

前言

本文的研究目的是在以 S3C2410A（ARM920T 核）芯片为核心的硬件平台上，进行嵌入式 Linux 2.6 系统内核的移植，以方便日后在此基础上进行二次开发。本文主要介绍了此嵌入式系统的硬件架构，Linux 2.6 内核的新特性；详细分析了作为操作系统的核心，运行级别最高的内核结构；分析了 Linux 内核的启动流程，移植要点；下载、配置、编译了交叉编译工具链，建立了嵌入式 Linux 交叉开发环境；详细分析了 BootLoader 的启动过程，以及从 NOR Flash 启动的原理，在此基础上对通用的 BootLoader 程序 u-boot 进行了移植研究并实现；实现了 Linux 2.6 内核的编译移植工作；并移植了一个 yaffs 文件系统。最后在选用的开发板上成功加载了 Linux 2.6 内核。

关键词：Linux 内核移植；S3C2410；BootLoader；u-boot；yaffs 文件系统

目录

前言.....	1
1 绪论.....	3
1.1 课题研究的背景、目的和意义.....	3
1.2 嵌入式系统现状及发展趋势.....	3
1.3 论文的主要工作.....	4
2 嵌入式 Linux 系统架构与移植环境搭建.....	6
2.1 嵌入式 Linux 系统的体系结构.....	6
2.2 嵌入式 Linux 系统硬件平台.....	6
2.3 移植环境搭建.....	8
2.4 本章小结.....	14
3 嵌入式 Linux 的 BootLoader 移植.....	15
3.1 BootLoader 概述.....	15
3.2 u-boot 移植分析.....	17
3.3 本章小结.....	26
4 Linux 内核的编译、移植.....	27
4.1 Linux2.6 内核的特性简介.....	27
4.2 Linux 内核的启动流程分析.....	27
4.3 内核移植的实现.....	30
4.4 本章小结.....	40
5 文件系统制作.....	41
5.1 用 Busybox 制作 yaffs2 根文件系统.....	41
5.2 移植过程中遇到的问题及处理.....	46
5.3 本章小结.....	47
6 测试.....	48
6.1 常用简单测试方法介绍.....	48
6.2 编写简单的 C 程序测试移植的系统.....	48
6.3 在开发板执行测试程序.....	49
7 结论.....	50
8 参考文献.....	51

1 绪论

1.1 课题研究的背景、目的和意义

嵌入式系统是当前热门的计算机应用领域之一，广泛应用于工业控制系统、仿真系统、医疗仪器、信息家电、同学设备等众多领域中。市场对嵌入式产品的巨大需求也证明了这个研究领域的活力，同时也对嵌入式系统开发技术，特别是软件设计技术提出了新的挑战，主要包括：支持日趋增长的功能密度、灵活的网络连接、轻便的移动应用和多媒体信息处理。

本课题研究的主要目的是：在以 S3C2410A 微处理器为核心的硬件平台上移植通用的嵌入式 Linux 2.6 内核的软件开发平台，以支持应用程序的开发。

研究嵌入式 Linux 系统的移植一方面可以增加对微处理器、内置外设以及系统接口的了解，熟悉嵌入式操作系统架构，另一方面也可以对内核的设备驱动模块及其实现有更深入的理解。最重要的是可以在实践的基础上增加对系统移植理论的理解并积累丰富的系统移植经验，为后续的嵌入式开发打下坚实的基础。

1.2 嵌入式系统现状及发展趋势

计算机与互联网技术的应用与普及，以及微电子技术的突破，正有力地推动着二十一世纪工业生产、商业活动、科学实验和家庭生活等领域的自动化和信息化进程。全自动化的产品制造、大规模的电子商务活动、高度协同的科学实验以及现代化智能家居等，为嵌入式产品造就了全新而巨大的商机，拥有广阔的市场前景。目前全球的工业产值也在逐年增加。

1.2.1 嵌入式系统的特点

嵌入式系统和通用型计算机系统相比具有以下特点：

1. 面向特定应用

嵌入式 CPU 大多工作在特定设计的系统中，具有低功耗、体积小、高集成等特点，大部分常用功能集成在芯片内部，设备日趋小型化，与网络的耦合也越来越紧密。

2. 对软件有严格要求

嵌入式软件一般固化在容量较小的 Flash 存储器中，这就要求软件代码具有较高质量和可靠性，故需对代码进行裁剪和调整。

3. 必须具备相关平台的开发环境和开发工具

嵌入式系统自身不具备开发能力，需与宿主机相连构成交叉开发环境，需要配置交叉编译工具链。

4. 生命周期长

当嵌入式系统应用到产品后，还可以进行空中升级，以适应需求的变更。

1.2.2 常见嵌入式处理器简介

目前全世界嵌入式处理器的品种总量已经超过 1000 多种，流行体系结构包括 MCU、MPU 等 30 几个系列，速度越来越快，性能越来越强，价格亦越来越低。市面上常用嵌入式处理器可分成以下几类：

1. 嵌入式微处理器（Embedded Microprocessor Unit, EMP）
2. 嵌入式微控制器（Microcontroller Unit, MCU）
3. 嵌入式 DSP 处理器（Embedded Digital Signal Processor）
4. 嵌入式片上系统（System on Chip, SOC）

1.2.3 嵌入式 Linux 2.6 操作系统介绍

Linux 操作系统是一种性能优良、开源、应用广泛的免费操作系统，因其体积小、可裁剪、运行速度快、网络性能优良等特点，适合作为嵌入式操作系统。嵌入式开发中操作系统并不是必需的，因为程序完全可以在裸板上运行。但对复杂的系统，为使其具有任务管理、定时器管理、存储器管理、资源管理、事件管理、消息管理、队列管理和中断处理的能力，提供多任务处理，更好地分配系统资源的功能，有必要对特定的硬件平台和实际应用移植操作系统。

嵌入式系统一般要求实施可靠性，尽管 Linux 2.6 并不是一个真正的实时操作系统，但其改进的响应特性已经能够满足大多数情况下的响应时间要求。Linux 2.6 有一个最显著的改进就是采用可抢占内核，提高了中断性能，采用更加有效的调度算法以及同步性的提高。此外 Linux 2.6 添加了对新的体系结构和处理器类型的支持（包括对没有 MMU 系统的支持），可支持大容量内存模型，同时改善了 I/O 子系统，增加了更多的多媒体应用。

1.3 论文的主要工作

本工程实践项目以 S3C2410 为核心的 ARM9 开发板为硬件开发平台，实现了 Linux 2.6 内核的移植，包括 u-boot 和 yaffs 文件系统的移植工作。主要工作如下：

1. 建立嵌入式 Linux 交叉开发环境

所谓搭建交叉编译环境，即安装、配置交叉编译工具链。在该环境下编译出嵌入式 Linux 系统所需的操作系统、应用程序等，然后再上传到目标机上。在此我使用的是 arm-linux-gcc。

2. 移植 BootLoader 引导启动程序

BootLoader 的工作是为 Linux 内核准备好合适的工作环境并加载内核，引导

内核的启动。BootLoader 在目标板上电的时候运行，主要完成板级初始化和 Linux 内核引导的任务。由于 BootLoader 和 CPU 及电路板的配置有关，开发时需根据具体情况进行移植。在 x86 架构下，有 GRUB 和 LILO 两款被广泛认可的启动加载器，arm 下的通用 BootLoader 有德国 DENX 小组开发的 u-boot，本文就是通过 u-boot 来引导的。

3. 配置、编译、移植 Linux 内核

详细分析了 Linux 移植的全流程和要点，并对目前较稳定的 2.6 版的 Linux 内核源码进行了移植、配置和编译。

4. 制作文件系统并对文件系统进行移植

Yaffs (yet another flash file system) 文件系统是专门为 NAND Flash 设计的文件系统，目前有 yaffs1 和 yaffs2 两个版本，yaffs1 对小页面 (512B+16B/页) 有很好的支持，yaffs2 是对 yaffs1 的扩展，支持更大页面 (2KB+64B/页) 的 NAND Flash 设备。Yaffs 基于日志结构，具备极高写性能，并且在异地更新时，保证当前数据和过期数据的顺序。Yaffs 是在当今嵌入式领域中 NAND Flash 文件系统的最优解决方案之一。

2 嵌入式 Linux 系统架构与移植环境搭建

对嵌入式操作系统以及驱动程序的移植，需要对目标硬件平台和软件结构有深入的理解，本章介绍嵌入式 Linux 系统的体系结构、硬件平台构成和相关软件开发环境。

2.1 嵌入式 Linux 系统的体系结构

除硬件系统外，嵌入式 linux 系统需要有如下三个基本元素：

1. 系统引导程序 BootLoader（用于设备加电后系统定位引导）
2. Linux 微内核（内存管理、进程管理、中断处理等）
3. 初始化进程

除此之外要使其成为一个完整的操作系统并保持小型化还必须加上硬件驱动程序、硬件接口程序和应用程序组。最终一个完善可用的嵌入式 Linux 系统体系结构如下表所示：

表 2-1 嵌入式 Linux 系统体系结构

应用软件	应用层
嵌入式 GUI 图形界面支持	支持层
BootLoader、Linux kernel、Drivers	系统层
嵌入式开发板（本项目使用三星的 S3C2410）	硬件层

硬件层是系统的基础，所有软件都建立在它的基础上，系统层的 BootLoader 是嵌入式系统软件的最底层，是上电后运行的第一个程序，类似于 PC 机上的 BIOS，完成对硬件的初始化和内核加载，驱动程序作为系统内核的一部分，实现操作系统内核和硬件设备之间的接口，为应用程序屏蔽硬件的细节，系统内核主要完成任务管理，调度算法等，GUI 图形支持库实现对硬件的抽象、提供基本的图形接口函数和与用户实现交互，而应用软件用来实现某一具体功能。

2.2 嵌入式 Linux 系统硬件平台

2.2.1 S3C2410A 处理器简介^{[1][6]}

三星公司推出的 16/32 位 RISC 处理器 S3C2410A，为手持设备和一般应用提供了低价格、低功耗、高性能的小型微控制器解决方案。为了降低整个系统的成本，S3C2410A 提供了以下的内部设备：

- 分离的 16KB 指令 Cache 和 16KB 数据 Cache，具有虚拟存储器管理单元 MMU，LCD 控制器(支持 STN&TFT)，8 通道 10 位 ADC 和触摸屏接口，支持 NAND Flash 系统引导，3 通道 UART，4 通道 DMA，4 通道 PWM 定时器，I/O 端口，RTC,IIS-BUS 接口，USB 主机，USB 设备，SD 主卡&MMC 卡接口，2 通道的 SPI 以及内部 PLL 时钟倍频器等等。
- 存储控制器

S3C2410A 以处理器内部集成了存储控制器，它可以为片外存储器访问提供必要的控制信号，它主要包括以下特点：

 1. 支持大、小端模式(通过软件选择)；
 2. 地址空间：包含 8 个地址空间，每个地址空间的大小为 128M 字节，总共有 1GB 的地址空间；
 3. 除 BANK0 以外的所有地址空间都可以通过编程设置为 8 位、16 位或 32 位访问。BANK0 可以设置为 16 位、32 位访问；
 4. 8 个地址空间中，6 个地址空间可以用于 ROM、SRAM 等存储器，2 个用于 ROM、SRAM、SDRAM 等存储器；
 5. 7 个地址空间的起始地址及空间大小是固定的；
 6. 1 个地址空间的起始地址和空间大小是可变的；
 7. 所有存储器空间的访问周期都可以通过编程配置；
 8. 提供外部扩展总线的等待周期；
 9. SDRAM 支持自动刷新和掉电模式。
- LCD 控制器

S3C2410A 内部集成了 LCD 控制器，可以很方便地去控制各种类型的 LCD 屏，如 STN 和 TFT 屏。
- NAND Flash 控制器

为了支持 NAND Flash 启动，S3C2410A 内建 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 中的内容复制到 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。
- I/O 端口

S3C2410A 有 117 个多功能 I/O 端口，可软件配置其特殊功能。
- USB 主控制器

S3C2410A 内嵌 2 个 USB 主控制器，有以下特点：

 1. 兼容 OHCI Rev1.0
 2. 兼容 USB Rev 1.1
 3. 两个 Two down stream ports
 4. 支持低速和全速 USB 设备

2.2.2 硬件系统整体结构

本文采用的硬件平台由底板和核心板组成，核心板上使用了 SAMSUNG 公司的 S3C2410A 处理器，并集成了 64MB 的 SDRAM，64MB NAND Flash 存储设备以及核心电压模块、实时时钟、系统跳线、系统时钟、核心板接口等；底板上提供了丰富的外设接口:CS8900A 以太网卡接口、2 个与 PC 机通信的 UART（10M/100M）、1 个 LCD 接口、触摸屏接口、128KB 的 NOR Flash 存储芯片、SD 接口、IDE 接口及 USB 接口等。核心板和底板配合即构成了一个完整的硬件系统，它能够装载和运行嵌入式 Linux 操作系统。也可以运行基于 ARM 核的其它操作系统。

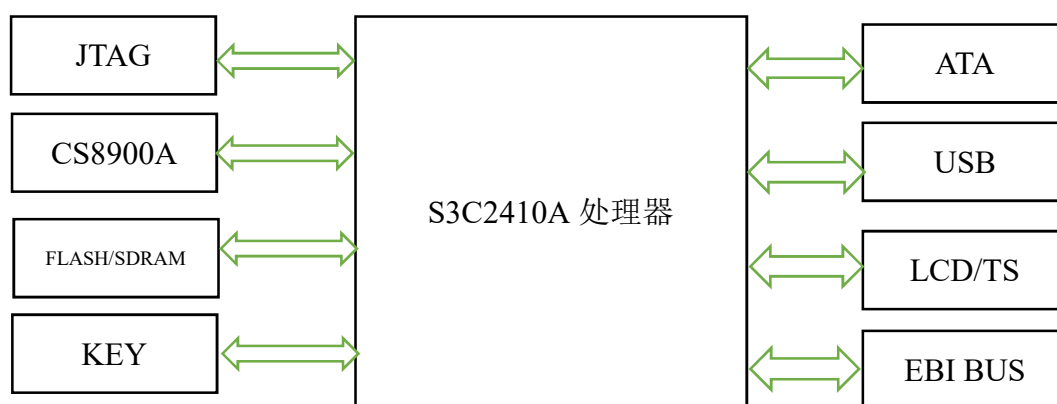


图 2-1 S3C2410 硬件平台体系结构图

2.3 移植环境搭建

与其他软件开发不同，嵌入式软件通常在一个平台开发，而在另一个平台运行。因此，需要一系列的跨平台开发工具（交叉开发工具），这些工具包括启动加载器（BootLoader）、构建工具、调试工具、Linux 内核以及其他的软件工具（如代码编辑器、ftp、终端工具等）。

2.3.1 Ubuntu 开发平台

选择 Ubuntu 作为开发平台的原因一是考虑到其是目前最流行的 Linux 发行版，二是它强大的基于网络的软件包管理机制，安装需要的软件包非常方便。在 Windows 系统上安装 Ubuntu 可以选择在虚拟机中安装，也可以安装与 Windows 并列的 Ubuntu 双系统。在实际开发操作中为了方便我两种方式都用上了，使用的是最新的 Ubuntu 18.04 系统。

2.3.2 搭建交叉编译环境

本文使用的交叉编译工具链是 arm-linux-gcc，最终选用了 4.2.2 版编译器来编译内核，使用 3.3.2 版编译的 u-boot，据前人的经验这两个版本编译比较稳定可靠，没有出现编译错误。安装步骤如下：

步骤 1 去官网下载文件 `obsolete-gcc-3.3.2.tar.bz2`，解压并安装到 `/usr/local/arm/` 目录下。使用如下命令：

```
# tar -jxvf obsolete-gcc-3.3.2.tar.bz2
```

步骤 2 配置系统环境变量，把交叉编译工具链的路径添加到环境变量 `PATH` 中，可在 “`~/.bashrc`” 中添加一行 “`export PATH=$PATH : /home/shengzx/usr/local/arm/3.3.2/bin`”。若是编译内核则可以把 `PATH` 改为：“`PATH=$PATH: /home/shengzx/usr/local/arm/4.2.2-eabi/usr/bin`”。

