“十二五”
国家重点图书出版规划项目

ARM公司鼎力推荐

嵌入式Linux开发实用教程

朱兆祺 李强 袁晋蓉 编著

U-boot-2013.04

Linux-3.8.3

Qt-4.8.4

设备驱动

基于ARM11，深入浅出

U-Boot-2013.04 + Linux-3.8.3 + Qt-4.8.4

立足初学者，快速入门

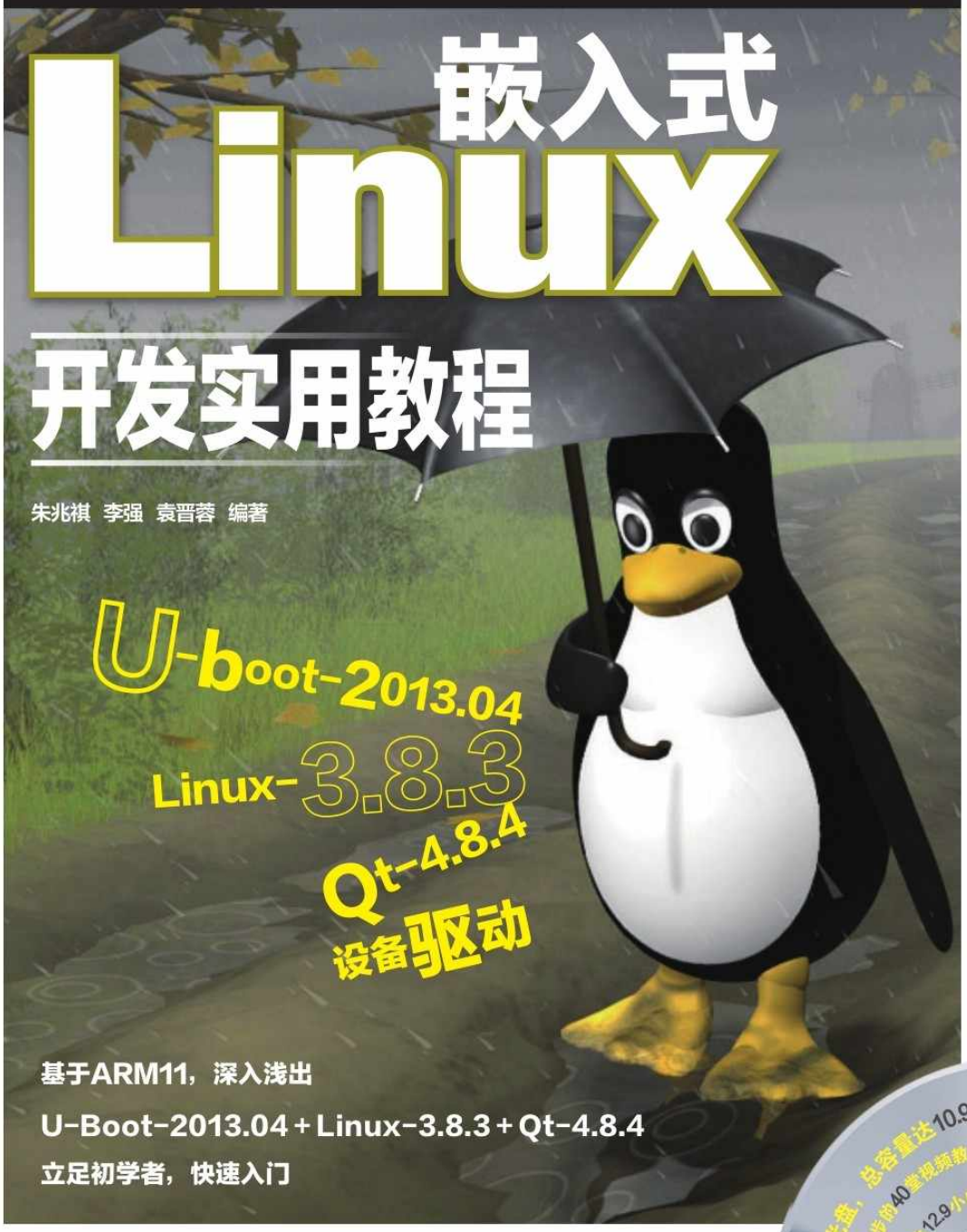
人民邮电出版社
POSTS & TELECOM PRESS

“十二五”
国家重点图书出版规划项目

ARM公司鼎力推荐

嵌入式Linux开发实用教程

朱兆祺 李强 袁晋蓉 编著



U-boot-2013.04

Linux-3.8.3

Qt-4.8.4

设备驱动

基于ARM11，深入浅出

U-Boot-2013.04＋Linux-3.8.3＋Qt-4.8.4

立足初学者，快速入门

多媒体教学光盘，总容量达10.9GB

与本书全程同步的40集视频教学课

总时长达12.9小时

人民邮电出版社
POSTS & TELECOM PRESS

目　录

嵌入式Linux开发实用教程

朱兆祺　李强　袁晋蓉　编著

人民邮电出版社

北京

来自ARM的问候与推荐

ARM Holdings 是全球领先的半导体知识产权（IP）提供商，并因此在数字电子产品的开发中处于核心地位。ARM 的总部位于英国剑桥，2000 多名员工分布在全球多个国家和地区。ARM 公司成立于1990年，目前已有超过250家公司在ARM处理器IP的基础上开发出了数以百计的各类芯片，至今已累计出货超过300亿颗，平均算下来地球上每个人都可以分得4颗ARM“芯”。由于ARM“芯”在各领域的广泛应用及ARM生态中丰富的资源，目前基本上所有的主流操作系统都提供了对ARM架构CPU的支持。目前，ARM技术已在90%的智能手机、80%的数码相机以及28%的电子设备中得到应用。

很高兴看到本书的出版。在嵌入式开发和教学中，软件的比重无疑变得越来越大。不同于PC上的软件开发，嵌入式软件开发者需要对硬件平台和操作系统具有一定的了解。对硬件平台和操作系统的选择经常困扰着很多人特别是初学者。本书作者结合自己的点滴经验为读者们做出了一个很好的范例。三星公司推出的基于 ARM 1176JZF-S 内核的 S3C6410 处理器，直至今日在工程项目和教学实践中仍被广泛采用；软件方面，Linux仍是嵌入式系统中的首选操作系统之一。

本书循序渐进，从Linux基础开始，覆盖了U-Boot移植、Linux移植、驱动开发等方面，并在最后以一个实际的系统设计为例，进行实战演练，对全书的内容进行巩固。

书的目标是帮助初学者快速进入嵌入式Linux学习的大门，听闻已有高校准备采用本书作为实验课教程，相信广大的同学和嵌入式的爱

好者们一定能够从本书中获益。也预祝您在嵌入式的学习和开发中获得更多的乐趣并取得成功。

时昕 博士
ARM公司中国区大学计划经理
2013年12月

推荐序

随着平板电脑与手机，乃至网络化电视等智能化电子产品的蓬勃发展，嵌入式系统及其应用获得了众多企业的青睐，以 ARM+Android 的嵌入式系统成为当今 IT 领域最热门的技术之一。Android是基于Linux内核的操作系统，要掌握Android的开发与应用，当然要先学好嵌入式Linux。但是嵌入式Linux是一门非常复杂的软件技术，入门较难，初学者在自学过程常常感到困惑，导致无法掌握，甚至不得不半途而废。

虽然讲授嵌入式Linux的书千千万万，但多数是专家、学者们的专著，或者是培训机构的教材。而这本书则是以一个嵌入式Linux学习者的角度，总结在自学嵌入式Linux过程中的种种体会，也是为众多苦苦跋涉在嵌入式Linux学习途中的自学者，描述成功入门的捷径。

学习嵌入式Linux目的是为了应用，因而作者从U-Boot移植入手，为初学者剖析U-Boot移植的难题，进而学习Linux驱动程序，然后通过Qt图形用户界面应用程序框架的学习，告诉初学者如何建立图形用户界面，以及实现嵌入式Linux在ARM系统中的应用。

特别要指出的是，该书的两位年轻作者，在江西理工大学自动化专业读书期间，专业课程中并没有关于Linux的课程，但是他们却能够独立进行研读Linux并对学习经验进行总结，为这本书今天的成型奠定基础。这不仅凝聚了他们在课余无数个日夜学习的艰辛，也说明了高等工科教育改革的成功。因为从入学起，和许许多多专业学生不一样的是，他们第一个学期已经开始学习“从晶体管到单片机”，第二个

学期已经学完了ARM嵌入式系统与μC/OS-II嵌入式操作系统。早期工程教育为后三年的“基于项目的学习”打下了坚实基础。

2006年起，学校与国内著名的嵌入式系统企业——广州周立功单片机科技有限公司通力合作，启动了“3+1”创新教育改革。在“面向工程、项目驱动、能力培养、全面发展”的教育改革理念的指导下，探索实施有效的高等工程教育的新路。每个自动化专业学生在国家级人才培养模式创新试验区（配备全套的计算机、电子仪器及嵌入式系统开发平台），可以日以继夜地学习自动化与嵌入式系统技术，暑期再前往公司强化嵌入式技术能力，大四再到企业进行一年的嵌入式系统工程实训（国家卓越工程师教育培养计划）。所以说，虽然他们当时不过是大三和大四的学生，却拥有在嵌入式技术领域3～4年的实践经验，在这个年轻与日新月异的技术领域，可以说是熟手了。

王祖麟

江西理工大学电气学院副院长

江西理工大学“3+1”创新教育创始人

2013年12月

前言

2012年11月，当我看到论坛中的同龄大学生在学习嵌入式Linux寸步难行，我就计划将我学习嵌入式Linux的点点滴滴记录下来，从一个学生的角度去写，或许更能让初学者接受。2013年1月，当写完初稿再重新审视的时候，总感觉不尽如意。2013年3月，我联系了我的师弟李强，两人打算以一个全新的思维重新完成这本书。

2013年6月，书稿终于定型。

本书一共有6章，从Linux指令基础到Linux常用软件；从U-Boot移植到Linux移植；从Linux驱动程序设计到Qt应用程序设计，全方位解析作为一个初学者该如何快速踏入嵌入式Linux学习的大门。

这本书大体结构如下：

第1章嵌入式Linux基础，为了让还没有接触过或者不太熟悉Linux的读者进一步认识Linux，介绍了两个在嵌入式Linux学习中使用频率很高的软件。有了这一章的知识作铺垫，后续的学习将更加顺畅。

第2章U-Boot-2013.04 分析与移植，本章覆盖 U-Boot 启动分析、SD 卡启动、NAND Flash启动移植、DM9000 网卡移植等内容。笔者从SD卡启动到NAND Flash 启动，解开众多厂家不愿公开的技术点。对于初学者来说，U-Boot的移植无疑是一座大山，笔者将一步步揭开U-Boot的神秘面纱。

