Linux 嵌入式系统设计与开发

1　第一部分嵌入式系统开发环境

第一章　嵌入式系统开发环境的搭建，以及相关工具的使用

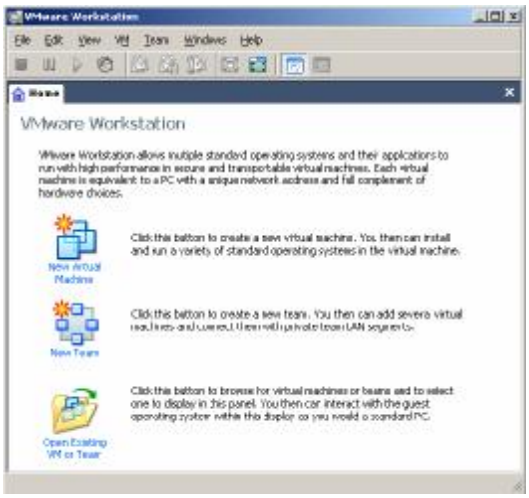
1.1　虚拟机+linux redhat9.0 安装

1.1.1　虚拟机软件的安装及配置。

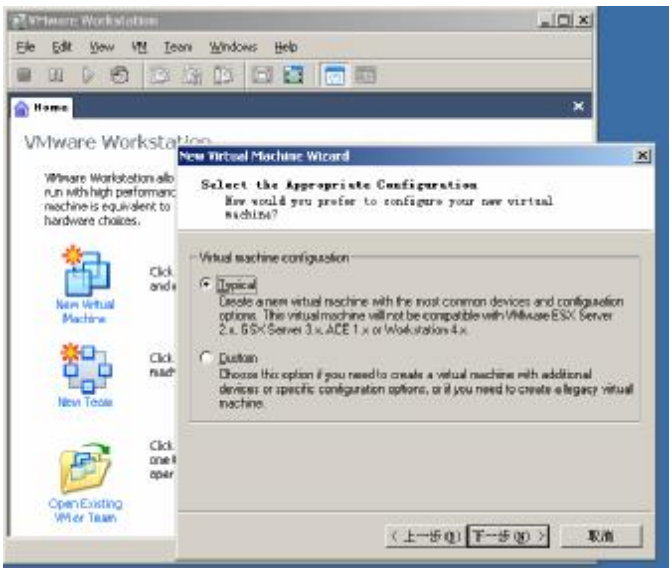
虚拟机软件是一种可以在一台电脑上模拟出来若干台 PC，每台 PC 可以运行单独操作系而互不干扰，实现一台电脑“同时”运行几个操作系统，还可以将这几个操作系统连成一个网络的软件。

采用 VMware Workstation5.5 例说明如何在 windows 创建一个虚拟机环境。

VMware Workstation 安装后的界面如下：

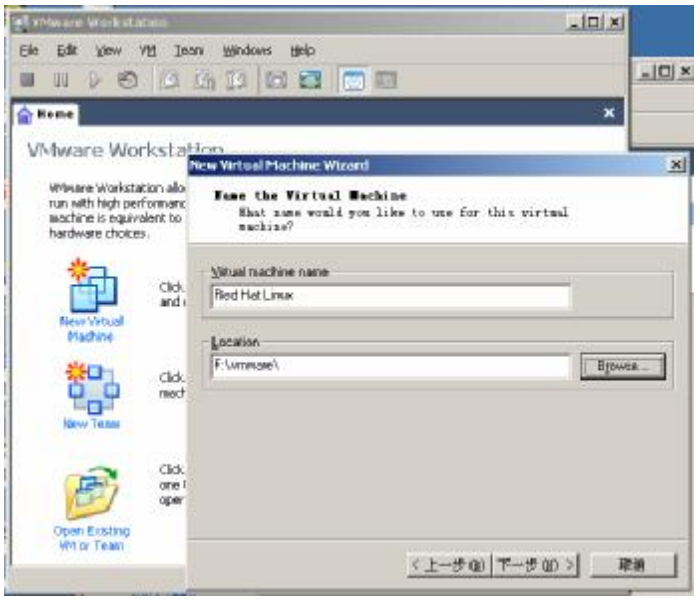


创建一个虚拟机



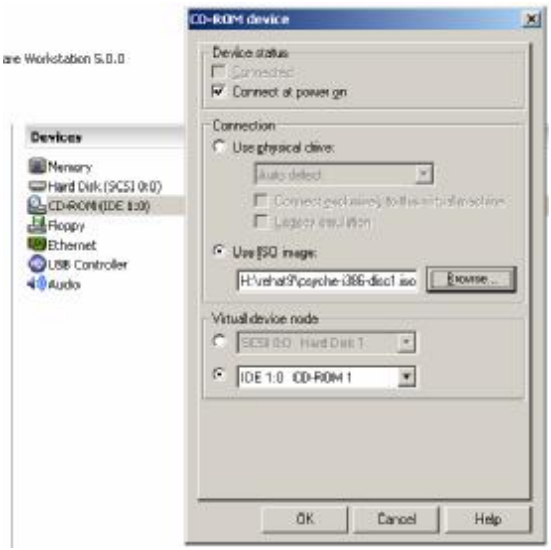
选择 linux

指定虚拟机存放的路径



选择网络配置

对于虚拟机的设备进行配置，这里采用虚拟光驱，指定安装 linux 镜像的路径



设备相关信息设置如下，启动虚拟机，开始安装。

进行 linux 安装

1.1.2 Redhat9.0 的安装。

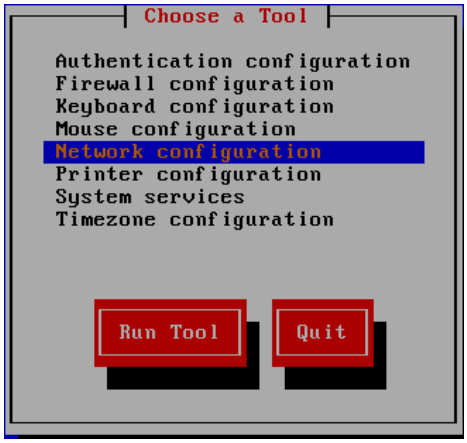
注意不安装防火墙，采用 workstation，确保服务 samba,tftp,tfp,nfs,SSH，DHCP，telnet 都安装上。

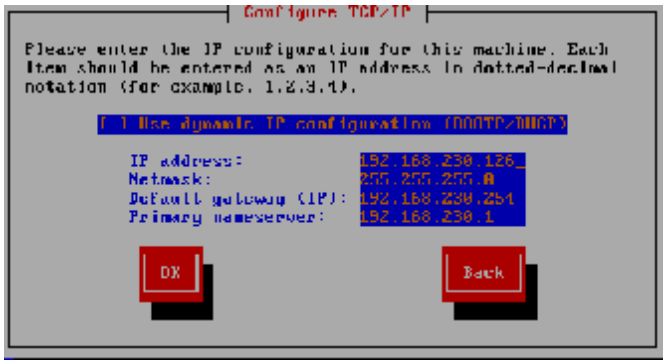
1.2 Linux 系统服务的配置

配置系统服务时，要使防火墙关闭，或使其为低。

1.2.1 网络配置

[root@localhost /]# setup　　进入网络配置菜单，设置 ip。





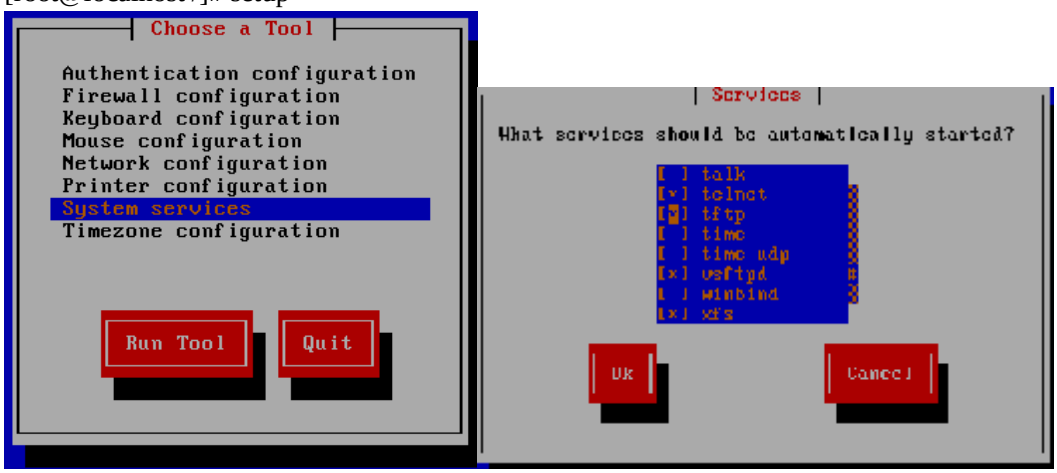
[root@localhost root]# service network restart　　重启网络服务

[root@localhost root]# ifconfig　　察看网络配置

1.2.2 Tftp 服务的配置

选择 tftp 服务

[root@localhost /]# setup



配置 tftp 服务目录

[root@localhost /]# 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
}

启动 tftp 服务

[root@localhost /]# service xinetd restart

Stopping xinetd: [OK]

Starting xinetd: [OK]

检测 tftp 服务

[root@localhost /]# netstat -a | grep tftp

udp 0 0 *:tftp *:*

1.2.3 Samba 服务器配置

选择 samba 服务

[root@localhost /]# setup

编辑 smb.conf 配置文件，

[root@localhost /]# vi /etc/samba/smb.conf

增加用户名以及共享路径，如下：

[homes]

comment = Home Directories

path = /

browseable = no

writable = yes

valid users = root

create mode = 0664

directory mode = 0775

[root@localhost /]#smbadduser root:admin

增加 samba 用户

[root@localhost /]# service smb restart

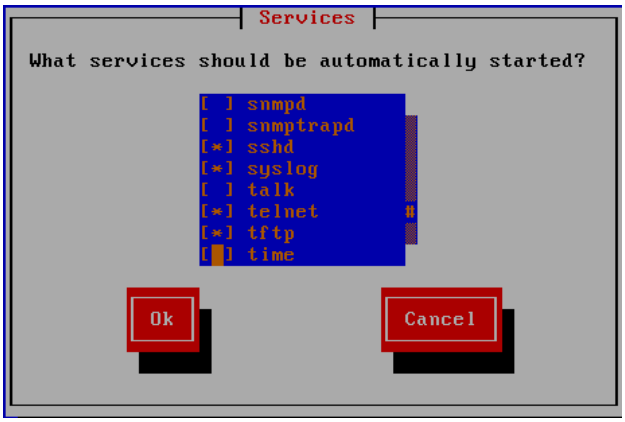
激活 samba 服务

[root@localhost /]# service smb status

察看 samba 服务

1.2.4 telnet 服务器配置

选择 telnet 服务



[root@localhost /]# vi /etc/pam.d/login

注释掉 auth required /lib/security/pam_securetty.so 这句话

重启服务

[root@localhost /]# service xinetd restart

至此可以以 root 用户 telnet 登录

1.2.5 nfs 服务器配置

选择 nfs 服务

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

增加 nfs 服务输出的路径以及供给的 ip 地址，例如增加如下条目：

/home/nfsroot 192.168.230.128(rw,sync)