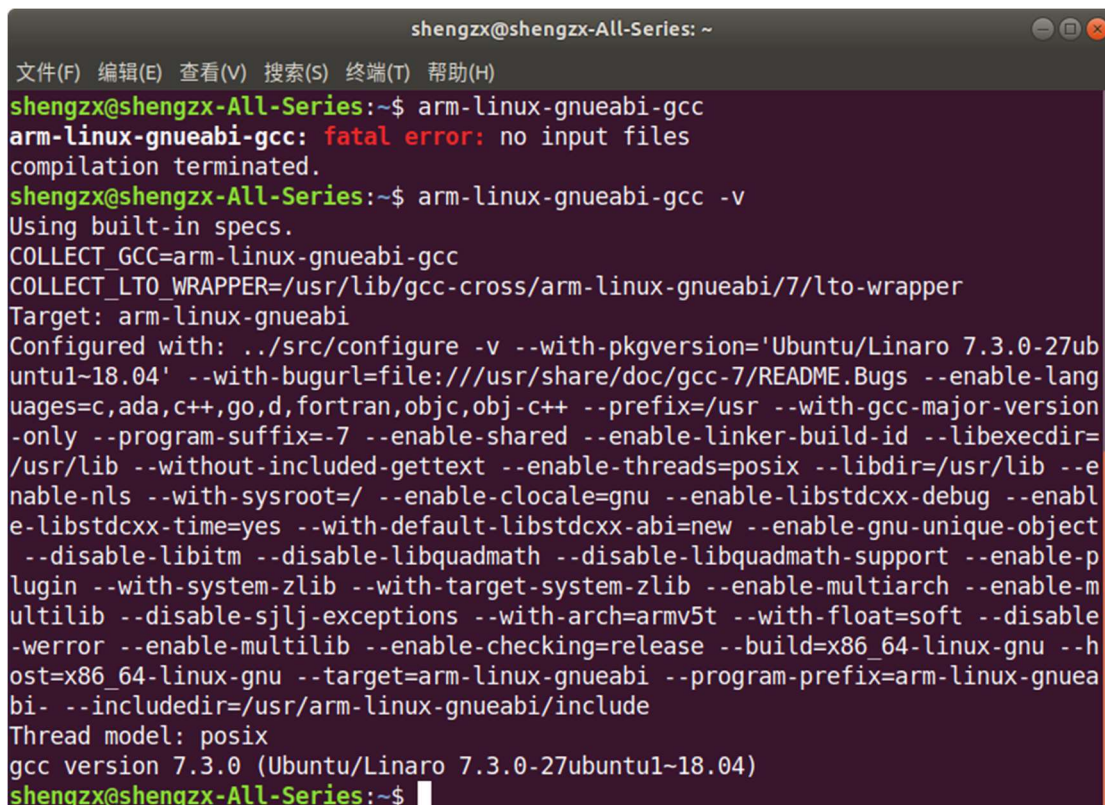


图 2-2 配置环境变量

步骤 3 使用命令 `source ~/.bashrc`，使环境变量生效。

步骤 4 在终端上输入命令 `arm-linux-`，再按 `TAB` 键，若能显示补全的命令，则说明环境变量设置成功了。

步骤 5 使用命令 `arm-linux-gcc -v`，查看软件配置和版本信息，若看到如图所示内容，则说明程序安装成功了。



```
shengzx@shengzx-All-Series: ~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
shengzx@shengzx-All-Series:~$ arm-linux-gnueabi-gcc  
arm-linux-gnueabi-gcc: fatal error: no input files  
compilation terminated.  
shengzx@shengzx-All-Series:~$ arm-linux-gnueabi-gcc -v  
Using built-in specs.  
COLLECT_GCC=arm-linux-gnueabi-gcc  
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/arm-linux-gnueabi/7/lto-wrapper  
Target: arm-linux-gnueabi  
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 7.3.0-27ubuntu1-18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libitm --disable-libquadmath --disable-libquadmath-support --enable-plugin --with-system-zlib --with-target-system-zlib --enable-multiarch --enable-multilib --disable-sjlj-exceptions --with-arch=armv5t --with-float=soft --disable-werror --enable-multilib --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=arm-linux-gnueabi --program-prefix=arm-linux-gnueabi- --includedir=/usr/arm-linux-gnueabi/include  
Thread model: posix  
gcc version 7.3.0 (Ubuntu/Linaro 7.3.0-27ubuntu1-18.04)  
shengzx@shengzx-All-Series:~$
```

图 2-3 安装 arm-linux-gcc 交叉编译工具链

2.3.3 获取内核

为了移植方便，我选用了 2.6.29 版本的内核，这款内核已经并入了对 S3C2410 处理器及其相近处理器公板的支持，从而可以大大降低移植的难度。可以使用如下命令来获取该版内核：

```
# wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.29.tar.bz2
```

2.3.4 获取 BootLoader

在选择 u-boot 的版本时，也需要像选择内核版本时一样考虑，不能太新也不能太旧，否则会给移植工作带来很大的困难。这里我选用的是 2009 年 11 月份发布的 u-boot-2009.11，和 Linux2.6.29 内核配合比较好。下载该版 u-boot 的命令如下：

```
# wget http://ftp.denx.de/pub/u-boot/u-boot.2009.11.tar.bz2
```

2.3.5 PuTTY 的安装和配置

PuTTY 是一个支持 Telnet、SSH、rlogin、纯 TCP 以及串口等多种通信方式的连接软件。可以替代 Windows 上的超级终端以及 Linux 上的 miniCOM，这里选用它是因为其简单易用，并且功能强大。在 Ubuntu 中安装 PuTTY 命令如下：

```
# apt install putty
```

完成安装后，打开 `putty` 运行，按照以下步骤配置后续会一直使用的一个串口连接会话。

步骤 1 如图 2-4 所示，在“Connection type”中选择连接类型为“Serial”串口，在“Saved Session”中输入会话名称，单击 Save 按钮保存该会话。

步骤 2 如图 2-5 所示，通过左边的类别栏，进入“Serial”选项卡，详细配置串口相关参数，这里需要修改“Flow Control”（硬件流控）为“None”。

步骤 3 切换回“Session”选项卡，再次保存会话，最后单击“Open”按钮即可打开会话。

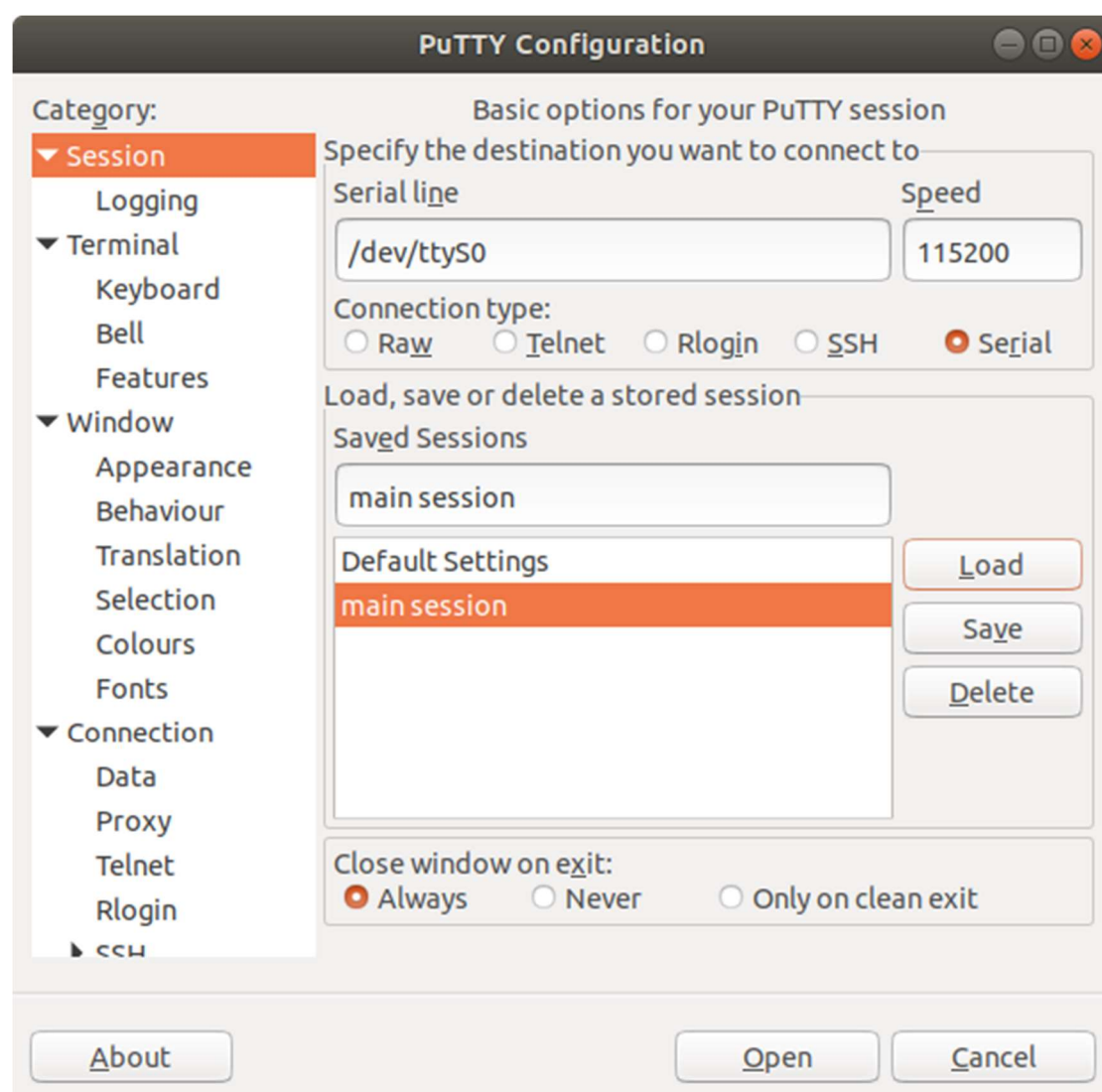


图 2-4 PuTTY session 设置

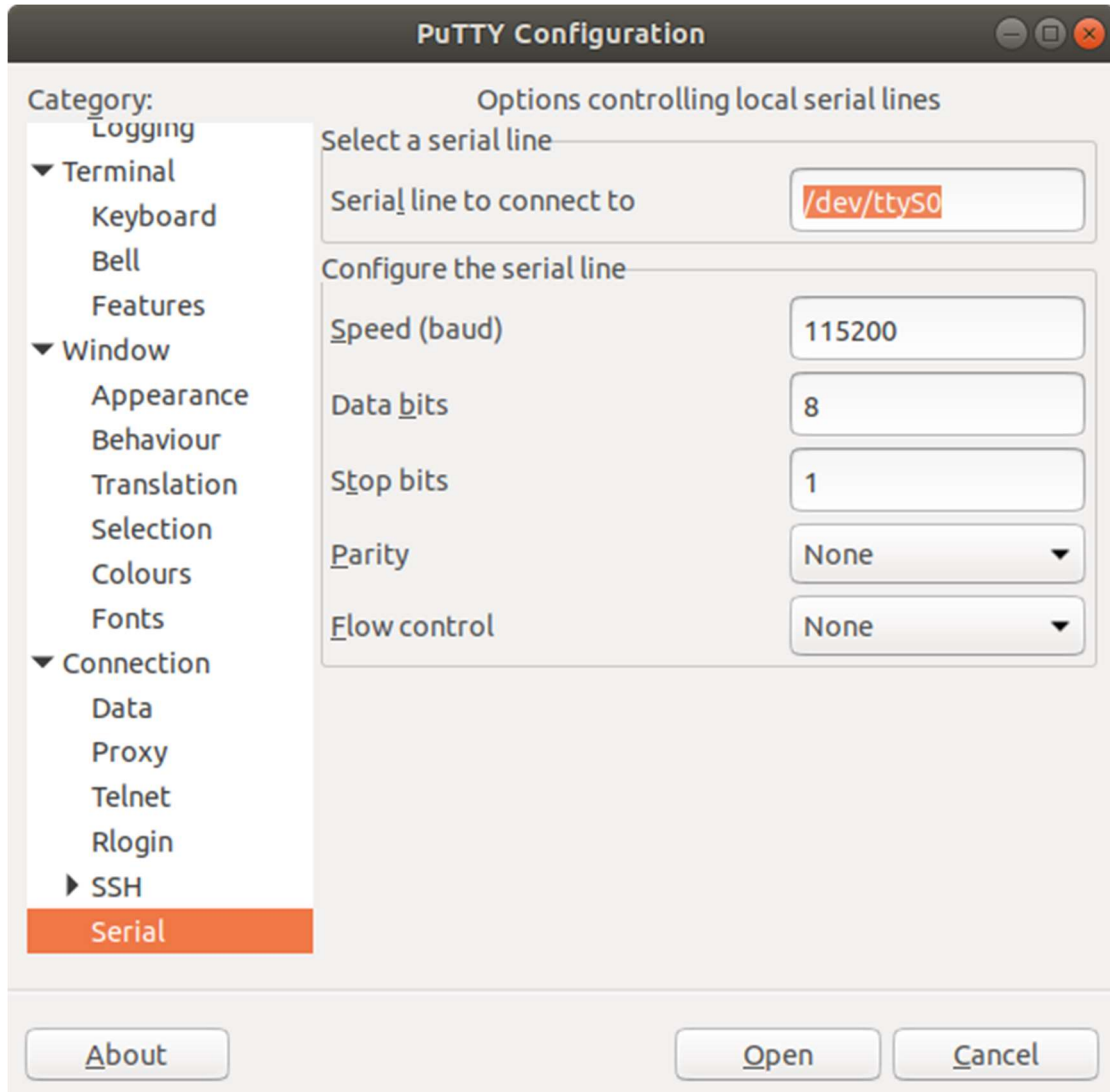


图 2-5 PuTTY 串口设置

2.3.6 配置主机的 TFTP 服务

在 Ubuntu 系统中，安装 TFTP 服务器和客户端的步骤如下：

步骤 1 使用如下命令，安装相关软件包 tftpd（服务器端）和 xinetd。

```
# apt install xinetd tftpd tftp
```

步骤 2 建立配置文件。在/etc/xinetd.d/下建立一个配置文件 tftp，在文件中输入以下内容：



图 2-6 tftp 服务设置

步骤 3 通过如下命令建立 TFTP 服务器目录（上传文件与下载文件的位置），并且更改其权限：

```
# mkdir /tftpboot
# chmod -R 777 /tftpboot
```

步骤 4 通过如下命令重新启动相关服务：

```
# service xinetd restart
```

至此 TFTP 服务已经安装完成了，下面可以对其进行测试。假设在当前目录下有一个测试文件 test.txt：

```
# tftp 127.0.0.1
tftp> put test.txt
Sent 1018 bytes in 0.0 seconds
tftp> get test.txt
Received 1018 bytes in 0.1 seconds
tftp> quit
```

通过 get 命令，可以把当前目录下的 test.txt 文件通过 TFTP 上传到它的服务文件目录，这时在/tftpboot 下会出现 test.txt 文件。通过 put 命令，可以从/tftpboot 中下载 test.txt 文件。这样就验证了 TFTP 服务配置的正确性。在文件上传与下载结束后，可以通过 quit 命令退出。

2.3.7 配置 NFS 服务

NFS 服务的主要任务是把本地的一个目录通过网络输出，相当于 Windows 系统

下的文件共享服务，其它计算机可以远程挂接这个目录并访问文件。下面介绍在 Ubuntu 下安装、配置 NFS 服务的步骤。

步骤 1 使用如下命令，安装 NFS 服务器端：

```
# apt install nfs-kernel-server
```

步骤 2 配置/etc/exports。NFS 允许挂载的目录及权限在文件/etc/exports 中进行了定义。这里采用的配置如下：

```
# vim /etc/exports
```

修改文件为：

```
/s3c2410_linux/nfs * (rw,insecure,no_root_squash,no_all_squash)
```

保存退出。

步骤 3 使用如下命令，重启服务：

```
# service portmap restart
```

```
# service nfs-kernel-server restart
```

步骤 4 通过如下的方法测试 NFS：

运行以下命令来显示一下共享的目录：

```
# showmount -e
```

或可使用以下命令把它挂载在本地磁盘上，如将/nfs/rootfs 挂载到 mnt 下：

```
# mount -t nfs localhost:/nfs/rootfs /mnt
```

可运行 df 命令查看是否挂载成功。查看后可使用如下命令卸载：

```
# umount /mnt
```

2.4 本章小结

本章首先介绍了嵌入式 Linux 系统的体系结构，其软件系统部分包括系统层、支撑层、应用层，这些是本文将要实现的具体内容，以下各章顺序也是按该体系结构自底向上安排的。接着描述了 S3C2410A 微处理器的结构，介绍了其内部的典型控制器，还简要介绍了硬件系统结构，这些内容和后面设备驱动的移植有关。最后，详述了嵌入式系统软件开发移植环境的建立方法、终端控制工具的配置及网络服务的配置，以供后面开发使用。

3 嵌入式 Linux 的 BootLoader 移植

3.1 BootLoader 概述^{[1][11]}

3.1.1 BootLoader 简介

对于计算机系统来说，从开机上电到操作系统启动需要一个引导过程。嵌入式 Linux 系统同样离不开引导程序，这个引导程序就叫启动加载器（BootLoader）。通过加载 BootLoader，可以初始化硬件设备、建立内存空间的映射表，从而建立适当的系统软硬件环境，为最终调用操作系统内核做好准备。一般系统上电或复位后 0x00000000 地址处存放的就是 BootLoader 程序。对于嵌入式系统，BootLoader 是基于特定硬件平台实现的。因此，几乎不可能为所有的嵌入式系统建立一个通用的 BootLoader，不同的处理器架构都有不同的 BootLoader。BootLoader 不但依赖于 CPU 的体系结构，而且依赖于嵌入式系统板级设备的配置。对于不同的嵌入式板而言，即使它们使用同一种处理器，要想让运行在一块板子上的 BootLoader 程序也能运行在另一块板子上，一般都需要修改 BootLoader 的源程序。