第3章Linux-3.8.3 内核移植，本书采用最新内核，涉及Linux 内核分析、NAND Flash 移植、DM9000网卡移植、LCD液晶屏移植、

YAFFS2文件系统制作等知识。从OK6410的内核移植，让初学者对Linux有个较为深入的了解和认识。

第4章Linux设备驱动程序设计，笔者截取了较为经典的字符设备驱动和块设备驱动程序对这部分知识进行讲解，给初学者在往后学习Linux设备驱动知识和从事Linux设备驱动工程师奠定扎实的基础。

第5章Qt-4.8.4移植，Qt4.8.4在Qt的发展具有重要地位，本章将带领读者将Qt4.8.4版本移植到OK6410开发板以及学习Qt程序的编写方法。

第6章嵌入式Linux学习拓展，笔者将前5章知识进行进一步拓展，所谓温故而知新、举一反三。

本书根据6章的内容分别录制了视频，联合OK6410-A开发板进行实验，一步一步带领读者深入学习。书中每一节内容都已经标注相对应的视频位置，请读者自行观看。

通过本书的学习，作者不能保证每一位读者都能成为嵌入式高手；但是我相信，一定可以带初学者进入嵌入式的大门。

完成本书的学习其实很简单：将少买一件衣服的钱买一块开发板，将每天玩游戏的1小时用于跟随本书一步步进行学习，我相信，3个月之后，你一定可以成功跨入嵌入式的大门。

在此感谢江西理工大学王祖麟教授大学四年对我的言传身教，并为本书作序；感谢 ARM 公司中国区大学计划经理时昕博士为本书撰写推荐序；感谢我的父母22年来对我含辛茹苦的培养；感谢我女朋友对我一直以来的关心和照顾。参与本书创作的还有谢贤斌、温如春、吴银凤、刘晖、张子明（飞凌嵌入式工程师），为本书做出宣传的电子发烧友陈锋和钱珊珊，在此对他们一并表示感谢。

笔者能力有限，如果有错误之处，还请各位读者指出。笔者邮箱：jxlgzzq@163.com 和jxustlq@163.com。笔者在2013年1月建立了嵌入式Linux学习手册QQ群：284013595、271641475。欢迎各位读者加

入群进行学习讨论。有关嵌入式Linux实用教程的相关视频、资料、软件、源代码、程序和C语言学习资料将在以下百度网盘中进行更新。

朱兆祺

2013年12月

第1章 嵌入式Linux基础

1.1 Linux基本命令

在学习嵌入式Linux开发的过程中，将经常使用到Linux的操作命令。实际上，Linux系统中的命令也是为实现特定的功能而编写的，而且绝大数的命令是用C语言编写的。有些实用性强的程序被广泛使用和传播，逐渐地演变成Linux的标准命令。但是Linux的操作命令繁多，本节将在U-Boot、Linux移植过程中常用到的Linux操作命令罗列出来进行讲解，为后续的学习做良好的铺垫。读者不要认为这是Linux简单命令则不屑一顾，嵌入式Linux学习是一个漫长的过程，循序渐进方能有所成就，这个过程是由每一小步累加而成的。天下难事，必作于易；天下大事，必作于细。所以读者务必要对待学习的每一个细节。

1.1.1 文件属性查询与修改

1．文件属性查询

“ls”命令在Linux目录中占据着重要地位，主要用于查看文件属性、查看目录下所包含的文件等。

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/busybox-1.20.2/_install$ ls
bin      dev  home  linuxrc  proc  sbin  tmp  var
creat_yaffs2.sh  etc  lib  mnt  root  sys  usr
```

通过“ls”命令可查看_install目录下有哪些东西。如果要进一步查看文件属性，则使用“ll”命令或者“ls -al”命令，这两个命令是等效的。

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/busybox-1.20.2/_install$ ll
总用量 64
drwxr-xr-x 15 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 .
drwxr-xr-x 35 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 15:34 ..
drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 15:34 bin
-rw-r--r-- 1 zhuzhaoqi zhuzhaoqi 393 2013-03-17 16:32 creat_yaffs2.sh
drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 dev
drwxr-xr-x 3 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 21:01 etc
drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 home
drwxr-xr-x 3 zhuzhaoqi zhuzhaoqi 4096 2013-03-18 09:57 lib
lrwxrwxrwx 1 zhuzhaoqi zhuzhaoqi 11 2013-03-17 15:34 linuxrc -> bin/busybox
drwxr-xr-x 5 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 mnt
drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 proc
```

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 root

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 15:34 sbin

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 sys

drwxrwxrwx 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 tmp

drwxr-xr-x 7 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 usr

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 var

这样每一个文件的属性将一目了然，而属性中的每一个数据都有特定的含义，如表1.1所示。

表1.1 文件属性含义

drwxr-xr-x	2	zhuzhaoqi	zhuzhaoqi	4096	2013-03-17 15:34	bin
文件权限	连接数	文件所有者	文件所属用户组	文件大小	文件最后一次被修改的时间	文件名称

其中文件权限的10个字符的含义如表1.2所示。

表1.2 文件权限含义

文件类型	文件所有者的权限			文件所属用户组的权限			其他人对此文件的权限		
d	r	w	x	r	-	x	r	-	x
目录	可读	可写	可执行	可读	无权限	可执行	可读	无权限	可执行

因此/bin目录的文件权限是：文件所有者对/bin目录可读可写可执行，文件所属用户组对/bin目录可读不可写可执行，其他人对/bin目录可读不可写可执行。

当对某个文件进行操作，要特别注意这个文件是否具有将要进行操作的权限。如果我们所在的用户组没有操作权限而又得进行操作，此时就得修改文件的权限。

2. 文件权限修改

“chmod”命令的作用是变更一个文件的权限。

zhuzhaoqi@zhuzhaoqi-desktop:~/linux/include$ ll

总用量 8

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-18 22:02 ./

drwxr-xr-x 3 zhuzhaoqi zhuzhaoqi 4096 2013-03-18 22:07 ../

-rw-r--r-- 1 zhuzhaoqi zhuzhaoqi 0 2013-03-18 22:02 s3c6410.h

从上一小节可知，“drwxr-xr-x”除了“d”是文件类型，剩下9个字符划分成3组，表示3个用户组的使用权限。而在Linux系统中，每一个用户组的3个字母分别可用数字进行描述其权限，r:4、w:2、x:1、-:0，将每一组的数字进行相加，即得到这组用户的权限。例如上面s3c6410.h的权限是：rw-r--r--，那么每一用户组权限分别是：6、4、4，那么组合起来即为644。每个文件的最高权限为777。

给予s3c6410.h最高权限，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/linux/include$ chmod 777 s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux/kernel/include$ ll

总用量 8

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-18 22:02 ./

drwxr-xr-x 3 zhuzhaoqi zhuzhaoqi 4096 2013-03-18 22:07 ../

-rwxrwxrwx 1 zhuzhaoqi zhuzhaoqi 0 2013-03-18 22:02 s3c6410.h*

通过“chmod”更改权限命令可以看到s3c6410.h的权限是最高权限。

1.1.2 目录与路径处理命令

1. 切换目录

“cd”命令的作用是从当前目录切换到另一个目录下。如从用户根目录进入/linux目录下，如下操作：

```
zhuzhaoqi@zhuzhaoqi-desktop:~$ cd linux/
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$
```

2. 创建新目录

“mkdir”命令的作用是创建一个新的目录，如在/linux目录下再创建一个/linux-3.8.3子目录，如下操作：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ mkdir linux-3.8.3
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
linux-3.8.3
```

mkdir 的用法很多，可以通过输入mkdir –help 查看，如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~$ mkdir --help
用法：mkdir [选项]... 目录...
若指定目录不存在则创建目录
长选项必须使用的参数对于短选项时也是必需使用的
-m, --mode=模式 设置权限模式(类似chmod)，而不是rwxrwxrwx减umask
-p, --parents 需要时创建目标目录的上层目录，但即使这些目录已存在也不当作错误处理
-v, --verbose 每次创建新目录都显示信息
-Z, --context=CTX 将每个创建的目录的SELinux 安全环境设置为CTX
--help 显示此帮助信息并退出
```

--version 显示版本并退出

mkdir -p 这个指令在U-Boot和Linux 内核源码中的Makefile 中的使用是相当频繁的。

3. 删除目录

如果是删除一个空目录，则使用“rmdir”命令即可；如果该目录下有东西，则不能使用“rmdir”命令删除。

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux/linux-3.6.7$ ls
arch
zhuzhaoqi@zhuzhaoqi-desktop:~/linux/linux-3.6.7$ cd ..
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
linux-3.6.7 linux-3.8.3
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ cd linux-3.8.3/
zhuzhaoqi@zhuzhaoqi-desktop:~/linux/linux-3.8.3$ ls
zhuzhaoqi@zhuzhaoqi-desktop:~/linux/linux-3.8.3$ cd ..
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
linux-3.6.7 linux-3.8.3
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ rmdir linux-3.8.3/
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
linux-3.6.7
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ rmdir linux-3.6.7/
rmdir: 删除 "linux-3.6.7/" 失败：目录非空
```

从上面操作可知，由于/linux-3.8.3 目录为空，则可使用“rmdir”删除；但是/ linux-3.6.7 目录下有一个子目录/arch，则不能使用“rmdir”删除。此时则应该使用“rm -r”命令删除。

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
linux-3.6.7
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls linux-3.6.7/
```

arch

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ rm -r linux-3.6.7/
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$
```

通过“ls”命令可知，linux目录下的linux-3.6.7/目录以及被删除。

1.1.3 文件操作

1. 新建文件

新建一个文件可以使用“vim”命令，但是使用“vim”命令退出打开的文件时需要保存退出，否则会视为没有创建文件。

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ vim s3c6410.h
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
s3c6410.h
```

2. 复制文件

复制文件命令为“cp”。如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
include s3c6410.c s3c6410.h
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ cp s3c6410.h include/
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
include s3c6410.c s3c6410.h
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls include/
s3c6410.h
```

如果要复制并且重命名，如下操作：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls
```

include kernel s3c6410.c s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ cp s3c6410.c include/s3c6400.c

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls include/

s3c6400.c s3c6410.h

当复制目录时，使用“cp -r”命令。如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls

include kernel s3c6410.c s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls kernel/

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ cp -r include/ kernel/

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls

include kernel s3c6410.c s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls kernel/

include

3. 移动文件

移动一个文件则使用“mv”命令，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls

include kernel s3c6410.c s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ mv s3c6410.c kernel/

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls

include kernel s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ ls kernel/

include s3c6410.c

编辑一个文件，建议使用“gedit”命令或者“vim”命令。

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ gedit s3c6410.h

zhuzhaoqi@zhuzhaoqi-desktop:~/linux$ vim s3c6410.c

1.1.4 打包与解包、压缩与解压缩

熟悉打包与解包、压缩与解压缩的操作命令是操作 Linux 文件的必备技能。Linux 下的打包与解包、压缩与解压缩的操作命令种类繁多，本节截取常用的8个格式进行讲解。本节中，FileName是指打包、压缩之后的文件名，DirName是指待打包、压缩的文件名。