表示 ip 地址为 192.168.230.128 可以读写本机的目录/home/nfsroot

[root@localhost /]# service nfs restart

激活 nfs 服务

[root@localhost /]# service nfs status

察看 nfs 服务的状态

1.2.6 DHCP 服务器配置

注意：当配置 DHCP 服务时会对于局域网有影响，建议在虚拟机的环境下，网络设备选择 Host-only 工作方式，如果不是虚拟机，请把网络连接与局域网断开。

选择 DHCP 服务器

[root@localhost /]# cp /usr/share/doc/dhcp-3.0pl1/dhcpd.conf.sample /etc/dhcpd.conf

创建一个 DHCP 服务的配置文件

[root@localhost /]# vi /etc/dhcpd.conf

编辑 dhcpd.conf 配置 DHCP 服务，下面的例子实现了基本的配置，目的是为在虚拟机上实现 linux 内核在网络上的启动。本机的 ip 是 192.168.230.129

```
ddns-update-style interim;
ignore client-updates;

subnet 192.168.230.0 netmask 255.255.255.0 {

# --- default gateway
        option routers                  192.168.230.129;
        option subnet-mask              255.255.255.0;

        option time-offset              -18000; # Eastern Standard Time

        range dynamic-bootp 192.168.230.120 192.168.230.128;
        default-lease-time 21600;
        max-lease-time 43200;
        filename "/pxelinux.0";
        #为实现通过网络启动加载的文件
        # we want the nameserver to appear at a fixed address
}
```

[root@localhost /]# service dhcpd start

开启 dhcp 服务

[root@localhost /]# service dhcpd status

察看服务是否运行

1.2.7 支持网络启动内核的配置方法

基本原理,实现网络启动必须支持 PXE，PXE(Pre-boot Execution Environment)是由 Intel 设计的协议，它可以使计算机通过网络启动。协议分为 client 和 server 两端，PXE client 在网卡的 ROM 中，当计算机引导时，BIOS 把 PXE client 调入内存执行，并显示出命令菜单，经用户选择后，PXE client 将放置在远端的操作系统通过网络下载到本地运行。

PXE 协议的成功运行需要解决以下两个问题：

- 既然是通过网络传输，那么计算机在启动时，它的 IP 地址由谁来配置；
- 通过什么协议下载 Linux 内核和根文件系统

对于第一个问题，可以通过 DHCP Server 解决，由 DHCP server 来给 PXE client 分配一个 IP 地址，DHCP Server 是用来给 DHCP Client 动态分配 IP 地址的协议，不过由于这里是给 PXE Client 分配 IP 地址，所以在配置 DHCP Server 时，需要增加相应的 PXE 特有配置。

至于第二个问题，在 PXE client 所在的 ROM 中，已经存在了 TFTP Client。PXE Client 使用 TFTP Client，通过 TFTP 协议到 TFTP Server 上下载所需的文件。

1.2.7.1 配置 DHCP 和 tfpt 服务，配置方式见以上描述。

1.2.7.2 配置 bootstrap

Bootstrap 文件是可执行程序，它向用户提供简单的控制界面，并根据用户的选择，下载合适的 Linux 内核以及 Linux 根文件系统。bootstrap 文件在 dhcpd.conf 中被指定为 pxelinux.0 文件，放置在/tftpboot。Linux 内核以及 Linux 根文件系统也放置在/tftpboot。pxelinux.0 在执行过程中，要读配置文件，所有的配置文件都放在/tftpboot/pxelinux.cfg/目录下。由于 PXElinux 具有为不同的 PXE Client 提供不同的 Linux 内核以及根文件系统的功能，所以要通过不同的配置文件名来区分出不同的 PXE Client 的需求。比如一个 PXE Client 由 DHCP Server 分配的 IP 地址为 192.168.0.22，那么相对应的配置文件名为 /tftpboot/pxelinux.cfg/C0A80016（注：C0A80016 为 IP 地址 192.168.0.22 的十六进制表示）。如果找不到，就按照顺序 C0A80016-> C0A8001-> C0A800-> C0A80-> C0A8-> C0A-> C0-> C->default 查找配置文件。

[root@localhost /]# cp /usr/lib/syslinux/pxelinux.0 /tftpboot/

[root@localhost /]# cd /tftpboot/

[root@localhost tftpboot]# mkdir pxelinux.cfg

[root@localhost tftpboot]# cd pxelinux.cfg/

[root@localhost pxelinux.cfg]# vi default

```
default linux-boot
prompt 1
timeout 30
label linux-boot
kernel vmlinuz
append initrd=initrd devfs=nomount ramdisk_size=9216
```

把内核 vmlinuz，initrd 拷贝到/tftp 下，这样当启动另外一个虚拟机，启动时按 F12，即可进入网络启动。启动界面如下：

1.3 交叉工具链的介绍与使用

1.3.1 基础知识

交叉编译，就是在一个平台上生成另一个平台上的可执行代码。这里需要注意的是所谓平台，实际上包含两个概念：体系结构（Architecture）、操作系统（Operating System）。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。

在主机平台上开发程序，并在这个平台上运行交叉编译器，编译我们的程序；而由交叉编译器生成的程序将在目的平台上运行。例如：

arm-linux，说明平台所使用的是 arm 体系结构，运行的操作系统是 linux

1.3.2 GNU 交叉工具链的下载

Arm 工具链的官方下载地址：

ftp://ftp.arm.linux.org.uk/pub/arm-linux-/toolchain/cross2.95.3.tar.bz2

ftp://ftp.arm.linux.org.uk/pub/arm-linux-/toolchain/cross3.0.tar.bz2

ftp://ftp.arm.linux.org.uk/pub/arm-linux-/toolchain/cross3.2.tar.bz2

http://www.handhelds.org/download/projects/toolchain/ 可以下载 arm-linux-gcc-3.4.1 编译 linux2.6 的内核

1.3.3 GNU 交叉工具链的介绍以及使用

1.3.3.1 常用工具介绍

arm-linux-as	编译 ARM 汇编程序
arm-linux-ar	把多个.o 合并成一个.o 或静态库(.a)，
arm-linux-ranlib	为库文件建立索引，相当于 arm-linux-ar –s
arm-linux-ld	连接器(Linker)，把多个.o 或库文件连接成一个可执行文件
arm-linux-objdump	查看目标文件(.o)和库(.a)的信息
arm-linux-objcopy	转换可执行文件的格式
arm-linux-strip	去掉 elf 可执行文件的信息. 使可执行文件变小
arm-linux-readelf	读 elf 可执行文件的信息
arm-linux-gcc	编译.c 或.S 开头的 C 程序或汇编程序
arm-linux-nm	用来列出目标文件的符号清单

1.3.3.2 主要工具的使用

1.3.3.2.1 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>
extern void hellosecond(void);
extern 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:

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

方法 2:

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

方法 3:

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

方法 4：

```
[root@localhost codetest]#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，适合看一个宏在源文件中产生的结果。

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

E:产生一个预处理文件.

1.3.3.2.2 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

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

2. 为静态库建立索引

```
[root@localhost codetest]#arm-linux-ar -s    libhello.a
[root@localhost codetest]#arm-linux-ranlib libhello.a
```

3. 由静态库产生可执行文件

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

1.3.3.2.3 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
```

1.3.3.2.4 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
.....
```

2. 读 elf 文件中所有 ELF 的头部:

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

.....

3. 显示整个文件的符号表

[root@localhost codetest]#arm-linux-readelf -s hello

.....

4. 显示使用的动态库

[root@localhost codetest]#arm-linux-readelf -d hello

.....

1.3.3.2.5 arm-linux-strip 的使用

1. 移除所有的符号信息

[root@localhost codetest]#arm-linux-strip —strip-all hello

strip-all:是移除所有符号信息

2. 移除调试符号信息

[root@localhost codetest]#arm-linux-strip -g hello

1.3.3.2.6 arm-linux-objcopy 的使用

生成可以执行的 2 进制代码

[root@localhost codetest]#arm-linux-objcopy -O binary hello hello.bin

1.3.3.2.7 arm-linux-nm 的使用

列出目标文件的符号清单

[root@localhost codetest]#arm-linux-nm hello.o

1.4 SSH Secure Shell 客户端软件的安装及使用

实现更安全的远程登录，可进行 ftp 操作。

1.5 串口终端 Procomm Plus 的使用

可以运行命令脚本，实现多个命令的自动运行

1.6 使用 Source insight 进行代码阅读

Linux 源代码阅读的利器

1.7 文件比较工具 Araxis Merge2001 的使用

可以方便对比文件之间的差异

1.8 文件编辑工具 UltraEdit 的使用

第二部分最小系统的启动

2 第二章 u-boot 的移植

2.1 u-boot 介绍

Uboot 是德国 DENX 小组的开发用于多种嵌入式 CPU 的 bootloader 程序, UBoot 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。UBoot 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。本文的代码取自于 uboot1.1.4。

2.2 获取 u-boot

从下面地址下载 uboot 的源代码。

http://sourceforge.net/projects/uboot

[uboot@localhost uboot]#tar xjvf uboot1.1.4.tar.bz2

[uboot@localhost uboot]#cd uboot1.1.4

2.3 u-boot 体系结构

1. 目录树

```
.
|board
|common
|cpu
|disk
|doc
|drivers
|dtt
|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 uboot 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 Uboot 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。Uboot 有非常完善的文档，推荐大家参考阅读。

7. drivers：Uboot 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，Uboot 现在支持 cramfs、fat、fdos、jffs2 和 registerfs。

9. include：Uboot 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 asmarm.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 Uboot 的工具，如：mkimage, crc 等等。

2.4 u-boot 编译及配置

2.4.1 u-boot 的 Makefile 分析

u-boot 的 Makefile 从功能上可以分成两个部分。一部分是用来编译生成 uboot.bin 文件；另一部分是用来执行每种 board 相关的配置。下面以 smdk2410 为例来说明作了哪些配置。

$make smdk2410_config

在 shell 执行以上命令，对应于 Makefile 执行的命令是

```
smdk2410_config :    unconfig
    @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

首先执行如下命令，删除文件 include/config.h include/config.mk board/*/config.tmp，后续会发现这些文件是如何建立的。