反过来，大部分 BootLoader 仍具有很多共性，某些 BootLoader 能够支持多种体系结构的嵌入式系统，例如，u-boot 就同时支持 PowerPC、ARM、MIPS 和 X86 等体系结构，支持的板子有上百种。

下表列出了几种比较流行并且支持 Linux 操作系统的 BootLoader 并对它们进行了简单的比较：

表 3-1 支持 Linux 的开源 BootLoader

BootLoader	基本描述
Lilo	X86 平台上 Linux 的 BootLoader
Grub	GNU 项目中 Lilo 的升级品
RedBoot	eCos 的 BootLoader
Blob	Lart 项目中的 Bootloader
U-Boot	通用 BootLoader，功能最全

3.1.2 u-boot 简介

u-boot 是德国 DENX 小组开发的用于多种嵌入式 CPU 的 BootLoader 程序，是在 ppcBoot 以及 ARMboot 的基础上发展而来的，u-boot 不仅支持嵌入式 Linux 系统的引导，它还支持 NetBSD、VxWorkS、QNX、RTEMS、ARTOS、LynxOS 等嵌入式操作系统。而且支持 PowerPC、MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。它的功能也比较强大，支持 tftp、nfs 等多种网络协议，所以

本论文将对 u-boot 进行移植及使用。

3.1.3 BootLoader 的启动模式及其应用

BootLoader 一般都要有两种启动模式：Flash 启动模式和下载模式。

Flash 启动模式：这种模式下，先要将内核映像和根文件系统写入 Flash，设备启动时 BootLoader 将 Flash 中的内核及根文件系统映像读入 SDRAM 指定位置并跳转到内核入口执行内核，整个过程没有用户的介入。

下载模式：这种模式下，Flash 中可以没有内核和根文件系统。设备启动时 BootLoader 通过串口或网络等通信方式从主机下载内核映像和根文件系统映像等。这种方式可以有以下用途：

- 作为安装或更新手段。通过 BootLoader 将要写入 Flash 的文件下载到内存，然后通过相应的命令，将其写到 Flash 指定的地址，在第一次安装内核与根文件系统时被使用，以后系统更新时也会使用 BootLoader 的这种工作模式。
- 作为调试手段。将内核和根文件下载到内存指定位置，并配置好传递给内核的参数，可用 BootLoader 提供的命令直接运行内核，此种方式，可以避免将内核和根文件频繁写入 Flash。
- 还可以直接以下载模式工作。以这种方式工作，内核或文件系统可存储在远程主机上，系统不需配置较大的存储介质，和无盘工作站有点类似。

在系统移植时采用如图所示的工作模式调试内核和文件系统：

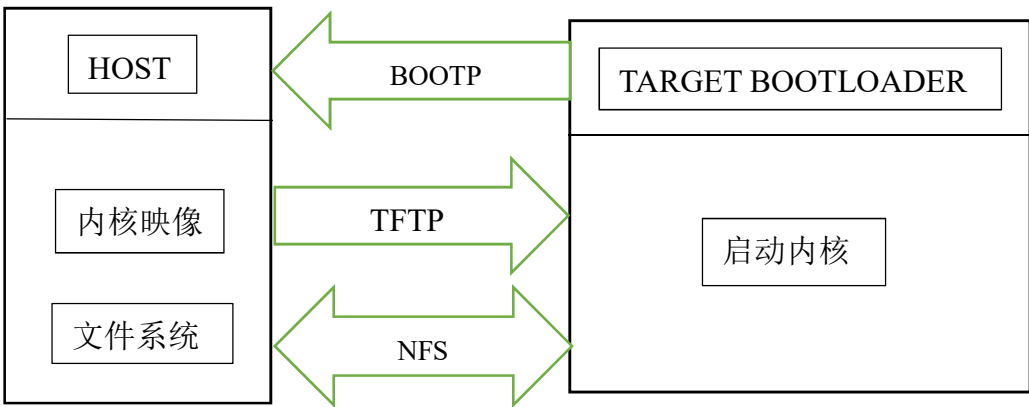


图 3-1 BootLoader 的下载模式的应用

3.2 u-boot 移植分析

3.2.1 u-boot 源码的顶层目录结构

下表列出了 u-boot 源码包下的子目录及其内容，移植过程中需要对其中部分文件进行修改。

表 3-2 u-boot 的源码顶层目录说明

目录	特性	解释说明
board	平台依赖	该目录包含一些对特定板子的初始化和操作代码
cpu	平台依赖	该目录下是针对特定处理器的初始化和操作代码，启动代码.S 文件也在这里
common	通用	此目录存放独立于处理器体系结构的通用代码，如内存大小探测与故障检测，且该目录下 main.c 可以看作是 Linux 中的主函数，负责接受用户输入并送给相应的处理函数执行
driver	通用	此目录下放的是各种驱动，如以太网驱动、LCD 屏驱动
doc	文档	U-Boot 的说明文档
examples	应用例程	目录下放的是可在 U-Boot 下运行的例子，相当于 Linux 中应用程序
fs	通用	目录下文件系统
include	通用	目录下存放各种头文件和配置文件
lib	平台依赖	处理器体系相关的文件，如 lib/arm 目录就包含 ARM 体系结构相关的文件
net	通用	与网络功能相关的文件目录，如 bootp、ntp、tftp
post	通用	上电自检文件目录
Rtc	通用	RTC 驱动程序
tools	通用	目录下的代码都是可供使用的“工具”。用于创建 U-Boot 的 bin 镜像文件。因为是在宿主机上跑的，需使用 gcc 编译

3.2.2 u-boot 的配置编译

u-boot 的源码是通过 GNU Makefile 组织编译的。顶层目录下的 Makefile 完成对

开发板整体配置，然后递归调用各级子目录下的 Makefile，最后把所有编译过的程序链接成 u-boot 映像。在此以 S3C2410 处理器公板 smdk2410 为例，介绍 u-boot 的配置编译方法，并简单分析其原理。

1. 基本的配置编译方法

将 u-boot 解压并进入源码包后，在 u-boot 顶层目录下执行如下两个命令：

```
# make <board_name>_config
```

```
# make
```

第一个命令用来配置 u-boot，其中<board_name>用具体的开发板来代替，如“make smdk2410_config”。执行第二个命令后即开始编译过程。

2. 顶层目录下的 Makefile

每一种开发板在顶层 Makefile 中都要有自己的配置规制，用 smdk2410 开发板的规制定义如下：

smdk2410_config: unconfig

 @\$(MKCONFIG) \$(@:_config=) arm arm920t smdk2410 samsung s3c24x0

执行配置 u-boot 的命令：make smdk2410_config，将通过 u-boot 顶层目录下的 mkconfig 脚本生成配置文件 include/config.mk，内容如下：

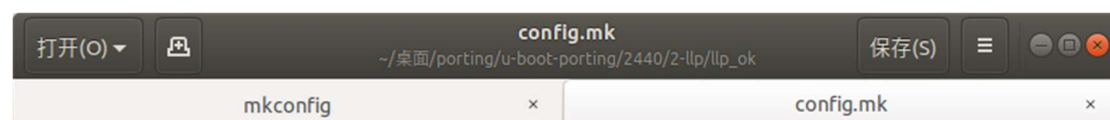
ARCH = arm

CPU = arm920t

BOARD = smdk2410

VENDOR = samsung

SOC = s3c24x0



```
# Load generated board configuration
sinclde $(OBJTREE)/include/autoconf.mk

ifdef ARCH
sinclde $(TOPDIR)/lib_$(ARCH)/config.mk      # include architecture dependend rules
endif
ifdef CPU
sinclde $(TOPDIR)/cpu/$(CPU)/config.mk      # include CPU specific rules
endif
ifdef SOC
sinclde $(TOPDIR)/cpu/$(CPU)/$(SOC)/config.mk # include SoC specific rules
endif
ifdef VENDOR
BOARDDIR = $(VENDOR)/$(BOARD)
else
BOARDDIR = $(BOARD)
endif
ifdef BOARD
sinclde $(TOPDIR)/board/$(BOARDDIR)/config.mk # include board specific rules
endif

#####

ifneq (,$(findstring s,$(MAKEFLAGS)))
ARFLAGS = cr
```

图 3-2 config.mk 文件内容

上面的 include/config.mk 文件定义了 ARCH、CPU、BOARD、VENDOR、SOC 等变量。这样硬件平台依赖的目录文件可以根据这些定义来确定。smdk2410 平台相关目录如下：

```
board/Samsung/smdk2410/
cpu/arm920t/
cpu/arm920t/s3c24x0
lib_arm/
include/asm-arm
include/config/smdk2410.h
```

再回到顶层目录的 Makefile 文件开始的部分，其中下列几行包含了对前面所述变量的定义：

```
# load ARCH,BOARD,and CPU configuration
include $(obj) include/config.mk
export ARCH CPU BOARD VENDOR SOC
```

Makefile 的编译选项和规则在顶层目录的 config.mk 文件中有定义。各种通用的规则则直接在该文件中定义。通过 ARCH、CPU、BOARD、SOC 等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在各自的 lib_XXX（如 lib_arm）目录下的 config.mk 文件中。

3. 开发板配置头文件

除了编译顶层 Makefile 外，还要在移植时为开发板定义配置选项或参数。这个头文件是 include/configs/<board_name>.h。这些头文件中定义的选项或参数宏以 CONFIG_ 为前缀，用来选择处理器、设备接口、命令、属性等。如：

```
# define CONFIG_ARM920T    1
# define CONFIG_DRIVER_CS8900    1
```

4. 编译结果

根据对 Makefile 的分析，编译分为两步：第一步配置，如 make smdk2410_config；第二步编译，执行 make 命令即可。

编译完成后可以得到 u-boot 各种格式的映像文件和符号表，如下图所示：

api	CREDITS	libfdt	net	u-boot.bin
board	disk	lib_generic	onenand_ipl	u-boot.lds
CHANGELOG	doc	LOG	post	u-boot.map
CHANGELOG-before-U-Boot-1.1.5	drivers	MAINTAINERS	README	u-boot.srec
common	examples	MAKEALL	rules.mk	
config.mk	fs	Makefile	System.map	
COPYING	include	mkconfig	tools	
cpu	lib_arm	nand_spl	u-boot	

图 3-3 编译生成的 u-boot 各种格式的映像文件

表 3-3 u-boot 编译生成的映像文件

文件名称	说明	文件名称	说明
System.map	u-boot 映像的符号表	u-boot.bin	u-boot 映像原始的二进制格式
u-boot	u-boot 格式的 ELF 格式	u-boot.srec	u-boot 映像的 S-Record 格式

u-boot 的 3 种映像格式都可以烧写到 Flash 中，但需要看加载器能否识别这些格式。一般 u-boot.bin 最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到 Flash 中即可。

3.2.3 u-boot 的启动流程

大多数 BootLoader 都分为 Stage1 和 Stage2 两部分，u-boot 也不例外。依赖于 CPU 体系结构的代码通常放在 Stage1 并用汇编语言来实现，而 Stage2 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。

1. Stage1 功能分析

u-boot 的 Stage1 代码通常放在 Start.s 文件中，u-boot 是从 cpu/ARM920t/start.s 开始执行的，这个文件的任务是设置处理器状态、初始化中断和内存时序等，并确定是否需要对整个 u-boot 代码重定位，最终从 Flash 中跳转到定位好的内存地址执行。

其主要代码功能如下：

1. 定义入口。由于一个可执行的 uImage 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 Flash 的 0x00000000 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。
2. 设置异常向量 (Exception vector)。
3. 设置 CPU 的速度、时钟频率及中断控制寄存器。
4. 初始化内存控制器。
5. 将 Flash 中的程序复制到 RAM 中。
6. 初始化堆栈。
7. 转到 RAM 中执行。

2. Stage2 功能分析

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 的主函数，该函数主要完成如下操作：

1. 调用一系列的初始化函数。

```

cpu_init();           //片级初始化代码
board_init();         //板级初始化相关代码
interrupt_init();     //中断初始化
env_init();           //事件初始化
init_baudrate();      //初始化波特率
Serial_init();        //串口初始化

```

2. 初始化 Flash 设备。

初始化 Flash 是指提供 Flash 的各 bank 的情况，是否擦写，是否上锁等信息，为以后 Flash 相关命令使用。

3. 初始化系统内存分配函数。

4. 如果目标系统有 NAND 设备，则初始化 NAND 设备。

5. 如果目标系统有显示设备，则初始化该类设备

6. 初始化相关网络设备，填写 IP、MAC 地址等。

7. 进入命令循环，接受用户从串口输入的命令，然后进行相应的工作。

其大致流程如图所示：

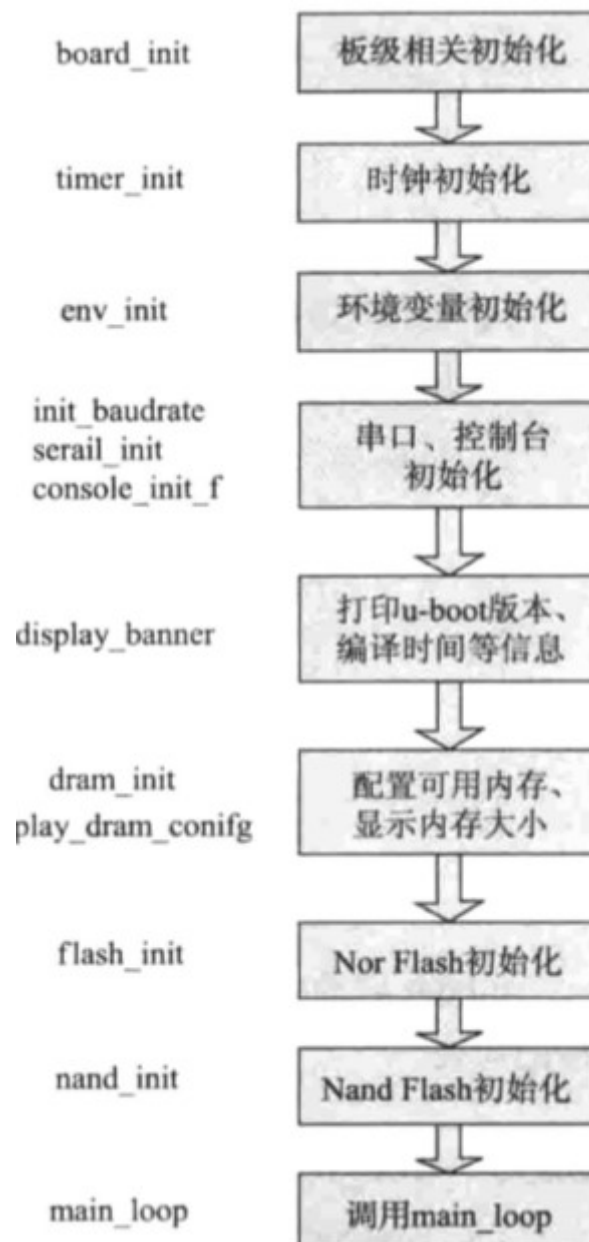


图 3-4 u-boot 第二阶段执行流程

3.2.4 NAND Flash 和 NOR Flash

NOR Flash 的特点是芯片内执行(XIP, eXecute In Place), 这样应用程序可以直接在 Flash 闪存内运行, 不必再把代码读到系统 RAM 中。NOR 的传输效率很高, 在 1M-4M 的小容量时具有很高的成本效益, 但是很低的写入和擦除速度大大影响了它的性能。

NAND FLASH 结构能提供极高的单元密度, 可以达到高存储密度, 并且写入和擦除速度也很快。应用 NAND 的困难在于 NAND Flash 的管理需要特殊的系统接口。擦除 NOR 器件时是以 64KB-128KB 的块进行的, 执行一个写入/擦除操作的时间为 5s, 与此相反, 擦除 NAND 器件是以 8KB~32KB 的块进行的, 执行相同的操作只需要 4ms。

NOR Flash 带有 SRAM 接口, 有足够的地址引脚来寻址, 可以很容易的存取其内部的每一个字节。

NAND 器件使用复杂的 I/O 口来串行地存取数据, 各个产品或厂商的方法可能不同。用 8 个引脚来传送控制、地址和数据信息。

由以上的区别可以看出, NOR Flash 适合于存储程序代码, 而 NAND Flash 适合于存储大量数据。但同样容量的存储芯片, NAND Flash 的价格是 NOR Flash 的 1/5。又因为 S3C2410A 芯片支持从 NAND Flash 直接启动, 所以很多开发板上 NOR Flash 容量很小甚至没有, 也有的开发板只有 NOR Flash 而没有 NAND Flash。