（1）.tar格式

单纯的tar功能其实仅仅是打包而已，也就是说将很多文件集结成一个文件，并没有进行压缩。

解包：tar xvf FileName.tar

打包：tar cvf FileName.tar DirName

（2）.gz格式

GZIP 最早由Jean-loup Gailly和Mark Adler 创建，用于UNIX系统的文件压缩。在Linux 中经常会碰到后缀名为.gz的文件，它们的原型即是GZIP格式。

解压1：gunzip FileName.gz

解压2：gzip -d FileName.gz

压缩：gzip FileName

（3）.tar.gz格式和.tgz格式

以.tar.gz和.tgz为后缀名的压缩文件在在Linux和OSX下是非常常见的，Linux和OSX都可以直接解压使用这种压缩文件。

解压：tar zxvf FileName.tar.gz

压缩：tar zcvf FileName.tar.gz DirName

（4）.bz2格式

压缩生成后缀名为.bz2 的压缩算法使用的是“Burrows-Wheeler block sorting text”，这类算法压缩比率比较高。

解压1：bzip2 -d FileName.bz2

解压2：bunzip2 FileName.bz2

压缩： bzip2 -z DirName

这里需要注意的是，当执行压缩指令之后，将会生成FileName.bz2压缩文件，同时DirName文件将会自动删除。

（5）.tar.bz2格式

bzip2是一个压缩能力非常强的压缩程序，以.bz2和.tar.bz2为后缀名的压缩文件都是bzip2压缩的结果。

解压：tar jxvf FileName.tar.bz2

压缩：tar jcvf FileName.tar.bz2 DirName

（6）.Z格式

compress 是一个相当古老的UNIX 压缩指令，压缩后的文件是以.Z 作为后缀名。

解压：uncompress FileName.Z

压缩：compress DirName

（7）.tar.Z格式

这个压缩格式可以认为是.tar打包加上.Z压缩。

解压：tar Zxvf FileName.tar.Z

压缩：tar Zcvf FileName.tar.Z DirName

（8）.zip格式

因为格式开放而且免费，越来越多的软件支持打开Zip文件。

解压：unzip FileName.zip

压缩：zip FileName.zip DirName

以上8种打包压缩算法都有所区别，最终导致的结果是压缩时间和压缩大小不一样。每一种压缩格式都有其优势和不足，在何种场合应

该使用何种压缩格式就得视实际情况而定了。

在程序设计当中，空间换取时间、时间换取空间的现象是非常常见的一种方法。比如在单片机的LED跑马灯中，经常使用在数组中取出想要的花样，这就是空间换取时间。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第一章01课（Linux基本操作指令）。

1.2 Makefile基本知识

Makefile如今能得以广泛应用，这还得归功于它被包含在UNIX系统中。在make诞生之前， UNIX系统的编译系统主要由“make”、“install”shell脚本程序和程序的源代码组成。它可以把不同目标的命令组成一个文件，而且可以抽象化依赖关系的检查和存档。这是向现代编译环境发展的重要一步。1977年，斯图亚特·费尔德曼在贝尔实验室里制作了这个软件。2003年，斯图亚特·费尔德曼因发明了这样一个重要的工具而接受了美国计算机协会（ACM）颁发的软件系统奖。

Makefile 文件可以实现自动化编译，只需要一个“make”命令，整个工程就能完全自动编译，极大地提高了软件开发的效率。目前虽有众多依赖关系检查工具，但是make是应用最广泛的一个。一个程序员会不会写 Makefile，从一个侧面说明了这个程序员是否具备完成大型工程的能力。

1.2.1 Makefile规则

一个简单的Makefile语句由目标、依赖条件、指令组成。

```
smdk6400_config :unconfig
    @mkdir -p $(obj)include $(obj)board/samsung/smdk6400
```

smdk6400_config：目标；

unconfig：先决条件；

@mkdir -p $(obj)include $(obj)board/samsung/smdk6400：指令。这里特别注意，“@”前面是Tab键，并且必须是Tab键，而不能是空格。

目标和先决条件是依赖关系，目标是依赖于先决条件生成的。

1.2.2 Makefile变量

1. 变量的引用方式

使用“$(OBJTREE)”或者“${ OBJTREE }”来引用OBJTREE 这个变量的定义。这个引用方式似乎很像C语言中的指针变量，使用*p来取存放在指针p中的值。

```
obj := $(OBJTREE)/
OBJTREE := $(if $(BUILD_DIR),$(BUILD_DIR),$(CURDIR))
export BUILD_DIR=/tmp/build
```

$(if $（BUILD_DIR)，$(BUILD_DIR)，$(CURDIR))的含义：如果“BUILD_DIR”变量值不为空，则将变量“BUILD_DIR”指定的目录作为一个子目录；否则将目录“CURDIR”作为一个子目录。

2. 递归展开式变量

这类变量的定义是通过“=”和“define”来定义的。

```
student = lilei
CLASS = $(student) $(teacher)
teacher = yang
all:
```

```
    @echo $(CLASS)
```

其优点是：这种类型递归展开式的变量在定义时，可以引用其他之前没有定义过的变量，这个变量可能在后续部分定义，或者是通过make的命令行选项传递的变量来定义。

其缺点是：其一，使用此风格的变量定义，可能会由于出现变量的递归定义而导致 make 陷入到无限的变量展开过程中，最终使make执行失败。

```
x = $(y)
y = $(z)
z = $(x)
```

这样的话会使得Makefile出错，因为到最终引用了自己。

其二，这种风格的变量定义中如果使用了函数，那么包含在变量值中的函数总会在变量被引用的地方执行。

3．直接展开式变量

为了避免递归展开式变量存在的问题和不方便，GNU make 支持另外一种风格的变量，称为直接展开式变量。这种风格的变量使用“:=”定义。在使用“:=”定义变量时，变量值中对其他变量或者函数的引用在定义变量时被展开，也就是对变量进行替换。

```
X := student
Y := $(X)
X := teacher
all:
    @echo $(X) $(Y)
```

这里的输出是：teacher student。

这个直接展开式变量在定义时就完成了对所引用变量和函数的展开，因此不能实现对其后定义变量的引用。

4．条件赋值

在对变量进行赋值之前，会对其进行判断，只有在这个变量之前没有进行赋值的情况下才会对这个变量进行赋值。

```
X := student
X ?= teacher
all:
    @echo $(X)
```

由于X在之前被赋值了，所以这里的输出是student。

5. 变量的替换引用

对于一个已经定义的变量，可以使用变量的替换引用将变量中的后缀字符使用指定的字符替换。格式为“$（X:a=b）”（或者“${X:a=b}”），即将变量“X”中所有以“a”字符结尾的字替换为以“b”结尾的字。

```
X := fun.o main.o
Y := $(X: .o=.c)
all:
    @echo $(X) $(Y)
```

特别注意的是，$（X: .o=.c）的“=”两边不能有空格。这里的输出是：fun.o main.o fun.c main.c。

6. 追加变量值

追加变量值是指一个通用变量在定义之后的其他一个地方，可以对其值进行追加。也就是说可以在定义时（也可以不定义而直接追加）给它赋一个基本值，后续根据需要可随时对它的值进行追加（增加它的值）。在Makefile中使用“+=”（追加方式）来实现对一个变量值的追加操作。

```
X = fun.o main.o
X += sub.o
all:
```

```
    @echo $(x)
```

这里的输出是：fun.o main.o sub.o。

1.2.3 Makfile常用关键字

1. ifneq关键字

这个关键字是用来判断两个参数是否不相等。格式为：

```
ifneq “Value1” “Value2”
ifneq (Value1,Value2)
```

在判断之前先要将Value1和Value2的值进行展开和替换，如在U-Boot-2013.04的顶层目录Makefile中，对U-Boot的版本参数就使用了ifneq关键字进行判断。

```
VERSION= 2013
PATCHLEVEL = 04
SUBLEVEL=
EXTRAVERSION =
ifneq "$(SUBLEVEL)" ""
U_BOOT_VERSION =
$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
else
U_BOOT_VERSION = $(VERSION).$(PATCHLEVEL)$(EXTRAVERSION)
endif
```

先将SUBLEVEL使用$()展开和替换，如果SUBLEVEL的值不是空，则执行：

```
U_BOOT_VERSION =
$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
```

也就是说，如果$（SUBLEVEL） = 1 的话，那么U_BOOT_VERSION = 2013.04.1。

如果SUBLEVEL的值是空，则执行：

U_BOOT_VERSION = $(VERSION).$(PATCHLEVEL)$(EXTRAVERSION)

那么此时U_BOOT_VERSION = 2013.04。

2. ifeq关键字

ifeq关键字和ifneq关键字是相对而言的，用来判断两个参数是否相等。格式为：

ifeq “Value1” “Value2”

ifeq (Value1,Value2)

和ifneq一样，先要将Value1和Value2展开替换之后，再进行比较。

```
ifeq ($(HOSTARCH),$(ARCH))
CROSS_COMPILE=/usr/local/arm/4.4.1/bin/arm-linux-
endif
```

如果HOSTARCH展开替换之后和ARCH展开替换之后相等，则：

CROSS_COMPILE=/usr/local/arm/4.4.1/bin/arm-linux-

否则CROSS_COMPILE不等于/usr/local/arm/4.4.1/bin/arm-linux-。

3. ifndef关键字

ifndef关键字用来判断一个变量是否没有进行定义。格式：

ifndef Value

由于在Makefile中，没有定义的变量的值为空。

```
ifndef CONFIG_SANDBOX
SUBDIRS += $(SUBDIR_EXAMPLES)
endif
```

如果CONFIG_SANDBOX值为空，条件成立，执行如下语句：

```
SUBDIRS += $(SUBDIR_EXAMPLES)
```

否则不执行。

4. ifdef关键字

ifdef关键字用来判断一个变量是否已经进行定义过。格式：

```
ifdef Value
```

如：

```
ifdef CONFIG_SYS_LDSCRIPT
    # need to strip off double quotes
    LDSCRIPT := $(subst ",,$(CONFIG_SYS_LDSCRIPT))
endif
```

如果CONFIG_SYS_LDSCRIPT定义过，则执行：

```
LDSCRIPT := $(subst ",,$(CONFIG_SYS_LDSCRIPT))
```

否则不执行。

1.2.4 Makefile常用函数

1. Makefile函数语法

在Makefile中，函数的调用和变量的调用类似，都是使用“$”进行标识。语法如下：

```
$(函数名 函数的参数)
${函数名 函数的参数}
```

函数名与函数的参数之间使用空格隔开，而函数的参数间使用逗号进行分隔。以上两种写法都是可以的，但是为了风格统一，请不要两者进行混合使用。

2. shell函数

make可以使用shell函数和外部通信。shell函数本身的返回值是其参数的执行结果，没有进行任何处理，对结果的处理是由 make 进

行的。当对函数的引用出现在规则的命令行中，命令行在执行时函数才被展开。展开时函数参数（shell命令）的执行是在另外一个shell进程中完成的，因此需要对出现在规则命令行的多级“shell”函数引用需要谨慎处理，否则会影响效率（每一级的“shell”函数的参数都会有各自的shell进程）。