```
unconfig:
    @rm -f include/config.h include/config.mk board/*/config.tmp
```

然后运行命令@./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0，mkconfig 是脚本文件，传入的参数$1 至$6 分别为：smdk2410 arm arm920t smdk2410 NULL s3c24x0，根据传入的参数执行如下命令

```
cd ./include
rm –f asm
ln –s asm-arm asm
rm –f asm-arm/arch
ln –s arch-s3c24x0 asm-arm/arch
rm –f asm-arm/proc
ln –s proc-armv asm-arm/proc
```

生成文件 config.mk，文件内容为：

```
ARCH    = arm
CPU     = arm920t
BOARD   = smdk2410
SOC     = s3c24x0
```

生成文件 config.h，文件内容为：

```
/* Automatically generated - do not edit */
```

#include <configs/smdk2410.h>

至此 make smdk2410_config 的命令全部执行完毕。配置完成与 board 相关的信息，下面就可以编译此 board 的 u-boot.bin 文件，执行如下命令：

$make CROSS_COMPILE=arm-linux-

Makefile 的执行首先包含 include include/config.mk 文件，获取 ARCH CPU BOARD VENDOR SOC 的定义，然后根据宏的配置编译指定的文件，最终生成 u-boot.bin 文件，执行流程请自行分析。

2.4.2 u-boot.bin 的生成

根据以上对于 makefile 的分析，u-boot.bin 的生成分为两步。如下：

对于 board 进行配置： $make smdk2410_config

进行编译生成 u-boot.bin： $make CROSS_COMPILE=arm-linux-

2.5 u-boot 的启动过程及工作原理

2.5.1 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 BootLoader 的正常工作模式，因此在嵌入式产品发布的时侯，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 BootLoader 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。BootLoader 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 BootLoader 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。

UBoot 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，uboot 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。

2.5.2 阶段 1 介绍

uboot 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：

2.5.2.1 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/smdk2410/uboot.lds: ENTRY(_start) ==> cpu/arm920t/start.o (.text)

2. uboot 在 ram 的代码区（TEXT_BASE = 0x33F80000）定义在 board/smdk2410/config.mk

2.5.2.2 设置异常向量

.globl _start

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

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

2.5.2.3 设置 CPU 的模式为 SVC 模式

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

2.5.2.4 关闭看门狗，禁掉所有中断，设置 CPU 的频率

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
ldr         r0, =pWTCON
mov         r1, #0x0
str         r1, [r0]
/*
 * mask all IRQs by setting all bits in the INTMR - default
 */
mov  r1, #0xffffffff
ldr     r0, =INTMSK
str     r1, [r0]
# if defined(CONFIG_S3C2410)
ldr     r1, =0x3ff
ldr     r0, =INTSUBMSK
str     r1, [r0]
# endif

/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
ldr     r0, =CLKDIVN
mov  r1, #3
str     r1, [r0]
#endif       /* CONFIG_S3C2400 || CONFIG_S3C2410 */
```

2.5.2.5 与内存管理相关寄存器的设置，cp15 协处理器，配置内存区控制寄存器

```
cpu_init_crit:
/*
 * flush v4 I/D caches
 */
mov  r0, #0
```

```
mcr   p15, 0, r0, c7, c7, 0     /*  失效 I/D cache，见 S3C2410 手册附录的 2-16 */
mcr   p15, 0, r0, c8, c7, 0     /*  失效 TLB，见 S3C2410 手册附录的 2-18 */

/*
 * disable MMU stuff and caches
 */
mrc   p15, 0, r0, c1, c0, 0
bic   r0, r0, #0x00002300   /*  清除  bits 13, 9:8 (--V- --RS)
                            * Bit 8: Disable System Protection
                            * Bit 7: Disable ROM Protection
                            * Bit 13:  异常向量表基地址: 0x0000 0000
                            */
bic   r0, r0, #0x00000087   /*  清除  bits 7, 2:0 (B--- -CAM)
                            * Bit 0: MMU disabled
                            * Bit 1: Alignment Fault checking disabled
                            * Bit 2: Data cache disabled
                            * Bit 7: 0 = Little-endian operation
                            */
orr   r0, r0, #0x00000002   @ set bit 2 (A) Align
orr   r0, r0, #0x00001000   @ set bit 12 (I) I-Cache
mcr   p15, 0, r0, c1, c0, 0

/*
 * before relocating, we have to setup RAM timing
 * because memory timing is board-dependend, you will
 * find a lowlevel_init.S in your board directory.
 */
mov  ip, lr
bl    lowlevel_init     /*寄存器的具体值的设置需要对总线周期及
                        外围芯片非常熟悉，根据所采用的内存芯片确定*/
mov  lr, ip
mov  pc, lr
```

2.5.2.6 把 u-boot.lds 定义的 text 段，rodata 段，data 段，got 段，__u_boot_cmd_start 段搬移到 ram 区

```
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
relocate:                       /* relocate U-Boot to RAM        */
      adr   r0, _start          /* r0 <- current position of code   */
      ldr   r1, _TEXT_BASE            /* test if we run from flash or RAM */
      cmp       r0, r1                    /* don't reloc during debug            */
      beq       stack_setup

      ldr   r2, _armboot_start
      ldr   r3, _bss_start
```

```
    sub   r2, r3, r2          /* r2 <- size of armboot                  */
    add   r2, r0, r2          /* r2 <- source end address               */


copy_loop:
    ldmiar0!, {r3-r10}            /* copy from source address [r0]      */
    stmia r1!, {r3-r10}           /* copy to     target address [r1]    */
    cmp   r0, r2              /* until source end addreee [r2]      */
    ble   copy_loop
#endif      /* CONFIG_SKIP_RELOCATE_UBOOT */
```

2.5.2.7 建立 stack 空间

```
stack_setup:
    ldr   r0, _TEXT_BASE              /* upper 128 KiB: relocated uboot     */
    sub   r0, r0, #CFG_MALLOC_LEN     /* malloc area                                */
    sub   r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                                  */
#ifdef CONFIG_USE_IRQ
    sub   r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
    sub   sp, r0, #12         /* leave 3 words for abort-stack      */
```

2.5.2.8 bss 段清 0

```
clear_bss:
    ldr   r0, _bss_start          /* find start of bss segment          */
    ldr   r1, _bss_end            /* stop here                              */
    mov  r2, #0x00000000      /* clear                                  */


clbss_l:str  r2, [r0]             /* clear loop...                          */
    add   r0, r0, #4
    cmp   r0, r1
    ble   clbss_l
```

2.5.2.9 进入 C 代码部分

```
ldr   pc, _start_armboot
_start_armboot:  .word start_armboot
```

2.5.3 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

uboot(armboot）的主函数，该函数主要完成如下操作：

2.5.3.1 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,
#if defined(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

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

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();
```

2.5.3.2 初始化网络设备

初始化相关网络设备，填写 IP、MAC 地址等。

2.5.3.3 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 */
}
```

