改连接器脚本来完成。

1. board/crane2410/u-boot.lds: ENTRY(_start) ==> cpu/arm920t/start.S: .globl _start
2. uboot 代码区 (TEXT_BASE = 0x33F80000) 定义在 board/crane2410/config.mk

2.2.2 设置异常向量

```
_start: b      reset                @ 0x00000000
        ldr     pc, _undefined_instruction @ 0x00000004
        ldr     pc, _software_interrupt   @ 0x00000008
        ldr     pc, _prefetch_abort       @ 0x0000000c
        ldr     pc, _data_abort           @ 0x00000010
        ldr     pc, _not_used              @ 0x00000014
        ldr     pc, _irq                   @ 0x00000018
        ldr     pc, _fiq                   @ 0x0000001c
```

当发生异常时，执行 `cpu/arm920t/interrupts.c` 中定义的中断处理函数。

2.2.3 设置CPU的模式为SVC模式

```
mrs    r0,cpsr
bic    r0,r0,#0x1f
orr    r0,r0,#0xd3
msr    cpsr,r0
```

2.2.4 关闭看门狗

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
    ldr    r0, =pWTCN
    mov    r1, #0x0    @ 根据三星手册进行调整。
    str    r1, [r0]
```

2.2.5 禁掉所有中断

```

        mov     r1, #0xffffffff
        ldr     r0, =INTMSK
        str     r1, [r0]
# if defined(CONFIG_S3C2410)
        ldr     r1, =0x3ff
        ldr     r0, =INISUBMSK
        str     r1, [r0]

```

2.2.6 设置以 CPU 的频率

默认频率为 FCLK:HCLK:PCLK = 1:2:4, 默认 FCLK 的值为 120 MHz, 该值为 S3C2410 手册的推荐值。

```
ldr    r0, =CLKDIVN
mov    r1, #3
str    r1, [r0]
```

2.2.7 设置CP15

设置 CP15, 失效指令(I)Cache 和数据(D)Cache 后, 禁止 MMU 与 Cache。

[illegible]

```

        */
        orr    r0, r0, #0x00000002    /* set bit 2 (A) Align, 1 = Fault checking enabled */
        orr    r0, r0, #0x00001000    /* set bit 12 (I) I-Cache, 1 = Instruction cache enabled
*/
        mcr    pl5, 0, r0, c1, c0, 0

```

2.2.8 配置内存区控制寄存器

配置内存区控制寄存器，寄存器的具体值通常由开发板厂商或硬件工程师提供。如果您对总线周期及外围芯片非常熟悉，也可以自己确定，在 U-BOOT 中的设置文件是 board/crane2410/lowlevel_init.S，该文件包含 lowleve_init 程序段。详细寄存器设置及值的解释见 3.2.2 启动 AXD 配置开发板一节中的第 5 点。

```

        mov    ip, lr
        bl     lowlevel_init
        mov    lr, ip

```

2.2.9 安装 U-BOOT 使用的栈空间

下面这段代码只对不是从 Nand Flash 启动的代码段有意义，对从 Nand Flash 启动的代码，没有意义。因为从 Nand Flash 中把 UBOOT 执行代码搬移到 RAM，由 2.1.9 中代码完成。

```

#ifdef CONFIG_SKIP_RELOCATE_UBOOT
...
#endif
stack_setup:
        ldr    r0, _TEXT_BASE          /* 代码段的起始地址 */
        sub    r0, r0, #CFG_MALLOC_LEN /* 分配的动态内存区 */
        sub    r0, r0, #CFG_GBL_DATA_SIZE /* UBOOT 开发板全局数据存放 */
#ifdef CONFIG_USE_IRQ
        /* 分配 IRQ 和 FIQ 栈空间 */
        sub    r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
        sub    sp, r0, #12              /* 留下 3 个字为 Abort */

```

2.2.10 BSS 段清 0

```

clear_bss:
        ldr    r0, _bss_start          /* BSS 段的起始地址 */
        ldr    r1, _bss_end            /* BSS 段的结束地址 */
        mov    r2, #0x00000000          /* BSS 段置 0 */

c1bss_1:str    r2, [r0]                /* 循环清除 BSS 段 */
        add    r0, r0, #4
        cmp    r0, r1
        ble    c1bss_1

```

2.2.11 搬移 Nand Flash 代码

从 Nand Flash 中，把数据拷贝到 RAM，是由 copy_myself 程序段完成，该程序段详细解释见：第七部分的 3.1 节。

```

#ifdef CONFIG_S3C2410_NAND_BOOT
        bl     copy_myself

        @ jump to ram
        ldr    r1, =on_the_ram
        add    pc, r1, #0
        nop
        nop

```

```

1:    b    1b        @ infinite loop

on_the_ram:
#endif

```

2.2.12 进入 C 代码部分

```

ldr    pc, _start_armboot
_start_armboot: .word start_armboot

```

2.3 阶段 2 的 C 语言代码部分

lib_arm/board.c 中的 start armboot 是 C 语言开始的函数，也是整个启动代码中 C 语言的主函数，同时还是整个 u-boot(armboot) 的主函数，该函数主要完成如下操作：

2.3.1 调用一系列的初始化函数

1. 指定初始函数表：

```

init_fnc_t *init_sequence[] = {
    cpu_init,           /* cpu 的基本设置          */
    board_init,         /* 开发板的基本初始化      */
    interrupt_init,     /* 初始化中断              */
    env_init,           /* 初始化环境变量          */
    init_baudrate,      /* 初始化波特率            */
    serial_init,        /* 串口通讯初始化          */
    console_init_f,     /* 控制台初始化第一阶段    */
    display_banner,     /* 通知代码已经运行到该处  */
    dram_init,          /* 配制可用的内存区        */
    display_dram_config,
#ifdef CONFIG_VCMA9 || defined (CONFIG_CMC_PU2)
    checkboard,
#endif
    NULL,
};

```

执行初始化函数的代码如下：

```

for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
    if ((*init_fnc_ptr)() != 0) {
        hang ();
    }
}

```

2. 配置可用的 Flash 区

```
flash_init ()
```

3. 初始化内存分配函数

```
mem_malloc_init()
```

4. nand flash 初始化

```

#ifdef CONFIG_COMMANDS & CFG_CMD_NAND
    puts ("NAND:");
    nand_init();           /* 初始化 NAND */

```

见第七部分 3.2.3 节中的第 3 点 nand_init() 函数。

5. 初始化环境变量

```
env_relocate ();
```

6. 外围设备初始化

```
devices_init()
```

7. I2C 总线初始化

```
i2c_init();
```

8. LCD 初始化

```
drv_lcd_init();
```

9. VIDEO 初始化

```
drv_video_init();
```

10. 键盘初始化

```
drv_keyboard_init();
```

11. 系统初始化

```
drv_system_init();
```

2.3.2 初始化网络设备

初始化相关网络设备，填写 IP、MAC 地址等。

1. 设置 IP 地址

```
/* IP Address */
gd->bd->bi_ip_addr = getenv_IPAddr ("ipaddr");

/* MAC Address */
{
    int i;
    ulong reg;
    char *s, *e;
    uchar tmp[64];

    i = getenv_r ("ethaddr", tmp, sizeof (tmp));
    s = (i > 0) ? tmp : NULL;

    for (reg = 0; reg < 6; ++reg) {
        gd->bd->bi_enetaddr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
        if (s)
            s = (*e) ? e + 1 : e;
    }
}
```

2.3.3 进入主 UBOOT 命令行

进入命令循环（即整个 boot 的工作循环），接受用户从串口输入的命令，然后进行相应的工作。

```
for (;;) {
    main_loop (); /* 在 common/main.c */
}
```

2.4 代码搬运

为了支持 NAND flash 启动, S3C2410 内建了内部的 4k 的 SRAM 缓存 “Steppingstone”。当启动时, NAND flash 最初的 4k 字节将被读入” Steppingstone”然后开始执行启动代码。通常启动代码会把 NAND flash 中的内容拷到 SDRAM 中以便执行主代码。

使用硬件的 ECC, NAND flash 中的数据的有效性将会得到检测。

功能

1. NAND flash 模式: 支持读/删除/编程 NAND Flash
2. 自动启动模式: 在复位时启动代码将被读入” Steppingstone”中, 然后开始执行启动代码。
3. 硬件 ECC 检测模块 (硬件检测, 软件纠正)
4. “Steppingstone” 4-KB 内部 SRAM 在启动后可以另外使用。

3 uboot 的移植过程

3.1 环境

详细环境设置参见:第一部分的 2.2.2.

1. 工作用户
uboot
2. u-boot 版本 1.1.4
获取 u-boot1.1.4 请看 1.2
3. 工具链 2.95.3

3.2 步骤

我们为开发板取名叫: crane2410, 并在 u-boot 中建立自己的开发板类型

3.2.1 修改 Makefile

```
[uboot@localhost uboot]#vi Makefile
```

#为 crane2410 建立编译项

```
crane2410_config : unconfig
```

```
@./mkconfig $(@:_config=) arm arm920t crane2410 NULL s3c24x0
```

各项的意思如下:

- arm: CPU 的架构(ARCH)
- arm920t: CPU 的类型(CPU), 其对应于 cpu/arm920t 子目录。
- crane2410: 开发板的型号(BOARD), 对应于 board/crane2410 目录。
- NULL: 开发者/或经销商(vender)。
- s3c24x0: 片上系统(SOC)。

3.2.2 在 board 子目录中建立 crane2410

```
[uboot@localhost uboot]#cp -rf board/smdk2410 board/crane2410
```

```
[uboot@localhost uboot]#cd board/crane2410
```

```
[uboot@localhost crane2410]#mv smdk2410.c crane2410.c
```

3.2.3 在 include/configs/中建立配置头文件

```
[uboot@localhost crane2410]#cd ../..
```

```
[uboot@localhost uboot]#cp include/configs/smdk2410.h include/configs/crane2410.h
```

3.2.4 指定交叉编译工具的路径

```
[uboot@localhost uboot]#vi ~/.bashrc
export PATH=/usr/local/arm/2.95.3/bin:$PATH
```

3.2.5 测试编译能否成功

```
[uboot@localhost uboot]#make crane2410_config
[uboot@localhost uboot]#make CROSS_COMPILE=arm-linux-
```

3.2.6 修改 lowlevel_init.S 文件

依照开发板的内存区的配置情况, 修改 board/crane2410/lowlevel_init.S 文件, 我的更改如下:

```
#include <config.h>
#include <version.h>

#define BWSCON 0x48000000

/* BWSCON */
#define DW8 (0x0)
#define DW16 (0x1)
#define DW32 (0x2)
#define WAIT (0x1<<2)
#define UBLB (0x1<<3)

#define B1_BWSCON (DW16)
#define B2_BWSCON (DW16)
#define B3_BWSCON (DW16 + WAIT + UBLB)
#define B4_BWSCON (DW16)
#define B5_BWSCON (DW16)
#define B6_BWSCON (DW32)
#define B7_BWSCON (DW32)

/* BANK0CON */
#define B0_Tacs 0x3 /* 0c1k */
#define B0_Tcos 0x3 /* 0c1k */
#define B0_Tacc 0x7 /* 14c1k */
#define B0_Tcoh 0x3 /* 0c1k */
#define B0_Tah 0x3 /* 0c1k */
#define B0_Tacp 0x3
#define B0_PMC 0x3 /* normal */

/* BANK1CON */
#define B1_Tacs 0x3 /* 0c1k */
#define B1_Tcos 0x3 /* 0c1k */
#define B1_Tacc 0x7 /* 14c1k */
#define B1_Tcoh 0x3 /* 0c1k */
#define B1_Tah 0x3 /* 0c1k */
#define B1_Tacp 0x3
#define B1_PMC 0x0

#define B2_Tacs 0x0
#define B2_Tcos 0x0
#define B2_Tacc 0x7
#define B2_Tcoh 0x0
#define B2_Tah 0x0
```

```

#define B2_Tacp      0x0
#define B2_PMC       0x0

#define B3_Tacs      0x0    /* 0c1k */
#define B3_Tcos      0x3    /* 4c1k */
#define B3_Tacc      0x7    /* 14c1k */
#define B3_Tcoh      0x1    /* 1c1k */
#define B3_Tah       0x0    /* 0c1k */
#define B3_Tacp      0x3    /* 6c1k */
#define B3_PMC       0x0    /* normal */

#define B4_Tacs      0x0    /* 0c1k */
#define B4_Tcos      0x0    /* 0c1k */
#define B4_Tacc      0x7    /* 14c1k */
#define B4_Tcoh      0x0    /* 0c1k */
#define B4_Tah       0x0    /* 0c1k */
#define B4_Tacp      0x0
#define B4_PMC       0x0    /* normal */

#define B5_Tacs      0x0    /* 0c1k */
#define B5_Tcos      0x0    /* 0c1k */
#define B5_Tacc      0x7    /* 14c1k */
#define B5_Tcoh      0x0    /* 0c1k */
#define B5_Tah       0x0    /* 0c1k */
#define B5_Tacp      0x0
#define B5_PMC       0x0    /* normal */

#define B6_MT        0x3    /* SDRAM */
#define B6_Trcd      0x1
#define B6_SCAN      0x1    /* 9bit */

#define B7_MT        0x3    /* SDRAM */
#define B7_Trcd      0x1    /* 3c1k */
#define B7_SCAN      0x1    /* 9bit */

/* REFRESH parameter */
#define REFEN        0x1    /* Refresh enable */
#define TREFMD       0x0    /* CBR(CAS before RAS)/Auto refresh */
#define Trp          0x0    /* 2c1k */
#define Trc          0x3    /* 7c1k */
#define Tchr         0x2    /* 3c1k */
#define REFCNT       1113    /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */
/* **** */

_TEXT_BASE:
    .word TEXT_BASE

.globl lowlevel_init
lowlevel_init:
    /* memory control configuration */
    /* make r0 relative the current location so that it */
    /* reads SMRDATA out of FLASH rather than memory ! */
    ldr    r0, =SMRDATA
    ldr    r1, _TEXT_BASE
    sub    r0, r0, r1

```

```

        ldr    r1, =BWSCON /* Bus Width Status Controller */
        add    r2, r0, #13*4
0:
        ldr    r3, [r0], #4
        str    r3, [r1], #4
        cmp    r2, r0
        bne    0b

        /* everything is fine now */
        mov    pc, lr

        .ltorg
/* the literal pools origin */

SMRDATA:
        .word
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(
B7_BWSCON<<28))
        .word
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<<2)+(B0_PMC))
        .word
((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp<<2)+(B1_PMC))
        .word
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tacp<<2)+(B2_PMC))
        .word
((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tacp<<2)+(B3_PMC))
        .word
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tacp<<2)+(B4_PMC))
        .word
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tacp<<2)+(B5_PMC))
        .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
        .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
        .word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
        .word 0x31
        .word 0x30
        .word 0x30

```

3.2.9 UBOOT 的 Nand Flash 移植

UBOOT 的 Nand Flash 支持见第七部分的第 3 节。

3.2.8 重新编译 u-boot

[uboot@localhost uboot1.1.4]make CROSS_COMPILE=arm-linux-

3.2.9 把 u-boot 烧入 flash

1. 通过仿真器烧入 u-boot

通过仿真器 u-boot 烧写到 flash 中就可以从 NAND flash 启动了。

2. 通过 JTAG 接口，由工具烧入 flash

4 U-BOOT 命令的使用

4.1 U-BOOT 命令的介绍

U-BOOT 常用命令

通常使用 `help`(或者只使用问号?), 来查看所有的 U-BOOT 命令。将会列出在当前配置下所有支持的命令。但是我们要注意, 尽管 U-BOOT 提供了很多配置选项, 并不是所有选项都支持各种处理器和开发板, 有些选项可能在你的配置中并没有被选上。

4.1.1 获得帮助信息

通过 `help` 可以获得当前开发板的 U-BOOT 中支持的命令。