建立一个测试程序，Makefile的内容：

```
zhu := $(shell cat func)
all:
    @ echo $(zhu)
```

Func文件中的内容：

```
juxst zhuzhaoqi
```

执行完成Makefile之后：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/u-boot/Makefile/shellfunction$ make
juxst zhuzhaoqi
```

在U-Boot和Linux内核源码中将会大量使用到shell函数。

3. subst函数

subst函数是字符串替换函数，语法为：

```
$(subst 被替换字串 替换字串 替换操作字符串)
```

执行subst函数之后，返回的是执行替换操作之后的字符串。如下：

```
name := zhu zhaoqi
Alphabet_befor := z
Alphabet_after := Z
Name := $(subst $(Alphabet_befor), $(Alphabet_after), $(name))
all:
```

```
    echo $(Name)
```

执行上面的Makefile，输出结果为：

```
echo Zhu Zhaoqi
Zhu Zhaoqi
```

即将“z”替换成了“Z”。

4. dir函数

dir函数作用为取出该文件的目录，其语法为：

```
$(dir 文件名称)
```

执行该函数之后返回文件目录部分。

Makefile中常用函数较多，笔者就不一一例举了，读者可参考相关书籍进行深入了解。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第一章 02 课（Makefile）。

1.3 arm-linux交叉编译链

平常我们做的编译叫本地编译，也就是在当前平台编译，编译得到的程序也是在本地执行。相对而言的交叉编译指的是在一个平台上生成另一个平台的可执行代码。

常见的交叉编译有以下3种。

在Windows PC上，利用ADS（ARM 开发环境），使用armcc 编译器，编译出针对ARM CPU的可执行代码。

在Linux PC 上，利用arm-linux-gcc 编译器，编译出针对Linux ARM 平台的可执行代码。

在Windows PC上，利用cygwin 环境，运行arm-elf-gcc 编译器，编译出针对ARM CPU的可执行代码。

1.3.1 arm-linux交叉编译工具链的制作方法

由于一般嵌入式开发系统的存储大小是有限的，通常都要在性能优越的 PC 上建立一个用于目标机的交叉编译工具链，用该交叉编译工具链在PC上编译目标机上要运行的程序，比如在PC平台（X86 CPU）上编译出能运行在以ARM为内核的CPU 平台上的程序。要生成在目标机上运行的程序，必须要用交叉编译工具链完成。交叉编译工具链是一个由编译器、连接器和解释器组成的综合开发环境，交叉编译工具链主要由binutils、gcc 和glibc 3 个部分组成。有时出于减小libc 库大小的考虑，也可以用别的 c 库来代替 glibc，例如 uClibc、dietlibc 和 newlib。建立交叉编译工具链是一个相当复杂的过程，如果不想自己经历复杂繁琐的编译过程，网上有一些编译好的可用的交叉编译工具链可以下载，但就以学习为目的来说，读者有必要学习自己制作一个交叉编译工具链。本节通过具体的实例讲述基于ARM的嵌入式Linux交叉编译工具链的制作过程。

制作arm-linux交叉编译工具链的一般通过crosstool工具或者crosstool_NG，前者使用方便，但是制作会受到一些限制，使用crosstool最多只能编译gcc 4.1.1、glibc 2.x 的版本。crosstool-NG是新的用来建立交叉工具链的工具，它是crosstool的替换者，crosstool_NG有更好的定制性，并且一直保持着更新，对新版本的编译工具链的支持比较好，当然也带来了一些麻烦，它并不是下载下来就可以使用的，必须先配置安装。我们这里选用crosstool_NG来制作我们的编译链。

1. 安装crosstool_NG

在crosstool_NG官网上下载最新版本。

zhuzhaoqi@zhuzhaoqi-desktop:~$ mkdir arm-linux-tools

zhuzhaoqi@zhuzhaoqi-desktop:~$ cd arm-linux-tools/

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ ls

获取源码操作命令：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ wget http://crosstool-ng.org/ download/crosstool-ng/crosstool-ng-1.18.0.tar.bz2

--2013-03-26 21:34:34-- http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-1.18.0.tar.bz2

正在解析主机 crosstool-ng.org... 140.211.15.107

正在连接 crosstool-ng.org|140.211.15.107|:80... 已连接。

已发出 HTTP 请求，正在等待回应... 200 OK

长度： 1884219 (1.8M) [application/x-bzip]

正在保存至: “crosstool-ng-1.18.0.tar.bz2”

100%[======================================>] 1,884,219223K/s 花时 8.8s

下载源码成功之后解压源码：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ tar jxvf crosstool-ng-1.18.0.tar.bz2 zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ ls

crosstool-ng-1.18.0 crosstool-ng-1.18.0.tar.bz2

考虑到后续将要使用到的各种目录，在这里先建立好后续所需目录。

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ mkdir crosstool-build crosstool-install src zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ ls

crosstool-buildcrosstool-ng-1.18.0 src

crosstool-install crosstool-ng-1.18.0.tar.bz2

由于Ubuntu操作系统的很多开发软件都没有安装，因此要先安装一些制作交叉编译链必备的软件。在Ubuntu 下安装软件的命令为：sudo apt-get install ***。

笔者建议arm-linux交叉编译工具链的制作最好在CentOS系统中完成，因为CentOS系统自带较为完善的开发软件，对于初学者不会造成不必要的麻烦。

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools$ sudo apt-get install sed bash cut dpkg-dev patch texinfom4 libtool statwebsvn tar gzip bzip2 lzmabison flex texinfo automake libtool patchcvs cvsd gawk - y

配置整个工程并且进行依赖检测：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$./configure--prefix /home/zhuzhaoqi/arm-linux-tools/crosstool-install

在安装过程中，提示如下错误：

……

checking how to run the C preprocessor... gcc -E

checking for ranlib... ranlib

checking for objcopy... objcopy

checking for absolute path to objcopy... /usr/bin/objcopy

checking for objdump... objdump

checking for absolute path to objdump... /usr/bin/objdump

checking for readelf... readelf

checking for absolute path to readelf... /usr/bin/readelf

checking for bison... no

configure: error: missing required tool: bison

输出错误提示缺失bison这个软件，安装：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo apt-get install bison

安装完成之后，再次进行配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$./configure--prefix /home/zhuzhaoqi/arm-linux-tools/crosstool-install

又一次输出错误：

……

checking for bison... bison

checking for flex... no

configure: error: missing required tool: flex

提示缺失flex这个软件，进行安装：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo apt-get install flex

安装完成之后，再一次进行配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$./configure --prefix /home/zhuzhaoqi/arm-linux-tools/crosstool-install

又一次提示错误：

checking for bison... bison

checking for flex... flex

checking for gperf... no

configure: error: missing required tool: gperf

提示缺失gperf这个软件，进行安装：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo apt-get install gperf

安装完成之后，再一次进行配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$./configure --prefix /home/zhuzhaoqi/arm-linux-tools/crosstool-install

再一次提示出错：

……

checking for bison... bison

checking for flex... flex

checking for gperf... gperf

checking for makeinfo... no

configure: error: missing required tool: makeinfo

缺失makeinfo软件，进行安装，如果安装的是makeinfo，则会有如下提示：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo apt-get install makeinfo

正在读取软件包列表... 完成

正在分析软件包的依赖关系树

E: 无法找到软件包makeinfo

此时应该安装texinfo软件：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo apt-get install makeinfo

安装完成之后，再一次进行配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$./configure --prefix /home/zhuzhaoqi/arm-linux-tools/crosstool-install

这次的配置成功，如果读者操作还会报错的话，依照上面方法找出其根源进行改正即可。成功配置之后会自动创建我们需要的Makefile文件。

checking for library containing initscr... -lncursesw

configure: creating ./config.status

config.status: creating Makefile

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ ls

bootstrap configure ct-ng.comp LICENSES patches steps.mk

config configure.ac ct-ng.in licenses.d README TODO

config.log contrib docs Makefile samples

config.status COPYING kconfig Makefile.in scripts

执行Makefile文件：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ make

SED 'ct-ng'

SED 'scripts/crosstool-NG.sh'

SED 'scripts/saveSample.sh'

SED 'scripts/showTuple.sh'

GEN 'config/configure.in'

GEN 'paths.mk'

GEN 'paths.sh'

DEP 'nconf.gui.dep'

DEP 'nconf.dep'

DEP 'lxdialog/yesno.dep'