2.6 u-boot 命令使用说明

2.6.1 命令配置

2.6.2 命令帮助获取

通过 help 可以获得当前开发板的 UBOOT 中支持的命令.

```
# 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

2.6.3 命令使用说明

- askenv(F)
  在标准输入（stdin）获得环境变量。
- 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).
  # autoscr 0x30008000
  ## Executing script at 30008000
- base
  打印或者设置当前指令与下载地址的地址偏移。
- bdinfo
  打印开发板信息
  # bdinfo
  -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）
- bootp
  通过网络使用 Bootp 或者 TFTP 协议引导境像文件。
  # help bootp
  bootp [loadAddress] [bootfilename]
- bootelf
  默认从 0x30008000 引导 elf 格式的文件(vmlinux)
  # help bootelf
  bootelf [address] - load address of ELF image.

- 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:

  # set bootcmd printenv

  # boot

  bootdelay=3

  baudrate=115200

  ethaddr=01:23:45:67:89:ab

  # bootd

  bootdelay=3

  baudrate=115200

  ethaddr=01:23:45:67:89:ab

- 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：

  # 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)

- bootm

  内核的入口地址开始引导内核。

  # bootm 0x30008000

  ## Booting image at 30008000 ...

  Starting kernel ...

  Uncompressing

  Linux.....

  done, .

- go

  直接跳转到可执行文件的入口地址，执行可执行文件。

  # go 0x30008000

  ## Starting application at 0x30008000 ...

- cmp

  对输入的两段内存地址进行比较。

cmp 0x30008000 0x30008040 64
word at 0x30008000 (0xe321f0d3) != word at 0x30008040 (0xc022020c)
Total of 0 words were the same
cmp 0x30008000 0x30008000 64
Total of 100 words were the same

- coninfo

  打印所有控制设备和信息，例如

  -List of available devices:

  -serial 80000003 SIO stdin stdout stderr

- cp

  内存拷贝，cp 源地址 目的地址 拷贝大小（字节）

  # help cp

  cp [.b, .w, .l] source target count

  ANE2410 # cp 0x30008000 0x3000f000 64

- date

  获得/设置/重设日期和时间

  # date

  Date: 2006-6-6 (Tuesday) Time: 06:06:06

- erase(F)

  擦除 FLASH MEMORY。

  # 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

- flinfo(F)

  打印 Nor Flash 信息。

- 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/teach2410.h 中定义.

  # iminfo

  ## Checking Image at 30008000 ...

  Image Name: Linux-2.6.14.1

  Created: 2006-010-01 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

- loadb

  从串口下载二进制文件

  # loadb

  ## Ready for binary (kermit) download to 0x30008000 at 115200 bps...

  ## Total Size = 0x00000000 = 0 Bytes

  ## Start Addr = 0x30008000

- md

  显示指定内存地址中的内容

  # md 0

  00000000: ea000012 e59ff014 e59ff014 e59ff014 .....

  00000010: e59ff014 e59ff014 e59ff014 e59ff014 .....

  00000020: 33f80220 33f80280 33f802e0 33f80340 ..3...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.....O.`.....P.

  000000a0: 0a000007 e51f2068 e51f3068 e0432002 ....h ..h0... C.

  000000b0: e0802002 e8b007f8 e8a107f8 e1500002 . .....P.

  000000c0: daffffb e51f008c e2400803 e2400080 .....@...@.

  000000d0: e240d00c e51f0094 e51f1094 e3a02000 ..@..... ..

  000000e0: e5802000 e2800004 e1500001 daffffb . .....P.....

  000000f0: eb000006 e59f13d0 e281f000 e1a00000 .....

- 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。

  # mm 0x30008000

  30008000: e1a00000 ? fffff

  30008004: e1a00000 ? eeeeee

  30008008: e1a00000 ? q

  # md 30008000

  30008000: 000fffff 00eeeeee e1a00000 e1a00000 .....

  30008010: e1a00000 e1a00000 e1a00000 e1a00000 .....

  30008020: ea000002 016f2818 00000000 000df1d8 .....(o.....

  30008030: e1a07001 e3a08000 e10f2000 e3120003 .p..... .....

- mtest

  简单的 RAM 检测

  # mtest

  Pattern FFFFFFFD Writing... Reading...

- mw

  向内存地址写内容

md 30008000
30008000: ffffdffd ffffdffc ffffdffb ffffdffa .....
mw 30008000 0 4
md 30008000
30008000: 00000000 00000000 00000000 00000000 .....

- nm

  修改内存地址，地址不递增

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

- printenv

  打印环境变量

  # 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

- ping

  ping 主机

  # ping 10.0.0.1
  host 10.0.0.1 is alive

- reset

  复位 CPU

- run

  运行已经定义好的 U-BOOT 的命令

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

- saveenv(F)

  保存设定的环境变量

- setenv

  设置环境变量

  # setenv ipaddr 10.0.0.254
  # printenv
  ipaddr=10.0.0.254

- sleep

  命令延时执行时间

sleep 1

- version

  打印 U-BOOT 版本信息

  # version

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

- nand info

  打印 nand flash 信息

  # nand info

  Device 0: Samsung K9F1208U0B at 0x4e000000 (64 MB, 16 kB sector)

- nand device <n>

  显示某个 nand 设备

  # nand device 0

  Device 0: Samsung K9F1208U0B at 0x4e000000 (64 MB, 16 kB sector)

  ... is now current device

- nand bad

  # nand bad

  Device 0 bad blocks:

- 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。

nand read 0x30008000 0 0x100000

NAND read: device 0 offset 0, size 1048576 ...

1048576 bytes read: OK

- nand erease

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

nand erase 0x100000 0x20000

NAND erase: device 0 offset 1048576, size 131072 ... OK

- 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

nand write 0x30f00000 0x100000 0x20000

NAND write: device 0 offset 1048576, size 131072 ...

131072 bytes written: OK

- 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

#setenv autostart yes

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 ...

2.7 u-boot 的移植过程

我们为当前移植的板子取名叫: teach2410，在 uboot 中建立自己的开发板类型

2.7.1 修改 Makefile 修改

修改 Makefile

[uboot@localhost uboot]#vi Makefile

#为 teach2410 建立编译项

teach2410_config : unconfig

@./mkconfig $(@:_config=) arm arm920t teach2410 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。

teach2410: 开发板的型号(BOARD)，对应于 teach2410 目录。

NULL: 开发者/或经销商(vender)。

s3c24x0: 片上系统(SOC)

2.7.2 在 board 子目录中建立 teach2410

$cp rf board/smdk2410 board/teach2410

$cd board/teach2410

$mv smdk2410.c teach2410.c

2.7.3 在 include/configs/中建立配置头文件

$cp include/configs/smdk2410.h include/configs/teach2410.h

2.7.4 编译

$make teach2410_config

$make CROSS_COMPILE=arm-linux-

2.7.5 u-boot 移植过程中的调试方法

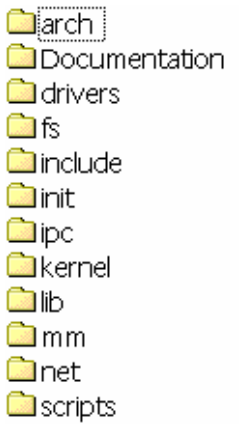
点灯

串口打印

使用 u-boot 的命令查看信息

3 第三章 linux 内核移植

3.1 Linux2.4.20 的代码布局



- arch 目录

arch 是 architecture 的缩写，内核中与具体 cpu 和体系结构相关的代码以单独目录进行存放，而相应的头文件.h 则分别放在 include/asm 目录下。

在每个 cpu 的子目录下，又进一步分解为 boot,mm,kernel 等子目录，分别包含与系统引导，内存管理，系统调用的进入与返回，中断处理以及其它内核代码依赖于 cpu 和系统结构的底层代码。

- kernel 目录

linux 大多数关键的核心功能都是在这个目录实现。（调度程序，进程控制，模块化，其它操作）

- mm 目录

mm 目录中的文件为 Linux 核心实现内存管理中体系结构无关的部分。这个目录包含换页及内存的分配和释放的函数，还有允许用户进程将内存区间映射到它们地址空间的各种技术。

- fs 目录

所有的文件系统实现的代码。每个目录分别对应一种文件系统的实现，公用的源程序则用于“虚拟文件系统”vfs。

- ipc 和 lib 目录

进程间通信和库函数各有一个小的专用目录。

- drivers

包括各种块设备与字符设备的驱动程序。

- net 目录

包含各种不同网卡和网络规程的设备驱动程序。

- include 目录

包含了所有的.h 文件，同时依据 arch 的目录结构作相应的组织。

3.2 Linux2.6.11 内核配置系统

首先分析了 Linux 内核中的配置系统结构，然后解释 Makefile 和配置文件的格式以及配置语句的含义。最后，通过一个简单的例子，具体说明如何将自行开发的代码加入到 Linux 内核中。

3.2.1 配置系统的基本结构

Linux 内核的配置系统由三个部分组成，分别是：

- Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则。
- 配置文件（Kconfig）：给用户提供配置选择的功能。
- 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。

这些配置工具都是使用脚本语言，如 Tcl/TK、Perl 编写的（也包含一些用 C 编写的代码）。本文并不是对配置系统本身进行分析，而是介绍如何使用配置系统。所以，除非是配置系统的维护者，内核开发者无须了解它们的原理，只需要知道如何编写 Makefile 和配置文件就可以。所以，在本文中，我们只对 Makefile 和配置文件进行讨论。另外，凡是涉及到与具体 CPU 体系结构相关的内容，我们都以 ARM 为例，这样不仅可以将讨论的问题明确化，而且对内容本身不产生影响。

3.2.2 Makefile

3.2.2.1 Makefile 概述

Makefile 的作用是根据配置的情况，构造出需要编译的源文件列表，然后分别编译，并把目标代码链接到一起，最终形成 Linux 内核二进制文件。由于 Linux 内核源代码是按照树形结构组织的，所以 Makefile 也被分布在目录树中。Linux 内核中的 Makefile 以及与 Makefile 直接相关的文件有：

- Makefile：顶层 Makefile，是整个内核配置、编译的总体控制文件。
- .config：内核配置文件，包含由用户选择的配置选项，用来存放内核配置后的结果（如 make menuconfig）。
- arch/*/Makefile：位于各种 CPU 体系目录下的 Makefile，如 arch/arm/Makefile，是针对特定平台的 Makefile。
- 各个子目录下的 Makefile：比如 drivers/Makefile，负责所在子目录下源代码的管理。
- scripts \Makefile.*：规则文件，被所有的 Makefile 使用。

用户通过 make menuconfig 配置后，产生了 .config。顶层 Makefile 读入 .config 中的配置选择。顶层 Makefile 有两个主要的任务：产生 vmlinux 文件和内核模块（module）。为了达到此目的，顶层 Makefile 递归的进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile。至于到底进入哪些子目录，取决于内核的配置。在顶层 Makefile 中，有一句：include arch/$(ARCH)/Makefile，包含了特定 CPU 体系结构下的 Makefile，这个 Makefile 中包含了平台相关的信息。位于各个子目录下的 Makefile 同样也根据 .config 给出的配置信息，构造出当前配置下需要的源文件列表，编译出与内核直接相连的目标文件，或者是模块。scripts/Makefile.* 为所有的 makefile 文件定义共有的规则。

3.2.2.2 Makefile 分析

3.2.2.2.1 目标定义

目标定义是子目录 Makefile 中最重要的部分，定义了在本子目录下那些目标被编译，根据特殊的编译选项如何链接目标文件，同时控制哪些子目录要递归进入进行编译。

- 被编译到 Linux 内核 vmlinux 中的目标文件列表：

Makefile 会编译所有的$(obj-y)中定义的文件，然后调用链接器将这些文件链接到 built-in.o 文件中。最终 built-in.o 文件通过顶层 Makefile 链接到 vmlinux 中。值得注意的是

$(obj-y)的文件顺序很重要。列表文件可以重复，文件第一次出现时将会链接到 built-in.o 中，后来出现的同名文件将会被忽略。文件顺序直接决定了他们被调用的顺序。

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN)                  += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)      += isdn_bsdcomp.o
```

当 CONFIG_ISDN，CONFIG_ISDN_PPP_BSDCOMP 定义为 y 时。会被编译到 vmlinux。

- 被编译到模块的目标文件列表

以$(obj-m)定义的目标文件，被编译成为可加载的模块，一个模块可以由一个或者几个文件组成。当以一个文件组成时可以定义成如下形式：

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

（假设 CONFIG_ISDN_PPP_BSDCOMP= m）

当一个模块有几个文件组成时，需要通过变量$(<module_name>-objs)来指定模块文件的组成，如下例：

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN) += isdn.o
isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
```

（假设 CONFIG_ISDN = m）

最终会生成模块 isdn.o，此模块由 isdn_net_lib.o isdn_v110.o isdn_common.o 链接而成。

- 被编译到库文件的目标文件列表

所有包含在 lib-y 定义中的目标文件都将会被编译到该目录下一个统一的库文件中。值得注意的是 lib-y 定义一般被限制在 lib 和 arch/$(ARCH)/lib 目录中。

Example:

```
#arch/i386/lib/Makefile
lib-y       := checksum.o delay.o
```

3.2.2.2.2 目录递归

Makefile 文件负责当前目录下的目标文件，子目录中的文件由子目录中的 makefile 负责编译，编译系统使用 obj-y 和 obj-m 来自动递归编译各个子目录中的文件。

Example:

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
```

如果在内核配置文件.config 中，CONFIG_EXT2_FS 被设置为 y 或者 m，则内核 makefile 会自动进入 ext2 目录来进行编译。内核 Makefile 只使用这些信息来决定是否需要编译这个目录，子目录中的 makefile 规定哪些文件编译为模块，哪些文件编译进内核

3.2.2.2.3 编译选项

- EXTRA_CFLAGS　　$(CC) 编译 c 文件
  EXTRA_AFLAGS,　　$(CC) 编译汇编文件
  EXTRA_LDFLAGS,　　$(LD) 链接

EXTRA_ARFLAGS　　$(AR) 库的管理

顶级的 makefile 定义的变量$(CFLAGS)适用于整个目录树，而变量 EXTRA_ 对于当前目录定义的 makefile 的所有命令都有效。

Example:

#arch/x86_64/kernel/Makefile

EXTRA_AFLAGS := -traditional

- CFLAGS_$@, AFLAGS_$@

  针对某个命令追加编译选项。

  Example:

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =    -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o    = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
                   -DGDTH_STATISTICS
CFLAGS_seagate.o =    -DARBITRATE -DPARITY -DSEAGATE_USE_ASM
```

在编译 aha152x.o，gdth.o，seagate.o 会分别增加对应的编译选项。

3.2.2.2.4 依赖关系

Linux Makefile 通过在编译过程中生成的 .文件名.o.cmd（比如对于 main.c 文件，它对应的依赖文件名为.main.o.cmd）来定义相关的依赖关系。一般文件的依赖关系由如下部分组成：

- 所有的依赖文件（包括所有相关的*.c 和 *.h）。
- 所有与 CONFIG_选项相关的文件。
- 编译目标文件所使用到的命令行。

Example:

#init/.main.o.cmd 中的内容如下,从中可以看出命令行中的命令，以及参数都加入到了依赖文件：

```
cmd_init/main.o := arm-linux-gcc -Wp,-MD,init/.main.o.d -nostdinc -isystem /home/work/usr/local/arm/3.4.1/bin/../lib/gcc/arm-linux/3.4.1/include -D__KERNEL__ -Iinclude -include include/linux/autoconf.h -mlittle-endian -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing -fno-common -Os -fno-omit-frame-pointer -fno-optimize-sibling-calls -fno-omit-frame-pointer -mapcs -mno-sched-prolog -mapcs-32 -mno-thumb-interwork -D__LINUX_ARM_ARCH__=4 -march=armv4 -mtune=arm9tdmi -malignment-traps -msoft-float -Uarm -Wdeclaration-after-statement -D"KBUILD_STR(s)=\#s" -D"KBUILD_BASENAME=KBUILD_STR(main)" -D"KBUILD_MODNAME=KBUILD_STR(main)" -c -o init/main.o init/main.c


deps_init/main.o := \
  init/main.c \
    $(wildcard include/config/x86/local/apic.h) \
    $(wildcard include/config/acpi.h) \
    … …
    $(wildcard include/config/cpu/cache/vipt.h) \
    $(wildcard include/config/cpu/cache/vivt.h) \

init/main.o: $(deps_init/main.o)
```

$(deps_init/main.o):

3.2.2.2.5 特殊规则

当需要使用在内核编译规则没有定义的规则时，就需要特殊规则。典型的例子如编译过程中头文件的产生规则。其他例子有体系 makefile 编译引导映像的特殊规则。特殊规则写法同普通的 makefile 规则。因为在 linux2.6 make 体系下不会自动搜索 makefile 所在的目录下的文件，因此所有的特殊规则需要提供依赖文件和目标文件的相对路径。

定义特殊规则时将使用到两个变量：

$(src)： $(src)是对于 makefile 文件目录的相对路径，当使用与 makefile 同级目录下的文件时使用该变量$(src)。

$(obj)： $(obj)是目标文件目录的相对路径。当生成文件在 makefile 同级目录下，使用$(obj)变量。

Example:

```
#drivers/scsi/Makefile
$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl
    $(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl
```

3.2.2.3 arch/$(ARCH)/Makefile 分析

3.2.2.4 Makefile 中的变量

Makefile 中通过 export 把一些变量输出，传给子目录的 makefile，进而达到统一控制编译的目的。

- 版本信息：VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION，版本信息定义了当前内核的版本，比如 VERSION=2，PATCHLEVEL=6，SUBLEVEL=17，它们共同构成内核的发行版本 KERNELRELEASE：2.6.17
- CPU 体系结构：ARCH ，在顶层 Makefile 的开头，用 ARCH 定义目标 CPU 的体系结构，比如 ARCH:=arm 等。许多子目录的 Makefile 中，要根据 ARCH 的定义选择编译源文件的列表。
- INSTALL_PATH：定义生成的内核镜像和 System.map 的安装路径。
- INSTALL_MOD_PATH, MODLIB：这两个变量共同定义了模块生成后的安装路径，MODLIB= $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)，Makefile 不会定义变量 INSTALL_MOD_PATH 的值，用户可以在编译参数中传入变量的定义。
- 编译信息：CPP, CC, AS, LD, AR，CFLAGS，AFLAGS 等，在编译过程中，具体到特定的场合，需要明确给出编译环境，编译环境就是在以上的变量中定义的。针对交叉编译的要求，定义了 CROSS_COMPILE。比如：

```
CROSS_COMPILE    = arm-linux-
CC               = $(CROSS_COMPILE)gcc
LD               = $(CROSS_COMPILE)ld
.....
```

CROSS_COMPILE 定义了交叉编译器前缀 arm-linux-，表明所有的交叉编译工具都是以 arm-linux- 开头的，所以在各个交叉编译器工具之前，都加入了$(CROSS_COMPILE)，以组成一个完整的交叉编译工具文件名，比如 arm-linux-gcc。CFLAGS 定义了传递给 C 编译器的参数。

3.2.3 配置文件

除了 Makefile 的编写，另外一个重要的工作就是把新功能加入到 Linux 的配置选项中，提供此项功能的说明，让用户有机会选择此项功能。所有的这些都需要在 Kconfig 文件中用配置语言来编写配置脚本，在 Linux 内核中，配置命令有多种方式，以菜单界面配置（make menuconfig）为例，顶层 Makefile 调用 scripts/kconfig/makefile，按照 arch/arm/config.in 来进行配置。命令执行完后产生文件 .config，其中保存着配置信息。下一次再做 make config 将产生新的 .config 文件，原 .config 被改名为 .config.old

3.2.3.1 配置条目

配置文件 Kconfig 描述了一系列菜单配置条目，以下关键字定义条目

- config
- menuconfig
- choice/endchoice
- comment
- menu/endmenu
- if/endif
- source

3.2.4 用户代码添加实例

对于一个开发者来说，将自己开发的内核代码加入到 Linux 内核中，需要有三个步骤。首先确定把自己开发代码放入到内核的位置；其次，把自己开发的功能增加到 Linux 内核的配置选项中，使用户能够选择此功能；最后，构建子目录 Makefile，根据用户的选择，将相应的代码编译到最终生成的 Linux 内核中去。下面，我们就通过一个简单的例子，结合前面学到的知识，来说明如何向 Linux 内核中增加新的功能。

3.3 Linux2.6.17 启动流程分析

3.3.1 linux/arch/arm/kernel/head.S 分析

linux/arch/arm/kernel/head.S 用汇编代码完成，是内核最先执行的一个文件（严格的说从 u_boot 跳到内核最先执行的代码是 linux/arch/arm/boot/compressed/head.S，这段代码分析暂时省略，待日后分析）。这一段汇编代码的主要作用，是检查 cpu id，architecture number，初始化页表、cpu、bbs 等操作，并跳到 start_kernel 函数。它在执行前，处理器的状态应满足：

r0	- should be 0
r1	- unique architecture number
MMU	- off
I-cache	- on or off
D-cache	- off

在代码的分析过程中略去一些条件编译的代码。

```
    __INIT
    .type stext, %function
ENTRY(stext)            /* 内核入口点 */
    msr   cpsr_c, #PSR_F_BIT | PSR_I_BIT | MODE_SVC @ ensure svc mode
                                @ and irqs disabled   /* 置当前程序状态寄存器，关中断，svc 模式 */
    mrc   p15, 0, r9, c0, c0        @ get processor id    /* 读 processor id 到 r9 */
    bl    __lookup_processor_type     @ r5=procinfo r9=cupid /* 查找运行的 cpu 的 id 值，和此
```

```
                                                        linux 编译支持的 id 值是否有相等*/
        movs r10, r5                    @ invalid processor (r5=0)?
        beq   __error_p                 @ yes, error 'p'
        bl    __lookup_machine_type     @ r5=machinfo /*  判断体系类型是否支持  */
        movs r8, r5                     @ invalid machine (r5=0)?
        beq   __error_a                 @ yes, error 'a'
        bl    __create_page_tables                      /*  创建临时页表，供初始化使用  */
```

下面分被对于上述的三个函数进行分析：

在文件 linux/arch/arm/kernel/head-common.S

```
/*
 * Read processor ID register (CP#15, CR0), and look up in the linker-built
 * supported processor list.    Note that we can't use the absolute addresses
 * for the __proc_info lists since we aren't running with the MMU on
 * (and therefore, we are not in the correct address space).    We have to
 * calculate the offset.
 *
 *     r9 = cpuid
 * Returns:
 *     r3, r4, r6 corrupted
 *     r5 = proc_info pointer in physical address space
 *     r9 = cpuid (preserved)
 */
        .type __lookup_processor_type, %function
__lookup_processor_type:
        adr   r3, 3f                    /*  取标号 3 的地址  */
        ldmda       r3, {r5 - r7}       /* r5：__proc_info_begin；r6：__proc_info_end；r7：标号 3 的地址*/
        sub   r3, r3, r7                @ get offset between virt&phys /*  由于 adr   r3, 3f 取得标号 3 的物理地址，此时 r7 是标号 3 的虚拟地址，通过此指令获得 virt&phys 的差值，此时为负值  */
        add   r5, r5, r3                @ convert virt addresses to
        add   r6, r6, r3                @ physical address space
        /*  把标号__proc_info_begin，__proc_info_end 从虚拟地址转化为物理地址，由于此时 mmu 没有开启，所以必须以物理地址来进行访问内存  */
1:      ldmiar5, {r3, r4}               @ value, mask /*  从 proc_info_list 获得 cpu_val（r3）和 cpu_mask（r4  ）*/
        and   r4, r4, r9                @ mask wanted bits
        teq   r3, r4                    /* r9 为从 cp15 读出来的 cpuid，判断 linux 是否支持  */
        beq   2f                        /*  相等则退出，r5 为匹配 proc_info 的物理地址，
                                        否则在下一个 proc_info_list 中查找  */
        add   r5, r5, #PROC_INFO_SZ             @ sizeof(proc_info_list)
        cmp   r5, r6
        blo   1b
        mov   r5, #0                    @ unknown processor   /*  没有找到匹配的，返回 r5 = 0 */
2:      mov   pc, lr
```

```
/*
 * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
 * more information about the __proc_info and __arch_info structures.
 */
        .long __proc_info_begin
        .long __proc_info_end
3:      .long .
        .long __arch_info_begin
        .long __arch_info_end