3.2.5 u-boot 从 NOR Flash 启动的具体实现

移植 u-boot 的难点在于在 u-boot 源码中编写对硬件平台的板级支持, 使硬件平台上的 FLASH、SRAM、串口控制芯片、网络控制芯片以及其他芯片可以正常工作。研究中使用的开发板和 smdk2410 开发板相近, 所以这里直接在 smdk2410 目录上修改, 根据需求配置好 u-boott 然后执行 make 生成可以移植的 u-boot.bin。移植 u-boot 到开发板上只需要修改和硬件相关的代码即可。主要修改的文件有:

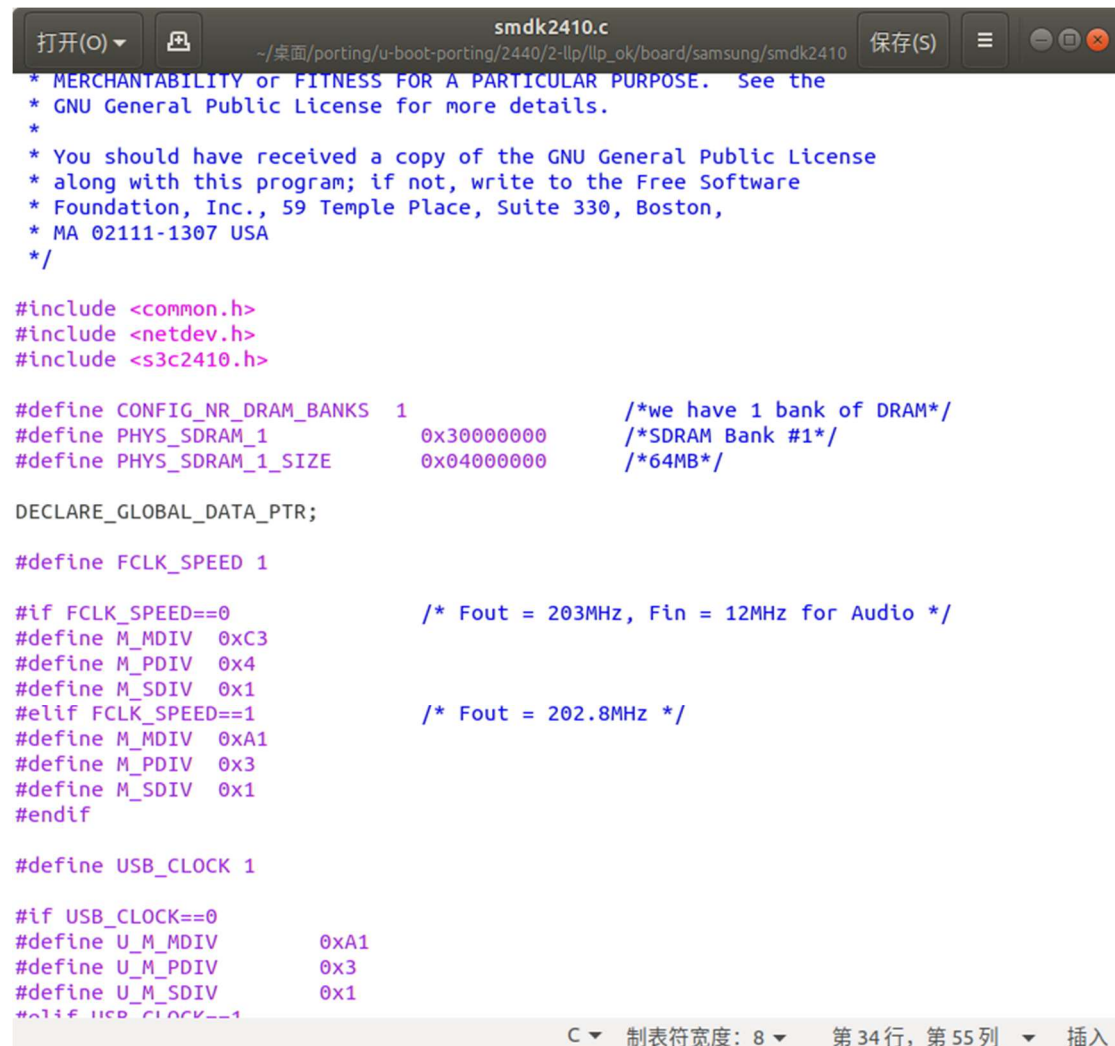
- Makefile 文件
- include 目录下的目标板.h 头文件(smdk2410.h)
- board 目录下的目标板 smdk2410.c 文件
- nand_flash.c 文件
- u-boot.lds 链接文件, 以及 cpu 目录下的串口驱动文件

具体修改如下:

1. cpu/arm920t 目录下的 Start.S, 修改 relocate 代码段, 初始化 NAND Flash 控制器, 并将 u-boot 自身代码复制到内存指定地址。
2. 编写 ./include/configs/smdk2410.h, 该文件主要设置目标板的硬件配置, 包括设置一些关键寄存器的值, 设置 FLASH 及 RAM 的起始地址以及片选, 设置串口波特率等。本文件还包含了一些定制 u-boot 的配置信息, 包括启动等待时间, 是否自动执行启动命令, u-boot 提示符以及 u-boot 所支持的用户交互命令等。

3. 编写./board/smdk2410/smdk2410.c, 此文件的主要作用是完成板卡的检测以及 SDRAM 的初始化和 SDRAM 容量大小的检测。

```
#define CONFIG_NR_DRAM_BANKS 1 /* we have 1 bank of DRAM */
#define PHYS_SDRAM_1 0x30000000 /* SDRAM Bank #1 */
#define PHYS_SDRAM_1_SIZE 0x04000000 /* 64MB */
```



```
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston,
* MA 02111-1307 USA
*/

#include <common.h>
#include <netdev.h>
#include <s3c2410.h>

#define CONFIG_NR_DRAM_BANKS 1 /*we have 1 bank of DRAM*/
#define PHYS_SDRAM_1 0x30000000 /*SDRAM Bank #1*/
#define PHYS_SDRAM_1_SIZE 0x04000000 /*64MB*/

DECLARE_GLOBAL_DATA_PTR;

#define FCLK_SPEED 1

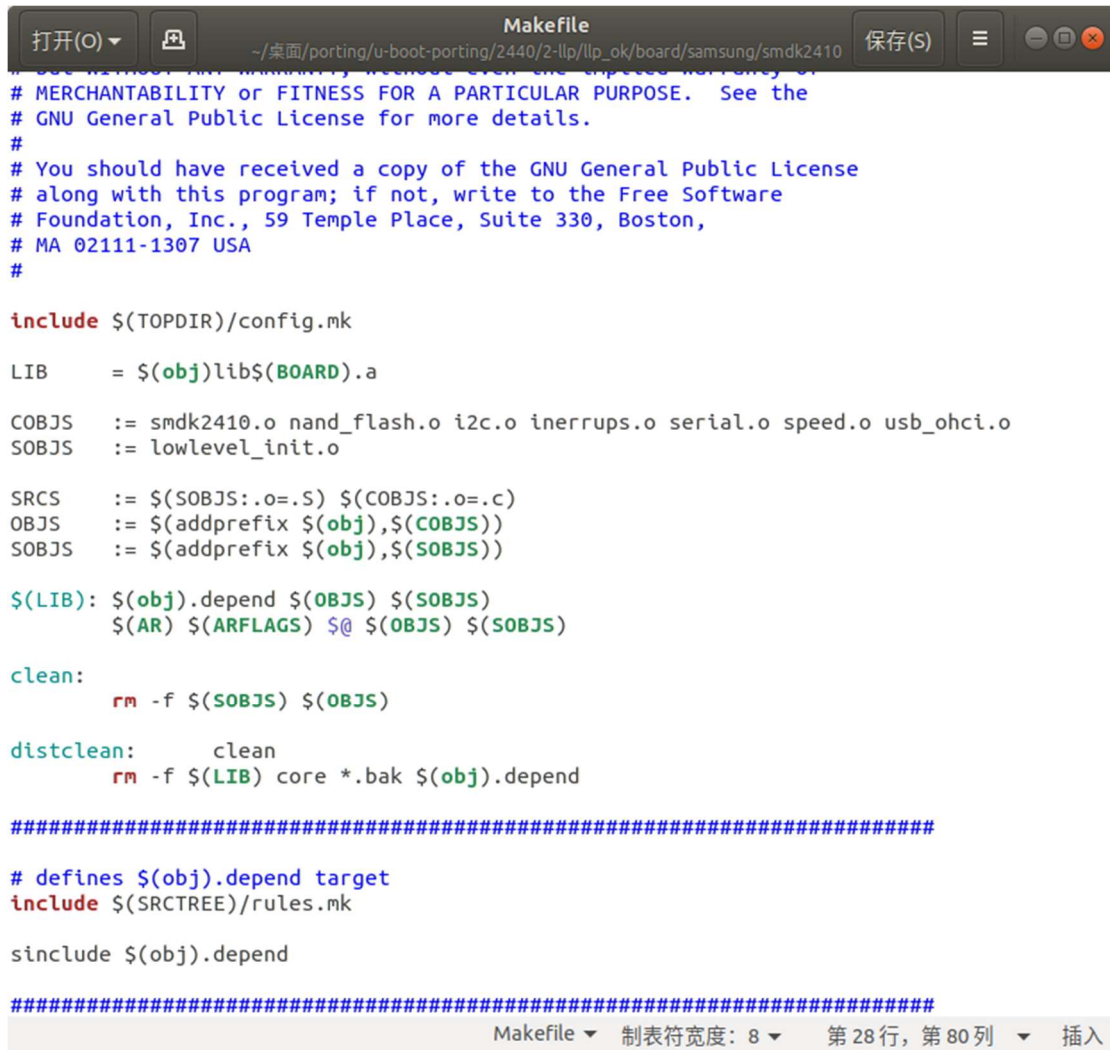
#if FCLK_SPEED==0 /* Fout = 203MHz, Fin = 12MHz for Audio */
#define M_MDIV 0xC3
#define M_PDIV 0x4
#define M_SDIV 0x1
#elif FCLK_SPEED==1 /* Fout = 202.8MHz */
#define M_MDIV 0xA1
#define M_PDIV 0x3
#define M_SDIV 0x1
#endif

#define USB_CLOCK 1

#if USB_CLOCK==0
#define U_MDIV 0xA1
#define U_PDIV 0x3
#define U_SDIV 0x1
#elif USB_CLOCK==1
```

图 3-5 修改 smdk2410.c

4. 增加对 NAND Flash 操作的相关文件: nand_flash.c, 这个文件在原来的 u-boot 里面没有需要自己加进去, 将写入的 C 代码放到 u-boot 下面 /cpu/arm920t/s3c24x0 下面。主要执行对 Nand Flash 的擦除、写入操作。然后还需要修改此文件夹下的 Makefile 使其能生成 nand_flash.o 文件。



```
# Use without any warranty; we make even the express warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston,
# MA 02111-1307 USA
#

include $(TOPDIR)/config.mk

LIB      = $(obj)lib$(BOARD).a

COBJS    := smdk2410.o nand_flash.o i2c.o inerrups.o serial.o speed.o usb_ohci.o
SOBJS    := lowlevel_init.o

SRCS     := $(SOBJS:.o=.S) $(COBJS:.o=.c)
OBJS     := $(addprefix $(obj),$(COBJS))
SOBJS    := $(addprefix $(obj),$(SOBJS))

$(LIB): $(obj).depend $(OBJS) $(SOBJS)
$(AR) $(ARFLAGS) $@ $(OBJS) $(SOBJS)

clean:
    rm -f $(SOBJS) $(OBJS)

distclean:
    clean
    rm -f $(LIB) core *.bak $(obj).depend

#####
# defines $(obj).depend target
include $(SRCTREE)/rules.mk

sinclude $(obj).depend

#####
```

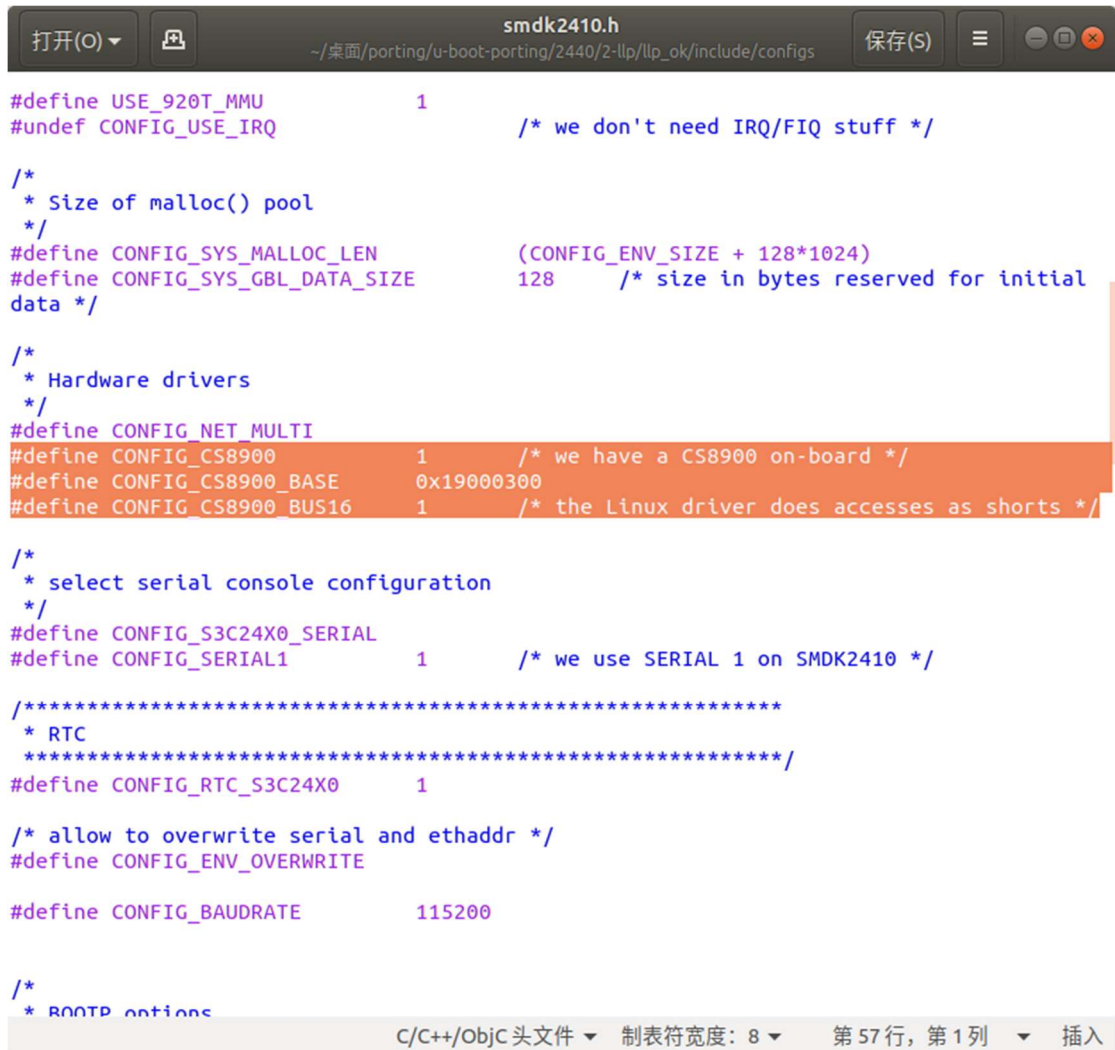
图 3-6 修改 Makefile 添加对 nand_flash.o 的支持

5. Makefile 文件。u-boot 的 Makefile 文件包括了 u-boot 所支持的目标板列表，若定义了新的目标板则要加进去。

6. 修改支持网卡驱动（可以选择支持多种网卡驱动）。执行：

```
# vim include/configs/smdk2410.h
```

修改支持 CS8900



```
#define USE_920T_MMU 1
#undef CONFIG_USE_IRQ /* we don't need IRQ/FIQ stuff */

/*
 * Size of malloc() pool
 */
#define CONFIG_SYS_MALLOC_LEN (CONFIG_ENV_SIZE + 128*1024)
#define CONFIG_SYS_GBL_DATA_SIZE 128 /* size in bytes reserved for initial
data */

/*
 * Hardware drivers
 */
#define CONFIG_NET_MULTI
#define CONFIG_CS8900 1 /* we have a CS8900 on-board */
#define CONFIG_CS8900_BASE 0x19000300
#define CONFIG_CS8900_BUS16 1 /* the Linux driver does accesses as shorts */

/*
 * select serial console configuration
 */
#define CONFIG_S3C24X0_SERIAL
#define CONFIG_SERIAL1 1 /* we use SERIAL 1 on SMDK2410 */

/*****
 * RTC
 *****/
#define CONFIG_RTC_S3C24X0 1

/* allow to overwrite serial and ethaddr */
#define CONFIG_ENV_OVERWRITE

#define CONFIG_BAUDRATE 115200

/*
 * BOOTP options
 */
```