```
CRANE2410 # help
?          - alias for 'help'
autoscr    - run script from memory
base       - print or set address offset
bdinfo     - print Board Info structure
boot       - boot default, i.e., run 'bootcmd'
bootd      - boot default, i.e., run 'bootcmd'
bootelf    - Boot from an ELF image in memory
bootm      - boot application image from memory
bootp      - boot image via network using BootP/TFTP protocol
bootvx     - Boot vxWorks from an ELF image
cmp        - memory compare
coninfo    - print console devices and information
cp         - memory copy
crc32      - checksum calculation
date       - get/set/reset date & time
dcache     - enable or disable data cache
echo       - echo args to console
erase      - erase FLASH memory
flinfo     - print FLASH memory information
go         - start application at address 'addr'
help       - print online help
icache     - enable or disable instruction cache
iminfo     - print header information for application image
imls       - list all images found in flash
itest      - return true/false on integer compare
loadb      - load binary file over serial line (kermit mode)
loads      - load S-Record file over serial line
loop       - infinite loop on address range
md         - memory display
mm         - memory modify (auto-incrementing)
mtest      - simple RAM test
mw         - memory write (fill)
nand       - NAND sub-system
nboot      - boot from NAND device
nfs        - boot image via network using NFS protocol
nm         - memory modify (constant address)
ping       - send ICMP ECHO_REQUEST to network host
printenv   - print environment variables
protect    - enable or disable FLASH write protection
rarpboot   - boot image via network using RARP/TFTP protocol
reset      - Perform RESET of the CPU
run        - run commands in an environment variable
saveenv    - save environment variables to persistent storage
setenv     - set environment variables
sleep      - delay execution for some time
```

```
tftpboot- boot image via network using TFTP protocol
version - print monitor version
```

4.2 常用命令使用说明

4.2.1 askenv(F)

在标准输入 (stdin) 获得环境变量。

4.2.2 autoscr

从内存 (Memory) 运行脚本。(注意, 从下载地址开始, 例如我们的开发板是从 0x30008000 处开始运行)。

```
CRANE2410 # autoscr 0x30008000
## Executing script at 30008000
```

4.2.3 base

打印或者设置当前指令与下载地址的地址偏移。

4.2.4 bdfinfo

打印开发板信息

```
CRANE2410 # bdfinfo
-arch_number = 0x000000C1 (CPU 体系结构号)
-env_t       = 0x00000000 (环境变量)
-boot_params = 0x30000100 (启动引导参数)
-DRAM bank   = 0x00000000 (内存区)
--> start    = 0x30000000 (SDRAM 起始地址)
--> size     = 0x04000000 (SDRAM 大小)
-ethaddr     = 01:23:45:67:89:AB (以太网地址)
-ip_addr     = 192.168.1.5 (IP 地址)
-baudrate    = 115200 bps (波特率)
```

4.2.5 bootp

通过网络使用 Bootp 或者 TFTP 协议引导镜像文件。

```
CRANE2410 # help bootp
bootp [loadAddress] [bootfilename]
```

4.2.6 bootelf

默认从 0x30008000 引导 elf 格式的文件(vmlinux)

```
CRANE2410 # help bootelf
bootelf [address] - load address of ELF image.
```

4.2.7 bootd(=boot)

引导的默认命令, 即运行 U-BOOT 中在 “include/configs/smdk2410.h” 中设置的 “bootcmd” 中的命令。如下:

```
#define CONFIG_BOOTCOMMAND "tftp 0x30008000 uImage; bootm 0x30008000";
```

在命令下做如下试验:

```
CRANE2410 # set bootcmd printenv
CRANE2410 # boot
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
```

```
CRANE2410 # bootd
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
```

4.2.8 tftp(tftpboot)

即将内核镜像文件从 PC 中下载到 SDRAM 的指定地址，然后通过 **bootm** 来引导内核，前提是所用 PC 要安装设置 **tftp** 服务。

下载信息：

```
CRANE2410 # tftp 0x30008000 zImage
TFTP from server 10.0.0.1; our IP address is 10.0.0.110
Filename 'zImage'.
Load address: 0x30008000
Loading: #####
          #####
          #####
done
Bytes transferred = 913880 (df1d8 hex)
```

4.2.9 bootm

内核的入口地址开始引导内核。

```
CRANE2410 # bootm 0x30008000
## Booting image at 30008000 ...
Starting kernel ...
Uncompressing
Linux.....
done, .
```

4.2.10 go

直接跳转到可执行文件的入口地址，执行可执行文件。

```
CRANE2410 # go 0x30008000
## Starting application at 0x30008000 ...
```

4.2.11 cmp

对输入的两段内存地址进行比较。

```
CRANE2410 # cmp 0x30008000 0x30008040 64
word at 0x30008000 (0xe321f0d3) != word at 0x30008040 (0xc022020c)
Total of 0 words were the same
```

```
CRANE2410 # cmp 0x30008000 0x30008000 64
Total of 100 words were the same
```

4.2.12 coninfo

打印所有控制设备和信息，例如

```
-List of available devices:
-serial    80000003 SIO stdin stdout stderr
```

4.2.13 cp

内存拷贝，**cp** 源地址 目的地址 拷贝大小（字节）

```
CRANE2410 # help cp
cp [.b, .w, .l] source target count
ANE2410 # cp 0x30008000 0x3000f000 64
```

4.2.14 date

获得/设置/重设日期和时间

```
CRANE2410 # date
Date: 2006-6-6 (Tuesday)    Time: 06:06:06
```

4.2.15 erase(F)

擦除 FLASH MEMORY, 由于该 ARM 板没有 Nor Flash, 所有不支持该命令.

```
CRANE2410 # help erase
erase start end
    - erase FLASH from addr 'start' to addr 'end'
erase start +len
    - erase FLASH from addr 'start' to the end of sect w/addr 'start'+ 'len'-1
erase N:SF[-SL]
    - erase sectors SF-SL in FLASH bank # N
erase bank N
    - erase FLASH bank # N
erase all
    - erase all FLASH banks
```

4.2.16 flinfo(F)

打印 Nor Flash 信息, 由于该 ARM 板没有 Nor Flash, 所有不支持该命令.

4.2.17 iminfo

打印和校验内核镜像头, 内核的起始地址由 CFG_LOAD_ADDR 指定:

```
#define CFG_LOAD_ADDR 0x30008000 /* default load address */
```

该宏在 include/configs/crane2410.h 中定义.

```
CRANE2410 # iminfo
## Checking Image at 30008000 ...
Image Name:   Linux-2.6.14.1
Created:      2006-06-28  7:43:01 UTC
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    1047080 Bytes = 1022.5 kB
Load Address: 30008000
Entry Point:  30008040
Verifying Checksum ... OK
```

4.2.18 loadb

从串口下载二进制文件

```
CRANE2410 # loadb
## Ready for binary (kermit) download to 0x30008000 at 115200 bps...
## Total Size      = 0x00000000 = 0 Bytes
## Start Addr      = 0x30008000
```

4.2.19 md

显示指定内存地址中的内容

```
CRANE2410 # md 0
00000000: ea000012 e59ff014 e59ff014 e59ff014 .....
00000010: e59ff014 e59ff014 e59ff014 e59ff014 .....
00000020: 33f80220 33f80280 33f802e0 33f80340 ..3...3@..3
00000030: 33f803a0 33f80400 33f80460 deadbeef ...3...3`..3...
00000040: 33f80000 33f80000 33f9c0b4 33fa019c ...3...3...3...3
00000050: e10f0000 e3c0001f e38000d3 e129f000 .....).
```

```

00000060: e3a00453 e3a01000 e5801000 e3e01000 S.....
00000070: e59f0444 e5801000 e59f1440 e59f0440 D.....@...@...
00000080: e5801000 e59f043c e3a01003 e5801000 ....<.....
00000090: eb000051 e24f009c e51f1060 e1500001 Q.....0.`.....P.
000000a0: 0a000007 e51f2068 e51f3068 e0432002 ....h ..h0... C.
000000b0: e0802002 e8b007f8 e8a107f8 e1500002 . ....P.
000000c0: daffffff e51f008c e2400803 e2400800 .....@...@.
000000d0: e240d00c e51f0094 e51f1094 e3a02000 ..@..... ..
000000e0: e5802000 e2800004 e1500001 daffffffb . ....P.....
000000f0: eb000006 e59f13d0 e281f000 e1a00000 .....

```

4.2.20 mm

顺序显示指定地址往后的内存中的内容,可同时修改,地址自动递增。

```

CRANE2410 # mm 0x30008000
30008000: e1a00000 ? fffff
30008004: e1a00000 ? eeeeeee
30008008: e1a00000 ? q

CRANE2410 # md 30008000
30008000: 000fffff 00eeeeee e1a00000 e1a00000 .....
30008010: e1a00000 e1a00000 e1a00000 e1a00000 .....
30008020: ea000002 016f2818 00000000 000df1d8 .....(o.....
30008030: e1a07001 e3a08000 e10f2000 e3120003 .p.....

```

4.2.21 mtest

简单的 RAM 检测

```

CRANE2410 # mtest
Pattern FFFFFFFD Writing... Reading...

```

4.2.22 mw

向内存地址写内容

```

CRANE2410 # md 30008000
30008000: fffffdfe fffffdfe fffffdfe fffffdfe .....

CRANE2410 # mw 30008000 0 4
CRANE2410 # md 30008000
30008000: 00000000 00000000 00000000 00000000 .....

```

4.2.23 nm

修改内存地址,地址不递增

```

CRANE2410 # nm 30008000
30008000: de4c457f ? 00000000
30008000: 00000000 ? 11111111
30008000: 11111111 ?

```

4.2.24 printenv

打印环境变量

```

CRANE2410 # printenv
bootdelay=3
baudrate=115200
ethaddr=01:23:45:67:89:ab
ipaddr=10.0.0.110
serverip=10.0.0.1
netmask=255.255.255.0
stdin=serial
stdout=serial

```

stderr=serial

Environment size: 153/65532 bytes

4.2.25 ping

ping 主机

```
CRANE2410 # ping 10.0.0.1  
host 10.0.0.1 is alive
```

4.2.26 reset

复位 CPU

4.2.27 run

运行已经定义好的 U-BOOT 的命令

```
CRANE2410 # set myenv ping 10.0.0.1  
CRANE2410 # run myenv  
host 10.0.0.1 is alive
```

4.2.28 saveenv(F)

保存设定的环境变量

4.2.29 setenv

设置环境变量

```
CRANE2410 # setenv ipaddr 10.0.0.254  
CRANE2410 # printenv  
ipaddr=10.0.0.254
```

4.2.30 sleep

命令延时执行时间

```
CRANE2410 # sleep 1
```

4.2.31 version

打印 U-BOOT 版本信息

```
CRANE2410 # version
```

U-Boot 1.1.4 (Jul 4 2006 - 12:42:27)

4.2.32 nand info

打印 nand flash 信息

```
CRANE2410 # nand info  
Device 0: Samsung K9F1208U0B at 0x4e000000 (64 MB, 16 kB sector)
```

4.2.33 nand device <n>

显示某个 nand 设备

```
CRANE2410 # nand device 0
```

```
Device 0: Samsung K9F1208U0B at 0x4e000000 (64 MB, 16 kB sector)  
... is now current device
```

4.2.34 nand bad

```
CRANE2410 # nand bad  
Device 0 bad blocks:
```

4.2.35 nand read

nand read InAddr FlAddr size
InAddr: 从 nand flash 中读到内存的起始地址。
FlAddr: nand flash 的起始地址。
size: 从 nand flash 中读取的数据的大小。

```
CRANE2410 # nand read 0x30008000 0 0x100000
NAND read: device 0 offset 0, size 1048576 ...
1048576 bytes read: OK
```

4.2.36 nand erase

nand erase FlAddr size
FlAddr: nand flash 的起始地址
size: 从 nand flash 中擦除数据块的大小

```
CRANE2410 # nand erase 0x100000 0x20000
NAND erase: device 0 offset 1048576, size 131072 ... OK
```

4.2.37 nand write

nand write InAddr FlAddr size
InAddr: 写到 Nand Flash 中的数据在内存的起始地址
FlAddr: Nand Flash 的起始地址
size: 数据的大小

```
CRANE2410 # nand write 0x30f00000 0x100000 0x20000
NAND write: device 0 offset 1048576, size 131072 ...
131072 bytes written: OK
```

4.2.37 nboot

u-boot-1.1.4 代码对于 nboot 命令的帮助不正确, 修改如下:

正确的顺序为:

nboot InAddr dev FlAddr

InAddr: 需要装载到的内存的地址。

FlAddr: 在 nand flash 上 uImage 存放的地址

dev: 设备号

需要提前设置环境变量, 否则 nboot 不会调用 bootm

```
CRANE2410 #setenv autostart yes
```

```
CRANE2410 # nboot 30008000 0 100000
```

```
Loading from device 0: <NULL> at 0x4e000000 (offset 0x100000)
```

```
Image Name:   Linux-2.6.14.3
```

```
Created:      2006-07-06   7:31:52 UTC
```

```
Image Type:   ARM Linux Kernel Image (uncompressed)
```

```
Data Size:    897428 Bytes = 876.4 kB
```

```
Load Address: 30008000
```

```
Entry Point:  30008040
```

```
Automatic boot of image at addr 0x30008000 ...
```

```
## Booting image at 30008000 ...
```

```
Starting kernel ...
```

4.3 命令简写说明

所以命令都可以简写, 只要命令前面的一部分不会跟其它命令相同, 就可以不用写全整个命令。

```
save 命令
CRANE2410 # sa
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...Erasing sector 10 ... Erased 1 sectors
```

4.4 把文件写入 NandFlash

如果把一个传到内存中的文件写入到 Nand Flash 中，如：新的 uboot.bin, zImage(内核), rootfs 等，如果做呢？我们可以用 Nand Flash 命令来完成。但是 Nand Flash 写时，必须先要把 Nand Flash 的写入区全部擦除后，才能写。下面以把内存 0x30008000 起长度为 0x20000 的内容写到 Nand Flash 中的 0x100000 为例。

```
CRANE2410 # nand erase 0x100000 20000
NAND erase: device 0 offset 1048576, size 131072 ... OK
```

```
CRANE2410 # nand write 0x30008000 0x100000 0x20000
NAND write: device 0 offset 1048576, size 131072 ...
131072 bytes written: OK
```

5 参考资料

1. u-boot 在 s3c2410 开发板上移植 (NAND Flash Boot) 过程

<http://dev.csdn.net/article/84/84538.shtml>

第五部分 linux 2.6 内核的移植

1 内核移植过程

1.1 下载 linux 内核

从 <http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.1.tar.bz2>

下载 linux-2.6.14.1 内核至 home/arm/dev_home/kernel.

```
[root@localhost ~]#su arm
[arm@localhost ~]#cd $KERNEL
[arm@localhost kernel]#tar -xzf linux-2.6.14.1.tar.gz
[arm@localhost kernel]# pwd
/home/arm/dev_home/kernel
[arm@localhost kernel]# cd linux-2.6.14
```

进入内核解压后的目录，以后示例中，只要是相对路径全部是相对于 /home/arm/dev_home/kernel/linux-2.6.14/此目录

1.2 修改 Makefile

修改内核目录树根下的的 Makefile,指明交叉编译器

```
[arm@localhost linux-2.6.14]# vi Makefile
找到 ARCH 和 CROSS_COMPILE, 修改
ARCH      ?= arm
CROSS_COMPILE  ?= arm-linux-
```

然后设置你的 PATH 环境变量，使其可以找到你的交叉编译工具链

```
[arm@localhost linux-2.6.14]# echo $PATH
```

```
/usr/local/arm/3.4.4/bin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/ly/bin
```

如果/usr/local/arm/3.4.4/bin 搜索路径, 加入下面语句在~/.bashrc 中

```
[arm@localhost linux-2.6.14]# vi ~/.bashrc
```

```
export PATH=/usr/local/arm/3.4.4/bin:$PATH
```

再重新登陆.