```
DEP  'lxdialog/util.dep'
DEP  'lxdialog/textbox.dep'
DEP  'lxdialog/menubox.dep'
DEP  'lxdialog/inputbox.dep'
DEP  'lxdialog/checklist.dep'
DEP  'mconf.dep'
DEP  'conf.dep'
BISON  'zconf.tab.c'
GPERF  'zconf.hash.c'
LEX  'lex.zconf.c'
DEP  'zconf.tab.dep'
CC  'zconf.tab.o'
CC  'conf.o'
LD  'conf'
CC  'lxdialog/checklist.o'
CC  'lxdialog/inputbox.o'
CC  'lxdialog/menubox.o'
CC  'lxdialog/textbox.o'
CC  'lxdialog/util.o'
CC  'lxdialog/yesno.o'
CC  'mconf.o'
LD  'mconf'
CC  'nconf.o'
CC  'nconf.gui.o'
LD  'nconf'
SED  'docs/ct-ng.1'
GZIP  'docs/ct-ng.1.gz'
```

编译成功之后进行安装：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ make install

成功安装之后，可以看到已经安装到指定的目录下，最后输出这么一句话：

……

For auto-completion, do not forget to install 'ct-ng.comp' into

your bash completion directory (usually /etc/bash_completion.d)

这是在提醒我们不要忘记了配置环境变量，多么人性化的提示。接下来配置环境变量。

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0$ sudo echo"PATH=$PATH:/home/zhuzhaoqi/arm-linux-tools/crosstool-install/bin" >> ~/.bashrc

执行使其生效：

zhuzhaoqi@zhuzhaoqi-desktop:~$ source /home/zhuzhaoqi/.bashrc

使用ct-ng -v 命令查看安装结果：

zhuzhaoqi@zhuzhaoqi-desktop:~$ ct-ng -v

GNU Make 3.81

Copyright (C) 2006 Free Software Foundation, Inc.

This is free software; see the source for copying conditions.

There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A

PARTICULAR PURPOSE.

这个程序创建为 i486-pc-linux-gnu

OK！ct-ng环境变量添加成功，也就意味着整个crosstool-NG安装成功。

2. 配置交叉编译链

现在需要去做的就是配置要编译的交叉编译工具链，在crosstool-NG中有很多已经做好的默认配置（位于crosstool-ng-X.Y.Z（crosstool-ng-1.18.0）/samples 目录下），这里只需要进行修改就好了。对于编译器组件部分的版本最好不要修改，因为那个应该是经过测试后的最高版本了，但内核版本可以修改。

可以看到samples目录下的一些默认配置，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0/samples$ ls

alphaev56-unknown-linux-gnu mips64el-n64-linux-uclibc

alphaev67-unknown-linux-gnu mips-ar2315-linux-gnu

arm-bare_newlib_cortex_m3_nommu-eabi mipsel-sde-elf

arm-cortex_a15-linux-gnueabi mipsel-unknown-linux-gnu

arm-cortex_a8-linux-gnueabi mips-malta-linux-gnu

arm-davinci-linux-gnueabi mips-unknown-elf

armeb-unknown-eabi mips-unknown-linux-uclibc

armeb-unknown-linux-gnueabi powerpc-405-linux-gnu

armeb-unknown-linux-uclibcgnueabi powerpc64-unknown-linux-gnu

arm-unknown-eabi powerpc-860-linux-gnu

arm-unknown-linux-gnueabi powerpc-e300c3-linux-gnu

arm-unknown-linux-uclibcgnueabi powerpc-e500v2-linux-gnuspe

armv6-rpi-linux-gnueabi powerpc-unknown-linux-gnu

avr32-unknown-none powerpc-unknown-linux-uclibc

bfin-unknown-linux-uclibc powerpc-unknown_nofpu-linux-gnu

i586-geode-linux-uclibc s390-ibm-linux-gnu

i586-mingw32msvc,i686-none-linux-gnu s390x-ibm-linux-gnu

i686-nptl-linux-gnu samples.mk

i686-unknown-mingw32 sh4-unknown-linux-gnu

m68k-unknown-elf x86_64-unknown-linux-gnu

m68k-unknown-uclinux-uclibc x86_64-unknown-linux-uclibc

mips64el-n32-linux-uclibc x86_64-unknown-mingw32

里面有很多默认配置，有arm、avr32、mips、powerpc等硬件平台，而arm平台有如下几个：

arm-unknown-eabi是基于裸板，也就是无操作系统。

arm-unknown-linux-gnueabi 是基于Linux。

arm-unknown-linux-uclibcgnueabi 这个应该能看出来了，是为uclinux 用的。

arm-cortex_a15-linux-gnueabi可从名字上看是为cortex-a15用的。

arm-cortex_a8-linux-gnueabi 这个也可从名字上看是为cortex-a8 用的。

arm-xxx$&#*&还有几个，这些暂且不去理会。

因为是制作 arm-linux 交叉编译链，所以选择 arm-unknown-linux-gnueabi 进行配置。将arm-unknown-linux-gnueabi文件夹复制到crosstool-build/目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-ng-1.18.0/samples$ cp -r arm-unknown-linux-gnueabi/

../../crosstool-build/

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-build$ ls

arm-unknown-linux-gnueabi

将默认配置文件拷贝到工作目录（crosstool-build）下并改名为.config，因为默认的配置文件为.config，完成之后可以加载需要的配置。

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-build$ cparm-unknown-linux-gnueabi/crosstool.config .config

执行ct-ng menuconfig 进入配置界面进行配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-build$ ct-ng menuconfig

```
    LN  config
    MKDIR config.gen
    IN  config.gen/arch.in
    IN  config.gen/kernel.in
    IN  config.gen/cc.in
    IN  config.gen/binutils.in
    IN  config.gen/libc.in
    IN  config.gen/debug.in
    CONF  config/config.in
#
# configuration saved
#
```

进入配置界面，如图1.1所示。

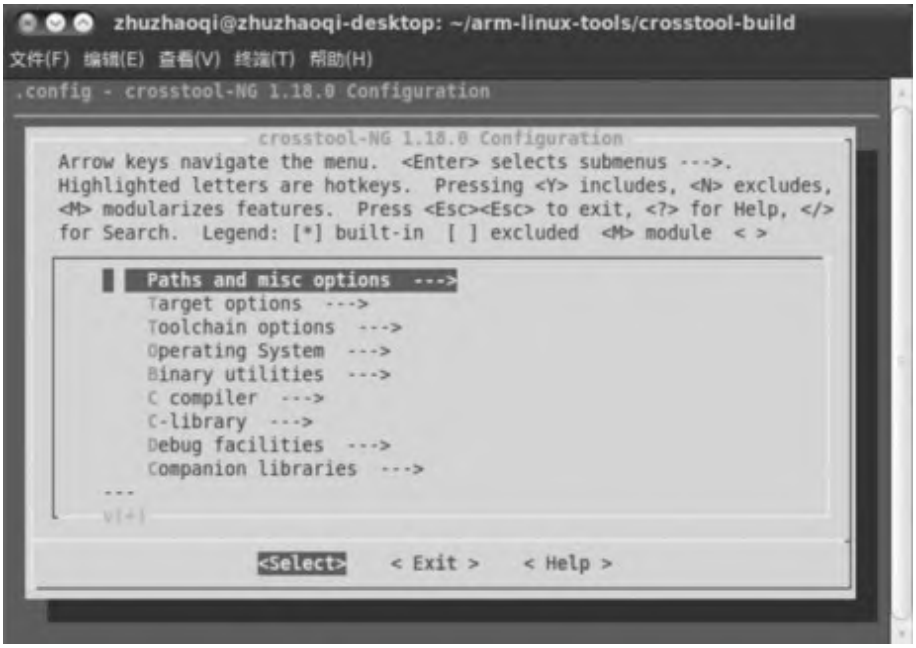


图1.1 ct-ng图形配置界面

下面设置源码目录和安装目录，这需要读者依据自己实际设定的情况来进行配置。

第一步，设定源码包路径和交叉编译器的安装路径。

Paths and misc options --->

(/home/zhuzhaoqi/arm-linux-tools/src) Local tarballs directory 保存源码包路径

(/home/zhuzhaoqi/arm-linux-tools/tools) Prefix directory 交叉编译器的安装路径

配置之后的结构如图1.2所示。

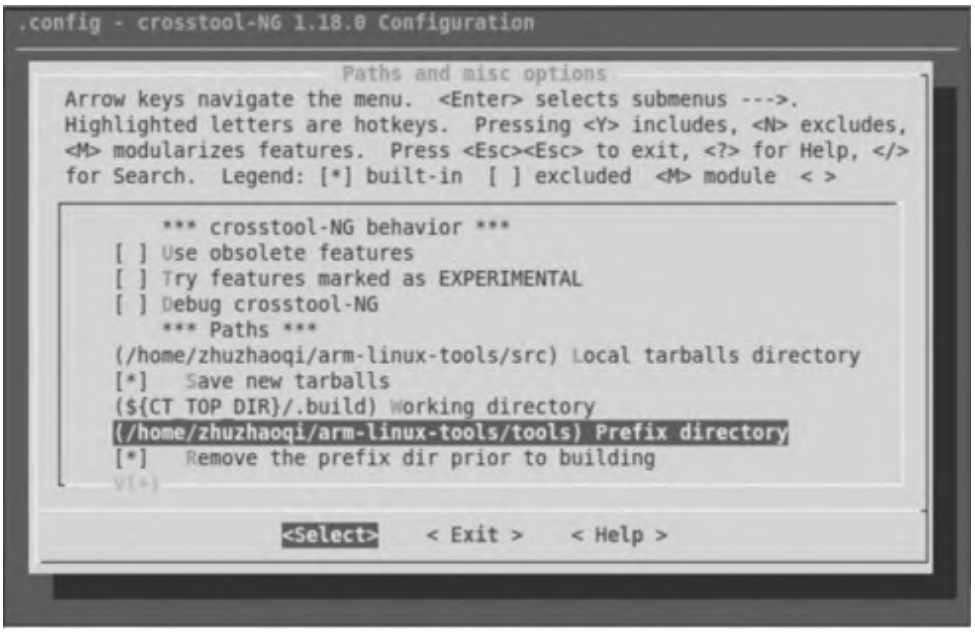


图1.2 添加源码包路径和交叉编译器的安装路径

第二步，修改交叉编译器针对的构架。

因为本次是针对OK6410制作编译链，那就依据s3c6410的硬件特性来制作。

Target options 是重点要修改的地方（以下配置均是基于已拷贝过来的配置）。

Target Architecture(arm) 这个不用管，已经是arm 了。

Default instruction set mode (arm) 这个也不管，也已经是 arm 了。

Architecture level()需要进行修改。

通过查找资料，这个应该是指令集的架构，对于 S3C6410 ARM1176JZF-S 核心使用的是armv6zk架构，就选armv6zk。那么，具体都支持哪些架构呢？可以用man gcc来查询，搜索arm，再搜索-march=就可以找到本gcc支持的处理器核心列表了：

-march=name

This specifies the name of the target ARM architecture. GCC uses

this name to determine what kind of instructions it can emit when

generating assembly code. This option can be used in conjunction

with or instead of the -mcpu= option. Permissible names are:

armv2, armv2a, armv3, armv3m, armv4, armv4t, armv5, armv5t, armv5e,

armv5te, armv6, armv6j, armv6t2, armv6z, armv6zk, armv6-m, armv7,

armv7-a, armv7-r, armv7-m, iwmmxt, iwmmxt2, ep9312.

Emit assembly for CPU()需要进行修改。

这个对应的是CPU的核心类型。同样，也和上面的选项一样，对应一个GCC选项。GCC的描述如下。

-mcpu=name

This specifies the name of the target ARM processor. GCC uses this

name to determine what kind of instructions it can emit when

generating assembly code. Permissible names are: arm2, arm250,

arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m, arm7d,

arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710c,

arm7100, arm720, arm7500, arm7500fe, arm7tdmi,arm7tdmi-s, arm710t,

arm720t, arm740t, strongarm, strongarm110, strongarm1100,

strongarm1110, arm8, arm810, arm9, arm9e, arm920, arm920t, arm922t,

arm946e-s, arm966e-s, arm968e-s, arm926ej-s, arm940t, arm9tdmi,

arm10tdmi, arm1020t, arm1026ej-s, arm10e, arm1020e, arm1022e,

arm1136j-s, arm1136jf-s, mpcore, mpcorenovfp, arm1156t2-s,

arm1176jz-s, arm1176jzf-s, cortex-a8, cortex-a9, cortex-r4,

cortex-r4f, cortex-m3, cortex-m1, xscale, iwmmxt, iwmmxt2, ep9312.

这样看简单一些了，如果是 S3C2410/S3C2440 就选 arm920t ，如果是 s3c6410 就选arm1176jzf-s。

Tune for CPU() ，对应的GCC 描述如下：

-mtune=name

This option is very similar to the -mcpu= option, except that

instead of specifying the actual target processor type, and hence

restricting which instructions can be used, it specifies that GCC

should tune the performance of the code as if the target were of

the type specified in this option, but still choosing the

instructions that it will generate based on the cpu specified by a

-mcpu= option. For some ARM implementations better performance can

be obtained by using this option.

意思是说这个选项和-mcpu 很类似，这里是指真实的CPU 型号。不过有读者是编译2440的工具链，这里选择的是arm9tdmi，如果不是，那就空着。这里的作用是如果arm920t处理不了，就用arm9tdmi的方式来编译。

与Floating point()浮点相关的选项s3c6410 有硬件VFP，所以这里选的是hardware FPU。这个是给有硬浮点的处理器强行选软浮点用的。

Use specific FPU()跟浮点有关，这里不选任何内容。至于怎么组合，读者可以跟据自己的CPU的实际情况进行相应的配置。