/*
 * Lookup machine architecture in the linker-build list of architectures.
 * Note that we can't use the absolute addresses for the __arch_info
 * lists since we aren't running with the MMU on (and therefore, we are
 * not in the correct address space).   We have to calculate the offset.
 *
 *   r1 = machine architecture number
 * Returns:
 *   r3, r4, r6 corrupted
 *   r5 = mach_info pointer in physical address space
 */
        .type __lookup_machine_type, %function
__lookup_machine_type:          /* 此函数的实现同__lookup_processor_type，分析略 */
        adr   r3, 3b
        ldmiar3, {r4, r5, r6}
        sub   r3, r3, r4                @ get offset between virt&phys
        add   r5, r5, r3                @ convert virt addresses to
        add   r6, r6, r3                @ physical address space
1:      ldr   r3, [r5, #MACHINFO_TYPE]      @ get machine type
        teq   r3, r1                    @ matches loader number?
        beq   2f                        @ found
        add   r5, r5, #SIZEOF_MACHINE_DESC      @ next machine_desc
        cmp   r5, r6
        blo   1b
        mov   r5, #0                    @ unknown machine
2:      mov   pc, lr
```

标号__proc_info_begin __proc_info_end 由链接脚本 vmlinux.lds.S 定义，在进行链接时会把其中内容填充。

```
__proc_info_begin = .;
    *(.proc.info.init)
__proc_info_end = .;
```

本例中 r5=__arm920_proc_info 定义在 linux/arch/arm/mm/proc-arm920.S 中

__create_page_tables 函数在 linux/arch/arm/kernel/head.S 内容

```
__arch_info_begin = .;
```

```
        *(.arch.info.init)
    __arch_info_end = .;
```

在 linux/arch/arm/mach-s3c2410/mach-smdk2410.c 宏

```
MACHINE_START(SMDK2410, "SMDK2410")
    /* Maintainer: Jonas Dietsche */
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C2410_SDRAM_PA + 0x100,
    .map_io         = smdk2410_map_io,
    .init_irq   = s3c24xx_init_irq,
    .init_machine   = smdk_machine_init,
    .timer          = &s3c24xx_timer,
MACHINE_END
```

进行初始化。宏 SMDK2410 在 include/asm-arm/mach-types.h 中定义，此文件是根据文件 arch/arm/tools/mach-types 生成的。

```
/*
 * Setup the initial page tables.   We only setup the barest
 * amount which are required to get the kernel running, which
 * generally means mapping in the kernel code.
 *
 * r8   = machinfo
 * r9   = cpuid
 * r10 = procinfo
 *
 * Returns:
 *   r0, r3, r6, r7 corrupted
 *   r4 = physical page table address
 */
    .type __create_page_tables, %function
__create_page_tables:           /* 假设 ram 的物理起始地址为 0x30000000 */
    pgtbl r4                    @ page table address /* r4=0x30004000 页表在物理地址中基址 */

    /*
     * Clear the 16K level 1 swapper page table
     */
    mov  r0, r4
    mov  r3, #0
    add  r6, r0, #0x4000
1:  str  r3, [r0], #4       /* 多条同样的指令在多级流水线体系结构中效率会很高 */
    str  r3, [r0], #4
    str  r3, [r0], #4
    str  r3, [r0], #4
    teq  r0, r6
    bne  1b                 /* 把物理地址 0x30004000 至 0x30008000 清零 */
```

```
ldr     r7, [r10, #PROCINFO_MMUFLAGS]   @ mmuflags

/*
 * Create identity mapping for first MB of kernel to
 * cater for the MMU enable.  This identity mapping
 * will be removed by paging_init().  We use our current program
 * counter to determine corresponding section base address.
 */
mov  r6, pc, lsr #20            @ start of kernel section /* r6 = 0x300 */
orr     r3, r7, r6, lsl #20     @ flags + kernel base     /* r3 = 0x30000000 | muflag = 0x30000c1e */
str     r3, [r4, r6, lsl #2]    @ identity mapping        /* 0x30004c00  中的内容为 0x30000c1e   */
/* 这一部分把物理地址 0x30000000，映射到 0x30000000，也就是完全映射，保证了 mmu 是否开启，都可以正常取指运行 */
/*
 * Now setup the pagetables for our kernel direct
 * mapped region.  We round TEXTADDR down to the
 * nearest megabyte boundary.  It is assumed that
 * the kernel fits within 4 contigous 1MB sections.
 */
add   r0, r4,   #(TEXTADDR & 0xff000000) >> 18 @ start of kernel /* r0 = 0x30007000 */
str     r3, [r0, #(TEXTADDR & 0x00f00000) >> 18]!   /* 0x30007000  中的内容为 0x30000c1e   */
add   r3, r3, #1 << 20
str     r3, [r0, #4]!           @ KERNEL + 1MB      /* 0x30007004  中的内容为 0x30100c1e   */
add   r3, r3, #1 << 20
str     r3, [r0, #4]!           @ KERNEL + 2MB      /* 0x30007008  中的内容为 0x30200c1e   */
add   r3, r3, #1 << 20
str     r3, [r0, #4]            @ KERNEL + 3MB      /* 0x3000700c  中的内容为 0x30300c1e   */

/*
 * Then map first 1MB of ram in case it contains our boot params.
 */
add   r0, r4, #PAGE_OFFSET >> 18
orr     r6, r7, #PHYS_OFFSET
str     r6, [r0]      /* r0 = 0x30007000 ， r6 = 0x30000c1e */
mov  pc, lr
```

下面以从虚拟地址 0xc0005123 到物理地址 0x30005123 的转换为例来说明 mmu 和页表共同的工作过程。cp15 的寄存器 c2 存放页表的基地址 0x30004000，

0x30004000 & 0xffffc000 | ((0xc0005123 >> 18) & 0x3ffc)组成地址 0x30007000，以此地址作为一级描述符地址，根据上述映射过程可以得到 0x30007000 中的内容是 0x30000c1e，其中 c1e 说明是段地址，有读写权限（详见一级描述符格式说明），

(0x30000c1e & 0xfff00000) | (0xc0005123 & 0xfffff) 的到地址 0x3005123，此地址就是虚拟地址 0xc0005123 对应的物理地址。

```
/*
```

```
 * The following calls CPU specific code in a position independent
 * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
 * xxx_proc_info structure selected by __lookup_machine_type
 * above.  On return, the CPU will be ready for the MMU to be
 * turned on, and r0 will hold the CPU control register value.
 */
ldr     r13, __switch_data      @ address to jump to after
                                @ mmu has been enabled
adr     lr, __enable_mmu        @ return (PIC) address
add     pc, r10, #PROCINFO_INITFUNC  /*此命令执行__arm920_setup 进行 mmu 寄存器的初始化
                                       可以参照 cp15 寄存器手册来分析实现的功能，执行完
                                       成__arm920_setup 后，执行函数__enable_mmu */
/*
 * Setup common bits before finally enabling the MMU.  Essentially
 * this is just loading the page table pointer and domain access
 * registers.
 */
        .type   __enable_mmu, %function
__enable_mmu:
#ifdef CONFIG_ALIGNMENT_TRAP
        orr     r0, r0, #CR_A
#else
        bic     r0, r0, #CR_A
#endif
#ifdef CONFIG_CPU_DCACHE_DISABLE
        bic     r0, r0, #CR_C
#endif
#ifdef CONFIG_CPU_BPREDICT_DISABLE
        bic     r0, r0, #CR_Z
#endif
#ifdef CONFIG_CPU_ICACHE_DISABLE
        bic     r0, r0, #CR_I
#endif
        mov     r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | \
                      domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
                      domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | \
                      domain_val(DOMAIN_IO, DOMAIN_CLIENT))
        mcr     p15, 0, r5, c3, c0, 0           @ load domain access register
        mcr     p15, 0, r4, c2, c0, 0           @ load page table pointer
        b       __turn_mmu_on

        /*
         * Enable the MMU.  This completely changes the structure of the visible
         * memory space.  You will not be able to trace execution through this.
```

```
 * If you have an enquiry about this, *please* check the linux-arm-kernel
 * mailing list archives BEFORE sending another post to the list.
 *
 *   r0   = cp#15 control register
 *   r13 = *virtual* address to jump to upon completion
 *
 * other registers depend on the function called upon completion
 */
        .align5
        .type   __turn_mmu_on, %function
__turn_mmu_on:
        mov     r0, r0
        mcr     p15, 0, r0, c1, c0, 0           @ write control reg
        mrc     p15, 0, r3, c0, c0, 0           @ read id reg
        mov     r3, r3                  /* 开启 mmu 清空流水线指令*/
        mov     r3, r3
        mov     pc, r13                 /* 对于以上针对 mmu 的操作可以参考 cp15 使用手册，分析略，
                                        当前指令会使程序跳转到__switch_data 处执行，此时标号
                                __switch_data 是虚拟地址，至此，内存访问操作进入虚拟地址空间*/
```

linux/arch/arm/kernel/head-common.S

```
        .type   __switch_data, %object
__switch_data:
        .long   __mmap_switched
        .long   __data_loc              @ r4
        .long   __data_start            @ r5
        .long   __bss_start             @ r6
        .long   _end                    @ r7
        .long   processor_id            @ r4
        .long   __machine_arch_type     @ r5
        .long   cr_alignment            @ r6
        .long   init_thread_union + THREAD_START_SP @ sp

/*
 * The following fragment of code is executed with the MMU on in MMU mode,
 * and uses absolute addresses; this is not position independent.
 *
 *   r0   = cp#15 control register
 *   r1   = machine ID
 *   r9   = processor ID
 */
        .type   __mmap_switched, %function
__mmap_switched:
        adr     r3, __switch_data + 4   /* r3 = __data_loc */
```