```
[arm@localhost linux-2.6.14]#su arm
```

1.3 设置 flash 分区

此处一共要修改 3 个文件, 分别是:

1.3.1 指明分区信息

在 arch/arm/mach-s3c2410/devs.c 文件中:

```
[arm@localhost linux-2.6.14]$ vi arch/arm/mach-s3c2410/devs.c
```

添加如下内容:

```
#include <linux/mtd/partitions.h>
```

```
#include <linux/mtd/nand.h>
```

```
#include <asm/arch/nand.h>
```

```
...
```

```
/* NAND Controller */
```

1.建立 Nand Flash 分区表

/* 一个 Nand Flash 总共 64MB, 按如下大小进行分区 */

```
static struct mtd_partition partition_info[] = {
```

```
    { /* 1MB */
```

```
        name: "bootloader",
```

```
        size: 0x00100000,
```

```
        offset: 0x0,
```

```
    }, { /* 3MB */
```

```
        name: "kernel",
```

```
        size: 0x00300000,
```

```
        offset: 0x00100000,
```

```
    }, { /* 40MB */
```

```
        name: "root",
```

```
        size: 0x02800000,
```

```
        offset: 0x00400000,
```

```
    }, { /* 20MB */
```

```
        name: "user",
```

```
        size: 0x00f00000,
```

```
        offset: 0x02d00000,
```

```
    }
```

```
};
```

name: 代表分区名字

size: 代表 flash 分区大小(单位: 字节)

offset: 代表 flash 分区的起始地址(相对于 0x0 的偏移)

目标板计划分 4 个区, 分别存放 bootloader, kernel, rootfs 以及以便以后扩展使用的用户文件系统空间。

各分区在 Nand flash 中起始地址. 分区大小. 记录如下:

```

bootloader:
    start: 0x00000000
    len: 0x00100000
    1MB
kernel:
    start: 0x00100000
    len: 0x00300000
    3MB
rootfs:
    start: 0x00400000
    len: 0x02800000
    40MB
User:
    start: 0x02c00000
    len: 0x01400000
    20MB

```

2. 加入 Nand Flash 分区

```

struct s3c2410_nand_set nandset = {
    nr_partitions: 4,          /* the number of partitions */
    partitions: partition_info, /* partition table */
};

```

nr_partitions: 指明 partition_info 中定义的分区数目

partitions: 分区信息表

3. 建立 Nand Flash 芯片支持

```

struct s3c2410_platform_nand superlpplatform = {
    tacs: 0,
    twrph0: 30,
    twrph1: 0,
    sets: &nandset,
    nr_sets: 1,
};

```

tacs, twrph0, twrph1 的意思见 S3C2410 手册的 6-3, 这 3 个值最后会被设置到 NFCONF 中, 见 S3C2410 手册 6-6.

sets: 支持的分区集

nr_set: 分区集的个数

4. 加入 Nand Flash 芯片支持到 Nand Flash 驱动

另外, 还要修改此文件中的 s3c_device_nand 结构体变量, 添加对 dev 成员的赋值

```

struct platform_device s3c_device_nand = {
    .name      = "s3c2410-nand", /* Device name */
    .id        = -1,             /* Device ID */
    .num_resources = ARRAY_SIZE(s3c_nand_resource),
    .resource   = s3c_nand_resource, /* Nand Flash Controller Registers */

    /* Add the Nand Flash device */
    .dev = {
        .platform_data = &superlpplatform
    }
};

```

name: 设备名称

id: 有效设备编号,如果只有唯一的一个设备为-1, 有多个设备从 0 开始计数.

num_resource: 有几个寄存器区

resource: 寄存器区数组首地址

dev: 支持的 Nand Flash 设备

1.3.2 指定启动时初始化

kernel 启动时依据我们对分区的设置进行初始配置

修改 arch/arm/mach-s3c2410/mach-smdk2410.c 文件

[arm@localhost linux-2.6.14]\$ vi arch/arm/mach-s3c2410/mach-smdk2410.c

修改 smdk2410_devices[],指明初始化时包括我们在前面所设置的 flash 分区信息

```
static struct platform_device *smdk2410_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,

    /* 添加如下语句即可 */
    &s3c_device_nand,
};
```

保存,退出。

1.3.3 禁止 Flash ECC 校验

我们的内核都是通过 UBOOT 写到 Nand Flash 的, UBOOT 通过的软件 ECC 算法产生 ECC 校验码, 这与内核校验的 ECC 码不一样, 内核中的 ECC 码是由 S3C2410 中 Nand Flash 控制器产生的. 所以, 我们在这里选择禁止内核 ECC 校验.

修改 drivers/mtd/nand/s3c2410.c 文件:

[arm@localhost linux-2.6.14]\$ vi drivers/mtd/nand/s3c2410.c

找到 s3c2410_nand_init_chip()函数, 在该函数体最后加上一条语句:

```
chip->eccmode = NAND_ECC_NONE;
```

保存,退出。

OK.我们的关于 flash 分区的设置全部完工.

1.4 配置内核

1.4.1 支持启动时挂载 devfs

为了我们的内核支持 devfs 以及在启动时并在/sbin/init 运行之前能自动挂载/dev 为 devfs 文件系统, 修改 fs/Kconfig 文件

[arm@localhost linux-2.6.14]\$ vi fs/Kconfig

找到 menu "Pseudo filesystems"

添加如下语句:

```
config DEVFS_FS
    bool "/dev file system support (OBSOLETE)"
    default y
```

```
config DEVFS_MOUNT
```

```
bool "Automatically mount at boot"
default y
depends on DEVFS_FS
```

1.4.2 配置内核产生.config 文件

```
[arm@localhost linux-2.6.14]$ cp arch/arm/configs/smdk2410_defconfig .config
[arm@localhost linux-2.6.14]$ make menuconfig
```

在 smdk2410_defconfig 基础上，我所增删的内核配置项如下：

```
Loadable module support --->
    [*] Enable loadable module support
        [*] Automatic kernel module loading
```

```
System Type ---> [*] S3C2410 DMA support
```

```
Boot options ---> Default kernel command string:
    noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0,115200
```

```
#说明: mtdblock2 代表我的第 3 个 flash 分区，它是我的 rootfs
# console=ttySAC0,115200 使 kernel 启动期间的信息全部输出到串口 0 上.
# 2.6 内核对于串口的命名改为 ttySAC0，但这不影响用户空间的串口编程。
# 用户空间的串口编程针对的仍是/dev/ttyS0 等
```

```
Floating point emulation --->
    [*] NWFPE math emulation
        This is necessary to run most binaries!!!
```

#接下来要做的是对内核 MTD 子系统的设置

```
Device Drivers --->
    Memory Technology Devices (MTD) --->
        [*] MTD partitioning support
            #支持 MTD 分区，这样我们在前面设置的分区才有意义
        [*] Command line partition table parsing
            #支持从命令行设置 flash 分区信息，灵活
            RAM/ROM/Flash chip drivers --->
                <*> Detect flash chips by Common Flash
                    Interface (CFI) probe
                <*> Detect non-CFI AMD/JEDEC-compatible flash chips
                <*> Support for Intel/Sharp flash chips
                <*> Support for AMD/Fujitsu flash chips
                <*> Support for ROM chips in bus mapping
            NAND Flash Device Drivers --->
                <*> NAND Device Support
                <*> NAND Flash support for S3C2410/S3C2440 SoC

    Character devices --->
        [*] Non-standard serial port support
        [*] S3C2410 RTC Driver
```

#接下来做的是针对文件系统的设置，本人实验时目标板上要上的文件系统是 cramfs,故做如下配置

File systems --->

<> Second extended fs support #去除对 ext2 的支持

Pseudo filesystems --->

[*] /proc file system support

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

[*] /dev file system support (OBSOLETE)

[*] Automatically mount at boot (NEW)

#这里会看到我们前修改 fs/Kconfig 的成果, devfs 已经被支持上了

Miscellaneous filesystems --->

<*> Compressed ROM file system support (cramfs)

#支持 cramfs

Network File Systems --->

<*> NFS file system support

保存退出, 产生.config 文件.

.config 文件能从提供的 2.4.14.1 的内核包中找到, 文件名为 config.back.

1.4.3 编译内核

```
[arm@localhost linux-2.6.14]$ make zImage
```

注意: 若编译内核出现如下情况

```
LD      .tmp_vmlinux1
```

```
arm-linux-ld:arch/arm/kernel/vmlinux.lds:1439: parse error
```

```
make: *** [.tmp_vmlinux1] Error 1
```

解决方法: 修改 arch/arm/kernel/vmlinux.lds

```
[arm@localhost linux-2.6.14]$ vi arch/arm/kernel/vmlinux.lds
```

将文件尾 2 条的 ASSERT 注释掉 (1439 行)

```
/* ASSERT((__proc_info_end - __proc_info_begin), "missing CPU support") */
```

```
/* ASSERT((__arch_info_end - __arch_info_begin), "no machine record defined") */
```

然后重新 make zImage 即可

1.4.4 下载 zImage 到开发板

```
CRANE2410 # tftp 0x30008000 zImage
```

```
TFTP from server 192.168.1.6; our IP address is 192.168.1.5
```

```
Filename 'zImage'.
```

```
Load address: 0x30008000
```

```
Loading: #####
```

```
#####
```

```
#####
```

```
#####
```

```
done
```

```
Bytes transferred = 1142856 (117048 hex)
```

```
CRANE2410 # bootm 0x30008000
```

1.4.5 目标板启动信息如下

```
IRQ Stack: 33fc149c
```

```
FIQ Stack: 33fc249c
```

1

1
DRAM Configuration:
Bank #0: 30000000 64 MB
1
NAND:64 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
zImage magic = 0x016f2818
NOW, Booting Linux.....
Uncompressing Linux..... don.Linux version 2.6.14.1 (arm@dozec) (gcc
version 3.3.2) #15 Thu Jul 6 14:26:29 CST 2006
CPU: ARM920Tid(wb) [41129200] revision 0 (ARMv4T)
Machine: SMDK2410
Warning: bad configuration page, trying to continue
Memory policy: ECC disabled, Data cache writeback
CPU S3C2410A (id 0x32410002)
S3C2410: core 202.800 MHz, memory 101.400 MHz, peripheral 50.700 MHz
S3C2410 Clocks, (c) 2004 Simtec Electronics
CLOCK: Slow mode (1.500 MHz), fast, MPLL on, UPLL on
CPU0: D VIVT write-back cache
CPU0: I cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets
CPU0: D cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets
Built 1 zonelists
Kernel command line: noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0,115200
irq: clearing subpending status 00000002
PID hash table entries: 128 (order: 7, 2048 bytes)
timer tcon=00500000, tcnt a509, tcfg 00000200,00000000, usec 00001e4c
Console: colour dummy device 80x30
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory: 16MB = 16MB total
Memory: 13712KB available (1927K code, 422K data, 104K init)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
softlockup thread 0 started up.
NET: Registered protocol family 16
S3C2410: Initialising architecture
SCSI subsystem initialized
usbcore: registered new driver usbfs
usbcore: registered new driver hub
S3C2410 DMA Driver, (c) 2003-2004 Simtec Electronics
DMA channel 0 at c1800000, irq 33
DMA channel 1 at c1800040, irq 34
DMA channel 2 at c1800080, irq 35
DMA channel 3 at c18000c0, irq 36
NetWinder Floating Point Emulator V0.97 (double precision)
devfs: 2004-01-31 Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
Console: switching to colour frame buffer device 80x25
fb0: Virtual frame buffer device, using 1024K of video memory

S3C2410 RTC, (c) 2004 Simtec Electronics
s3c2410_serial0 at MMIO 0x50000000 (irq = 70) is a S3C2410
s3c2410_serial1 at MMIO 0x50004000 (irq = 73) is a S3C2410
s3c2410_serial2 at MMIO 0x50008000 (irq = 76) is a S3C2410
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Cirrus Logic CS8900A driver for Linux (Modified for SMDK2410)
eth0: CS8900A rev E at 0xe0000300 irq=53, no eeprom , addr: 08: 0:3E:26:0A:5B
S3C24XX NAND Driver, (c) 2004 Simtec Electronics
s3c2410-nand: mapped registers at c1980000
s3c2410-nand: timing: Tacls 10ns, Twrph0 30ns, Twrph1 10ns
NAND device: Manufacturer ID: 0xec, Chip ID: 0x76 (Samsung NAND 64MiB 3,3V 8-bit)
NAND_ECC_NONE selected by board driver. This is not recommended !!
Scanning device for bad blocks
Creating 4 MTD partitions on "NAND 64MiB 3,3V 8-bit":
0x00000000-0x00100000 : "bootloader"
0x00100000-0x00500000 : "kernel"
0x00500000-0x02d00000 : "root"
0x02d00000-0x03c00000 : "User"
usbmon: debugfs is not available
116x: driver isp116x-hcd, 05 Aug 2005
s3c2410-ohci s3c2410-ohci: S3C24XX OHCI
s3c2410-ohci s3c2410-ohci: new USB bus registered, assigned bus number 1
s3c2410-ohci s3c2410-ohci: irq 42, io mem 0x49000000
usb usb1: Product: S3C24XX OHCI
usb usb1: Manufacturer: Linux 2.6.14.1 ohci_hcd
usb usb1: SerialNumber: s3c24xx
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 2 ports detected
sl811: driver sl811-hcd, 19 May 2005
usbcore: registered new driver cdc_acm
drivers/usb/class/cdc-acm.c: v0.23:USB Abstract Control Model driver for USB modems and ISDN
adaptesdrivers/usb/class/bluetty.c: USB Bluetooth support registered
usbcore: registered new driver bluetty
drivers/usb/class/bluetty.c: USB Bluetooth tty driver v0.13
usbcore: registered new driver usblp
drivers/usb/class/usblp.c: v0.13: USB Printer Device Class driver
Initializing USB Mass Storage driver...
usbcore: registered new driver usb-storage
USB Mass Storage support registered.
mice: PS/2 mouse device common for all mice
NET: Registered protocol family 2
IP route cache hash table entries: 256 (order: -2, 1024 bytes)
TCP established hash table entries: 1024 (order: 0, 4096 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP reno registered
TCP bic registered
NET: Registered protocol family 1

```

NET: Registered protocol family 17
Reading data from NAND FLASH without ECC is not recommended
VFS: Mounted root (cramfs filesystem) readonly.
Mounted devfs on /dev
Freeing init memory: 104K
Reading data from NAND FLASH without ECC is not recommended
mount /etc as ramfs
re-create the /etc/mntab entries
-----mount /dev/shm as tmpfs
-----mount /proc as proc
-----mount /sys as sysfs
init started: BusyBox v1.1.3 (2006.07.03-03:43+0000) multi-call binary
Starting pid 28, console /dev/tts/0: '/etc/init.d/rcS'
in /etc/init.d/rcS
-----/sbin/ifconfig eth0 192.168.1.5

```

Please press Enter to activate this console.

#

1.5 Linux 下 cs8900a 的移植说明

1.5.1 为 cs8900a 建立编译菜单

1. 拷贝到文件

把 cs8900a 的压缩包拷贝到 arm 用户下的 dev_home/localapps/

```
[arm@localhost localapps]$ tar -xvzf cs8900a.tar.gz
```

```
[arm@localhost localapps]$ cd cs8900a
```

```
[arm@localhost cs8900a]$ cp cs8900a.c $KERNEL/linux-2.6.14.1/drivers/net/
```

```
[arm@localhost cs8900a]$ cp cs8900.h $KERNEL/linux-2.6.14.1/drivers/net/
```

2. 修改 Kconfig 文件

```
[arm@localhost cs8900a]$ vi $KERNEL/linux-2.6.14.1/drivers/net/Kconfig
```

#加入如下内容

```

config CS8900a
tristate "CS8900a support"
depends on NET_ETHERNET && ARM && ARCH_SMDK2410
---help---
Support for CS8900A chipset based Ethernet cards. If you have a network (Ether
net) card of this type, say Y and read the Ethernet-HOWTO, available from as
well as.
To compile this driver as a module, choose M here and read.
The module will be called cs8900.o.

```

3. 修改 Makefile 文件

```
[arm@localhost cs8900a]$ vi $KERNEL/linux-2.6.14.1/drivers/net/Makefile
```

#加入如下内容

```
obj-$(CONFIG_CS8900a) += cs8900a.o
```

1.5.2 修改 S3C2410 相关信息

1. 加入 CS8900A 在内存中的起始位置