```
C compiler --->
  *** Additional supported languages: ***
  [ ] Java //不用这个编译器来编译java
```

当然，如果读者需要用它来编译java那就不用去除。

其他选项采用默认设置存盘然后退出，这样就配置完了。

zhuzhaoqi@zhuzhaoqi-desktop:~/arm-linux-tools/crosstool-build$ ct-ng build

开始编译，此编译过程需要花费大约两个小时，最终编译出arm-linux-gcc-4.4.1编译链。

1.3.2 交叉编译链在宿主机上的安装

这里要安装的交叉编译链版本是arm-linux-gcc 4.4.1（交叉编译链的版本很多，读者可以自行安装版本更高的编译链）。采用的Linux系统环境是Ubuntu10.04.4。具体的操作步骤如下。

1. 在/usr/local下面创建一个文件夹：mkdir arm，将arm-linux-gcc 4.4.1 放在arm文件夹里面。然后解压缩，命令根据压缩包的后缀不同而不同。

2. 添加环境变量，vim /etc/profile。

3. 在最后一行添加：export PATH=$PATH:/usr/local/arm/4.4.1/bin。

4. 退出执行命令：source /etc/profile。使其生效。

5. 检测安装是否成功：arm-linux-gcc -v ；如果成功，输出最后一行则会提示：gcc version 4.4.1 （Sourcery G++ Lite 2009q3-67）。描述如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/u-boot/Makefile/shellfunction$ arm-linux-gcc -v

Using built-in specs.

Target: arm-none-linux-gnueabi

Configured with: /scratch/julian/2009q3-respin-linux-lite/src/gcc-4.4/configure --build= i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi --enable-threads --disable-libmudflap --disable-libssp --disable-libstdcxx-pch --enable-extra-sgxxlite-multilibs --with-arch=armv5te --with-gnu-as --with-gnu-ld --with-specs='%{funwind-tables|fno-unwind-tables|mabi=*|ffreestanding|nostdlib:;:-funwind-tables} %{O2:%{!fno-remove-local-statics: -fremove-local-statics}} %{O*:%{0|00|01|02|0s:;:%{!fno-remove-local-statics: -fremove-

local- statics}}}' --enable-languages=c,c++ --enable-shared --disable-lto --enable-symvers=gnu --enable-__cxa_atexit --with-pkgversion='Sourcery G++ Lite 2009q3-67' --with-bugurl=https: //support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery --with-sysroot=/opt/codesourcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/scratch/julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/libc--with-gmp=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/u sr --with-mpfr=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none- linux-gnueabi-i686-pc-linux-gnu/usr --with-ppl=/scratch/julian/2009q3-respin-linux-lite/ obj/ host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-host-libstdcxx='-static- libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --with-cloog=/scratch/julian/2009q3-respin-linux-lite/obj/host-libs-2009q3-67-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr

--disable-libgomp --enable-poison-system-directories --with-build-time-tools=/scratch/ julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/bin --with-build-time-tools=/scratch/julian/2009q3-respin-linux-lite/install/arm-none-linux-gnueabi/bin

Thread model: posix

gcc version 4.4.1 (Sourcery G++ Lite 2009q3-67)

笔者建议最好不要在root用户下进行安装，否则使用交叉编译链可能会存在权限限制。

1.4 映像文件的生成和运行

德国罕见的科学大师莱布尼茨，在他的手迹里留下这么一句话：“1与0，一切数字的神奇渊源。这是造物的秘密美妙的典范，因为，一切无非都来自上帝。”二进制0和1两个简单的数字，构造了神奇的计算机世界，对人类的生产活动和社会活动产生了极其重要的影响，并以强大的生命力飞速发展。在嵌入式系统移植过程中，不管文件数量多么庞大，经过编译工具的层层处理后，最终生成一个可以加载到存储器内执行的二进制映像文件（.bin）。本节内容将会探讨映像文件的生成过程，以及它在存储设备的不同位置对程序运行产生的影响，为本书后文嵌入式系统的移植打下坚定的基础。

1.4.1 编译过程

GNU提供的编译工具包括汇编器as、C编译器gcc、C++编译器g++、链接器ld、二进制转换工具objcopy和反汇编的工具objdump等。它们被称作GNU编译器集合，支持多种计算机体系类型。基于ARM平台的工具分别为arm-linux-gcc、arm-linux-g++、arm-linux-ld、arm-linux-objcopy和arm-linux-objdump。arm-linux交叉编译工具链的制作方法已经详细介绍过了，编译程序直接使用前面制作好的工具链。

GNU 编译器的功能非常强大，程序可以用 C 文件、汇编文件编写，甚至是二者的混合。如图 1.3 所示是程序编译的大体流程，源文件经过预处理器、汇编器、编译器、链接器处理后生成可执行文件，再由二进制转换工具转换为可用于烧写到 Flash 的二进制文件，同时为了调试的方便还可以用反汇编工具生成反汇编文件。图中双向箭头的含义是，当gcc增加一些参数时可以相互调用汇编器和链接器进行工

作。例如输入命令行“gcc -O main.c”后，直接就得到可执行文件a.out（elf）。

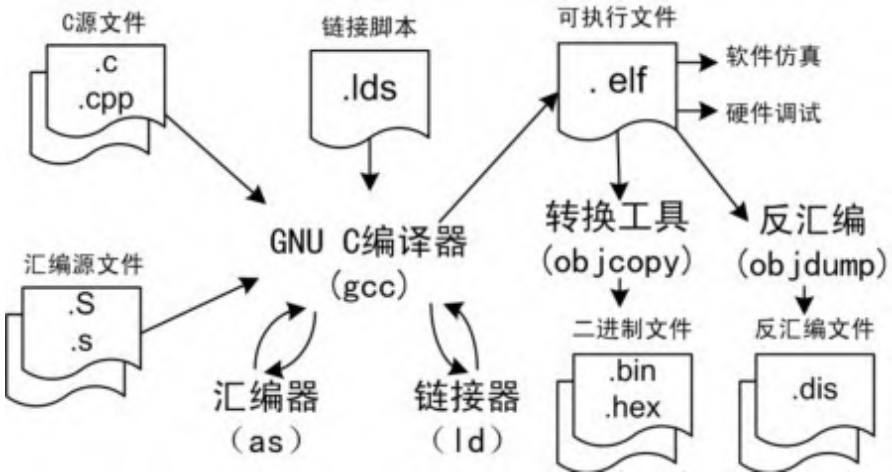


图1.3 程序编译流程

程序编译大体上可以分为编译和链接两个步骤：把源文件处理成中间目标文件.o（linux）、obj （windows）的动作称为编译；把编译形成的中间目标文件以及它们所需要的库函数.a（linux）、lib（windows）链接在一起的动作称为链接。现用一个简单的test工程来分析程序的编译流程。麻雀虽小，五脏俱全，它由启动程序start.S、应用程序main.c、链接脚本test.lds和Makefile四个文件构成。test工程中的程序通过操作单板上的LED灯的状态来判定程序的运行结果，它除了用于理论研究之外，没有其他的实用价值。

1．编译

在编译阶段，编译器会检查程序的语法、函数与变量的声明情况等。如果检查到程序的语法有错误，编译器立即停止编译，并给出错误提示。如果程序调用的函数、变量没有声明原型，编译器只会抛出一个警告，继续编译生成中间目标文件，待到链接阶段进一步确定调用的变量、函数是否存在。

start.S文件的内容如程序清单1.1所示，文件中的_start函数，为程序能够在C语言环境下运行做了最低限度的初始化：将S3C6410处

理外设端口的地址范围告知ARM内核，关闭看门狗，清除bss段，初始化栈。初始化工作完毕后，跳转到main()。start.S是用汇编语言编写的代码文件，文件中定义了一个WATCHDOG宏，用于寄存器的赋值。在汇编文件中出现#define宏定义语句，对于初学者可能会有些迷惑。

程序清单1.1 start.S中汇编代码

```
/*
  *This is a part of the test project
  *Author: LiQiang Date: 2013/04/01
  *Licensed under the GPL-2 or later.
  */
.globl _start
_start:
  #define REG32 0x70000000
  ldr r0, =REG32
  orr r0, r0, #0x13
  mcr p15,0,r0,c15,c2,4
  /*关闭看门狗*/
  #define WATCHDOG 0x7E004000
  ldr r0, =WATCHDOG
  mov r1, #0
  str r1, [r0]
clean_bss:
  ldr r0, =bss_start
  ldr r1, =bss_end
  mov r3, #0
  cmp r0, r1
  beq clean_done
```

```
clean_loop:
  str r3, [r0], #4
  cmp r0, r1
  bne clean_loop
clean_done:
  /* 初始化栈 S3C6410 8K 的SRAM 映射到0 地址处*/
  ldr sp, =8*1024
  bl main
halt:
  b halt
```

事实上，汇编文件有“.S”和“.s”两种后缀，在以“.s”为后缀的汇编文件中，程序完全是由纯粹的汇编代码编写的。所谓的纯粹是相对以“.S”为后缀的汇编文件而言的，由于现代汇编工具引入了预处理的概念，允许在汇编代码（.S）中使用预处理命令。预处理命令以符号“#”开头，包括宏定义、文件包含和条件编译。在U-Boot和Linux内核源码中，这种编程方式运用非常广泛。

main.c文件内容如程序清单1.2所示，main.c中的main函数是运行完_start函数的跳转点。main()中首先定义了一个静态局部变量，初值为12，然后配置S3C6410处理器的GPM端口为输出、下拉模式，并将GPM端口低四位对应的管脚设为高电平（LED驱动管脚的电平为高时，LED熄灭）。最后判断flag是否等于12，如果等于就点亮LED，否则不点亮。从程序上看，这个判断语句好像多此一举、莫名其妙，因为flag期间并没有做任何改变。其实，这个变量是为讲解程序的运行地址和加载地址的概念而定义的，它与程序运行的位置有关。

程序清单1.2 main.c 文件内容

```
/*
  * This is a part of the test project
```

```
 * Author: LiQiang Date: 2013/04/01
 * Licensed under the GPL-2 or later.
 */
#define GPMCON *((volatile unsigned long*)0x7F008820)
#define GPMDAT *((volatile unsigned long*)0x7F008824)
#define GPMPUD *((volatile unsigned long*)0x7F008828)
int main()
{
  static int flag = 12;
  GPMCON = 0x1111; /* 输出模式 */
  GPMPUD = 0x55; /* 使能下拉 */
  GPMDAT = 0x0f; /* 关闭LED */
  if(12 == flag)
    GPMDAT = 0x00;
  else
    GPMDAT = 0x0f;
  while(1);
  return 0;
}
```

将上面两个源码文件处理成中间目标文件，分别输入如下命令行：