```
        ldmiar3!, {r4, r5, r6, r7}      /* r4=__data_loc,r5=__data_start,r6=__bss_start,r7=_end */
        cmp   r4, r5                    @ Copy data segment if needed
1:      cmpne       r5, r6
        ldrne fp, [r4], #4
        strne  fp, [r5], #4
        bne    1b

        mov   fp, #0                    @ Clear BSS (and zero fp)
1:      cmp   r6, r7
        strcc  fp, [r6],#4
        bcc    1b

        ldmiar3, {r4, r5, r6, sp}
        str     r9, [r4]          @ Save processor ID /* processorID 保存到 processor_id */
        str     r1, [r5]          @ Save machine type /*arch type  保存到 machine_arch_type */
        bic     r4, r0, #CR_A     @ Clear 'A' bit          /*把禁止和使能地址对齐检查的*/
        stmia r6, {r0, r4}        @ Save control register values /*控制寄存器分别放入变量  */
                                          /* cr_alignment 和 cr_no_alignment  中*/
        b       start_kernel                      /*  进入 c 代码  */
```

3.3.2 c 代码启动流程分析（linux2.4.20）

3.3.2.1 基本的核心环境的建立--内核引导第一部分

start_kernel()中调用了一系列初始化函数，以完成 kernel 本身的设置。这些初始化有的是公共的，有的则是需要配置的才会执行的。

- 输出 Linux 版本信息（printk(linux_banner)）
- 设置与体系结构相关的环境（setup_arch()）页表结构初始化（paging_init()）
- 提取并分析核心启动参数（从环境变量中读取参数，设置相应标志位等待处理，（parse_options()）
- 设置系统自陷入口（trap_init()）
- 初始化系统中断 IRQ（init_IRQ()）
- 核心进程调度器初始化（包括初始化几个缺省的 Bottom-half，sched_init()）
- 软中断初始化（softirq_init()）
- 时间、定时器初始化（包括读取 CMOS 时钟、估测主频、初始化定时器中断等，time_init()）
- 控制台初始化（为输出信息而先于 PCI 初始化，console_init()）
- 模块初始化（init_modules()）
- 剖析器数据结构初始化（prof_buffer 和 prof_len 变量）
- 核心 Cache 初始化（描述 Cache 信息的 Cache，kmem_cache_init()）
- 延迟校准（获得时钟 jiffies 与 CPU 主频 ticks 的延迟，calibrate_delay()）
- 内存初始化（设置内存上下界和页表项初始值，mem_init()）
- 创建和设置内部及通用 cache（"slab_cache"，kmem_cache_sizes_init()）
- 相关 cache 初始化（）
- SMP 机器其余 CPU（除当前引导 CPU）初始化（对于没有配置 SMP 的内核，此函数为空，smp_init()）
- 启动 init 过程（创建第一个核心线程，调用 init()函数，原执行序列调用 cpu_idle() 等

待调度，init()）

至此 start_kernel()结束，基本的核心环境已经建立起来了

3.3.2.2 外设初始化--内核引导第二部分

init()函数作为核心线程，首先锁定内核（仅对 SMP 机器有效），然后调用 do_basic_setup()完成外设及其驱动程序的加载和初始化。过程如下：

- 网络初始化（sock_init()）
- 创建事件管理核心线程初始化（start_context_thread()）
- 其他所需要的初始化都使用__initcall()宏包含，在 do_initcalls()函数中启动执行。

3.3.3 最小系统启动需要关注的初始化部分

控制台初始化（console_init()）

3.4 Linux2.6.11 内核移植

4 第四章嵌入式设备的文件系统

4.1 ramdisk 文件系统

本文以 initrd 为例来介绍 linux 关于 ramdisk 的相关问题。Linux 初始 RAM 磁盘（initrd）是在系统引导过程中挂载的一个临时根文件系统，initrd 与内核绑定在一起，并作为内核引导过程的一部分进行加载。用来支持两阶段的引导过程。initrd 文件中包含了各种可执行程序和驱动程序，它们可以用来挂载实际的根文件系统，然后再将这个 initrd RAM 磁盘卸载，并释放内存。在很多嵌入式 Linux 系统中，initrd 就是最终的根文件系统。

4.1.1 initrd 内容剖析及制作

Linux2.4 内核的 initrd 的格式是文件系统镜像文件，本文将其称为 image-initrd，以区别后面介绍的 linux2.6 内核的 cpio 格式的 initrd。

4.1.1.1 Linux2.4 内核的 initrd

4.1.1.1.1 察看 initrd 的内容

[root@localhost mnt]# cp /boot/initrd-2.4.18-14.img . 拷贝 initrd 到当前目录

[root@localhost mnt]# mv initrd-2.4.18-14.img initrd-2.4.18-14.img.gz

由于 initrd-2.4.18-14.img 是压缩文件，要使用 gunzip 必须加后缀.gz

[root@localhost mnt]# gunzip initrd-2.4.18-14.img.gz 解压缩到当前文件夹

[root@localhost mnt]# mount -t ext2 -o loop initrd-2.4.18-14.img ./mnt

Mount 到一个目录下

[root@localhost mnt]# ls ./mnt/ 察看内容

bin dev etc lib linuxrc loopfs proc sbin sysroot

4.1.1.1.2 创建定制 initrd 的 shell 脚本

```
#!/bin/bash

# Housekeeping...
rm -f /tmp/ramdisk.img
rm -f /tmp/ramdisk.img.gz

# Ramdisk Constants
RDSIZE=4000
BLKSIZE=1024

# Create an empty ramdisk image
dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE

# Make it an ext2 mountable file system
/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE

# Mount it so that we can populate
mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0

# Populate the filesystem (subdirectories)
```

```
mkdir /mnt/initrd/bin
mkdir /mnt/initrd/sys
mkdir /mnt/initrd/dev
mkdir /mnt/initrd/proc

# Grab busybox and create the symbolic links
pushd /mnt/initrd/bin
cp /usr/local/src/busybox-1.1.1/busybox .
ln -s busybox ash
ln -s busybox mount
ln -s busybox echo
ln -s busybox ls
ln -s busybox cat
ln -s busybox ps
ln -s busybox dmesg
ln -s busybox sysctl
popd

# Grab the necessary dev files
cp -a /dev/console /mnt/initrd/dev
cp -a /dev/ramdisk /mnt/initrd/dev
cp -a /dev/ram0 /mnt/initrd/dev
cp -a /dev/null /mnt/initrd/dev
cp -a /dev/tty1 /mnt/initrd/dev
cp -a /dev/tty2 /mnt/initrd/dev

# Equate sbin with bin
pushd /mnt/initrd
ln -s bin sbin
popd

# Create the init file
cat >> /mnt/initrd/linuxrc << EOF
#!/bin/ash
echo
echo "Simple initrd is active"
echo
mount -t proc /proc /proc
mount -t sysfs none /sys
/bin/ash --login
EOF

chmod +x /mnt/initrd/linuxrc
```

```
# Finish up...
umount /mnt/initrd
gzip -9 /tmp/ramdisk.img
```

为了创建 initrd，我们最开始创建了一个空文件，这使用了 /dev/zero（一个由零组成的码流）作为输入，并将其写入到 ramdisk.img 文件中。所生成的文件大小是 4MB（4000 个 1K 大小的块）。然后使用 mke2fs 命令在这个空文件上创建了一个 ext2（即 second extended）文件系统。现在这个文件变成了一个 ext2 格式的文件系统，我们使用 loop 设备将这个文件挂载到 /mnt/initrd 上了。在这个挂载点上，我们现在就有了一个目录，它以 ext2 文件系统的形式呈现出来，我们可以对自己的 initrd 文件进行拼装了。接下来的脚本提供了这种功能。

下一个步骤是创建构成根文件系统所需要的子目录：/bin、/sys、/dev 和 /proc。这里只列出了所需要的目录（例如没有库），但是其中包含了很多功能。

为了可以使用根文件系统，我们使用了 BusyBox。这个工具是一个单一映像，其中包含了很多在 Linux 系统上通常可以找到的工具（例如 ash、awk、sed、insmod 等）。BusyBox 的优点是它将很多工具打包成一个文件，同时还可以共享它们的通用元素，这样可以极大地减少映像文件的大小。这对于嵌入式系统来说非常理想。将 BusyBox 映像从自己的源目录中拷贝到自己根目录下的 /bin 目录中。然后创建了很多符号链接，它们都指向 BusyBox 工具。BusyBox 会判断所调用的是哪个工具，并执行这个工具的功能。我们在这个目录中创建了几个链接来支持 init 脚本（每个命令都是一个指向 BusyBox 的链接。）

下一个步骤是创建几个特殊的设备文件。我从自己当前的 /dev 子目录中直接拷贝了这些文件，这使用了 -a 选项（归档）来保留它们的属性。

倒数第二个步骤是生成 linuxrc 文件。在内核挂载 RAM 磁盘之后，它会查找 init 文件来执行。如果没有找到 init 文件，内核就会调用 linuxrc 文件作为自己的启动脚本。我们在这个文件中实现对环境的基本设置，例如挂载 /proc 文件系统。除了 /proc 之外，我还挂载了 /sys 文件系统，并向终端打印一条消息。最后，我们调用了 ash（一个 Bourne Shell 的克隆），这样就可以与根文件系统进行交互了。linuxrc 文件然后使用 chmod 命令修改成可执行的。

最后，我们的根文件系统就完成了。我们将其卸载掉，然后使用 gzip 对其进行压缩。

4.1.1.1.3 Linux 对 Initrd 的处理流程

Linux2.4 内核的 initrd 的格式是文件系统镜像文件，linux2.4 内核对 initrd 的处理流程如下：

1. boot loader 把内核以及/dev/initrd 的内容加载到内存，/dev/initrd 是由 boot loader 初始化的设备，存储着 initrd。

2. 在内核初始化过程中，内核把 /dev/initrd 设备的内容解压缩并拷贝到 /dev/ram0 设备上。

3. 内核以可读写的方式把 /dev/ram0 设备挂载为原始的根文件系统。

4. 如果 /dev/ram0 被指定为真正的根文件系统，那么内核跳至最后一步正常启动。

5. 执行 initrd 上的 /linuxrc 文件，linuxrc 通常是一个脚本文件，负责加载内核访问根文件系统必须的驱动， 以及加载根文件系统。

6. /linuxrc 执行完毕，真正的根文件系统被挂载。

7. 如果真正的根文件系统存在 /initrd 目录，那么 /dev/ram0 将从 / 移动到 /initrd。否则如果 /initrd 目录不存在， /dev/ram0 将被卸载。

8. 在真正的根文件系统上进行正常启动过程 ，执行 /sbin/init。 linux2.4 内核的

initrd 的执行是作为内核启动的一个中间阶段，也就是说 initrd 的 /linuxrc 执行以后，内核会继续执行初始化代码，这是 linux2.4 内核同 2.6 内核的 initrd 处理流程的一个显著区别。

4.1.1.2 Linux2.6 内核的 initrd

4.1.1.2.1 察看 initrd 的内容

```
[root@localhost temp]# gunzip initrd.img.gz
[root@localhost temp]# cpio -i --make-directories < initrd.img
[root@localhost temp]# ls
bin  bootsplash  dev  etc  init  lib  proc  root  sbin  sys  tmp
```

4.1.1.2.2 创建 initrd

cpio-initrd 的制作非常简单，通过两个命令就可以完成整个制作过程，假设当前目录位于准备好的 initrd 文件系统的根目录下