```
[arm@localhost cs8900a]$cp reg-cs8900.h $KERNEL/linux-2.6.14.1/include/asm-arm/arch-s3c2410/
cs8900.h 的内容如下:
#ifndef _INCLUDE_CS8900A_H_
#define _INCLUDE_CS8900A_H_
```

```
#include <linux/config.h>
```

```
#define pSMDK2410_ETH_IO      0x19000000 /* S3C2410_CS3 0x18000000 */
#define vSMDK2410_ETH_IO      0xE0000000
#define SMDK2410_ETH_IRQ      IRQ_EINT9
```

```
#endif
```

2. 加入 cs8900A 的物理地址到虚拟地址的映射

```
[arm@localhost cs8900a]$vi $KERNEL/linux-2.6.14.1/arch/arm/mach-s3c2410/mach-smdk2410.c
/* 加入如下内容 */
static struct map_desc smdk2410_iodesc[] __initdata = {
    {vSMDK2410_ETH_IO, 0x19000000, SZ_1M, MT_DEVICE} /* Add this line */
};
```

2 创建 *uImage*

2.1 相关技术背景介绍

前面已经介绍了内核编译后，生成 *zImage* 的内核镜像文件。该镜像文件可以通过 *U-BOOT* 提供的 *go* 命令，跳转执行，引导内核。同时在 *u-boot-1.1.4* 的 *tools* 目录下，提供了生成 *uImage* 的工具 *mkimage* 命令，在生成 *u-boot* 的二进制镜像文件的同时，*mkimage* 命令会同时编译生成，无需另外编译。通过 *mkimage* 命令，在 *zImage* 中加入头文件（镜像头长 *0x40*，真正的内核入口向后偏移了 *0x40* 大小），生成 *uImage* 镜像文件，该文件就是执行 *bootm* 所需的内核镜像文件。

2.2 在内核中创建 *uImage* 的方法

2.2.1 获取 *mkimage* 工具

2.6 内核树的 *Makefile* 提供了创建 *uImage* 的方法，但需要我们提供相应的 *mkimage* 命令。

所以首先拷贝 *u-boot* 中 *tools* 目录下编译后生成的 *mkimage* 到 */usr/bin/* 下，然后便可以在内核根目录下通过 *make uImage*

来创建 *uImage* 文件。该文件生成在 *arch/arm/boot/* 下。

2.2.2 修改内核的 *Makefile* 文件

```
[arm@localhost linux-2.6.14.1]$ vi arch/arm/boot/Makefile
```

```
#MKIMAGE 变量记录 mkimage 命令的路径 mkuboot.sh 脚本文件可以在 scripts 目录中找到
MKIMAGE      := $(srctree)/scripts/mkuboot.sh
```

```
#zreladdr-y 与 params_phys-y 可以在 arch/arm/mach-s3c2410/Makefile.boot 当中找到
ZRELADDR     := $(zreladdr-y)
PARAMS_PHYS := $(params_phys-y)
INITRD_PHYS := $(initrd_phys-y)
```

#生成 uImage 的 mkImage 命令行，其中需要关注的就是 -a 与 -e 参数。
#参数 -a: 指明 uImage 加载的 SDRAM 地址，内核默认指定加载地址为 0x30008000。
u-boot 引导时，bootm 命令跳到与上相同位置执行，检查完镜像头后，它会跳到内核真正的入口点开始执行。
#参数 -e: 指明 uImage 中刨去镜像头后真正的内核入口地址。
镜像头为 0x40 长，故此处指定为 0x30008040。
u-boot 引导时，go 命令可以直接指定此位置。go 命令不检查镜像头。
quiet_cmd_uimage = UIMAGE \$@
cmd_uimage = \$(CONFIG_SHELL) \$(MKIMAGE) -A arm -O linux -T kernel \
-C none -a \$(ZRELADDR) -e 0x30008040 \
-n 'Linux-\$(KERNELRELEASE)' -d \$< \$@

3 追加实验记录

以同样方式移植其他 2.6 主线内核，出现问题如下：

3.1 移植 linux-2.6.15.7

编译通过，启动时显示：

```
VFS: Cannot open root device "mtdblock2" or unknown-block(31,2)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)
```

3.2 移植 linux-2.6.16.21

编译通过，启动时显示：

```
VFS: Cannot open root device "mtdblock2" or unknown-block(31,2)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)
```

3.3 移植 linux-2.6.17

编译失败

4 参考资料

1. Porting kernel 2.6.11.7 to S3C2410

<http://superlp.blogchina.com/1391393.html>

非常感谢此篇文档的作者

2. devfs 介绍

<http://www-128.ibm.com/developerworks/cn/linux/filesystem/l-fs4/index.html#resources>

3. <<BUILDING EMBEDDED LINUX SYSTEMS>>

中文名：<<构建嵌入式 Linux 系统>>

第六部分 应用程序的移植

1 构造目标板的根目录及文件系统

1.1 建立一个目标板的空根目录

我们将在这里构建构建根文件系统，创建基础目录结构. 存放交叉编译后生成的目标应用程序（BUSYBOX，TINYLOGIN），存放库文件等。

```
[arm@localhost rootfs]# mkdir my_rootfs
[arm@localhost rootfs]# pwd
/home/arm/dev_home/rootfs/my_rootfs
[arm@localhost rootfs]# cd my_rootfs
[arm@localhost my_rootfs]#
```

1.2 在 my_rootfs 中建立 Linux 目录树

```
[arm@localhost my_rootfs]#mkdir bin dev etc home lib mnt proc sbin sys tmp root usr
[arm@localhost my_rootfs]#mkdir mnt/etc
[arm@localhost my_rootfs]#mkdir usr/bin usr/lib usr/sbin
[arm@localhost my_rootfs]#touch linuxrc
[arm@localhost my_rootfs]#tree
|-- bin
|-- dev
|-- etc
|-- home
|-- lib
|-- linuxrc /* 此文件为启动脚本，是一 shell 脚本文件。本文后面有专门介绍 */
|-- mnt
| `-- etc
|-- proc
|-- sbin
|-- sys
|-- tmp
|-- root
`-- usr
    |-- bin
    |-- lib
    `-- sbin
```

权限参照你的 linux 工作站即可，基础目录介绍参见本文参考资料(末尾)。

需要说明的一点就是 etc 目录存放配置文件，这个目录通常是需要修改的，所以在 linuxrc 脚本当中将 etc 目录挂载为 ramfs 文件系统，然后将 mnt/etc 目录中的所有配置文件拷贝到 etc 目录当中，这在下一节的 linuxrc 脚本文件当中会有体现。

1.3 创建 linuxrc 文件

1. 创建 linuxrc,加入如下内容:

```
[arm@localhost my_rootfs]#vi linuxrc
#!/bin/sh
```

#挂载/etc 为 ramfs, 并从/mnt/etc 下拷贝文件到/etc 目录当中

```
echo "mount /etc as ramfs"
```

```
/bin/mount -n -t ramfs ramfs /etc
```

```
/bin/cp -a /mnt/etc/* /etc
```

```
echo "re-create the /etc/mtab entries"
```

```
# re-create the /etc/mtab entries
```

```
/bin/mount -f -t cramfs -o remount,ro /dev/mtdblock/2 /
```

```
#mount some file system
```

```
echo "-----mount /dev/shm as tmpfs"
```

```
/bin/mount -n -t tmpfs tmpfs /dev/shm
```

```
#挂载/proc 为 proc 文件系统
```

```
echo "-----mount /proc as proc"
```

```
/bin/mount -n -t proc none /proc
```

```
#挂载/sys 为 sysfs 文件系统
```

```
echo "-----mount /sys as sysfs"
```

```
/bin/mount -n -t sysfs none /sys
```

```
exec /sbin/init
```

2. 修改权限

```
[arm@localhost my_rootfs]#chmod 775 linuxrc
```

```
[arm@localhost my_rootfs]#ls linuxrc -al
```

```
-rwxrwxr-x 1 root root 533 Jun 4 11:19 linuxrc
```

当编译内核时, 指定命令行参数如下

Boot options ---> Default kernel command string: 我的命令行参数如下

```
noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0,115200
```

其中的 init 指明 kernel 执行后要加载的第一个应用程序, 缺省为/sbin/init, 此处指定为/linuxrc

2 移植 *Busybox*

2.1 下载 busybox

从 <http://www.busybox.net/downloads/busybox-1.1.3.tar.gz> 下载 busybox-1.1.3 到/tmp 目录当中, 并解压.

2.2 进入解压后的目录, 配置 Busybox

```
[arm@localhost busybox-1.1.3]$ make menuconfig
```

Busybox Settings --->

General Configuration --->

[*] Support for devfs

Build Options --->

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

/* 将 busybox 编译为静态连接, 少了启动时找动态库的麻烦 */

[*] Do you want to build BusyBox with a Cross Compiler?

(/usr/local/arm/3.3.2/bin/arm-linux-) Cross Compiler prefix

/* 指定交叉编译工具路径 */

Init Utilities --->

[*] init

[*] Support reading an inittab file

/* 支持 init 读取/etc/inittab 配置文件，一定要选上 */

Shells --->

Choose your default shell (ash) --->

/* (X) ash 选中 ash，这样生成的时候才会生成 bin/sh 文件

* 看看我们前头的 linuxrc 脚本的头一句：

* #!/bin/sh 是由 bin/sh 来解释执行的

*/

[*] ash

Coreutils --->

[*] cp

[*] cat

[*] ls

[*] mkdir

[*] echo (basic SuSv3 version taking no options)

[*] env

[*] mv

[*] pwd

[*] rm

[*] touch

Editors ---> [*] vi

Linux System Utilities --->

[*] mount

[*] umount

[*] Support loopback mounts

[*] Support for the old /etc/mtab file

Networking Utilities --->

[*] inetd

/*

* 支持 inetd 超级服务器

* inetd 的配置文件为/etc/inetd.conf 文件，

* "在该部分的 4: 相关配置文件的创建"一节会有说明

*/

2.3 编译并安装 Busybox

```
[arm@localhost busybox-1.1.3]$ make TARGET_ARCH=arm CROSS=arm-linux- \
PREFIX=/home/arm/dev_home/rootfs/my_rootfs/ all install
```

PREFIX 指明安装路径：就是我们根文件系统所在路径。

*这里需要注意一点的是，只要 install busybox，我们根文件系统下先前建好的 linuxrc 就会被覆盖为一同名二进制文件。

所以要事先备份我们自己的 linuxrc，在安装完 busybox 后，将 linuxrc 复制回去就好。

3 移植 TinyLogin

3.1 下载

从 <http://tinylogin.busybox.net/downloads/tinylogin-1.4.tar.bz2> 下载 tinylogin-1.4 到/tmp 目录当中，并解压。

3.2 修改 tinyLogin 的 Makefile

```
[arm@localhost tinylogin-1.4]$ vi Makefile
```

修改记录如下：

指明静态编译，不连接动态库

```
DOSTATIC = true
```

指明 tinyLogin 使用自己的算法来处理用户密码

```
USE_SYSTEM_PWD_GRP = false
```

```
USE_SYSTEM_SHADOW = false
```

3.3 编译并安装

```
[root@localhost tinylogin-1.4]# make CROSS=arm-linux- PREFIX=/home/arm/dev_home/rootfs/my_rootfs all install
```

PREFIX 指明根文件路径

4 相关配置文件的创建

进入 mnt/etc 中， 这里是我们存放配置文件的路径

```
[arm@localhost my_rootfs]$ cd mnt/etc
```

4.1 创建帐号及密码文件

```
[arm@localhost etc]$ cp /etc/passwd .
```

```
[arm@localhost etc]$ cp /etc/shadow .
```

```
[arm@localhost etc]$ cp /etc/group .
```

这 3 个文件是从你工作站当中拷贝过来的，删除其中绝大部分不需要的用户，经过删减后的上述 3 个文件如下。

那么现在 root 的登陆密码

和你工作站上的登陆口令一致了，这可能透露你工作站的信息

```
[arm@localhost etc]$ cat passwd
```

```
root:x:0:0:root:/root:/bin/sh    /* 改为/bin/sh */
```

```
bin:x:1:1:bin:/bin:/sbin/nologin
```

```
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

```
[arm@localhost etc]$ cat shadow
```

```
root:$1$2LG20u89$UCEEUzBhEIYpKMNZQPU.e1:13303:0:99999:7:::
```

```
bin:!:13283:0:99999:7:::
```

```
daemon:!:13283:0:99999:7:::
```

```
[arm@localhost etc]$ cat group
```

```
root:x:0:root
```

```
bin:x:1:root,bin,daemon
```

```
daemon:x:2:root,bin,daemon
```

4.2 创建 profile 文件

```
[arm@localhost etc]$ vi profile
# Set search library path
# 这条语句设置动态库的搜索路径，极其重要！！！
echo "Set search library path int /etc/profile"
export LD_LIBRARY_PATH=/lib:/usr/lib

# Set user path
echo "Set user path in /etc/profile"
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

4.4 创建 fstab 文件

```
[arm@localhost etc]$ vi fstab
none      /proc      proc  defaults    0 0
none      /dev/pts   devpts mode=0622    0 0
tmpfs     /dev/shm   tmpfs  defaults    0 0
```

4.5 创建 inetd.conf 配置文件

此处只是一个经过修改的示例配置文件，用于代理监听 telnetd 的 23 端口。

读者可以根据自己需求进行修改。该配置文件可以从 busybox 的 examples 目录中获得

```
[arm@localhost etc]$ vi inetd.conf
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#ftp          stream      tcp        nowait  root    /usr/sbin/ftpd
telnet        stream      tcp        nowait  root    /usr/sbin/telnetd
```

5 移植 inetd

5.1 inetd 的选择及获取

Busybox 1.1.3 提供了 inetd 支持。如果读者使用的是较低版本的不提供 inetd 的 Busybox，那么可以考虑使用 netkit 套件来提供网络服务。强烈建议使用高版本的 Busybox。此节后半部分介绍如果编译布署 netkit 当中的 inetd。

5.1.1 获取 inetd

Netkit 套件可以从 <ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/> 下载。

其中 netkit-base-0.17 中包括 inetd 程序。下载 netkit-base-0.17 到/tmp 目录并解压。

5.2 编译 inetd

5.2.1 修改 configure 文件

开始配置 netkit-base 之前需要先修改 configure 脚本以免它在主机上执行测试程序。

```
[arm@localhost netkit-base-0.17]# vi configure
```

将每一行出现的 `./__confest || exit 1;`
修改成:
`# ./__confest || exit 1;`

5.2.2 编译

```
[arm@localhost netkit-base-0.17]$ CC=arm-linux-gcc ./configure  
[arm@localhost netkit-base-0.17]$ make
```

5.3 配置 inetd

5.3.1 拷贝 *inetd* 到根文件系统的 *usr/sbin* 目录中

```
[arm@localhost netkit-base-0.17]$ cp inetd/inetd /home/arm/dev_home/rootfs/my_rootfs/usr/sbin/
```

拷贝 *inetd* 的配置文件 *inetd.conf* 到根文件系统的 */mnt/etc* 目录中

```
[arm@localhost netkit-base-0.17]$ cp etc.sample/inetd.conf /home/arm/dev_home/rootfs/my_rootfs/mnt/etc
```

5.3.2 根据需要，修改 *inetd.conf* 配置文件

例如：支持 *telnetd* 的 *inetd.conf* 配置文件如下

```
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>  
telnet          stream      tcp      nowait  root    /usr/sbin/telnetd
```

5.3.3 拷贝配置文件

etc.sample 目录下有许多网络相关配置文件，其中有一些需要拷贝到根文件系统的 *etc* 目录当中，记录如下：