```
arm-linux-gcc -o mian.o main.c  -c
arm-linux-gcc -o start.o start.S  -c
```

得到main.o、Start.o两个中间目标文件，供链接器使用。

2. 链接

链接是汇编阶段生成的中间目标文件，相互查找自己所需要的函数与变量，重定向数据，完成符号解析的过程。包括对所有目标文件

进行重定位、建立符号引用规则，同时为变量、函数等分配运行地址。函数与变量可能来源于其他中间文件或者库文件，如果没有找到所需的实现，链接器立即停止链接，给处错误提示。

利用一个链接脚本（.lds 后缀）来指导链接器工作。控制输出节（Outpat section）在映像文件中的布局。fortest.lds 是一个简单的链接脚本，指示了程序的运行地址（又称链接地址）为0x5000_0000以及text段、data段和bss段在映像文件中的空间排布顺序。

fortest.lds文件的内容如下：

```
ENTRY(_start)
SECTIONS
{
  . = 0x50000000;
  . = ALIGN(4);
  .text : {
    start.o (.text)
    * (.text)
  }
  .data : {
    * (.data)
  }
  bss_start = .;
  .bss : {
    * (.bss)
  }
  bss_end = .;
}
```

（1）text段代码段（text segment），通常是用来存放程序执行代码的内存区域。这块区域的大小在程序编译时就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量。

（2）data段数据段（data segment），数据段是存放已经初始化不为0的静态变量的内存区域，静态变量包括全局变量和局部变量，它们与程序有着相同的生存期。

（3）bss 段bss segment，bbs 段与data 段类似，也是存放的静态变量的内存区域。与data 段不同的是，bbs段存放的是没有初始化或者初始化为0 的静态变量，并且bbs段不在生成的可执行二进制文件内。bss_start表示这块内存区域的起始地址，bss_end表示结束地址，它们由编译系统计算得到。未初始化的静态变量默认为 0，因此程序开始执行的时候，在 bss_start 到 bss_end内存中存储的数据都必须是0。

（4）其他段，上面3个段是编译系统预定义的段名，用户还能通过.section伪操作自定义段，在后面的移植过程中会发现，Linux 内核源码中为了合理地排布数据实现特定的功能，定义了各种各样的段。

在宿主机上输入以下命令行，完成中间的目标文件的链接和可执行二进制文件的格式转换。

```
arm-linux-ld  -T test.lds -o test.elf start.o main.o
arm-linux-objcopy -O binary test.elf test.bin
arm-linux-objdump -D test.elf > test.dis
```

如图1.4所示是使用arm-linux-objcopy格式转换工具得到的二进制文件test.bin的内容，这些内容是处理器能够识别的机器码，我们

往往难以直接阅读、理解它们的含义。使用arm-linux-objdump工具生成便于我们阅读的反汇编文件test.dis。

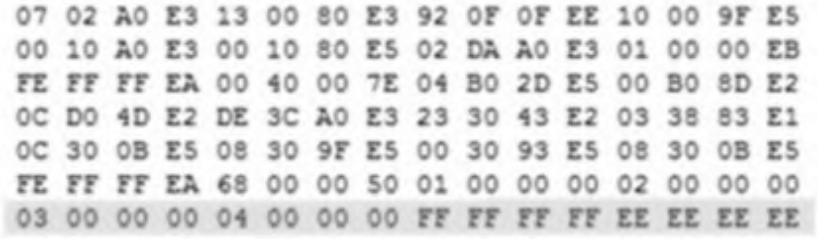


图1.4 二进制镜像文件内容

对比二进制文件test.bin的内容，耐心细致地分析反汇编文件，如程序清单1.3所示，可以提炼出大量的信息。

程序清单1.3 text.dis 文件内容

```
50000000 <_start>: /* 代码段起始位置程序的运行地址为
0x5000_0000*/
    50000000:  e3a00207  mov r0, #1879048192  ; 0x70000000
    50000004:  e3800013  orr r0, r0, #19  ; 0x13
    50000008:  ee0f0f92  mcr 15, 0, r0, cr15, cr2, {4}
    5000000c:  e59f0030  ldr r0, [pc, #48]; 50000044
<halt+0x4>
    50000010:  e3a01000  mov r1, #0  ; 0x0
    50000014:  e5801000  str r1, [r0]
    50000018 <clean_bss>: /* 清除bss段 */
    50000018:  e59f0028  ldr r0, [pc, #40]; 50000048
<halt+0x8>
    5000001c:  e59f1028  ldr r1, [pc, #40]; 5000004c
<halt+0xc>
    50000020:  e3a03000  mov r3, #0  ; 0x0
    50000024:  e1500001  cmp r0, r1
    50000028:  0a000002  beq 50000038 <clean_done>
```

```
5000002c <clean_loop>:
5000002c:  e4803004  str r3, [r0], #4
50000030:  e1500001  cmp r0, r1
50000034:  1afffffc  bne 5000002c <clean_loop>
50000038 <clean_done>:
50000038:  e3a0da02  mov sp, #8192  ; 0x2000 /* 初始化sp
*/
5000003c:  eb000003  bl  50000050 <main> /* 跳转至mian()
*/
50000040 <halt>:
50000040:  eafffffe  b  50000040 <halt>
50000044:  7e004000  .word  0x7e004000
50000048:  500000e0  .word  0x500000e0
5000004c:  500000e0  .word  0x500000e0
50000050 <main>: /* main()*/
50000050:  e52db004  push{fp}  ; (str fp, [sp, #-4]!)
50000054:  e28db000  add fp, sp, #0  ; 0x0
50000058:  e3a0247f  mov r2, #2130706432  ; 0x7f000000
5000005c:  e2822b22  add r2, r2, #34816  ; 0x8800
50000060:  e2822020  add r2, r2, #32  ; 0x20
50000064:  e3a03c11  mov r3, #4352  ; 0x1100
50000068:  e2833011  add r3, r3, #17  ; 0x11
5000006c:  e5823000  str r3, [r2]  /*  GPMCON = 0x1111 */
50000070:  e3a0347f  mov r3, #2130706432  ; 0x7f000000
50000074:  e2833b22  add r3, r3, #34816  ; 0x8800
50000078:  e2833028  add r3, r3, #40  ; 0x28
5000007c:  e3a02055  mov r2, #85  ; 0x55
```

```
50000080:  e5832000  str r2, [r3]  /*  GPMPUD = 0x55 */
50000084:  e3a0347f  mov r3, #2130706432  ; 0x7f000000
50000088:  e2833b22  add r3, r3, #34816  ; 0x8800
5000008c:  e2833024  add r3, r3, #36  ; 0x24
50000090:  e3a0200f  mov r2, #15  ; 0xf
50000094:  e5832000  str r2, [r3]  /* GPMDAT = 0x0f */
50000098:  e59f3038  ldr r3, [pc, #56]; 500000d8
<main+0x88>  /* 读取flag变量存储地址 */
5000009c:  e5933000  ldr r3, [r3]/* 读取flag变量的值 */
500000a0:  e353000c  cmp r3, #12  ; 0xc
500000a4:  1a000005  bne 500000c0 <main+0x70>
500000a8:  e3a0347f  mov r3, #2130706432  ; 0x7f000000
500000ac:  e2833b22  add r3, r3, #34816  ; 0x8800
500000b0:  e2833024  add r3, r3, #36  ; 0x24
500000b4:  e3a02000  mov r2, #0  ; 0x0
500000b8:  e5832000  str r2, [r3]
500000bc:  ea000004  b  500000d4 <main+0x84>
500000c0:  e3a0347f  mov r3, #2130706432  ; 0x7f000000
500000c4:  e2833b22  add r3, r3, #34816  ; 0x8800
500000c8:  e2833024  add r3, r3, #36  ; 0x24
500000cc:  e3a0200f  mov r2, #15  ; 0xf
500000d0:  e5832000  str r2, [r3]
500000d4:  eafffffe  b  500000d4 <main+0x84>
500000d8:  500000dc  .word  0x500000dc
Disassembly of section .data:
500000dc <flag.1245>: /* flag变量的地址为0x5000_00dc，值
为12 */
```

500000dc: 0000000c .word 0x0000000c

从test.dis反汇编文件中可知，test.bin包含了代码段和数据段，并没有包含bss段。我们知道，bbs 内存区域的数据初始值全部为零，区域的起始位置和结束位置在程序编译的时候预知。很容易想到在程序开始运行时，执行一小段代码将这个区域的数据全部清零即可，没必要在 test.bin包含全为0的bss段。编译器的这种机制有效地减小了镜像文件的大小，节约了磁盘容量。

main()函数的核心功能是验证flag变量是否等于12，现在追踪下这个操作的实现过程。要想读取flag的值，必须知道它的存储位置，首先执行指令“ldrr3, [pc, #56]”得到flag变量的地址（指针）。pc与56相加合成一个地址，它是相对pc偏移56产生的。pc+56地址处存放了flag变量的指针0x5000_00dc，读取出来存放到r3寄存器。然后执行指令“ldrr3, [r3]”，将内存0x5000_00dc地址处的值读出，这个值就是 flag，并覆盖 r3 寄存器。最后，判断 r3 寄存器是否等于12。flag变量的地址在链接阶段已经被分配好了，固定在0x5000_00dc处，但是从代码中，我们没有找到对flag 变量赋初值的语句，尽管在main 函数已经用C 语句“flag = 12”对它赋初值。

现提供一个验证程序效果的简单方法：将 S3C6410 处理器设置为SD 卡启动方式，使用SD_Writer软件将test.bin烧写至SD卡中，然后将SD卡插入单板的卡槽，复位启动即可。实际上，启动的时候test.bin被加载到内部SRAM中，SRAM映射到0地址处。这个简单方法可以用来验证一些裸板程序，方法实现的原理和SD_Writer软件用法现在不展开讨论，目前只要会使用即可。复位后，LED并没有点亮。

如果每次编译都要重复输入编译命令，操作起来很麻烦，为此test工程中建立了一个Makefile文件，内容如下：