图 3-7 配置网卡驱动

网卡 CS8900 的访问基地址是 0x19000000, 之所以再偏移 0x300 是由它来的特性决定。此外, 在此文件里还可以修改网卡 MAC 地址和 IP 地址。

7. config.mk 文件。此文件用于设置程序链接的起始地址。

3.2.3 u-boot 的调试

新移植的 u-boot 如不能正常工作, 就需要进行调试。

硬件调试器:

可以用仿真器通过 JTAG 端口直接把程序下载到目标板内存, 或者进行 Flash 编程。仿真器还可以在线调试程序。对于 u-boot 的调试可以采用 BDI2000。BDI2000 可以反汇编地跟踪 Flash 中的程序, 也可以进行源码级的调试。

调试方法如下:

1. 配置 BDI2000 和目标板初始化程序, 连接目标板。

2. 添加 u-boot 的调试编译选项，重新编译。

u-boot 的程序代码是位置相关的，尽量在内存中进行调试，可以通过修改 board/<board_name>/econfig.mk 中定义的 TEXT_BASE 来实现连接地址的定位。

另外，如果有复位向量也需要先从链接脚本中去掉。连接脚本为：

```
board/(board_name)/u-boot.lds
```

添加调试选项：在 config.mk 文件中查找 DBGFLAGS，加上 -g 选项。然后重新编译 u-boot。

3. 下载 u-boot 到目标板内存。

通过 BDI2000 的下载命令 LOAD，把程序加载到目标板内存中，然后跳转到 U-Boot 入口。

4. 启动 GDB 调试。

启动 GDB 调试，这里是交叉调试的 GDB。GDB 与 BDI2000 建立链接，然后就可以设断点执行了：

```
# arm-Linux-gdb u-boot
(gdb) target remote 192.168.1.222:2010
(gdb) stepi
(gdb) start_armboot
```

软件跟踪：

如果 u-boot 没有任何串口打印信息，手头又没有硬件调试工具，那么可以通过开发板上的 LED 指示灯判断。执行到不同的阶段时，将指示灯设置成不同的状态。在项目实践过程中，并没有使用以上两种方法，而是使用串口输出信息进行调试。

3.3 本章小结

BootLoader 是系统最先运行的程序代码，它是内核正常启动的前提，它的移植成败直接影响后面的工作，所以，这里对 BootLoader 做了比较深入的研究。因所使用硬件平台的可用 NOR Flash 容量有 512KB，而 u-boot 的目标代码在 95KB 左右，可以直接在 NOR FLASH 启动。

本章首先介绍了 BootLoader 的两种工作模式：启动模式和下载模式以及下载模式在调试、升级系统时的具体应用，接着分析了 u-boot 移植到 NOR Flash 的要点以及 u-boot 的两种调试方法，最终，通过调试和烧写，使 u-boot 可以从 NOR Flash 启动运行。

4 Linux 内核的编译、移植

4.1 Linux2.6 内核的特性简介^[5]

1. 支持更多处理器，比如 AMD64、一些大型机及嵌入式等，同时改进了对已有处理器的支持；
2. 采用抢占式内核，使交互式操作的响应速度大大提高；
3. 修改了 I/O 子系统部分，保证在各种工作负荷下 I/O 都有很好的响应速度；
4. 增加了 IDE/ATA、SCSI 等存储总线，解决和改善了以前的一些问题。比如 2.6 版内核可以直接通过 IDE 驱动程序来支持 IDE CD/RW 设备，而不必像以前一样要使用一个特别的 SCSI 模拟驱动程序；
5. 大量改进文件系统。比如支持 Windows 的逻辑卷管理器、重写对 NTFS 文件系统的支持、改进 HPFS 等；
6. 改进和部分重写了 Modules 功能，使之更稳定；
7. 改进对 USB 的支持，使之能够支持当前多数主流的 USB 设备；
8. 加强对无线设备的支持；
9. 增加了 ALSA(Advanced Linux Sound Architecture)。ALSA 是有希望取代旧式 OSS(Open Sound System)的另一种声音系统，能支持全杜比录音及回放、无缝混音、支持声音合成设备、USB 声卡等；
10. 支持更多种类和型号的多媒体设备；
11. 网络方面新增了对 IPSec 协议的支持，改进了对 IPv6 的支持。

由于 Linux2.6 具有以上新特性，与 Linux 2.4 相比更适合嵌入式应用，所以本文将对 2.6 版 Linux 内核进行移植。

4.2 Linux 内核的启动流程分析

理解 Linux 内核的启动流程，是理解 Linux 工作原理的基础，也是掌握 Linux 移植方法的必备知识。Linux 内核启动代码大体上可以分为用汇编编写的体系结构相关部分、用 C 语言编写的体系结构相关部分，以及用 C 语言编写的体系结构无关部分。本节将分析这三个部分的启动代码。但在此之前，需先了解内核镜像的生成过程。

4.2.1 内核镜像生成

在编译内核时，通常会使用“make zImage”命令生成 zImage 镜像，事实上该镜像是一个经过压缩，并带有自解压代码的内核镜像。一个非压缩的内核镜像才是真正的 Linux 内核代码。压缩内核镜像执行时，先解压内部包含的数据块（即非压缩内核镜像），再执行非压缩内核镜像。

非压缩内核镜像由“make Image”命令产生。其生成过程如下：

1. 内核的各个模块经过编译、链接，在内核源代码的顶层目录下生成 `vmlinux` 文件，这是一个 ELF 格式的镜像。
 2. 用 `arm-linux-objcopy` 命令把 `vmlinux` 转换为二进制格式镜像 `arch/arm/boot/Image`。
- 压缩内核镜像由“`make zImage`”命令产生，其生成过程如下：
- 用 `gzip` 对非压缩内核二进制镜像 `arch/arm/boot/Image` 进行压缩，生成 `arch/arm/boot/compressed/piggy.gz` 文件。
 - `arch/arm/boot/compressed` 目录下有 3 个文件：`piggy.S` 定义了一个包含 `piggy.gz` 文件的数据段；`head.S` 包含对 `gzip` 压缩过的内核进行解压的代码；`vmlinux-lds` 是链接脚本。这几个文件经过编译链接在 `arch/arm/boot/compressed` 目录下产生 `vmlinux` 文件，这是一个 ELF 格式的镜像。
 - 用 `arm-linux-objcopy` 命令将 `arch/arm/boot/compressed/vmlinux` 转换为二进制格式镜像：`arch/arm/boot/vompressed/zImage`。

两种内核镜像的产生过程如图所示：

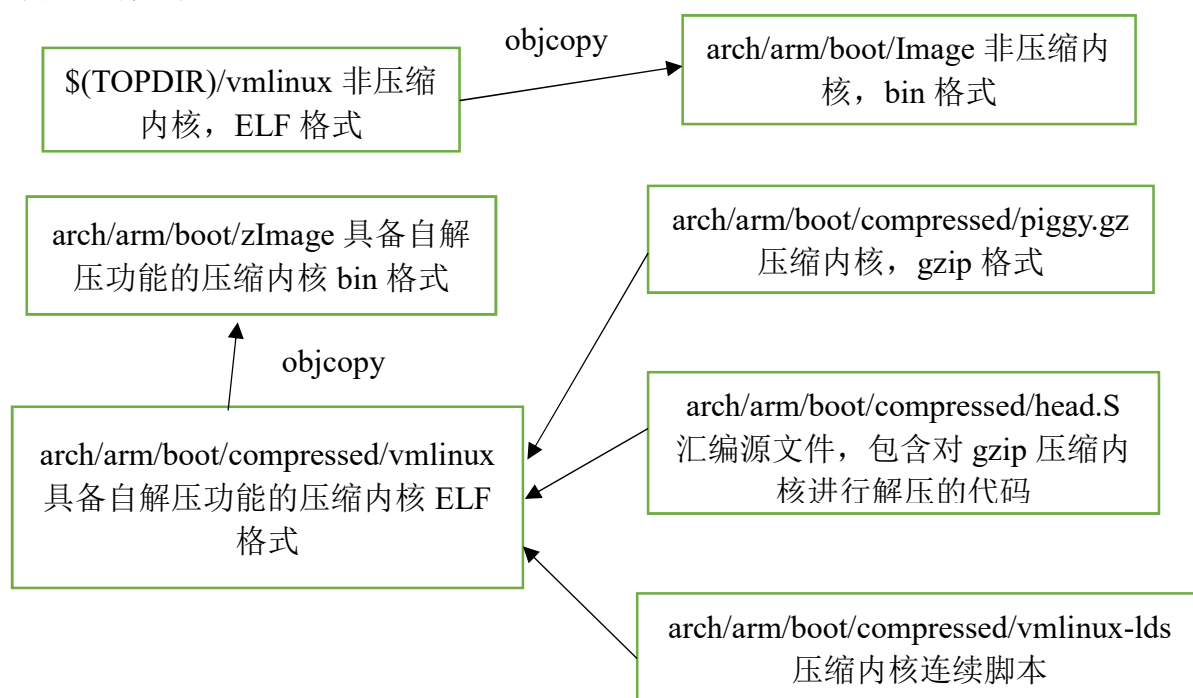


图 4-1 内核镜像生成过程

4.2.2 内核启动流程——汇编部分

内核汇编部分启动流程如下：

1. 设置处理器为 SVC 模式，关中断
2. 读取 CPUID，调用 `_lookup_processor_type`，查找处理器信息结构 `proc_info`
3. 调用 `_lookup_machine_type`，查找机器类型信息结构 `machine_desc`

4. 调用 `_create_page_tables` 函数为内核创建页面映射表
5. 调用处理器底层初始化函数，初始化 MMU、Cache、TLB
6. 跳转到 `_enable_mmu` 函数，打开 MMU
7. 跳转到 `_mmap_switched` 函数 `u`，建立 C 语言入口函数 `start_kernel()`

4.2.3 内核启动流程——C 语言部分

此阶段的初始化由 `start_kernel` 函数开始，至第一个用户进程 `init` 结束，过程中调用了一系列的初始化函数对所有内核组件进行初始化。其中，`start_kernel`、`rest_init`、`kernel_init`、`init_post` 等 4 个函数构成了整个初始化过程的主线，如图所示：

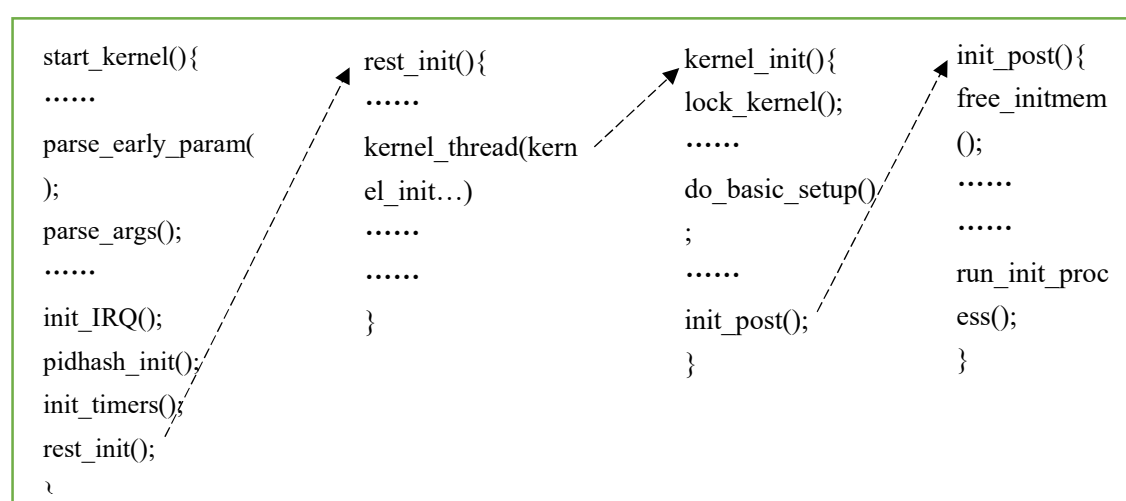


图 4-2 内核启动流程 C 语言部分初始化流程主线

4.2.4 内核启动流程——Busybox 的 `init` 进程

Busybox 是后面用来制作文件系统的一个工具集，可以用来替换 GNU `fileutils` `shutils` 等工具集。Busybox 中的各种命令与相应的 GNU 工具相比，能提供的选项较少，但是能够满足一般的应用。Busybox 为各种小型的嵌入式系统提供了比较完全的工具集。

Busybox 提供的核心程序中包括了用户空间的 `init` 进程。用户空间的 `init` 进程是整个系统启动流程中的最后一个阶段，经过该进程的初始化，整个系统将进入服务状态，提供诸如系统调用、任务管理服务以及设备管理等服务。

Busybox 的 `init` 进程会根据配置文件决定启动哪些程序，如执行某些脚本、启动 shell、运行用户指定的程序等。总之，该 `init` 进程将成为后续所有进程的发起者，如在 `init` 进程启动 `“/bin/sh”` 程序后，才能在控制台上输入各种命令。

Busybox 的 `init` 进程对应的代码在 Busybox 根目录下的 `init/init.c` 文件中。其初始化的流程如图所示：

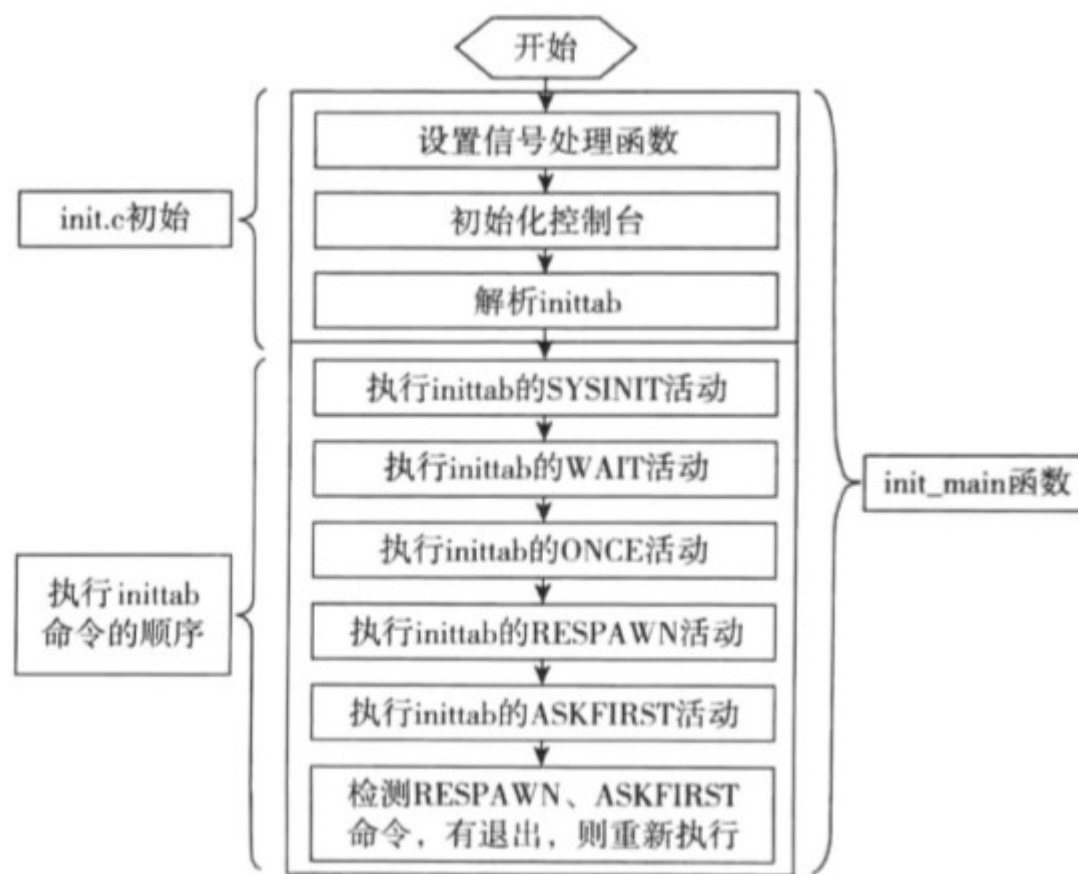


图 4-3 Busybox 的 init 进程初始化流程图

其中与构建根文件系统关系密切的是控制台的初始化、对 inittab 文件的解释及执行。从图中可以看出，Busybox init 进程的主函数是 init_main()，在其中设置了信号的处理函数，初始化控制台，以及解析 inittab 中内容的工作。

4.3 内核移植的实现

在 Linux 内核移植的初始阶段，应尽可能屏蔽不相关的设备驱动以及内核功能配置选项，使内核支持的选项尽可能的少，构造最小内核。在确保已经进行的内核移植操作正确的情况下逐步的添加相应的硬件支持和功能支持。