```
[arm@localhost netkit-base-0.17]$ cd etc.sample/  
[arm@localhost etc.sample]$ cp host.conf /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/  
[arm@localhost etc.sample]$ cp hosts /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/  
[arm@localhost etc.sample]$ cp networks /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/  
[arm@localhost etc.sample]$ cp protocols /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/  
[arm@localhost etc.sample]$ cp resolv.conf /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/  
[arm@localhost etc.sample]$ cp services /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/
```

以上重要配置文件说明如下：

host.conf: 在系统中同时存在着 DNS 域名解析和 */etc/hosts* 的主机表机制时,由文件 */etc/host.conf* 来说明了解析器的查询顺序

hosts: 记录主机名到 IP 地址的映射

protocols: 记录常用网络协议及端口别名关系，网络应用程序依赖于此文件

resolv.conf: 指定 DNS 服务器

services: 记录知名网络服务及端口，网络编程依赖于此文件

6 移植 *thttpd* Web 服务器

6.1 下载

从 <http://www.acme.com/software/thttpd/> 下载 *thttpd* 到/tmp 目录当中，并解压。

6.2 编译 *thttpd*

```
[arm@localhost thttpd-2.25b]$ CC=arm-linux-gcc ./configure --host=arm-linux
```

```
[arm@localhost thttpd-2.25b]$ vi Makefile
```

指定静态链接二进制文件

```
LDFLAGS = -static
```

```
[arm@localhost thttpd-2.25b]$ make LDFLAGS="-static"
```

6.3 配置

6.3.1 拷贝 *thttpd* 二进制可执行文件到根文件系统/usr/sbin/目录中

```
[arm@localhost thttpd-2.25b]$ cp thttpd /home/arm/dev_home/rootfs/my_rootfs/usr/sbin/
```

6.3.2 修改 *thttpd* 配置文件

```
[arm@localhost thttpd-2.25b]$ vi contrib/redhat-rpm/thttpd.conf
```

```
# This section overrides defaults
```

```
dir=/etc/thttpd/html          #指明 WebServer 存放网页的根目录路径
```

```
chroot
```

```
user=root                    #以 root 身份运行 thttpd
```

```
logfile=/etc/thttpd/log/thttpd.log #日志文件路径
```

```
pidfile=/etc/thttpd/run/thttpd.pid #pid 文件路径
```

拷贝 *thttpd.conf* 配置文件到根文件系统的 *mnt/etc/* 目录，

系统加载后，*linuxrc* 脚本会自动将 *mnt/etc/* 下的所有文件拷贝到 *etc* 目录中。

```
[arm@localhost thttpd-2.25b]$ cp contrib/redhat-rpm/thttpd.conf /home/arm/dev_home/rootfs/my_rootfs/mnt/etc/
```

6.3.3 转移到根文件系统目录，创建相应的文件

```
[arm@localhost etc]$ cd /home/arm/dev_home/rootfs/my_rootfs
```

```
[arm@localhost my_rootfs]$ cd mnt/etc/
```

创建 *thttpd* 目录

```
[arm@localhost etc]$ mkdir thttpd
```

```
[arm@localhost etc]$ cd thttpd
```

thttpd 目录下的目录结构

```
|-- html
```

```
| `-- index.html Web Server 网页根目录下的默认 HTML 文件
```

```
|-- log
```

```
| `-- thttpd.log 创建一个空文件就可
```

```
`-- run
```

```
    |-- thttpd.pid 创建一个空文件就可
```

html 目录下的 *index.html* 文件内容如下：

```

<html>
<head>
    <title> Welcome to here^^ </title>
</head>
<body>
    <marquee>
        <font color=red>
            Welcome to here^^!!!
        </font>
    </marquee>
</body>
</html>

```

7 建立根目录文件系统包

7.1 建立 CRAMFS 包

7.1.1 下载 *cramfs* 工具

从 <http://prdownloads.sourceforge.net/cramfs/cramfs-1.1.tar.gz> 下载源代码包。

把下载包拷贝到 dev_home/tools 下。

```
[arm@localhost tools]$tar -xzvf cramfs-1.1.tar.gz
```

```
[arm@localhost tools]$cd cramfs-1.1
```

```
[arm@localhost tools]$make
```

```
[arm@localhost tools]$su root
```

```
[root@localhost tools]$cp mkcramfs /usr/bin
```

```
[arm@localhost tools]$exit
```

注意：如果你的系统中已经安装了 mkcramfs 工具，则在 /usr/bin 目录下是一个软 link，请先删除该文件之后，再拷贝该 mkcramfs 到 /usr/bin 下。

7.1.2 制作 *cramfs* 包

```
[arm@localhost tools]$mkcramfs my_rootfs my_rootfs.cramfs
```

7.1.3 写 *cramfs* 包到 *Nand Flash*

```
[arm@localhost tools]$su root
```

```
[root@localhost tools]$cp my_rootfs.cramfs /tftpboot/
```

打开 minicom, 进行 ARM 板的终端模式:

```
CRANE2410 #
```

8 参考资料

1. linux 目录结构介绍

<http://www.uplooking.com/content/view/1487/2/>

2. <<BUILDING EMBEDDED LINUX SYSTEMS>>

中文名: <<构建嵌入式 Linux 系统>>

第七部分 Nand flash 驱动的编写与移植

1 Nand flash 工作原理

S3C2410 板的 Nand Flash 支持由两部分组成: Nand Flash 控制器(集成在 S3C2410 CPU)和 Nand Flash 存储芯片(K9F1208U0B)两大部分组成。当要访问 Nand Flash 中的数据时,必须通过 Nand Flash 控制器发送命令才能完成。所以, Nand Flash 相当于 S3C2410 的一个外设,而不位于它的内存地址区。

1.1 Nand flash 芯片工作原理

Nand flash 芯片型号为 Samsung K9F1208U0B, 数据存储容量为 64MB, 采用块页式存储管理。8 个 I/O 引脚充当数据、地址、命令的复用端口。

1.1.1 芯片内部存储布局及存储操作特点

一片 Nand flash 为一个设备(device), 其数据存储分层为:

1 设备(Device) = 4096 块(Blocks)

1 块(Block) = 32 页/行(Pages/rows) ;页与行是相同的意思,叫法不一样

1 块(Page) = 528 字节(Bytes) = 数据块大小(512Bytes) + OOB 块大小(16Bytes)

在每一页中, 最后 16 个字节(又称 OOB)用于 Nand Flash 命令执行完后设置状态用, 剩余 512 个字节又分为前半部分和后半部分。可以通过 Nand Flash 命令 00h/01h/50h 分别对前半部、后半部、OOB 进行定位通过 Nand Flash 内置的指针指向各自的首地址。

存储操作特点:

1. 擦除操作的最小单位是块。
2. Nand Flash 芯片每一位(bit)只能从 1 变为 0, 而不能从 0 变为 1, 所以在对其进行写入操作之前要一定将相应块擦除(擦除即是将相应块得位全部变为 1)。
3. OOB 部分的第六字节(即 517 字节)标志是否是坏块, 如果不是坏块该值为 FF, 否则为坏块。
4. 除 OOB 第六字节外, 通常至少把 OOB 的前 3 个字节存放 Nand Flash 硬件 ECC 码(关于硬件 ECC 码请参看 Nandflash 控制器一节)。

1.1.2 重要芯片引脚功能

I/O0-I/O7: 复用引脚。可以通过它向 nand flash 芯片输入数据、地址、nand flash 命令以及输出数据和操作状态信息。

CLE(Command Latch Enable): 命令锁存允许

ALE(Address Latch Enable): 地址锁存允许

-CE: 芯片选择

-RE: 读允许

-WE: 写允许

-WP: 在写或擦除期间, 提供写保护

R/-B: 读/忙输出

1.1.3 寻址方式

Samsung K9F1208U0B Nand Flash 片内寻址采用 26 位地址形式。从第 0 位开始分四次通过 I/O0-I/O7 进行传送, 并进行片内寻址。具体含义如下:

0-7 位: 字节在上半部、下半部及 OOB 内的偏移地址

8 位: 值为 0 代表对一页内前 256 个字节进行寻址

值为 1 代表对一页内后 256 个字节进行寻址

9-13 位: 对页进行寻址

14—25 位：对块进行寻址

当传送地址时,从位 0 开始

1.1.4 Nand flash 主要内设命令详细介绍

Nand Flash 命令执行是通过将命令字送到 Nand Flash 控制器的命令寄存器来执行。

Nand Flash 的命令是分周期执行的，每条命令都有一个或多个执行周期，每个执行周期都有相映代码表示该周期将要执行的动作。

主要命令有：Read 1、Read 2、Read ID、Reset、Page Program、Block Erase、Read Status。

详细介绍如下：

1. Read 1:

功能：表示将要读取 Nand flash 存储空间中一个页的前半部分，并且将内置指针定位到前半部分的第一个字节。

命令代码：00h

2. Read 2:

功能：表示将要读取 Nand flash 存储空间中一个页的后半部分，并且将内置指针定位到后半部分的第一个字节。

命令代码：01h

3. Read ID:

功能：读取 Nand flash 芯片的 ID 号

命令代码：90h

4. Reset:

功能：重启芯片。

命令代码：FFh

5. Page Program:

功能：对页进行编程命令,用于写操作。

命令代码：首先写入 00h(A 区)/01h(B 区)/05h(C 区),表示写入那个区;再写入 80h 开始编程模式(写入模式),接下来写入地址和数据;最后写入 10h 表示编程结束。

6. Block Erase

功能：块擦除命令。

命令代码：首先写入 60h 进入擦写模式，然后输入块地址;接下来写入 D0h,表示擦写结束。

7. Read Status

功能：读取内部状态寄存器值命令。

命令代码：70h

1.2 Nand Flash 控制器工作原理

对 Nand Flash 存储芯片进行操作,必须通过 Nand Flash 控制器的专用寄存器才能完成。所以,不能对 Nand Flash 进行总线操作。而 Nand Flash 的写操作也必须块方式进行。对 Nand Flash 的读操作可以按字节读取。

1.2.1 Nand Flash 控制器特性

1. 支持对 Nand Flash 芯片的读、检验、编程控制
2. 如果支持从 Nand Flash 启动,在每次重启后自动将前 Nand Flash 的前 4KB 数据搬运到 ARM 的内部 RAM 中
3. 支持 ECC 校验

1.2.2 Nand Flash 控制器工作原理

Nand Flash 控制器在其专用寄存器区(SFR)地址空间中映射有属于自己的特殊功能寄存器,就是通过将 Nand Flash 芯片的内设命令写到其特殊功能寄存器中,从而实现对 Nand flash 芯片读、检验和编程控制的。特殊功能寄存器有: NFNCONF、NFCMD、NFADDR、NFDATA、NFSTAT、NFECC。寄存详细说明见下一节。

1.3 Nand flash 控制器中特殊功能寄存器详细介绍

1. 配置寄存器(NFNCONF)

功能：用于对 Nand Flash 控制器的配置状态进行控制。

在地址空间中地址:0x4E000000,其中:

Bit15: Nand Flash 控制器使能位, 置 0 代表禁止 Nand Flash 控制器, 置 1 代表激活 Nand Flash 控制器; 要想访问 Nand Flash 芯片上存储空间, 必须激活 Nand Flash 控制器。在复位后该位自动置 0, 因此在初始化时必须将该位置为 1。

Bit12: 初始化 ECC 位, 置 1 为初始化 ECC; 置 0 为不初始化 ECC。

Bit11: Nand Flash 芯片存储空间使能位, 置 0 代表可以对存储空间进行操作; 置 1 代表禁止对存储空间进行操作。在复位后, 该位自动为 1。

Bit10-8: TACLS 位。根据此设定 CLE&ALE 的周期。TACLS 的值范围在 0-7 之间。

Bit6-4、2-0 分别为: TWRPH0、TWRPH1 位。设定写操作的访问周期。其值在 0-7 之间。

2. 命令寄存器(NFCMD)

功能：用于存放 Nand flash 芯片内设的操作命令。

在地址空间中地址: 0x4E000004, 其中:

Bit0-7: 存放具体 Nand flash 芯片内设的命令值。其余位保留以后用。

3. 地址寄存器(NFADDR)

功能：用于存放用于对 Nand flash 芯片存储单元寻址的地址值。

在地址空间中地址: 0x4E000008, 其中:

Bit0-7: 用于存放地址值。因为本款 Nand flash 芯片只有 I/O0-7 的地址/数据复用引脚且地址是四周期每次 8 位送入的, 所以这里只用到 8 位。其余位保留待用。

4. 数据寄存器(NFDATA)

功能：Nand flash 芯片所有内设命令执行后都会将其值放到该寄存器中。同时, 读出、写入 Nand flash 存储空间的值也是放到该寄存器。

在地址空间中地址: 0x4E00000C, 其中:

Bit0-7: 用于存放需要读出和写入的数据。其余位保留代用。

5. 状态寄存器(NFSTAT)

功能：用于检测 Nand flash 芯片上次对其存储空间的操作是否完成。

在地址空间中地址: 0x4E000010, 其中:

Bit0: 置 0 表示 Nand flash 芯片正忙于上次对存储空间的操作; 置 1 表示 Nand flash 芯片准备好接收新的对存储空间操作的请求。

6. ECC 校验寄存器(NFECC)

功能：ECC 校验寄存器

在地址空间中地址: 0x4E000014, 其中:

Bit0-Bit7: ECC0

Bit8-Bit15: ECC1

Bit16-Bit23: ECC2

1.4 Nand Flash 控制器中的硬件 ECC 介绍

1.4.1 ECC 产生方法

ECC 是用于对存储器之间传送数据正确进行校验的一种算法, 分硬件 ECC 和软件 ECC 算法两种, 在 S3C2410 的 Nand Flash 控制器中实现了由硬件电路 (ECC 生成器) 实现的硬件 ECC。

1.4.2 ECC 生成器工作过程

当写入数据到 Nand flash 存储空间时，ECC 生成器会在写入数据完毕后自动生成 ECC 码，将其放入到 ECC0—ECC2。当读出数据时 Nand Flash 同样会在读数据完毕后，自动生成 ECC 码将其放到 ECC0—ECC2 当中。

1.4.3 ECC 的运用

当写入数据时，可以在每页写完数据后将产生的 ECC 码放入到 OOB 指定的位置(Byte 6)去，这样就完成了 ECC 码的存储。这样当读出该页数据时，将所需数据以及整个 OOB 读出，然后将指定位置的 ECC 码与读出数据后在 ECC0—ECC1 的实际产生的 ECC 码进行对比，如果相等则读出正确，若不相等则读取错误需要进行重读。

2 在 ADS 下 flash 烧写程序

2.1 ADS 下 flash 烧写程序原理及结构

基本原理：在 windows 环境下借助 ADS 仿真器将在 SDRAM 中的一段存储区域中的数据写到 Nand flash 存储空间中。烧写程序在纵向上分三层完成：

第一层：主烧写函数（完成将在 SDRAM 中的一段存储区域中的数据写到 Nand flash 存储空间中）；

第二层：为第一层主烧写函数提供支持的 Nand flash 进行操作的页读、写，块擦除等函数；

第三层：为第二层提供具体 Nand flash 控制器中对特殊功能寄存器进行操作的核心函数，该层也是真正的将数据能够在 SDRAM 和 Nand flash 之间实现传送的函数。

下面对其三层进行分述：

2.2 第三层实现说明

2.2.1 特殊功能寄存器定义

```
#define rNFCONF    (*(volatile unsigned int *)0x4e000000)
#define rNFCMD    (*(volatile unsigned char *)0x4e000004)
#define rNFADDR    (*(volatile unsigned char *)0x4e000008)
#define rNFDATA    (*(volatile unsigned char *)0x4e00000c)
#define rNFSTAT    (*(volatile unsigned int *)0x4e000010)
#define rNFECC     (*(volatile unsigned int *)0x4e000014)
#define rNFECC0    (*(volatile unsigned char *)0x4e000014)
#define rNFECC1    (*(volatile unsigned char *)0x4e000015)
#define rNFECC2    (*(volatile unsigned char *)0x4e000016)
```

2.2.2 操作的函数实现

1. 发送命令