```
[root@localhost temp]# find . | cpio -c -o > ../initrd-2.6.17.img
[root@localhost temp]# gzip -9 initrd-2.6.17.img
```

4.1.1.2.3 Linux 对 Initrd 的处理流程

linux2.6 内核支持两种格式的 initrd，一种是介绍的 linux2.4 内核那种传统格式的文件系统镜像－image-initrd，它的制作方法同 Linux2.4 内核的 initrd 一样，其核心文件就是 /linuxrc。另外一种格式的 initrd 是 cpio 格式的，这种格式的 initrd 从 linux2.5 起开始引入，使用 cpio 工具生成，其核心文件不再是 /linuxrc，而是 /init，本文将这种 initrd 称为 cpio-initrd。尽管 linux2.6 内核对 cpio-initrd 和 image-initrd 这两种格式的 initrd 均支持，但对其处理流程有着显著的区别，下面分别介绍 linux2.6 内核对这两种 initrd 的处理流程。

- cpio-initrd 的处理流程

1. boot loader 把内核以及 initrd 文件加载到内存的特定位置。
2. 内核判断 initrd 的文件格式，如果是 cpio 格式。
3. 将 initrd 的内容释放到 rootfs 中。
4. 执行 initrd 中的/init 文件，执行到这一点，内核的工作全部结束，完全交给/init 文件处理。

- image-initrd 的处理流程

1. boot loader 把内核以及 initrd 文件加载到内存的特定位置。
2. 内核判断 initrd 的文件格式，如果不是 cpio 格式，将其作为 image-initrd 处理。
3. 内核将 initrd 的内容保存在 rootfs 下的/initrd.image 文件中。
4. 内核将/initrd.image 的内容读入/dev/ram0 设备中，也就是读入了一个内存盘中。
5. 接着内核以可读写的方式把/dev/ram0 设备挂载为原始的根文件系统。
6. .如果/dev/ram0 被指定为真正的根文件系统，那么内核跳至最后一步正常启动。
7. 执行 initrd 上的/linuxrc 文件，linuxrc 通常是一个脚本文件，负责加载内核访问根文件系统必须的驱动， 以及加载根文件系统。
8. /linuxrc 执行完毕，常规根文件系统被挂载。
9. 如果常规根文件系统存在/initrd 目录，那么/dev/ram0 将从/移动到/initrd。否则如果/initrd 目录不存在， /dev/ram0 将被卸载。
10. 在常规根文件系统上进行正常启动过程 ，执行/sbin/init。

通过上面的流程介绍可知，Linux2.6 内核对 image-initrd 的处理流程同 linux2.4 内核相比并没有显著的变化， cpio-initrd 的处理流程相比于 image-initrd 的处理流程却有很大的区

别，流程非常简单，cpio-initrd 并没有使用额外的 ramdisk,而是将其内容输入到 rootfs 中，其实 rootfs 本身也是一个基于内存的文件系统。这样就省掉了 ramdisk 的挂载、卸载等步骤。cpio-initrd 启动完/init 进程，内核的任务就结束了，剩下的工作完全交给/init 处理；而对于 image-initrd，内核在执行完/linuxrc 进程后，还要进行一些收尾工作，并且要负责执行真正的根文件系统的/sbin/init。

4.1.1.3 u-boot 启动参数如何设置 ramdisk 作为文件系统

当使用 u-boot 启动内核时，如果 ramdisk 作为根文件系统设置参数如下：

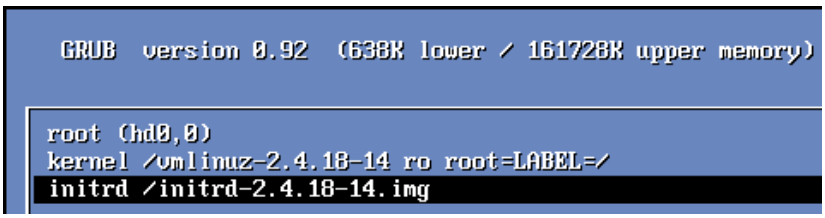
setenv bootargs root=/dev/ram rw;

tftp 1000000 zImage;

tftp 2000000 ramdisk.image;

bootm 1000000

4.1.1.4 grub 关于 initrd 参数的设置



以上是 grub 的截图，其中 initrd 的参数/initrd-2.4.18-14.img 就是镜像，通过此参数可以测试用户定制的 initrd。

4.1.1.5 嵌入式系统 RamDisk 综述

Ram：内存，Disk：磁盘，在 Linux 中可以将一部分内存当作分区来使用，称之为 RamDisk。对于一些经常被访问、并且不会被更改的文件，可以将它们通过 RamDisk 放在内存中，能够明显地提高系统性能。RamDisk 工作于虚拟文件系统（VFS）层，不能格式化，但可以创建多个 RamDisk。虽然现在硬盘价钱越来越便宜，但对于一些我们想让其访问速度很高的情况下，RamDisk 还是很好用的。

如果对计算速度要求很高，可以通过增加内存来实现，使用 ramdisk 技术。一个 RamDisk 就是把内存假设为一个硬盘驱动器，并且在它的上面存储文件。假设有几个文件要频繁的使用，如果将它们加到内存当中，程序运行速度会大幅度提高，因为内存的读写速度远高于硬盘。划出部分内存提高整体性能，不亚于更换新的 CPU。像 Web 服务器这样的计算机，需要大量读取和交换特定的文件。因此，在 Web 服务器上建立 RamDisk 会大大提高网络读取速度。

RamDisk 就是将内存模拟为硬盘空间。无论什么时候你使用 RamDisk，实际上你是在使用内存而不是硬盘。在这一点上既有优点又有缺点。最基本的，最大的优点是你是在使用内存，你所做的一切都会快一些，因为硬盘的速度较内存慢。最大的缺点是如果你改变了数据库服务器的内容并且重新启动机器时，所做的一切改动都将丢失。

4.2 日志闪存文件系统（JFFS2）

4.3 NFS 文件系统

4.4 第二版扩展文件系统（Ext2fs）

5 第五章 busyBox 移植与应用

BusyBox 是为构建内存有限的嵌入式系统的一个优秀工具。BusyBox 通过将很多必需的工具放入一个可执行程序，并让它们可以共享代码中相同的部分，从而对它们的大小进行了很大程度的缩减，BusyBox 对于嵌入式系统来说是一个非常有用的工具，因此值得我们花一些时间进行探索。

5.1 Busybox 获取

http://www.busybox.net/downloads/可以下载到最新的代码。

以下以 busybox-1.1.3 为例进行说明

$ tar xvfz busybox-1.1.1.tar.gz

$ cd busybox-1.1.1

5.2 Busybox 的编译与配置

5.2.1 BusyBox 编译选项

BusyBox 包括了几个编译选项，可以帮助为我们编译和调试正确的 BusyBox。

目标	说明
help	显示 make 选项的完整列表
defconfig	启用默认的（通用）配置
allnoconfig	禁用所有的应用程序（空配置）
allyesconfig	启用所有的应用程序（完整配置）
allbareconfig	启用所有的应用程序，但是不包括子特性
config	基于文本的配置工具
menuconfig	N-curses（基于菜单的）配置工具
all	编译 BusyBox 二进制文件和文档（./docs）
busybox	编译 BusyBox 二进制文件
clean	清除源代码树
distclean	彻底清除源代码树
sizes	显示所启用的应用程序的文本/数据大小

在定义配置时，我们只需要输入 make 就可以真正编译 BusyBox 二进制文件。例如，要为所有的应用程序编译 BusyBox，我们可以执行下面的命令：

$ make allyesconfig

$ make

5.2.2 定制与编译 busybox

```
$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?
        (/home/arm/3.3.2/bin/armlinux)
        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
        */
$make TARGET_ARCH=arm CROSS=armlinux-  all install
# ls _install/
bin  lib  linuxrc  sbin  usr  此目录就是默认的安装目录。
```

5.3 如何向 Busybox 添加命令

将这个新命令称为 newcmd，并将它放到了 ./miscutils 目录中。这个新命令的源代码如下：

```
#include "busybox.h"
int newcmd_main( int argc, char *argv[] )
```

```
{
  int i;
  printf("newcmd called:\n");
  for (i = 0 ; i < argc ; i++) {
    printf("arg[%d] = %s\n", i, argv[i]);
  }
  return 0;
}
```

- 第一个步骤是为新命令的源代码选择一个位置。我们要根据命令的类型（网络，shell 等）来选择位置，并与其他命令保持一致。这一点非常重要，因为这个新命令最终会在 menuconfig 的配置菜单中出现（在这个例子中，是 Miscellaneous Utilities 菜单）。
- 修改 makefile，我们要将这个新命令的源代码添加到所选子目录中的 Makefile.in 中。在本例中，我更新了 ./miscutils/Makefile.in 文件。按照字母顺序来添加新命令，以便维持与现有命令的一致性：

  将命令添加到 Makefile.in 中

  ```
  MISCUTILS-$(CONFIG_MT)              += mt.o
  MISCUTILS-$(CONFIG_NEWCMD)    += newcmd.o
  MISCUTILS-$(CONFIG_RUNLEVEL)      += runlevel.o
  ```
- 修改 config.in，更新 ./miscutils 目录中的配置文件，以便让新命令在配置过程中是可见的。这个文件名为 Config.in，新命令是按照字母顺序添加的：

  将命令添加到 Config.in 中

  ```
  config CONFIG_NEWCMD
      bool "newcmd"
      default n
      help
        newcmd is a new test command.
  ```

这个结构定义了一个新配置项（通过 config 关键字）以及一个配置选项（CONFIG_NEWCMD）。新命令可以启用，也可以禁用，因此我们对配置的菜单属性使用了 bool （Boolean）值。这个命令默认是禁用的（n 表示 No），我们可以最后放上一个简短的 Help 描述。在源代码树的 ./scripts/config/Kconfig-language.txt 文件中，我们可以看到配置语法的完整文法。

- 更新 ./include/applets.h 文件，使其包含这个新命令。将下面这行内容添加到这个文件中，记住要按照字母顺序。维护这个次序非常重要，否则我们的命令就会找不到。

将命令添加到 applets.h 中

```
USE_NEWCMD(APPLET(newcmd, newcmd_main, _BB_DIR_USER_BIN, _BB_SUID_NEVER))
```

这定义了命令名（newcmd），它在 Busybox 源代码中的函数名（newcmd_main），应该在哪里会为这个新命令创建链接（在这种情况中，它在 /usr/bin 目录中），最后这个命令是否有权设置用户 id（在本例中是 no）。

- 向 ./include/usage.h 文件中添加详细的帮助信息。正如您可以从这个文件的例子中看到的一样，使用信息可能非常详细。在本例中，只添加了一点信息，这样就可以编译这个新命令了：

  向 usage.h 添加帮助信息

```
#define newcmd_trivial_usage      "None"
#define newcmd_full_usage "None"
```

- 最后一个步骤是启用新命令（通过 make menuconfig，然后在 Miscellaneous Utilities 菜单中启用这个选项）然后使用 make 来编译 BusyBox。

  使用新的 BusyBox，我们可以对这个新命令进行测试，如

```
$ ./busybox newcmd arg1
newcmd called:
arg[0] = newcmd
arg[1] = arg1
$ ./busybox newcmd --help
BusyBox v1.1.1 (2006.04.12-13:47+0000) multi-call binary

Usage: newcmd None

None
```

5.4 采用 busybox 建立根文件系统

参照，创建定制 initrd 的 shell 脚本，一节

第三部分设备驱动程序开发

6　第 6 章 linux 字符设备驱动程序