4.3.1 准备工作

建立工作目录，下载源码，安装交叉工具链，步骤如下：

```
# mkdir /root/build_kernel
# cd /root/build_kernel
# wget -c http://www.kernel.org/pub/linux/kernel/v2.6/linux2.6.29.tar.bz2
# tar jxvf linux2.6.29.tar.bz2
# export PATH=/home/shengzx/usr/local/arm/4.2.2-eabi/binPATH
```


4.3.2 修改顶层 Makefile

(1) 修改内核目录树根下的的 Makefile, 指明体系结构是 arm, 交叉编译工具是 arm-linux-gcc:

```
# vim Makefile
```

找到 ARCH 和 CROSS_COMPILE, 修改:

```
ARCH ?= arm
```

```
CROSS_COMPILE ?=/home/shengzx/usr/local/4.2.2-eabi/bin/ arm-linux-
```

添加对 S3C2410 处理器的支持。其代码如下:

```
ifeq ( #(CONFIG_ARCH_S3C2410), y)
```

```
TEXTADDR= 0xC0008000
```

```
MACHINE= s3c2410
```

```
endif
```

TEXTADDR 为内核解压的起始地址, 决定内核起始运行地址, 即内核映像应下载的位置, 根据开发板的电路设计, 这个地址是 0xC0008000。这里 0xC0008000 的含义是, 从地址 0xC0000000 开始, 总共 32M 字节的空间。

保存退出。

(2) compressed/Makefile

添加对本文开发板的支持。通过这个文件, 将从 vmLinux 创建一个压缩的 vmlinuz 镜像。

```
ifeq ( #(CONFIG_ARCH_S3C2410), y)
```

```
objs += head-s3c2410.o
```

```
Endif
```

4.3.3 设置 Flash 分区

内核是使用 MTD 驱动来实现分区功能的, Flash 的分区表是通过 mtd_partition 结构体来描述的。分区信息既可以通过引导程序向内核传递, 也可以直接在程序中定义。只有定义了分区表信息, 内核启动后才能正确挂载文件系统。

此处一共要修改 3 个文件, 分别是:

arch/arm/mach-s3c2410/devs.c	//指明分区信息
arch/arm/mach-s3c2410/mach-smdk2410.c	//指定启动时初始化
drivers/mtd/nand/s3c2410.c	//禁止 Flash ECC 校验

1. 指明分区信息

在 arch/arm/mach-s3c2410/devs.c 文件中:

```
# vim arch/arm/mach-s3c2410/devs.c
```

在 arch/arm/mach-s3c2410/devs.c 文件添加的内容包括:

(1) 添加包含头文件。

- (2) 建立 Nand Flash 分区表。
- (3) 加入分区信息
- (4) 建立 Nand Flash 芯片支持
- (5) 加入 Nand Flash 芯片支持到 Nand Flash 驱动

(1) 添加包含头文件。

```
#include <linux/mtd/partitions.h>
#include <linux/mtd/nand.h>
#include <asm/arch/nand.h>
```

...

(2) 建立 Nand Flash 分区表

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

```
static struct mtd_partition partition_info[] = {
    { /* 256kB */
        name: "boot",
        size: 0x00040000,
        offset: 0x0,
    }, { /* 1.75MB */
        name: "kernel",
        size: 0x001C0000,
        offset: 0x00040000,
    }, { /* 30MB */
        name: "root",
        size: 0x01e00000,
        offset: 0x00200000,
    }, { /* 32MB */
        name: "user",
        size: 0x02000000,
        offset: 0x02000000,
    }
};
```

};

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 boot, kernel, rootfs 以及以便以后扩展使用的用户文件系统空间。

(3) 加入 Nand Flash 分区

```
struct s3c2410_nand_set nandset = {
    nr_partitions: 4, /* 指明 partition_info 中定义的分区数目 */
    partitions: partition_info, /* 分区信息表 */
};
```



```
};
```

(4) 建立 Nand Flash 芯片支持

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

sets: 支持的分区集

nr_set:分区集的个数

(5) 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

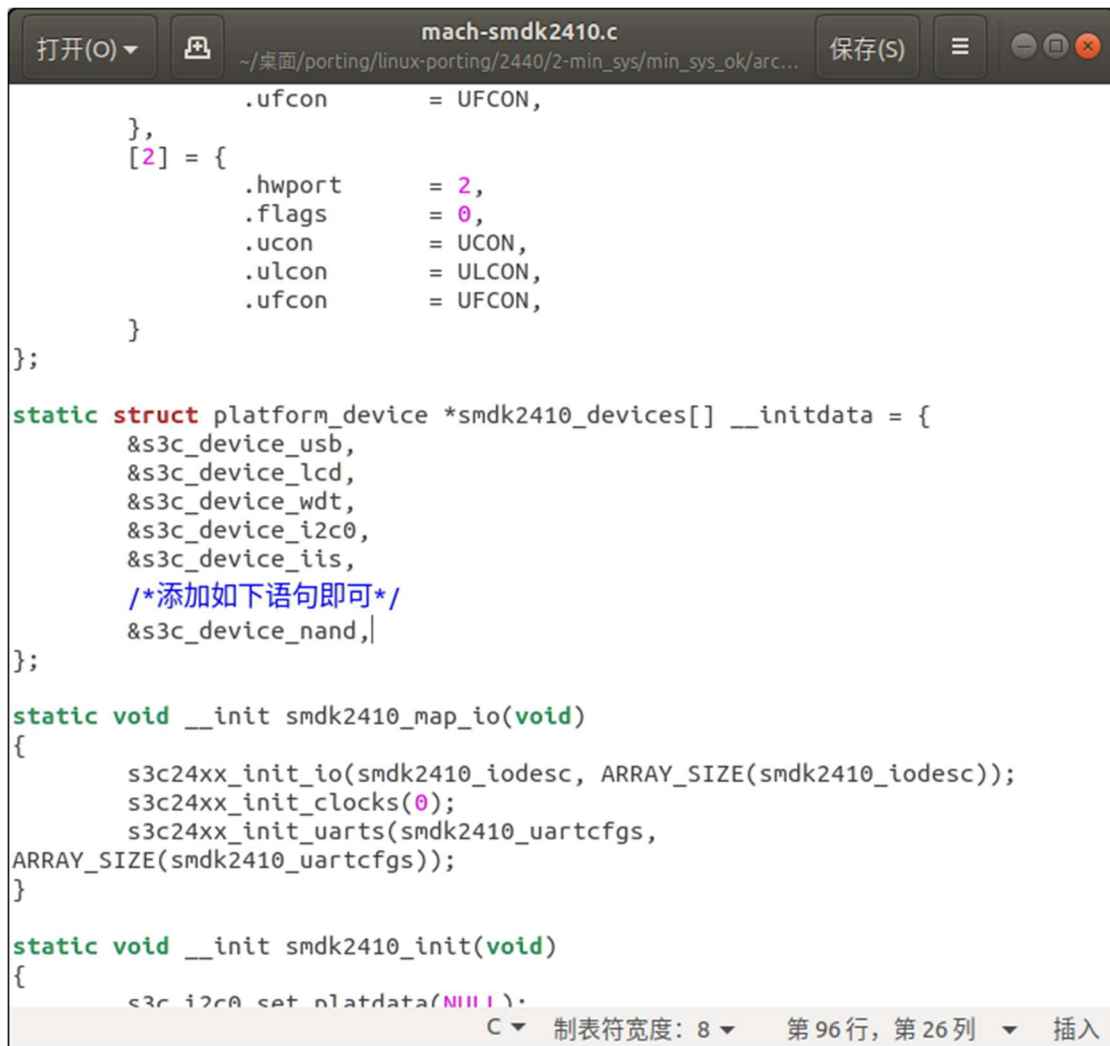
dev: 支持的 Nand Flash 设备

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。

```
# vim 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：



```
mach-smdk2410.c
~/桌面/porting/linux-porting/2440/2-min_sys/min_sys_ok/arc... 保存(S)

        .ufcon      = UFCON,
    },
    [2] = {
        .hwport      = 2,
        .flags        = 0,
        .ucon         = UCON,
        .ulcon        = ULCON,
        .ufcon        = UFCON,
    }
};

static struct platform_device *smdk2410_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    /*添加如下语句即可*/
    &s3c_device_nand,
};

static void __init smdk2410_map_io(void)
{
    s3c24xx_init_io(smdk2410_iodesc, ARRAY_SIZE(smdk2410_iodesc));
    s3c24xx_init_clocks(0);
    s3c24xx_init_uarts(smdk2410_uartcfgs,
ARRAY_SIZE(smdk2410_uartcfgs));
}

static void __init smdk2410_init(void)
{
    s3c_i2c0_set_platdata(NULL);
}
```

图 4-4 设置 flash 启动信息

保存,退出。

3. 禁止 Flash ECC 校验

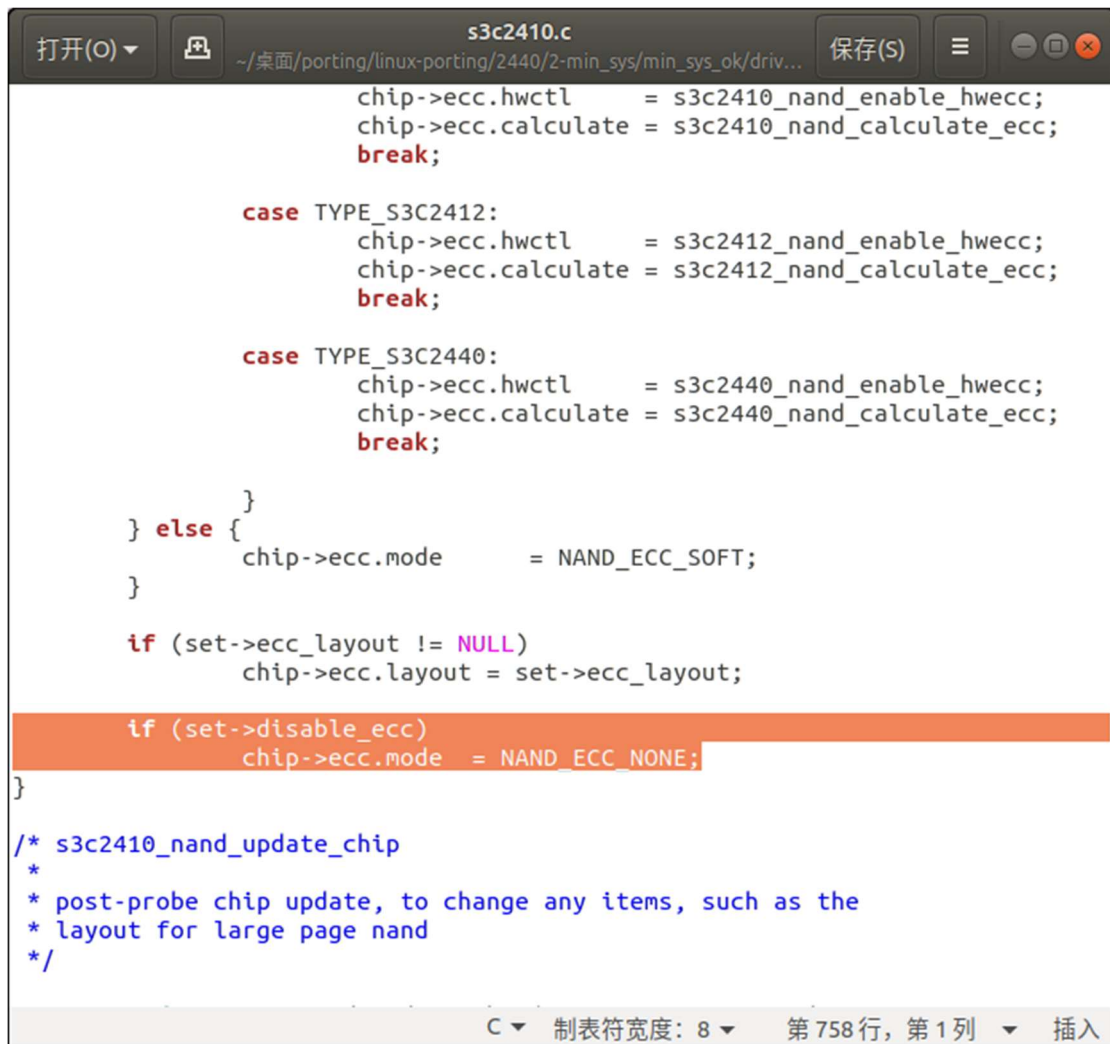
我们的内核都是通过 u-boot 写到 Nand Flash 的, u-boot 通过软件 ECC 算法产生 ECC 校验码, 这与内核校验的 ECC 码不一样, 内核中的 ECC 码是由 S3C2410 中 Nand Flash 控制器产生的。所以, 在这里选择禁止内核 ECC 校验。

修改 drivers/mtd/nand/s3c2410.c 文件:

vim 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;



```
s3c2410.c
~/桌面/porting/linux-porting/2440/2-min_sys/min_sys_ok/driv... 保存(S)

    chip->ecc.hwctl      = s3c2410_nand_enable_hwecc;
    chip->ecc.calculate = s3c2410_nand_calculate_ecc;
    break;

    case TYPE_S3C2412:
        chip->ecc.hwctl      = s3c2412_nand_enable_hwecc;
        chip->ecc.calculate = s3c2412_nand_calculate_ecc;
        break;

    case TYPE_S3C2440:
        chip->ecc.hwctl      = s3c2440_nand_enable_hwecc;
        chip->ecc.calculate = s3c2440_nand_calculate_ecc;
        break;

    } else {
        chip->ecc.mode      = NAND_ECC_SOFT;
    }

    if (set->ecc_layout != NULL)
        chip->ecc.layout = set->ecc_layout;