```
#define NF_CMD(cmd)    {rNFCMD=cmd;}
```

2. 写入地址

```
#define NF_ADDR(addr)    {rNFADDR=addr;}
```

3. Nand Flash 芯片选中

```
#define NF_nFCE_L()    {rNFCONF&=~(1<<11);}
```

4. Nand Flash 芯片不选中

```
#define NF_nFCE_H()    {rNFCONF|= (1<<11);}
```

5. 初始化 ECC

```
#define NF_RSTTECC()    {rNFCONF|= (1<<12);}
```

6. 读数据

```
#define NF_RDDATA()      (rNFDATA)
7. 写数据
#define NF_WRDATA(data) {rNFDATA=data;}
8. 获取 Nand Flash 芯片状态
#define NF_WAITRB()      {while(!(rNFSTAT&(1<<0)));}
0/假: 表示 Nand Flash 芯片忙状态
1/真: 表示 Nand Flash 已经准备好
```

2.3 第二层实现说明

2.3.1 Nand Flash 初始化

```
void NF_Init(void)
{
    /* 设置 Nand Flash 配置寄存器, 每一位的取值见 1.3 节 */
    rNFCONF=(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
    /* 复位外部 Nand Flash 芯片 */
    NF_Reset();
}
```

2.3.2 Nand flash 复位

```
static void NF_Reset(void)
{
    int i;

    NF_nFCE_L();    /* 片选 Nand Flash 芯片*/
    NF_CMD(0xFF);   /* 复位命令 */
    for(i=0;i<10;i++); /* 等待 tWB = 100ns. */
    NF_WAITRB();    /* wait 200~500us; */
    NF_nFCE_H();    /* 取消 Nand Flash 选中*/
}
```

2.3.3 获取 Nand flash ID

返回值为 Nand flash 芯片的 ID 号

```
unsigned short NF_CheckId(void)
{
    int i;
    unsigned short id;

    NF_nFCE_L();    /* 片选 Nand Flash 芯片*/
    NF_CMD(0x90);   /* 发送读 ID 命令到 Nand Flash 芯片 */
    NF_ADDR(0x0);   /* 指定地址 0x0, 芯片手册要求 */
    for(i=0;i<10;i++); /* 等待 tWB = 100ns. */
    id=NF_RDDATA()<<8; /* 厂商 ID(K9S1208V:0xec) */
    id|=NF_RDDATA(); /* 设备 ID(K9S1208V:0x76) */
    NF_nFCE_H();    /* 取消 Nand Flash 选中*/
    return id;
}
```

2.3.4 Nand flash 写入

以页为单位写入.

参数说明: block 块号

page 页号

buffer 指向内存中待写入 Nand flash 中的数据起始位置

返回值: 0: 写错误

1: 写成功

```
static int NF_WritePage(unsigned int block, unsigned int page, unsigned char *buffer)
{
    int i;
    unsigned int blockPage = (block<<5)+page;
    unsigned char *bufPt = buffer;

    NF_RSTTECC(); /* 初始化 ECC */
    NF_nFCE_L(); /* 片选 Nand Flash 芯片*/
    NF_CMD(0x0); /* 从 A 区开始写 */
    NF_CMD(0x80); /* 写第一条命令 */
    NF_ADDR(0); /* A0~A7 位(Column Address) */
    NF_ADDR(blockPage&0xff); /* A9-A16, (Page Address) */
    NF_ADDR((blockPage>>8)&0xff); /* A17-A24, (Page Address) */
    NF_ADDR((blockPage>>16)&0xff); /* A25, (Page Address) */

    for(i=0;i<512;i++)
    {
        NF_WRDATA(*bufPt++); /* 写一个页 512 字节到 Nand Flash 芯片 */
    }

    /*
    * OOB 一共 16 Bytes, 每一个字节存放什么由程序员自己定义, 通常,
    * 我们在 Byte0-Byte2 存 ECC 检验码. Byte6 存放坏块标志.
    */
    seBuf[0]=rNFECC0; /* 读取 ECC 检验码 0 */
    seBuf[1]=rNFECC1; /* 读取 ECC 检验码 1 */
    seBuf[2]=rNFECC2; /* 读取 ECC 检验码 2 */
    seBuf[5]=0xff; /* 非坏块标志 */

    for(i=0;i<16;i++)
    {
        NF_WRDATA(seBuf[i]); /* 写该页的 OOB 数据块 */
    }

    NF_CMD(0x10); /* 结束写命令 */

    /* 等待 Nand Flash 处于准备状态 */
    for(i=0;i<10;i++);
    NF_WAITRB();

    /* 发送读状态命令给 Nand Flash */
    NF_CMD(0x70);
    for(i=0;i<3;i++);
```

```

if (NF_RDDATA() & 0x1)
{ /*如果写有错,则标示为坏块 */
    NF_nFCE_H(); /* 取消 Nand Flash 选中*/
    NF_MarkBadBlock(block);
    return 0;
} else { /* 正常退出 */
    NF_nFCE_H(); /* 取消 Nand Flash 选中*/
    return 1;
}
}

```

2.3.5 Nand flash 读取

参数说明: block: 块号

page: 页号

buffer: 指向将要读取到内存中的起始位置

返回值: 1: 读成功

0: 读失败

```

static int NF_ReadPage(unsigned int block, unsigned int page, unsigned char *buffer)
{
    int i;
    unsigned int blockPage;
    unsigned char ecc0, ecc1, ecc2;
    unsigned char *bufPt=buffer;
    unsigned char se[16];

    page=page&0x1f;
    blockPage=(block<<5)+page;
    NF_RSTCECC(); /* 初始化 ECC */
    NF_nFCE_L(); /* 片选 Nand Flash 芯片*/
    NF_CMD(0x00); /* 从 A 区开始读 */
    NF_ADDR(0); /* A0~A7 位(Column Address) */
    NF_ADDR(blockPage&0xff); /* A9-A16, (Page Address) */
    NF_ADDR((blockPage>>8)&0xff); /* A17-A24, (Page Address) */
    NF_ADDR((blockPage>>16)&0xff); /* A25, (Page Address) */

    /* 等待 Nand Flash 处于再准备状态 */
    for(i=0;i<10;i++);
    NF_WAITRB(); /*等待 tR(max 12us) */
    /* 读整个页, 512 字节 */
    for(i=0;i<512;i++)
    {
        *bufPt++=NF_RDDATA();
    }

    /* 读取 ECC 码 */
    ecc0=rNFECC0;
    ecc1=rNFECC1;
    ecc2=rNFECC2;
}

```

```

/* 读取该页的 OOB 块 */
for(i=0;i<16;i++)
{
    se[i]=NF_RDDATA();
}

NF_nFCE_H(); /* 取消 Nand Flash 选中*/

/* 校验 ECC 码, 并返回 */
if(ecc0==se[0] && ecc1==se[1] && ecc2==se[2])
    return 1;
else
    return 0;
}

```

2.3.6 Nand flash 标记坏块

如果是坏块, 通过写 OOB 块的 Byte6 把该块标记为坏块。

参数说明: block 块号

返回值: 1: ok, 成功完成标记。

0: 表示写 OOB 块正确.

```

static int NF_MarkBadBlock(unsigned int block)
{
    int i;
    unsigned int blockPage=(block<<5);

    seBuf[0]=0xff;
    seBuf[1]=0xff;
    seBuf[2]=0xff;
    seBuf[5]=0x44; /* 设置坏块标记 */

    NF_nFCE_L(); /* 片选 Nand Flash 芯片*/
    NF_CMD(0x50); /* 从 C 区开始写 */
    NF_CMD(0x80); /* 发送编程命令, 让 Nand Flash 处理写状态 */
    NF_ADDR(0x0); /* A0~A7 位(Column Address) */
    NF_ADDR(blockPage&0xff); /* A9-A16, (Page Address) */
    NF_ADDR((blockPage>>8)&0xff); /* A17-A24, (Page Address) */
    NF_ADDR((blockPage>>16)&0xff); /* A25, (Page Address) */

    /* 写 OOB 数据块 */
    for(i=0;i<16;i++)
    {
        NF_WRDATA(seBuf[i]);
    }

    NF_CMD(0x10); /* 结束写命令 */

    /* 等待 NandFlash 准备好 */
    for(i=0;i<10;i++); /* tWB = 100ns. */
    NF_WAITRB();
}

```

```

/*读 NandFlash 的写状态 */
NF_CMD(0x70);
for(i=0;i<3;i++); /* twhr=60ns */
if (NF_RDDATA()&0x1)
{
    NF_nFCE_H(); /* 取消 Nand Flash 选中*/
    return 0;
} else {
    NF_nFCE_H(); /* 取消 Nand Flash 选中*/
}
return 1;
}

```

2.3.7 Nand Flash 检查坏块

检查指定块是否是坏块。

参数说明: block: 块号

返回值: 1: 指定块是坏块

0: 指定块不是坏块。

static int NF_IsBadBlock(U32 block)

```

{
    int i;
    unsigned int blockPage;
    U8 data;

    blockPage=(block<<5);
    NF_nFCE_L(); /* 片选 Nand Flash 芯片*/
    NF_CMD(0x50); /* Read OOB 数据块 */
    NF_ADDR(517&0xf); /* A0~A7 位(Column Address) */
    NF_ADDR(blockPage&0xff); /* A9-A16, (Page Address) */
    NF_ADDR((blockPage>>8)&0xff); /* A17-A24, (Page Address) */
    NF_ADDR((blockPage>>16)&0xff); /* A25, (Page Address) */

    /* 等待 NandFlash 准备好 */
    for(i=0;i<10;i++); /* wait tWB(100ns) */
    NF_WAITRB();

    /* 读取读出值 */
    data=NF_RDDATA();
    NF_nFCE_H(); /* 取消 Nand Flash 选中*/
    /* 如果 data 不为 0xff 时, 表示该块是坏块 */
    if(data != 0xff)
        return 1;
    else
        return 0;
}

```

2.3.8 擦除指定块中数据

参数说明: block 块号

返回值: 0: 擦除错误。(若是坏块直接返回 0; 若擦除出现错误则标记为坏块然后返回 0)

1: 成功擦除。

```
static int NF_EraseBlock(unsigned int block)
{
    unsigned int blockPage=(block<<5);
    int i;

    /* 如果该块是坏块, 则返回 */
    if(NF_IsBadBlock(block))
        return 0;

    NF_nFCE_L(); /* 片选 Nand Flash 芯片*/
    NF_CMD(0x60); /* 设置擦写模式 */
    NF_ADDR(blockPage&0xff); /* A9-A16, (Page Address) , 是基于块擦*/
    NF_ADDR((blockPage>>8)&0xff); /* A17-A24, (Page Address) */
    NF_ADDR((blockPage>>16)&0xff); /* A25, (Page Address) */
    NF_CMD(0xd0); /* 发送擦写命令, 开始擦写 */

    /* 等待 NandFlash 准备好 */
    for(i=0;i<10;i++); /* tWB(100ns) */
    NF_WAITRB();

    /* 读取操作状态 */
    NF_CMD(0x70);
    if (NF_RDDATA()&0x1)
    {
        NF_nFCE_H(); /* 取消 Nand Flash 选中*/
        NF_MarkBadBlock(block); /* 标记为坏块 */
        return 0;
    } else {
        NF_nFCE_H(); /* 取消 Nand Flash 选中*/
        return 1;
    }
}
```

2.4 第一层的实现

2.4.1 NandFlash 烧写主函数说明

参数说明: block 块号

srcAddress SDRAM 中数据起始地址

fileSize 要烧写的数据长度

返回值: 无

```
void K9S1208_Program(unsigned int block, unsigned int srcAddress, unsigned int fileSize)
{
    int i;
    int programError=0;
    U32 blockIndex;
    U8 *srcPt, *saveSrcPt;

    srcPt=(U8 *)srcAddress; /* 文件起始地址 */
    blockIndex = block; /* 块号 */
```

```

while(1)
{
    saveSrcPt=srcPt;
    /* 如果当前块是坏块, 跳过当前块 */
    if(NF_IsBadBlock(blockIndex))
    {
        blockIndex++; /* 到下一个块 */
        continue;
    }
    /* 在写之前, 必须先擦除, 如果擦除不成功, 跳过当前块 */
    if(!NF_EraseBlock(blockIndex))
    {
        blockIndex++; /* 到下一个块 */
        continue;
    }

    /* 写一个块, 一块有 32 页 */
    for(i=0;i<32;i++)
    {
        /* 写入一个页, 如果出错, 停止写当前块 */
        if(!NF_WritePage(blockIndex,i,srcPt))
        {
            programError=1;
            break;
        }
        /* 如果操作正常, 文件的写位置加上 1 页偏移,到下一页的起始位置 */
        srcPt+=512;
        /* 如果写地址没有超过文件长度, 继续; 超出则终止写 */
        if((U32)srcPt>=(srcAddress+fileSize))
            break;
    }

    /* 如果写一个块时, 其中某一页写失败, 则把写地址恢复写该块之前, 并跳过当前块 */
    if(programError==1)
    {
        blockIndex++;
        srcPt=saveSrcPt;
        programError=0;
        continue;
    }
    /* 如果写地址没有超过文件长度, 继续; 超出则终止写 */
    if((U32)srcPt >= (srcAddress+fileSize))
        break;

    /* 如果正常写成功, 继续写下一个块 */
    blockIndex++;
}
}

```

3 在 U-BOOT 对 Nand Flash 的支持

3.1 U-BOOT 对从 Nand Flash 启动的支持

3.1.1 从 Nand Flash 启动 U-BOOT 的基本原理

1. 前 4K 的问题

如果 S3C2410 被配置成从 Nand Flash 启动(配置由硬件工程师在电路板设置), S3C2410 的 Nand Flash 控制器有一个特殊的功能, 在 S3C2410 上电后, Nand Flash 控制器会自动的把 Nand Flash 上的前 4K 数据搬移到 4K 内部 RAM 中, 并把 0x00000000 设置内部 RAM 的起始地址, CPU 从内部 RAM 的 0x00000000 位置开始启动。这个过程不需要程序干涉。

程序员需要完成的工作, 是把最核心的启动程序放在 Nand Flash 的前 4K 中。

2. 启动程序的安排

由于 Nand Flash 控制器从 Nand Flash 中搬移到内部 RAM 的代码是有限的, 所以, 在启动代码的前 4K 里, 我们必须完成 S3C2410 的核心配置以及把启动代码(UBOOT)剩余部分搬到 RAM 中运行。以 UBOOT 为例, 前 4K 完成的主要工作, 见第四部分的 2.2 节。

3.1.2 支持 Nand Flash 启动代码说明

首先在 include/configs/crane2410.h 中加入 CONFIG_S3C2410_NAND_BOOT, 如下:

```
#define CONFIG_S3C2410_NAND_BOOT 1
```

支持从 Nand Flash 中启动.

1. 执行 Nand Flash 初始化

下面代码在 cpu/arm920t/start.S 中

```
#ifdef CONFIG_S3C2410_NAND_BOOT
```

```
copy_myself:
```

```
    mov r10, lr
```

```
    ldr sp, DW_STACK_START @安装栈的起始地址
```

```
    mov fp, #0             @初始化帧指针寄存器
```

```
    bl nand_reset          @跳到复位 C 函数去执行
```

```
...
```

```
DW_STACK_START:
```

```
    .word  STACK_BASE+STACK_SIZE-4
```

2. nand_reset C 代码

下面代码被加在/board/crane2410/crane2410.c 中

```
void nand_reset(void)
```

```
{
```

```
    int i;
```

```
    /* 设置 Nand Flash 控制器 */
```

```
    rNFCNF=(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
```

```
    /* 给 Nand Flash 芯片发送复位命令 */
```

```
    NF_nFCE_L();
```

```
    NF_CMD(0xFF);
```

```
    for(i=0; i<10; i++);
```

```
    NF_WAITRB();
```

```

        NF_nFCE_H();
    }

```

3. 从 Nand Flash 中把 UBOOT 拷贝到 RAM

@read U-BOOT from Nand Flash to RAM

```

    ldr r0, =UBOOT_RAM_BASE @ 设置第 1 个参数: UBOOT 在 RAM 中的起始地址
    mov r1, #0x0             @ 设置第 2 个参数:Nand Flash 的起始地址
    mov r2, #0x20000         @ 设置第 3 个参数: UBOOT 的长度(128KB)
    bl nand_read_whole       @ 调用 nand_read_whole(), 该函数在 board/crane2410/crane2410.c 中
    tst r0, #0x0             @ 如果函数的返回值为 0,表示执行成功.
    beq ok_nand_read         @ 执行内存比较

```

4. 从 Nand Flash 中把数据读入到 RAM 中

int nand_read_whole(unsigned char *buf, unsigned long start_addr, int size)

```

{
    int i, j;

    /* 如果起始地址和长度不是 512 字节(1 页)的倍数, 则返回错误代码 */
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
        return -1;
    }

    /* 激活 Nand Flash */
    NF_nFCE_L();
    for(i=0; i<10; i++);

    i = start_addr;
    while(i < start_addr + size) {
        /* 读 A 区 */
        rNFCMD = 0;

        /* 写入读取地址 */
        rNFADDR = i & 0xff;
        rNFADDR = (i >> 9) & 0xff;
        rNFADDR = (i >> 17) & 0xff;
        rNFADDR = (i >> 25) & 0xff;

        NF_WAITRB();
        /* 读出一页(512 字节) */
        for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
            *buf = (rNFDATA & 0xff);
            buf++;
        }
    }

    /* 停止驱动 Nand Flash */
    NF_nFCE_H();

    return 0;
}

```

5. 检查搬移后的数据

把 RAM 中的前 4K 与内部中前 4K 进行比较, 如果完全相同, 则表示搬移成功.