    if (set->disable_ecc)
        chip->ecc.mode = NAND_ECC_NONE;
}

/* s3c2410_nand_update_chip
 *
 * post-probe chip update, to change any items, such as the
 * layout for large page nand
 */
```

图 4-5 禁止 flash ECC 校验

至此，关于 flash 分区的设置全部完工。

4.3.4 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：

vim 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

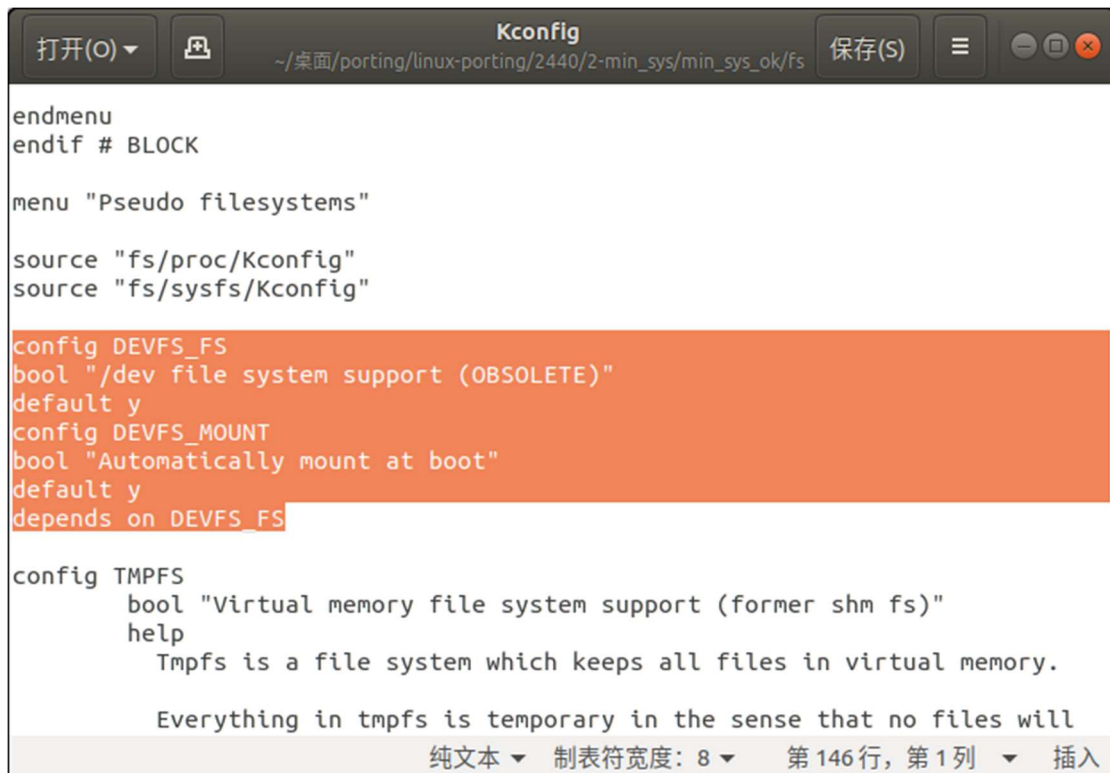


图 4-6 支持启动时挂载 devfs

4.3.5 配置、编译内核

- 编译内核之前首先要配置内核，配置内核的命令包括：

(1) make s3c2410_defconfig

这种方法将配置选项以命令行的形式列出，会恢复内核的默认配置。如果已经存在有.config 的配置文件，那么就会以该文件中的配置选项作为默认配置设置。

(2) make menuconfig

与 make config 类似，不过这种方法的显示方式是以菜单模式进行显示的。Make xconfig 使用鼠标选择对应的选项，make menuconfig 使用空格选择相应的选项，每个选项前的括号可以是 ()、< > 和 ()。中括号中要么是空格，要么是*，尖括号里可以是空格、*和 M；小括号里的内容是在所提供的几个选项里选择一个。空格表示不将该功能编译进内核，*表示将该功能编译进内核，M 表示将该功能编译成模块，在需要时将其动态插入到内核。

(3) make xconfig

用于 X Window 下的配置，将配置选项以图形菜单的形式显示出来。

在进行相应的配置的时候，有三种方式选择：

Y:将该功能编译进内核；

N:不将该功能编译进内核；

M:将该功能编译成可译载需要时动态插入到内核的模块。

`make xconfig` 使用鼠标选择对应的选项，`make menuconfig` 使用空格选择相应的选项，每个选项前的括号可以是[]、<>和()。中括号中要么是空格，要么是*，尖括号里可以是空格、*和M；小括号里的内容是在所提供的几个选项里选择一个。空格表示不将该功能编译进内核，*表示将该功能编译进内核，M表示将该功能编译成模块，在需要时将其动态插入到内核。

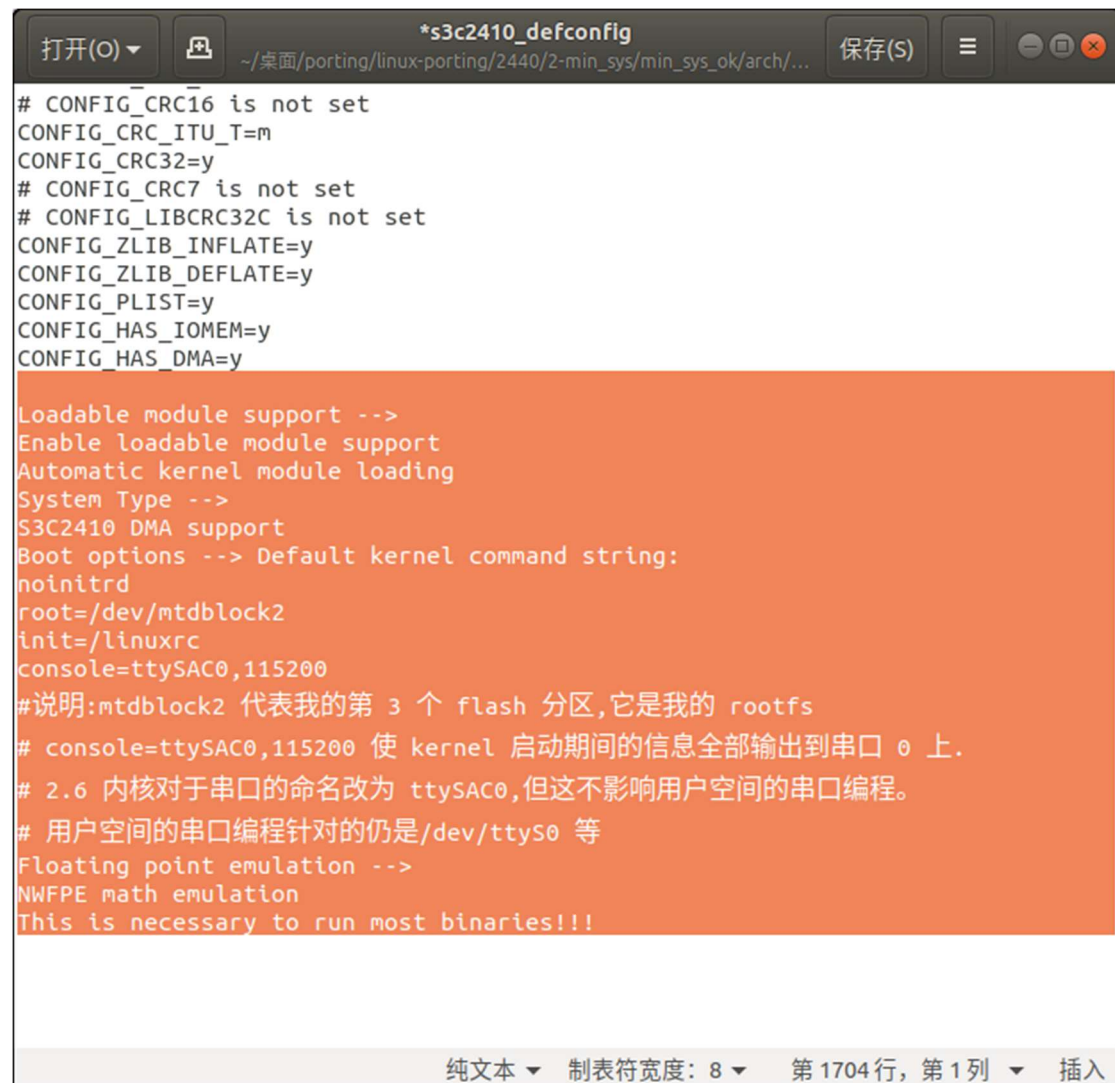
我的配置如下：

```
# cp arch/arm/configs/smdk2410_defconfig .config
```

```
# make menuconfig
```

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

这里约定“#”后面的是注释部分。

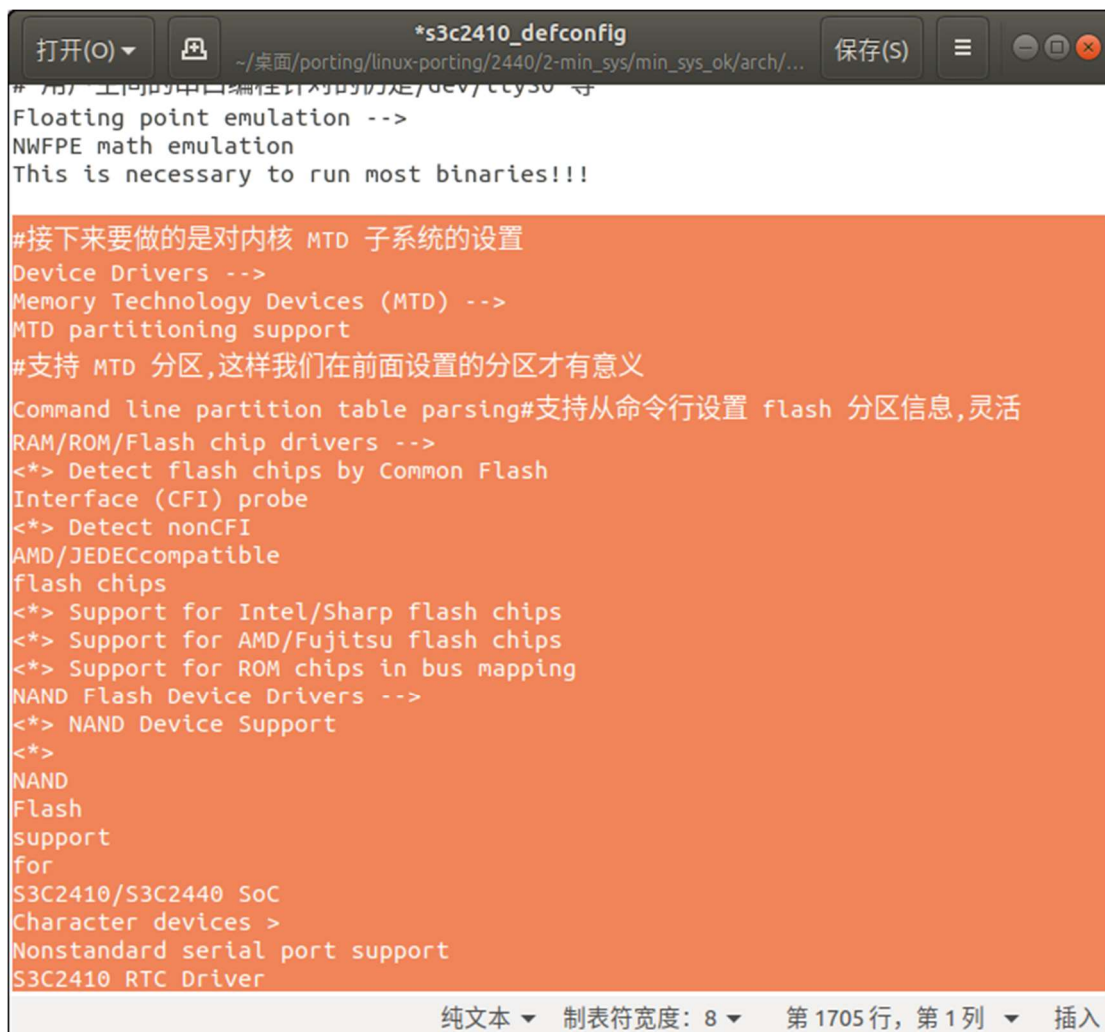


```
# CONFIG_CRC16 is not set
CONFIG_CRC_ITU_T=m
CONFIG_CRC32=y
# CONFIG_CRC7 is not set
# CONFIG_LIBCRC32C is not set
CONFIG_ZLIB_INFLATE=y
CONFIG_ZLIB_DEFLATE=y
CONFIG_PLIST=y
CONFIG_HAS_IOMEM=y
CONFIG_HAS_DMA=y

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!!!
```

图 4-7 增删内核配置项

接下来要做的是对内核 MTD 子系统的设置：



```
*s3c2410_defconfig
# 7.5) 上面的串口编程针对的是/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 nonCFI
AMD/JEDECcompatible
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 >
Nonstandard serial port support
S3C2410 RTC Driver
```

图 4-8 内核 MTD 子系统设置

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：



```
#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
```

图 4-9 针对文件系统设置

保存退出，产生.config 文件。

● 编译内核，生成可以移植到开发板的 uImage:

1. # make zImage

注意：若编译内核出现如下情况：

```
LD .tmp_vmlinux1
```

```
armlinuxld:
```

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

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

解决方法：修改 arch/arm/kernel/vmlinux.lds：

```
# vim arch/arm/kernel/vmlinux.lds
```

将文件尾 2 条的 ASSERT 注释掉：

```
/* 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。

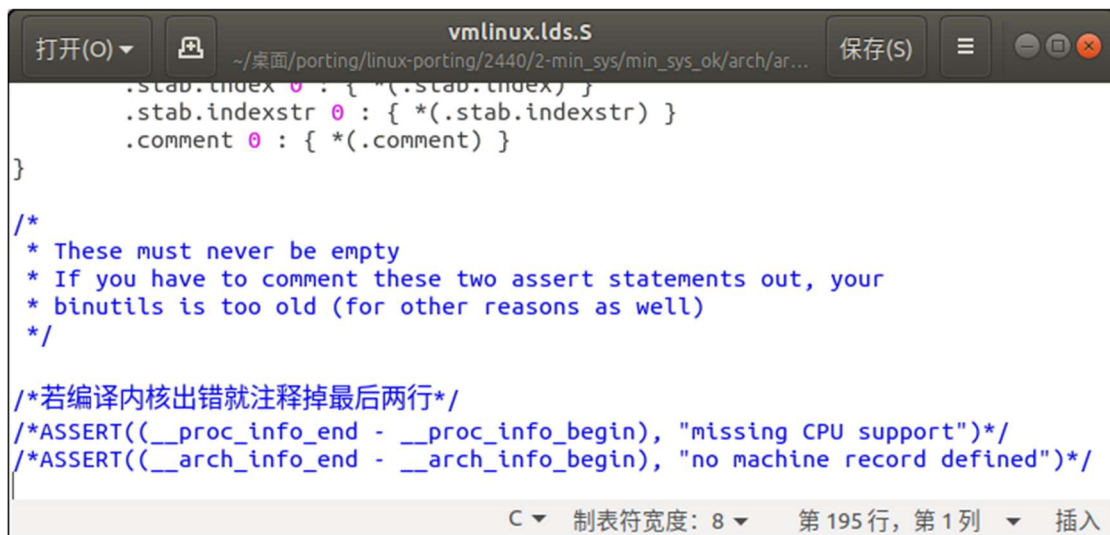


图 4-10 注释掉最后两行 ASSERT

在编译内核的命令中，当加上参数 uImage 后，则必须把编译好的 u-boot 文件下面的 mkimage 拷贝到 /root/usr/bin 下面否则会出现提示错误找不到 mkimage。

编译完成后会在 arch/arm/boot/目录下产生 zImage 内核映像。zImage 映像是可引导的，压缩的内核映像，就是以后要移植到开发板上的内核映像文件。

2. # make modules

此命令可以完成对所选的内核选项进行组件化编译，所需时间取决于所选的内核组件选项的多少。还可以使用命令：

```
# make modules SUBDIRS = drivers/x
```

将编译好的 u-boot.bin、uImage、root 文件系统放在 linux 系统下的 /tftpboot 下面以便烧写，tftp 和 nfs 配置在第二章已经讲过，配置好可以执行下面操作进行对内核、文件系统的烧写：

（1）移植 uboot:

连接好 JTAG 并口线，用 ./JLASH_2410EP_Nor /f:u-boot.bin;

（2）移植内核:

```
tftp 0x300080000 uImage /*下载内核到开发板*/  
bootm 0x300080000 /*启动内核*/
```

（3）移植文件系统：

```
tftp 0x30000000 root /*下载文件系统到开发板*/  
nand 0x30000000 0x200000 /*将 root 烧入开发板*/
```

4.4 本章小结

关于内核移植，主要介绍了 Linux 2.6 的新特性，内核的启动流程以及内核移植要修改的几个重点文件，在内核添加 Flash 分区信息，使内核的 mtd 驱动能识别 Flash 分区，最后，介绍了内核的配置和编译。最终在选用的开发板上移植了 u-boot 并成功加载了 Linux 2.6 内核，系统配置、编译正确，能够正常运行。

5 文件系统制作

5.1 用 Busybox 制作 yaffs2 根文件系统

原始的 linux 内核是不支持 yaffs2 文件系统的，我们首先需要下载 yaffs2 的内核补丁，给内核打上 yaffs2 补丁才能使内核支持该文件系统。

从相关网站下载下来源码之后，在根目录下存在一个 patch-ker.sh，这是一个给 linux 打补丁的脚本，打上这个补丁之后，内核的源代码就支持 yaffs2 了。

5.1.1 给内核打补丁

- yaffs 源代码下载完后，放到某个目录下(但不要放在内核目录下!)进入 yaffs2 源代码目录：

```
#cd yaffs2
```

打补丁（注意参数顺序不能错）：

```
#!/patch-kernel.sh c m ../linux-2.6.29
```

- 然后配置内核：

```
#cd ../linux-2.6.29 //返回内核根目录
```

```
# make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

File systems -->

Miscellaneous filesystems -->

<*> YAFFS2 file system support

- 再重新编译内核：

```
# make ARCH=arm CROSS_COMPILE=arm-linux-
```

然后制作新的 uImage，加载或者烧写到 FLASH，如果能正确引导并加载 yaffs 文件系统则移植成功。

需要注意：因为 windows 中下载导致文件编码和 linux 的不同。所以，如果是在 windows 下用 git 下载并传递到 linux 下的，则需要修改两个文件的编码：

(1) 在 linux 下，进入 yaffs2 源代码目录

```
#vim patch-kernel.sh
```

然后在 vim 中执行如下命令：

```
:set ff=unix
```

保存退出

(2) 然后修改权限使 patch-kernel.sh 具有可执行权限：

```
#chmod -R 777 patch-kernel.sh
```

(3)修改 fs/yaffs2/Kconfig 的编码:

```
#vim fs/yaffs2/Kconfig
```

在 vim 中执行命令:

```
:set ff=unix
```

保存退出, 按照上述步骤进行打补丁。

5.1.2 用 Busybox 制作 yaffs2 根文件系统

1. 根文件系统的目录结构

表 5-1 根文件系统目录结构

目录	说明
bin	存放所有用户都可以使用的、基本的命令
sbin	存放的是基本的系统命令, 它们用于启动系统、修复系统等
usr	里面存放的是共享、只读的程序和数据
proc	这是个空目录, 常作为 proc 文件系统的挂载点
dev	该目录存放设备文件和其它特殊文件
etc	存放系统配置文件, 包括启动文件
lib	存放共享库和可加载块(即驱动程序), 共享库用于启动系统、运行根文件系统中的可执行程序
boot	引导加载程序使用的静态文件
home	用户主目录, 包括供服务账号锁使用的主目录, 如 FTP
mnt	用于临时挂接某个文件系统的挂接点, 通常是空目录。也可以在里面创建空的子目录
opt	给主机额外安装软件所摆放的目录
root	root 用户的主目录
tmp	存放临时文件, 通常是空目录
var	存放可变的数据

2. 建立根文件系统的目录

进入工作目录, 创建一个 shell 的脚本用于构建根文件系统的各个目录: mkrootfs.sh, 并且改变执行的权限。

```
# chmod 777 mkrootfs.sh
```

脚本内容如下:



```
#!/bin/sh
echo "-----Create rootfs directons start...-----"
mkdir rootfs
cd rootfs
echo "-----Create root,dev....-----"
mkdir root dev etc boot tmp var sys proc lib mnt home
mkdir etc/init.d etc/rc.d etc/sysconfig
mkdir usr/sbin usr/bin usr/lib usr/modules
mkdir proc/sys mkdir proc/sys/kernel
mkdir proc/sys/kernel/
echo "make node in dev/console dev/null"mknod -m 600 dev/console c 5 1
mknod -m 600 dev/null c 1 3
mkdir mnt/etc mnt/jffs2 mnt/yaffs mnt/data mnt/temp
mkdir var/lib var/lock var/run var/tmp
chmod 1777 tmp
chmod 1777 var/tmp
echo "-----make direction done-----"
```

图 5-1 mkrootfs.sh

改变了 tmp 目录的使用权，让它开启 sticky 位，为 tmp 目录的使用权开启此位，可确保 tmp 目录底下建立的文件，只有建立它的用户有权删除。

3. 编译、安装 Busybox

Busybox 是一个遵循 GPL v2 协议的开源项目，它在编写过程总对文件大小进行优化，并考虑了系统资源有限(比如内存等)的情况，使用 Busybox 可以自动生成根文件系统所需的 bin、sbin、usr 目录和 linuxrc 文件。

首先下载 busybox，下载地址：www.busybox.net

下载链接：<http://www.busybox.net/downloads/busybox-1.18.3.tar.bz2>

解压源代码：

```
#tar -jxvf busybox-1.18.3.tar.bz2
```

修改 Makefile 中的交叉链和系统架构：

```
CROSS_COMPILE ?=arm-linux-
```

```
ARCH ?=arm
```

配置编译选项：

如果修改了 Makefile 则使用如下命令：

```
#make menuconfig
```

如果未修改 Makefile 则使用如下命令：

```
# make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

(1) 指定安装位置：

Busybox Settings --->

Installation Options ("make install" behavior) --->

BusyBox installation prefix-->

输入：../rootfs //实际中，要根据计划的文件系统根设定！

(2) 指定 mdev 动态文件系统:

Linux System Utilities --->

[*]Support /etc/mdev.conf

[*]Support command execution at device addition/removal

编译 busybox:

```
# make ARCH=arm CROSS_COMPILE=arm-linux- install
```

在 rootfs 目录下会生成目录 bin、sbin、usr 和文件 linuxrc 的内容。

4. 建立 etc 目录

init 进程根据/etc/inittab 文件来创建其他的子进程，比如调用脚本文件配置 IP 地址，挂载其他的文件系统，最后启动 shell 等。

(1)、拷贝主机 etc 目录下的 passwd、group、shadow 文件到 rootfs/etc 目录下。

(2) etc/sysconfig 目录下新建文件 HOSTNAME，内容为” smdk2410”。

(3) etc/inittab 文件:

仿照 Busybox 的 examples/inittab 文件，在 etc/目录下创建一个 inittab 文件:



图 5-2 inittab 文件

5. 创建 ext/init.d/rcS 文件

使用如下脚本文件，可以在里面添加自动执行的命令:

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S //运行的级别
prevlevel=N
umask 022 //文件夹的掩码
export PATH runlevel prevlevel
mount -a //挂载/etc/fstab/文件指定的所有的文件系统
mount -t tmpfs none /tmp
mount -t tmpfs none /var
mkdir -p /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev>/proc/sys/kernel/hotplug
mdev -s
/bin/hostname -F /etc/sysconfig/HOSTNAME //主机的名字
```

图 5-3 创建 ext/init.d/rcS 文件的脚本

最后，还要改变它的属性，使它能够运行：

```
# chmod -R 777 etc/init.d/rcS
```

6. 创建 etc/fstab 文件

内容如下，表示执行完，“mount -a”命令后将挂载 proc，tmpfs 等包含在该文件中的所有的文件系统：

```
#device mount-point type options dump fsck order
proc      /proc          proc   defaults 0 0
sysfs     /sys           sysfs  defaults 0 0
tmpfs     /tmp            tmpfs  defaults 0 0
tmpfs     /var           tmpfs  defaults 0 0
tmpfs     /dev           tmpfs  defaults 0 0
```

/etc/fstab/文件被用来定义文件系统的“静态信息”，这些信息被用来控制 mount 命令的行为，各个字段的含义以如下：

表 5-2 etc/fstab 文件字段说明

字段	含义
device	要挂载的设备。比如/dev/hda2 /dev/mtdblock1 等设备文件，也可以是其他格式的，比如对于 proc 文件系统这个字段就没有意义，可以就任意的值，对于 NFS 文件系统，这个字段是,<host>:<dir>.
proc	文件系统这个字段就没有意义，可以就任意的值，对于 NFS 文件系统，这个字段是,<host>:<dir>.
mount-point	挂载点
type	文件系统类型。比如 proc,jffs2,yaffs,ext2,nfs 等，也可以是 auto，表示自动检测文件系统类型
options	挂接参数，以逗号隔开

/etc /fstab 的作用不仅仅是用来控制“mount -a”的行为，即使是一般的 mount 命令，也受它的控制，常用的取值还有：

auto noauto user: 只允许普通用户挂载设备

nouser exec: 允许运行所挂载设备上的程序

noexec Ro: 只读方式

rw: 以读写的方式

sync: 修改文件时，它会同步写入设备中

async: 不同步

defaults rw suid dev exec auto nouser async 等的组合。

dump 是一个用来备份的文件的程序，fsck 是一个用来检查磁盘的程序，

7. 创建 etc/profile 文件



```
#Ash profile
#vim:syntax=sh
#No core file by defaults
#ulimit -S -c 0>/dev/null 2>&1
USER="id -un"
LOGNAME=$USER
PS1='[\u@\h\W]#'
PATH=$PATH
HOSTNAME='/bin/hostname'
export USER LOGNAME PS1 PATH
```

图 5-4 etc/profile 文件内容

8. 制作根文件系统映像文件

我的目标板的 NandFlash 是 64MB 的，所以要使用 mkyaffs2image 的 64M 版本这个可执行的文件生成映像文件。

使用命令：

```
# mkyaffs2image rootfs rootfs.img
```

生成根文件系统映像文件。把生成的 rootfs.img 文件烧写到 nandFlash 中的根文件系统区。重新引导操作系统即可实现文件系统的正确挂载。

5.2 移植过程中遇到的问题及处理

- 如果出现 “Kernel panic - not syncing: Attempted to kill init!” 错误，则在编译内核时选择 EABI 支持：

Kernel Features --->

[*] Use the ARM EABI to compile the kernel

[*] Allow old ABI binaries to run with this kernel (EXPERIMENTA)

把这个选上重新编译就可以了。如果文件系统镜像也是新做的则也要考虑文件系统本身有问题的可能性。此外要注意 mkyaffs2image 工具是否正确，这和产商提供的工具有关。

- 如果出现 “Failed_to_execute_/linuxrc” 可以根据下面的建议逐个检查：

1. bin/busybox 文件是可以执行的。
2. 在配置 busybox 的时候要选中 shell 选项中的一个选项
3. linuxrc 是可执行的。
4. 制作文件系统的时候利用的工具也要留意区分：

mkcramfs 制作 cramfs 镜像的工具

mkimage	制作 jffs2 镜像的工具
mkyaffs2image	制作 2.6 的 yaffs2 的镜像工具(针对的 Nand Flash 是 128MB 到 1G 的)
mkyaffsimage	制作 2.6 的 yaffs2 的镜像工具
mkyaffsimage_2	制作 2.6.29 或 2.6.30 或更高版本内核的 yaffs2 的镜像工具(针对的 Nand Flash 是 64MB 的)

5、配置内核时是否取消 ECC(我遇到此问题是通过取消 ECC 解决此问题。)

5.3 本章小结

本章重点介绍了内核支持的 yaffs 文件格式，其实我们内核能够支持的文件系统有很多种，本论文研究选择 yaffs 文件格式进行编译烧写，待以后有时间再对其它的文件系统进行研究。

6 测试

6.1 常用简单测试方法介绍

将编译好的可执行文件下载到目标板目前主要四种方式：

1. 复制到介质(如优盘)
2. 通过网络传送文件到开发板
3. 通过串口传送文件到开发板
4. 通过 NFS 挂载(网络文件系统)

介绍部分测试方法：

1. 使用优盘

先把编译好的可执行程序复制到优盘，再把优盘插到目标板上并挂载它，然后把优盘插入到开发板的 USB Host 接口，优盘会自动挂载到/udisk 目录，通过命令运行 hello.c。

2. 使用 ftp 传送文件

使用 ftp 登录目标板，把编译好的程序上传；然后修改上传后目标板上的程序的可执行属性，并执行(put 命令传送文件)。

3. 通过网络文件系统 NFS 执行

Linux 中最常用的方法就是采用 NFS 来执行各种程序，这样可以不必花费很多时间下载程序，虽然在此下载 hello 程序用不了多久，应用程序如果越来越大，就会发现使用 NFS 运行的方便所在，所以推荐使用。

6.2 编写简单的 C 程序测试移植的系统

用编辑器编写一个简单的 C 程序如下：



```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    printf("Thanks Everybody!\n");
    return 0;
}
```

图 6-1 helloworld 测试程序

编写好之后保存退出 hello.c，执行以下命令：

```
# arm-linux-gcc -o hello hello.c （生成可运行的二进制 hello 文件）  
# cp hello /udisk
```

```
shengzx@shengzx-All-Series:~$ arm-linux-gnueabi-gcc -o hello hello.c  
shengzx@shengzx-All-Series:~$ cp hello /udisk
```

图 6-2 生成测试程序

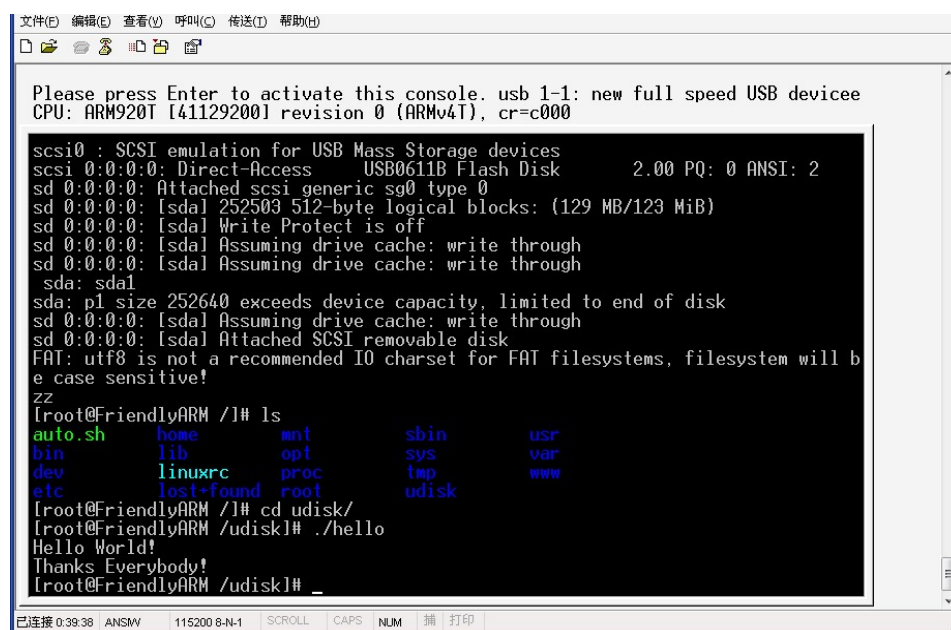
将会生成一个可以在开发板运行的二进制文件 hello，将 hello 拷贝到 U 盘以便测试。

6.3 在开发板执行测试程序

将 U 盘插入开发板的 USB 接口，并在终端下面执行如下命令：

```
# cd /udisk  
# ./hello
```

此时可以在终端下看到如下测试信息：



```
Please press Enter to activate this console. usb 1-1: new full speed USB device  
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c000  
  
scsi0 : SCSI emulation for USB Mass Storage devices  
scsi 0:0:0:0: Direct-Access    USB0611B Flash Disk        2.00 PQ: 0 ANSI: 2  
sd 0:0:0:0: Attached scsi generic sg0 type 0  
sd 0:0:0:0: [sdal] 252503 512-byte logical blocks: (129 MB/129 MiB)  
sd 0:0:0:0: [sdal] Write Protect is off  
sd 0:0:0:0: [sdal] Assuming drive cache: write through  
sd 0:0:0:0: [sdal] Assuming drive cache: write through  
sda: sdal  
sda: p1 size 252640 exceeds device capacity, limited to end of disk  
sd 0:0:0:0: [sdal] Assuming drive cache: write through  
sd 0:0:0:0: [sdal] Attached SCSI removable disk  
FAT: utf8 is not a recommended IO charset for FAT filesystems, filesystem will be case sensitive!  
ZZ  
[root@FriendlyARM /]# ls  
auto.sh  home  mnt      sbin     usr  
bin      lib   opt      sys      var  
dev      linuxrc  proc    tmp      www  
etc      lost+found  root   udisk  
  
[root@FriendlyARM /]# cd /udisk/  
[root@FriendlyARM /udisk]# ./hello  
Hello World!  
Thanks Everybody!  
[root@FriendlyARM /udisk]# _
```

图 6-3 测试结果

在终端下面可以看到 hello.c 在开发板上实现的内容，这说明内核移植成功，故在此基础上可以进行关于嵌入式的应用开发。

7 结论

本课题研究了 Linux 系统移植理论、探索了嵌入式软件系统构建的方法、实践了软件系统构建的整个过程，最终在以 S3C2410A 为核心的硬件平台上，搭建了完整的嵌入式 Linux 软件开发平台。

主要完成了以下工作：下载、配置编译了交叉工具链，在主机上搭建了嵌入式 Linux 交叉开发环境；详细分析了 BootLoader 的启动过程，深入地研究了 BootLoader 从 NOR Flash 启动的原理，在此基础上，对 u-boot 进行了移植并实现了从 NOR Flash 启动；分析了 Linux2.6 内核的启动流程，移植、配置编译了 2.6 版 Linux 内核。最后在移植的内核基础上再移植了一个 yaffs 文件系统。

编写简单的 HelloWorld 测试程序，并烧写到了目标板上，验证了该嵌入式 Linux 软件平台已基本建立，并可用于二次开发。

在本文的研究成果上，可以直接进行应用程序开发，大大加快了嵌入式系统的开发过程。

在本项目的实践过程中，对嵌入式软件系统的整体结构和具体实现有了更深的理解，为日后做嵌入式系统开发和制定嵌入式系统实现方案积累了宝贵的经验。很多嵌入式开发人员因对嵌入式软件系统没有整体上的认识，掌握的知识过于片面，所以在遇到问题时不能快速准确的找到问题的根源。本人虽已基本完成对嵌入式 Linux 系统的移植，达到了课题的预期目的，但由于时间等因素的限制，仍存在一些问题有待进一步研究，如：

1. 提高移植后 Linux 内核的稳定性，增强内核的实时性；
2. 编译安装 Qt/Embedded 库和 Qtopia；
3. 开发嵌入式应用程序，如数据库，GPS 导航；
4. 发挥 ARM 处理器体积小，低功耗，高性能的优势，设计实现高端智能手持设备等。

8 参考文献

- [1].于明, 范书瑞, 曾详烨.ARM9 嵌入式系统设计与开发教程[M].北京.电子工业出版社.
- [2].魏平等.Linux 体系结构及嵌入式 Linux 的移植方法[J].东南大学学报 (自然科学版).
- [3].青静.嵌入式系统设计与开发实例详解[M].北京.北京航空航天大学出版社.
- [4].杨延军.用 busybox 制作嵌入式 Linux 的文件系统[C].单片机与嵌入式系统.
- [5].嵌入式 Linux 系统开发 标准教程 (第二版) [M]武汉.华清远见嵌入式培训中心.
- [6].韦东山. 嵌入式 linux 应用开发完全手册[M]. 北京.人民邮电出版社.
- [7].s3c2410 yaffs 制作文件系统.Doc[C]. 百度文库资料.
<http://wenku.baidu.com/view/ed5cdc84b9d528ea81c77912.html>.
- [8].最新内核 linux2.6.33 移植到 S3C2410[C]. 百度文库资料.
<http://wenku.baidu.com/view/f04fe5fdc8d376eeaeaa317f.html>.
- [9].刘兵. linux 实用教程[M]. 北京.中国水利水电出版社.
- [10].于明、范书瑞、曾详烨. ARM9 嵌入式系统设计与开发教程[M].北京.电子工业出版社, 2006.
- [11].魏平等. Linux 体系结构及嵌入式 Linux 的移植方法[J].东南大学学报.
- [12].范展源、刘韬. 深度实践嵌入式 Linux 系统移植[M].北京.机械工业出版社.