```
ok_nand_read:
    mov r0, #0x00000000        @内部 RAM 的起始地址
    ldr r1, =UBOOT_RAM_BASE    @UBOOT 在 RAM 中的起始地址
    mov r2, #0x400              @比较 1024 次, 每次 4 字节, 4 bytes * 1024 = 4K-bytes
go_next:    @ 比较 1024 次, 每次 4 个字节
    ldr r3, [r0], #4
    ldr r4, [r1], #4
    teq r3, r4
    bne notmatch
    subs r2, r2, #4
    beq done_nand_read
    bne go_next

notmatch:
    1:b    1b
done_nand_read:
    mov pc, r10
```

3.2 U-BOOT 对 Nand Flash 命令的支持

在 U-BOOT 下对 Nand Flash 的支持主要是在命令行下实现对 nand flash 的操作。对 nand flash 实现的命令为: nand info、nand device、nand read、nand write、nand erase、nand bad。

用到的主要数据结构有: struct nand_flash_dev、struct nand_chip。前者包括主要的芯片型号、存储容量、设备 ID、I/O 总线宽度等信息; 后者是具体对 nand flash 进行操作时用到的信息。

3.2.1 主要数据结构介绍

1. struct nand_flash_dev 数据结构

该数据结构在 include/linux/mtd/nand.h 中定义, 在 include/linux/mtd/nand_ids.h 中赋初值。

```
struct nand_flash_dev {
    char *name;                /* 芯片名称 */
    int manufacture_id;        /* 厂商 ID */
    int model_id;              /* 模式 ID */
    int chipshift;             /* Nand Flash 地址位数 */
    char page256;              /* 表明是否时 256 字节一页。1: 是; 0: 否。*/
    char pageadrln;            /* 完成一次地址传送需要往 NFADDR 中传送几次。*/
    unsigned long erasesize;    /* 一次块擦除可以擦除多少字节 */
    int bus16;                 /* 地址线是否是 16 位, 1: 是; 0: 否 */
};
```

2. struct nand_chip 数据结构

该数据结构在 include/linux/mtd/nand.h 中定义. 该结构体定义出一个 Nand Flash 设备数组:

```
struct nand_chip nand_dev_desc[CFG_MAX_NAND_DEVICE];
```

该数组在 nand_probe()中对其进行初始化.

```
struct nand_chip {
    int    page_shift;        /* Page 地址位数 */
    u_char *data_buf;         /* 本次读出的一页数据 */
    u_char *data_cache;       /* 读出的一页数据 */
    int    cache_page;        /* 上次操作的页号 */
};
```

```

u_char ecc_code_buf[6]; /* ECC 校验码 */
u_char reserved[2];
char ChipID; /* 芯片 ID 号 */
struct Nand *chips; /* Nand Flash 芯片列表, 表示支持几个芯片为一个设备*/
int chipshift;
char* chips_name; /* Nand Flash 芯片名称 */
unsigned long erasesize; /* 块擦写的大小 */
unsigned long mfr; /* 厂商 ID */
unsigned long id; /* 模式 ID */
char* name; /* 设备名称 */
int numchips; /* 有几块 Nand Flash 芯片 */
char page256; /* 一页是 256 字节, 还是 512 字节 */
char pageadrlen; /* 页地址的长度 */
unsigned long IO_ADDR; /* 用于对 nand flash 进行寻址的地址值存放处 */
unsigned long totlen; /* Nand Flash 总共大小 */
uint oobblock; /* 一页的大小。本款 nand flash 为 512 */
uint oobsize; /* spare array 大小。本款 nand flash 为 16 */
uint eccsize; /* ECC 大小 */
int bus16; /* 地址线是否是 16 位, 1: 是; 0: 否 */
};

```

3.2.2 支持的命令函数说明

1. nand info/nand device

功能: 显示当前 nand flash 芯片信息。

函数调用关系如下(按先后顺序):

```
static void nand_print(struct nand_chip *nand);
```

2. nand erase

功能: 擦除指定块上的数据。

函数调用关系如下(按先后顺序):

```
int nand_erase(struct nand_chip* nand, size_t ofs, size_t len, int clean);
```

3. nand bad

功能: 显示坏块。

函数调用关系如下(按先后顺序):

```
static void nand_print_bad(struct nand_chip* nand);
```

```
int check_block (struct nand_chip *nand, unsigned long pos);
```

4. nand read

功能: 读取 nand flash 信息到 SDRAM。

函数调用关系如下(按先后顺序):

```
int nand_rw (struct nand_chip* nand, int cmd, size_t start, size_t len, size_t * retlen, u_char * buf);
```

```
static int nand_read_ecc(struct nand_chip *nand, size_t start, size_t len,
                        size_t * retlen, u_char *buf, u_char *ecc_code);
```

```
static void Nand_ReadBuf (struct nand_chip *nand, u_char * data_buf, int cntr);
```

```
READ_NAND(adr);
```

5. nand write

功能: 从 SDRAM 写数据到 nand flash 中。

函数调用关系如下(按先后顺序):

```
int nand_rw (struct nand_chip* nand, int cmd, size_t start, size_t len, size_t * retlen, u_char * buf);
static int nand_write_ecc (struct nand_chip* nand, size_t to, size_t len,
                          size_t * retlen, const u_char * buf, u_char * ecc_code);
static int nand_write_page (struct nand_chip *nand, int page, int col, int last, u_char * ecc_code);
WRITE_NAND(d , adr);
```

3.2.3 U-BOOT 支持 Nand Flash 命令移植说明

1. 设置配置选项

在 CONFIG_COMMANDS 中, 打开 CFG_CMD_NAND 选项.

```
#define CONFIG_COMMANDS \
    (CONFIG_CMD_DFL      | \
     CFG_CMD_CACHE      | \
     CFG_CMD_NAND        | \
     /*CFG_CMD_EEPROM   | \
     /*CFG_CMD_I2C       | \
     /*CFG_CMD_USB       | \
     CFG_CMD_PING        | \
     CFG_CMD_REGINFO     | \
     CFG_CMD_DATE        | \
     CFG_CMD_ELF)

#if (CONFIG_COMMANDS & CFG_CMD_NAND)
#define CFG_NAND_BASE      0x4E000000 /* Nand Flash 控制器在 SFR 区中起始寄存器地址 */
#define CFG_MAX_NAND_DEVICE 1 /* 支持的最在 Nand Flash 数据 */
#define SECTORSIZE         512 /* 1 页的大小 */
#define NAND_SECTOR_SIZE   SECTORSIZE
#define NAND_BLOCK_MASK    (NAND_SECTOR_SIZE - 1) /* 页掩码 */

#define ADDR_COLUMN        1 /* 一个字节的 Column 地址 */
#define ADDR_PAGE          3 /* 3 字节的页块地址, A9-A25*/
#define ADDR_COLUMN_PAGE   4 /* 总共 4 字节的页块地址 */

#define NAND_ChipID_UNKNOWN 0x00 /* 未知芯片的 ID 号 */
#define NAND_MAX_FLOORS     1
#define NAND_MAX_CHIPS      1

/* Nand Flash 命令层底层接口函数 */
#define WRITE_NAND_COMMAND(d, adr) do {rNFCMD = d;} while(0)
#define WRITE_NAND_ADDRESS(d, adr) do {rNFADDR = d;} while(0)
#define WRITE_NAND(d, adr) do {rNFDATA = d;} while(0)
#define READ_NAND(adr) (rNFDATA)
#define NAND_WAIT_READY(nand) {while(!(rNFSTAT&(1<<0)));}
#define NAND_DISABLE_CE(nand) {rNFCONF |= (1<<11);}
#define NAND_ENABLE_CE(nand) {rNFCONF &= ~(1<<11);}

/* 下面一组操作对 Nand Flash 无效 */
#define NAND_CTL_CLRALE(nandptr)
#define NAND_CTL_SETALE(nandptr)
#define NAND_CTL_CLRCLE(nandptr)
#define NAND_CTL_SETCLE(nandptr)
```

```
/* 允许 Nand Flash 写校验 */
#define CONFIG_MTD_NAND_VERIFY_WRITE 1

#endif /* CONFIG_COMMANDS & CFG_CMD_NAND*/
```

2. 加入自己的 Nand Flash 芯片型号

在 include/linux/mtd/nand_ids.h 中的对如下结构体赋值进行修改:

```
static struct nand_flash_dev nand_flash_ids[] = {
    .....
    {"Samsung K9F1208U0B", NAND_MFR_SAMSUNG, 0x76, 26, 0, 4, 0x4000, 0},
    .....
}
```

这样对于该款 Nand Flash 芯片的操作才能正确执行。

3. 编写自己的 Nand Flash 初始化函数

在 board/crane2410/crane2410.c 中加入 nand_init()函数.

```
void nand_init(void)
{
    /* 初始化 Nand Flash 控制器, 以及 Nand Flash 芯片 */
    nand_reset();
    /* 调用 nand_probe()来检测芯片类型 */
    printf ("%4lu MB\n", nand_probe(CFG_NAND_BASE) >> 20);
}
```

该函数在启动时被 start_armboot()调用.

4 在 Linux 对 Nand Flash 的支持

4.1 Linux 下 Nand Flash 调用关系

4.1.1 Nand Flash 设备添加时数据结构包含关系

```
struct mtd_partition      partition_info[]
--> struct s3c2410_nand_set nandset
--> struct s3c2410_platform_nand super1pplatfrom
--> struct platform_device s3c_device_nand
    在该数据结构的 name 字段的初始化值"s3c2410-nand",必须与 Nand Flash 设备驱动注册时
    struct device_driver 结构中的 name 字段相同,因为 platfrom bus 是依靠名字来匹配的.
--> struct platform_device *smdk2410_devices[]
```

4.1.2 Nand Flash 设备注册时数据结构包含关系

```
struct device_driver s3c2410_nand_driver
-->struct device *dev
    该数据构由系统分配.
-->struct platform_device *pdev
-->struct s3c2410_platform_nand *plat
-->struct s3c2410_nand_set nset
-->struct mtd_partition
```

4.1.3 当发生系统调用时数据结构调用关系

```
struct mtd_info
```

它的*priv指向chip
 -->struct nand_chip
 它的*priv指向nmt
 -->struct s3c2410_nand_mtd
 它是 s3c2410_nand_info 的一个字段
 -->s3c2410_nand_info
 它被设为 Nand Flash 设备驱动的私有数据结构,在 Nand Flash 设备驱动注册时分配空间.
 -->struct device

4.2 Linux 下 Nand Flash 驱动主要数据结构说明

4.2.1 s3c2410 专有数据结构

1. s3c2410_nand_set

```
struct s3c2410_nand_set {
    int            nr_chips;        /* 芯片的数目 */
    int            nr_partitions; /* 分区的数目 */
    char           *name;          /* 集合名称 */
    int            nr_map;         /* 可选, 底层逻辑到物理的芯片数目 */
    struct mtd_partition partitions; /* 分区列表 */
};
```

2. s3c2410_platform_and

```
struct s3c2410_platform_nand {
    /* timing information for controller, all times in nanoseconds */

    int    tacls; /* 从 CLE/ALE 有效到 nWE/nOE 的时间 */
    int    twrph0; /* nWE/nOE 的有效时间 */
    int    twrph1; /* 从释放 CLE/ALE 到 nWE/nOE 不活动的时间 */

    int    nr_sets; /* 集合数目 */
    struct s3c2410_nand_set sets; /* 集合列表 */

    /* 根据芯片编号选择有效集合 */
    void (*select_chip)(struct s3c2410_nand_set , int chip);
};
```

3. s3c2410_nand_mtd

在 drivers/mtd/nand/s3c2410.c 中,

```
struct s3c2410_nand_mtd {
    struct mtd_info mtd; /* MTD 信息 */
    struct nand_chip chip; /* nand flash 芯片信息 */
    struct s3c2410_nand_set set; /* nand flash 集合 */
    struct s3c2410_nand_info *info; /* nand flash 信息 */
    int scan_res;
};
```

4. s3c2410_nand_info

```
struct s3c2410_nand_info {
    /* mtd info */
    struct nand_hw_control controller; /* 硬件控制器 */
    struct s3c2410_nand_mtd *mtds; /* MTD 设备表 */
    struct s3c2410_platform_nand platform; /* Nand 设备的平台 */
};
```

```

/* device info */
struct device          *device;    /* 设备指针 */
struct resource        *area;      /* 资源指针 */
struct clk             *clk;       /* Mtd Flash 时钟 */
void __iomem          *regs;      /* 寄存器基地址(map后的逻辑地址) */
int                   mtd_count;   /* MTD的数目 */

unsigned char          is_s3c2440;
};

```

5. struct clk

在 arch/arm/mach-s3c2410/clock.h 中

```

struct clk {
    struct list_head    list;       /* clock 列表结点 */
    struct module       *owner;     /* 所属模块 */
    struct clk          *parent;    /* 父结点 */
    const char          *name;      /* 名称 */
    int                 id;         /* 编号 */
    atomic_t            used;       /* 使用者计数 */
    unsigned long        rate;      /* 时钟速率 */
    unsigned long        ctrlbit;   /* 控制位 */
    int                 (*enable)(struct clk *, int enable); /* Clock 打开方法 */
};

```

4.2.2 Linux 通用数据结构说明

1. device_driver

include/linux/device.h

```

struct device_driver {
    const char          * name;     /* 驱动名称 */
    struct bus_type      * bus;     /* 总线类型 */

    struct completion    unloaded;  /* 卸载事件通知机制 */
    struct kobject        kobj;     /* sys 中的对象 */
    struct klist          klist_devices; /* 设备列表 */
    struct klist_node     knode_bus; /* 总线结点列表 */

    struct module        * owner;   /* 所有者 */

    /* 设备驱动通用方法 */
    int      (*probe)      (struct device * dev); /* 探测设备 */
    int      (*remove)     (struct device * dev); /* 移除设备 */
    void     (*shutdown)   (struct device * dev); /* 关闭设备 */
    /* 挂起设备 */
    int      (*suspend)    (struct device * dev, pm_message_t state, u32 level);
    int      (*resume)     (struct device * dev, u32 level); /* 恢复 */
};

```

2. platform_device

include/linux/device.h

```

struct platform_device {
    const char          * name;     /* 名称 */
    u32                 id;         /* 设备编号, -1 表示不支持同类多个设备 */
    struct device        dev;       /* 设备 */
    u32                 num_resources; /* 资源数 */
    struct resource      * resource; /* 资源列表 */
};

```

```
};
```

3. resource

```
struct resource {  
    const char name;          /* 资源名称 */  
    unsigned long start, end; /* 开始位置和结束位置 */  
    unsigned long flags;       /* 资源类型 */  
    /* 资源在资源树中的父亲,兄弟和孩子 */  
    struct resource *parent, *sibling, *child;  
};
```

4. device

```
include/linux/device.h  
struct device {  
    struct klist          klist_children; /* 在设备列表中的孩子列表 */  
    struct klist_node     knode_parent;   /* 兄弟结点 */  
    struct klist_node     knode_driver;   /* 驱动结点 */  
    struct klist_node     knode_bus;      /* 总线结点 */  
    struct device         parent;        /* 父亲 */  
  
    struct kobject kobj;                  /* sys结点 */  
    char    bus_id[BUS_ID_SIZE];  
  
    struct semaphore    sem; /* 同步驱动的信号量 */  
  
    struct bus_type * bus; /* 总线类型 */  
    struct device_driver *driver; /* 设备驱动 */  
    void    *driver_data; /* 驱动的私有数据 */  
    void    *platform_data; /* 平台指定的数据,为 device 核心驱动保留 */  
    void    *firmware_data; /* 固件指定的数据,为 device 核心驱动保留 */  
    struct dev_pm_info power; /* 设备电源管理信息 */  
  
    u64    *dma_mask; /* DMA掩码 */  
    u64    coherent_dma_mask;  
    struct list_head dma_pools; /* DMA缓冲池 */  
  
    struct dma_coherent_mem *dma_mem; /* 连续 DMA 内存的起始位置 */  
  
    void    (*release)(struct device * dev); /* 释放设置方法 */  
};
```

5. nand_hw_control

```
include/linux/mtd/nand.h  
struct nand_hw_control {  
    spinlock_t    lock; /* 自旋锁,用于硬件控制 */  
    struct nand_chip *active; /* 正在处理 MTD 设备 */  
    wait_queue_head_t wq; /* 等待队列 */  
};
```

6. nand_chip

```
include/linux/mtd/nand.h  
struct nand_chip {  
    void    __iomem    *IO_ADDR_R; /* 读地址 */  
    void    __iomem    *IO_ADDR_W; /* 写地址 */
```

```

/* 字节操作 */
u_char      (*read_byte)(struct mtd_info *mtd); /* 读一个字节 */
void        (*write_byte)(struct mtd_info *mtd, u_char byte); /* 写一个字节 */
/* 双字节操作 */
u16         (*read_word)(struct mtd_info *mtd); /* 读一个字 */
void        (*write_word)(struct mtd_info *mtd, u16 word); /* 写一个字 */
/* buffer 操作 */
void        (*write_buf)(struct mtd_info *mtd, const u_char *buf, int len);
void        (*read_buf)(struct mtd_info *mtd, u_char *buf, int len);
int         (*verify_buf)(struct mtd_info *mtd, const u_char *buf, int len);
/* 选择一个操作芯片 */
void        (*select_chip)(struct mtd_info *mtd, int chip);
/* 坏块检查操作 */
int         (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
/* 坏块标记操作 */
int         (*block_markbad)(struct mtd_info *mtd, loff_t ofs);
/* 硬件控制操作 */
void        (*hwcontrol)(struct mtd_info *mtd, int cmd);
/* 设备准备操作 */
int         (*dev_ready)(struct mtd_info *mtd);
/* 命令发送操作 */
void        (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column, int
page_addr);
/* 等待命令完成 */
int         (*waitfunc)(struct mtd_info *mtd, struct nand_chip *this, int state);
/* 计算 ECC 码操作 */
int         (*calculate_ecc)(struct mtd_info *mtd, const u_char *dat, u_char
*ecc_code);
/* 数据纠错操作 */
int         (*correct_data)(struct mtd_info *mtd, u_char *dat, u_char *read_ecc,
u_char *calc_ecc);
/* 开启硬件 ECC */
void        (*enable_hwecc)(struct mtd_info *mtd, int mode);
/* 擦除操作 */
void        (*erase_cmd)(struct mtd_info *mtd, int page);
/* 检查坏块表 */
int         (*scan_bbt)(struct mtd_info *mtd);
int         eccmode; /* ECC 模式 */
int         eccsize; /* ECC 计算时使用的字节数 */
int         eccbytes; /* ECC 码的字节数 */
int         eccsteps; /* ECC 码计算的步骤数 */
int         chip_delay; /* 芯片的延迟时间 */
spinlock_t  chip_lock; /* 芯片访问的自旋锁 */
wait_queue_head_t wq; /* 芯片访问的等待队列 */
nand_state_t state; /* Nand Flash 状态 */
int         page_shift; /* 页右移的位数,即 column 地址位数 */
int         phys_erase_shift; /* 块右移的位数,即 column 和页一共的地址位数 */
int         bbt_erase_shift; /* 坏块页表的位数 */
int         chip_shift; /* 该芯片总共的地址位数 */
u_char      *data_buf; /* 数据缓冲区 */
u_char      *oob_buf; /* oob 缓冲区 */
int         oobdirty; /* oob 缓冲区是否需要重新初始化 */
u_char      *data_poi; /* 数据缓冲区指针 */
unsigned int options; /* 芯片专有选项 */
int         badblockpos; /* 坏块标示字节在 OOB 中的位置 */
int         numchips; /* 芯片的个数 */

```

```

unsigned long    chipsize; /* 在多个芯片组中, 一个芯片的大小 */
int             pagemask; /* 每个芯片页数的屏蔽字, 通过它取出每个芯片包含多少个页 */
int             pagebuf; /* 在页缓冲区中的页号 */
struct nand_oobinfo *autooob; /* oob 信息 */
uint8_t         *bbt;      /* 坏块页表 */
struct nand_bbt_descr *bbt_td; /* 坏块表描述 */
struct nand_bbt_descr *bbt_md; /* 坏块表镜像描述 */
struct nand_bbt_descr *badblock_pattern; /* 坏块检测模板 */
struct nand_hw_control *controller; /* 硬件控制 */
void            *priv; /* 私有数据结构 */
/* 进行附加错误检查 */
int             (*errstat)(struct mtd_info *mtd, struct nand_chip *this, int state, int
status, int page);
};

```

7. mtd_info

```
include/linux/mtd/mtd.h
```

```

struct mtd_info {
    u_char type; /* 设备类型 */
    u_int32_t flags; /* 设备标志位组 */
    u_int32_t size; /* 总共设备的大小 */
    u_int32_t erasesize; /* 擦除块的大小 */

    u_int32_t oobblock; /* OOB块的大小,如: 512 个字节有一个 OOB */
    u_int32_t oobsize; /* OOB数据的大小,如: 一个 OOB 块有 16 个字节 */
    u_int32_t ecctype; /* ECC 校验的类型 */
    u_int32_t eccsize; /* ECC 码的大小 */

    char *name; /* 设备名称 */
    int index; /* 设备编号 */

    /* oobinfo 信息,它可以通过 MEMSETOOBINFO ioctl 命令来设置 */
    struct nand_oobinfo oobinfo;
    u_int32_t oobavail; /* OOB 区的有效字节数,为文件系统提供 */

    /* 数据擦除边界信息 */
    int numeraseregions;
    struct mtd_erase_region_info *eraseregions;

    u_int32_t bank_size; /* 保留 */
    /* 擦除操作 */
    int (*erase)(struct mtd_info *mtd, struct erase_info *instr);
    /* 指向某个执行代码位置 */
    int (*point)(struct mtd_info *mtd, loff_t from,
                 size_t len, size_t *retlen, u_char **mtdbuf);
    /* 取消指向 */
    void (*unpoint)(struct mtd_info *mtd, u_char * addr, loff_t from, size_t len);
    /* 读/写操作 */
    int (*read)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*write)(struct mtd_info *mtd, loff_t to, size_t len,
                 size_t *retlen, const u_char *buf);
    /* 带 ECC 码的读/写操作 */
    int (*read_ecc)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
                    u_char *buf, u_char *eccbuf, struct nand_oobinfo *oobsel);
    int (*write_ecc)(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
                     const u_char *buf, u_char *eccbuf, struct nand_oobinfo *oobsel);
};

```

```

/* 带 OOB 码的读/写操作 */
int (*read_oob) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
                u_char *buf);
int (*write_oob) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
                 const u_char *buf);

/* 提供访问保护寄存器区的方法 */
int (*get_fact_prot_info) (struct mtd_info *mtd, struct otp_info *buf, size_t len);
int (*read_fact_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len,
                           size_t *retlen, u_char *buf);
int (*get_user_prot_info) (struct mtd_info *mtd, struct otp_info *buf, size_t len);
int (*read_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len,
                           size_t *retlen, u_char *buf);
int (*write_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len,
                            size_t *retlen, u_char *buf);
int (*lock_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len);

/* 提供 readv 和 writev 方法 */
int (*readv) (struct mtd_info *mtd, struct kvec *vecs, unsigned long count,
             loff_t from, size_t *retlen);
int (*readv_ecc) (struct mtd_info *mtd, struct kvec *vecs, unsigned long count,
                 loff_t from, size_t *retlen, u_char *eccbuf,
                 struct nand_oobinfo *oobsel);
int (*writev) (struct mtd_info *mtd, const struct kvec *vecs,
              unsigned long count, loff_t to, size_t *retlen);
int (*writev_ecc) (struct mtd_info *mtd, const struct kvec *vecs,
                  unsigned long count, loff_t to, size_t *retlen,
                  u_char *eccbuf, struct nand_oobinfo *oobsel);

/* 同步操作 */
void (*sync) (struct mtd_info *mtd);

/* 芯片级支持的加/解锁操作 */
int (*lock) (struct mtd_info *mtd, loff_t ofs, size_t len);
int (*unlock) (struct mtd_info *mtd, loff_t ofs, size_t len);

/* 电源管理操作 */
int (*suspend) (struct mtd_info *mtd);
void (*resume) (struct mtd_info *mtd);

/* 坏块管理操作 */
int (*block_isbad) (struct mtd_info *mtd, loff_t ofs);
int (*block_markbad) (struct mtd_info *mtd, loff_t ofs);

/* 重启前的通知事件 */
struct notifier_block reboot_notifier;

void *priv; /* 私有数据结构 */

struct module *owner; /* 模块所有者 */
int usecount; /* 使用次数 */
};

```

4.3 Linux 下 Nand Flash 驱动说明

4.3.1 注册 driver_register

通过 module_init(s3c2410_nand_init);注册 Nand Flash 驱动. 在 s3c2410_nand_init ()中通过 driver_register()注册 s3c2410_nand_driver 驱动程序,如下所示:

```
static struct device_driver s3c2410_nand_driver = {
    .name          = "s3c2410-nand",
    .bus           = &platform_bus_type, /* 在 drivers/base/platform.c 中定义 */
    .probe         = s3c2410_nand_probe,
    .remove        = s3c2410_nand_remove,
};
```

4.3.2 探测设备 probe

在注册的 Nand Flash 驱动程序中, probe 方法为 s3c2410_nand_probe(). s3c2410_nand_probe()再调用 s3c24xx_nand_probe(). 在该函数中, 把*info 作为 Nand Flash 驱动的私有数据结构, 并通过 dev_set_drvdata(dev, info)把*info 保存在*device 的*driver_data 字段中. 然后通过 clk_get(dev, "nand")获取 Nand Flash 的时钟资源, clk_use(info->clk)增加时钟资源的使用计数, clk_enable(info->clk)开启资源. 填写*info 的其它字段, 其中包括:

1. 通过 request_mem_region()为 Nand Flash 寄存器区申请 I/O 内存地址空间区,并通过 ioremap()把它映射到虚拟地址空间.
2. 调用 s3c2410_nand_inithw()初始化 Nand Flash 控制器.
3. 为 mtd 设备分配设备信息的存储空间.
4. 对当前 mtd 设备,调用 s3c2410_nand_init_chip()进行初始化.
5. 对当前 mtd 设备,调用 nand_scan()检测 Nand Flash 芯片, nand_scan()函数在 drivers/mtd/nand/nand_base.c 中定义.该函数的作用是初始化 struct nand_chip 中一些方法, 并从 Nand Flash 中读取芯片 ID, 并初始化 struct mtd_info 中的方法.
6. 对当前 mtd 设备,加入其分区信息.
7. 如果还有更多 mtd 设备,到 4 执行.

4.3.3 初始化 Nand Flash 控制器

s3c2410_nand_inithw()函数会初始化 Nand Flash 控制器, 通过设置 Nand Flash 控制寄存器(S3C2410_NFCONF)来完成, 这里最重要的是根据 S3C2410 的 PCLK 计算出 tacls, twrph0 以及 twrph1 值.

4.3.4 移除设备

s3c2410_nand_remove()当设备被移除时,被 device 核心驱动调用.它完成的主要工作如下:

1. 把*device 的*driver_data 字段置空.
2. 释放 mtd 设备信息.
3. 释放 clk 资源.
4. 通过 iounmap()取消映射地址空间.
5. 释放申请的 I/O 内存资源.
6. 释放设备私有数据*info 的空间.

4.3.5 Nand Flash 芯片初始化

s3c2410_nand_init_chip()初始化 struct nand_chip 中的一些主要字段以及方法.其中主要包括的方法有:

1. s3c2410_nand_hwcontrol(); 硬件控制
2. s3c2410_nand_devready(); 设备是否准备好
3. s3c2410_nand_write_buf(); 写一个 buffer 到 nand flash
4. s3c2410_nand_read_buf(); 读一个 buffer 到 nand flash
5. s3c2410_nand_select_chip(); 选择操作芯片

如果支持 ECC 硬件校验,还设置如下方法:

1. s3c2410_nand_correct_data(); 通过 ECC 码校正数据
2. s3c2410_nand_enable_hwecc(); 开启硬件 ECC 检查
3. s3c2410_nand_calculate_ecc(); 计算 ECC 码

4.3.6 读 Nand Flash

当对 nand flash 的设备文件(nand flash 在/dev 下对应的文件)执行系统调用 read(),或在某个文件系统中对该设备进行读操作时. 会调用 struct mtd_info 中的 read 方法,他们缺省调用函数为 nand_read(),在 drivers/mtd/nand/nand_base.c 中定义. nand_read()调用 nand_do_read_ecc(),执行读操作. 在 nand_do_read_ecc()函数中,主要完成如下几项工作:

1. 会调用在 nand flash 驱动中对 struct nand_chip 重载的 select_chip 方法,即 s3c2410_nand_select_chip()选择要操作的 MTD 芯片.
2. 会调用在 struct nand_chip 中系统缺省的方法 cmdfunc 发送读命令到 nand flash.
3. 会调用在 nand flash 驱动中对 struct nand_chip 重载的 read_buf(),即 s3c2410_nand_read_buf()从 Nand Flash 的控制器的数据寄存器中读出数据.
4. 如果有必要的话,会调用在 nand flash 驱动中对 struct nand_chip 重载的 enable_hwecc,correct_data 以及 calculate_ecc 方法,进行数据 ECC 校验。

4.3.7 写 Nand Flash

当对 nand flash 的设备文件(nand flash 在/dev 下对应的文件)执行系统调用 write(),或在某个文件系统中对该设备进行读操作时, 会调用 struct mtd_info 中 write 方法,他们缺省调用函数为 nand_write(),这两个函数在 drivers/mtd/nand/nand_base.c 中定义. nand_write()调用 nand_write_ecc(),执行写操作.在 nand_do_write_ecc()函数中,主要完成如下几项工作:

1. 会调用在 nand flash 驱动中对 struct nand_chip 重载的 select_chip 方法,即 s3c2410_nand_select_chip()选择要操作的 MTD 芯片.
2. 调用 nand_write_page()写一个页.
3. 在 nand_write_page()中,会调用在 struct nand_chip 中系统缺省的方法 cmdfunc 发送写命令到 nand flash.
4. 在 nand_write_page()中,会调用在 nand flash 驱动中对 struct nand_chip 重载的 write_buf(),即 s3c2410_nand_write_buf()从 Nand Flash 的控制器的数据寄存器中写入数据.
5. 在 nand_write_page()中,会调用在 nand flash 驱动中对 struct nand_chip 重载 waitfunc 方法,该方法调用系统缺省函数 nand_wait(),该方法获取操作状态,并等待 nand flash 操作完成.等待操作完成,是调用 nand flash 驱动中对 struct nand_chip 中重载的 dev_ready 方法,即 s3c2410_nand_devready()函数.