```
test.bin: start.o main.o
```

```
    arm-linux-ld -T fortest.lds -o test.elf start.o
main.o
    arm-linux-objcopy -O binary test.elf test.bin
    arm-linux-objdump -D test.elf > test.dis
  start.o : start.S
    arm-linux-gcc -o start.o start.S -c
  main.o : main.c
    arm-linux-gcc -o main.o main.c -c
  clean:
    rm *.o test.*
```

当将链接脚本中的运行地址修改为0时，进入test目录，输入“make clean”命令清除旧的文件，再输入“make”重新编译程序，验证新生成的test.bin文件的效果，发现LED全部点亮，产生这个现象的原因在下一个小节讲述。

1.4.2 代码搬运

当程序执行时，必须把代码搬运到链接时所指定的运行地址空间，以保证程序在执行过程中对变量、函数等符号的正确引用。在带有操作系统的嵌入式系统中，这个过程由操作系统负责完成。而在裸机环境下，镜像文件的运行地址由程序员根据具体平台指定，加载地址又与处理器的设计密切相关。通常情况下，启动代码最先执行一段位置无关码，这段代码实现程序从加载地址到运行地址的重定位，或者将程序从外部存储介质直接拷贝至其运行地址。

1．位置无关码

位置无关码必须具有位置无关的跳转、位置无关的常量访问等特点，不能访问静态变量，都是相对pc的偏移量来进行函数的跳转或者

常量的访问。在ARM 体系中，使用相对跳转指令b/bl实现程序跳转。指令中所跳转的目标地址用基于当前 PC 的偏移量来表示，与链接时分配给地址标号的绝对地址值无关，因而代码可以在任何位置正确的跳转，实现位置无关性。

使用ldr伪指令将一个常量读取到非pc的其他通用寄存器中，可实现位置无关的常量访问。例如：

ldr r0, =WATCHDOG

如果使用ldr伪指令将一个函数标号读取到pc，这是一条与位置有关的跳转指令，执行的结果是跳转到函数的运行地址处。

2．运行地址与加载地址

试想一下，当系统上电复位的时候，如果test.bin刚好位于0x5000_0000地址（flag的初值12位于0x5000_00dc），PC指向0x5000_0000地址，那么这段代码按照上述flag变量的读取步骤，能够准确无误地得到结果。但是，如果test.bin位于0地址（flag的初值12位于0xdc，LED不亮时的情况），PC指向0地址，程序依然从0x5000_00dc地址读取flag变量，实际上它的初值位于0xdc。这时从C语言的角度看，出现一个flag不等于它的初值的现象（期间没有改变flag）。出现错误的原因是在程序中使用了位置相关的变量，但运行地址与加载地址不一致（加载地址为 0，运行地址为0x5000_0000）。由此，能够容易理解运行地址和加载地址的含义：

加载地址是系统上电启动时，程序被加载到可直接执行的存储器的地址，也就是程序在RAM或者 Flash ROM 中的地址。因为有些存储介质只能用来存储数据不能执行程序，例如 SD 卡和NAND Flash 等，必须把程序从这些存储介质加载到可以执行的地址处。运行地址就是程序在链接时候确定的地址，比如fortest.lds链接脚本指定了程序的运行地址为0x5000_0000，那么链接器在为变量、函数等分配地址的时候就会以0x5000_0000作为参考。当加载地址和运行地址不相等时，必

须使用与位置无关码把程序代码从它的加载地址搬运至运行地址，然后使用“ldr pc, =label”指令跳转到运行地址处执行。

1.4.3 混合编程

在嵌入式系统底层编程中，C语言和汇编两种编程语言的使用最广泛。C语言开发的程序具有可读性高，容易修改、移植和开发周期短等特点。但是，C语言在一些场合很难或无法实现特定的功能：底层程序需要直接与CPU内核打交道，一些特殊的指令在C语言中并没有对应的成分，例如关闭看门狗、中断的使能等；被系统频繁调用的代码段，对代码的执行效率要求严格的时候。事实上，CPU 体系结构并不一致，没有对内部寄存器操作的通用指令。汇编语言与 CPU的类型密切相关，提供的助记符指令能够方便直接地访问硬件，但要求开发人员对CPU的体系结构十分熟悉。在早期的微处理器中，由于处理器速度、存储空间等硬件条件的限制，开发人员不得不选用汇编语言开发程序。随着微处理器的发展，这些问题已经得到很好的解决。如果依然完全使用汇编语言编写程序，工作量会非常大，系统很难维护升级。大多数情况下，充分结合两种语言的特点，彼此相互调用，以约定规则传递参数，共享数据。

1．汇编函数与C语言函数相互调用

C 程序函数与汇编函数相互调用时必须严格遵循ATPCS（ARMThumb Procedure Call Standard）。函数间约定R0、R1和R2为传入参数，函数的返回值放在R0 中。GNU ARM编译环境中，在汇编程序中要使用.global伪操作声明改汇编程序为全局的函数，可被外部函数调用。在C程序中要被汇编程序调用的C函数，同样需要用关键字extern声明。

程序清单1.4是从arch\arm\cpu\arm1176\start.S文件（U-Boot）中截取的代码片段，relocate_code函数用于重定位代码。它在C 程序中，通过relocate_code（addr_sp, id, addr）被调用。变量addr_sp、id和addr分别通过寄存器R0、R1和R3传递给汇编程序，实现了C函数和汇编函数数据的共享。

程序清单1.4 代码重定位函数

```
.globl relocate_code
relocate_code:
  mov r4, r0  /* save addr_sp */
  mov r5, r1  /* save addr of gd */
  mov r6, r2  /* save addr of destination */
  /* Set up the stack  */
stack_setup:
  mov sp, r4
  adr r0, _start
  cmp r0, r6
  beq clear_bss    /* skip relocation */
  mov r1, r6      /* r1 <- scratch for copy_loop */
  ldr r3, _bss_start_ofs
  add r2, r0, r3    /* r2 <- source end address    */
……
```

2. C语言内嵌汇编

当需要在C语言程序中内嵌汇编代码时，可以使用gcc提供的asm语句功能。

程序清单1.5是从Linux源码文件arch/arm/include/asm/atomic.h截取的一段代码，本节内容不分析函数的具体实现。对于初学者，这

段代码看起来晦涩难懂，因为这不是标准C所定义的形式，而是gcc对C语言扩充的asm功能语句，用以在C语言程序中嵌入汇编代码。

程序清单1.5 整数原子加操作的实现

```
  /*
    * ARMv6 UP and SMP safe atomic ops. We use load
exclusive and
    * store exclusive to ensure that these are atomic. We
may loop
    * to ensure that the update happens.
    */
  static inline void atomic_add(int i, atomic_t *v)
  {
    unsigned long tmp;
    int result;
    __asm__ __volatile__("@ atomic_add\n"
  "1: ldrex  %0, [%3]\n"
  "  add %0, %0, %4\n"
  "  strex  %1, %0, [%3]\n"
  "  teq %1, #0\n"
  "  bne 1b"
    : "=&r" (result), "=&r" (tmp), "+Qo" (v->counter)
    : "r" (&v->counter), "Ir" (i)
    : "cc");
  }
```

asm语句最常用的格式为：

```
  __asm __ __volatile__(" inst1 op1, op2, . \n"
  " inst2 op1, op2, . \n"    /* 指令部分必选*/
```

```
    ...
  " instN op1, op2, . \n"
    : output_operands  /* 输出操作数可选 */
    : input_operands  /* 输入操作数可选 */
    : clobbered_operands /* 损坏描述部分可选*/
  );
```

它由4个部分组成：指令部分、输出部分、输入部分和损坏描述部分。各部分使用“:”隔开，指令部分必不可少，其他3部分可选。但是如果使用了后面的部分，而前面部分为空，也需要用“:”分隔，相应部分内容为空。__asm__表示汇编语句的起始，__volatile__是一个可选项，加上它可以防止编译器优化时对汇编语句删除、移动。

指令部分，指令之间使用“\n”（也可以使用“;”或者“\n\t”）分隔。嵌入汇编指令的格式与标准汇编指令的格式大体相同，嵌入汇编指令的操作数使用占位符“%”预留位置，用以引用 C语言程序中的变量。操作数占位符的数量取决于 CPU 中通用寄存器的总数量，占位符的格式为%0，%1，……，%n。

输出、输入部分，这两部分用于描述操作数，不同的操作数描述语句之间用逗号分隔，每个操作数描述符由限定字符串和 C 语言表达式组成，当限定字符串中带有“=”时表示该操作数为输出操作数。限定字符串用于指示编译器如何处理C语言表达式与指令操作数之间的关系，限定字符串中的限定字母有很多种，有些是通用的，有些跟特定的体系相关。在程序清单1.5中：result、tmp和v->counter是输出操作数，分别赋给%0、%1和%2；v->counter和i是输入操作数，分别赋给%3和%4。其中，“r”表示把变量放入通用寄存器中；“I”表示0-31之间的常数。

1.5 嵌入式Linux移植常用软件

在进行嵌入式Linux学习与开发的过程中，需要使用到一些常用的开发工具，熟练使用这些软件，能让学习与开发达到事半功倍的效果。

1.5.1 SecureCRT

SecureCRT 是可以在 Window 环境下登录 UNIX 和 Linux 服务器主机的软件，它不仅支持SSH1、SSH2，而且支持TeInet和rlogin协议。

在Ubuntu宿主机上安装SSH。

zhuzhaoqi@zhuzhaoqi-desktop:~/sudo apt-get install openssh-server openssh-client

安装完成之后SSH功能自动打开。从网上下载SecureCRT软件，完成之后安装。

打开软件，弹出如图1.5所示的对话框。在“主机名”中输入Ubuntu虚拟机的ip地址，用户名即为Ubuntu宿主机的用户名。

图1.5 SecureCRT快速链接

连接上之后输入Ubuntu 宿主机密码即可进入SecureCRT连接界面。可以看到界面是单色且不支持中文，那是因为默认的字符编码不支持中文。字符编码是把字符集中的字符编码为指定集合中的某一对象，以便文本在计算机中存储和通过通信网络传递。

单击“选项”菜单，选择“会话选项”命令，打开“会话选项”对话框，如图1.6所示。

单击“字体”可以进行字体的任意设置。在“字符编码”中选择UTF-8，UTF-8是一种针对Unicode 的可变长度字符编码，它逐渐成为电子邮件、网页及其他存储或传送文字的应用优先采用的编码。

单击“终端”下的“仿真”进行颜色设置，如图1.7所示。

图1.6 SecureCRT字符编码配置

图1.7 SecureCRT颜色设置

在仿真终端中选择xterm，并且选择“ANSI颜色”复选框。

在Ubuntu宿主机中编辑/etc/profile，为vim编辑添加颜色显示效果。

zhuzhaoqi@zhuzhaoqi-desktop:~/sudo vim /etc/profile

在/etc/profile文末添加如下内容：

export TERM=xterm-color

添加完毕后执行如下内容，使之生效：

zhuzhaoqi@zhuzhaoqi-desktop:~/source /etc/profile

至此就完成了SecureCRT软件的安装。

1.5.2 Source Insight

由于U-Boot、Linux内核源码等都是相当庞大的工程，文件成千上万，为了方便编写和阅读代码，特此向读者推荐Source Insight 编辑器。

有时候，源码分析的难度不只在于源码本身，而在于如何使用更合适的分析代码的工具和手段。Source Insight非常好用，支持几乎所有的语言，如C、C++、ASM、PAS、ASP、HTML等。Source Insight与其他的编辑器产品相比较，增添了分析源代码，并在编辑的同时立刻提供给您有用的信息和分析等众多人性化功能。

目前能找到的最新版本是Source Insight 3.5.0072。从网上下载该软件，并进行安装。过程很简单，读者可自行完成。

安装完成之后打开Source Insight 软件，如图1.8 所示。

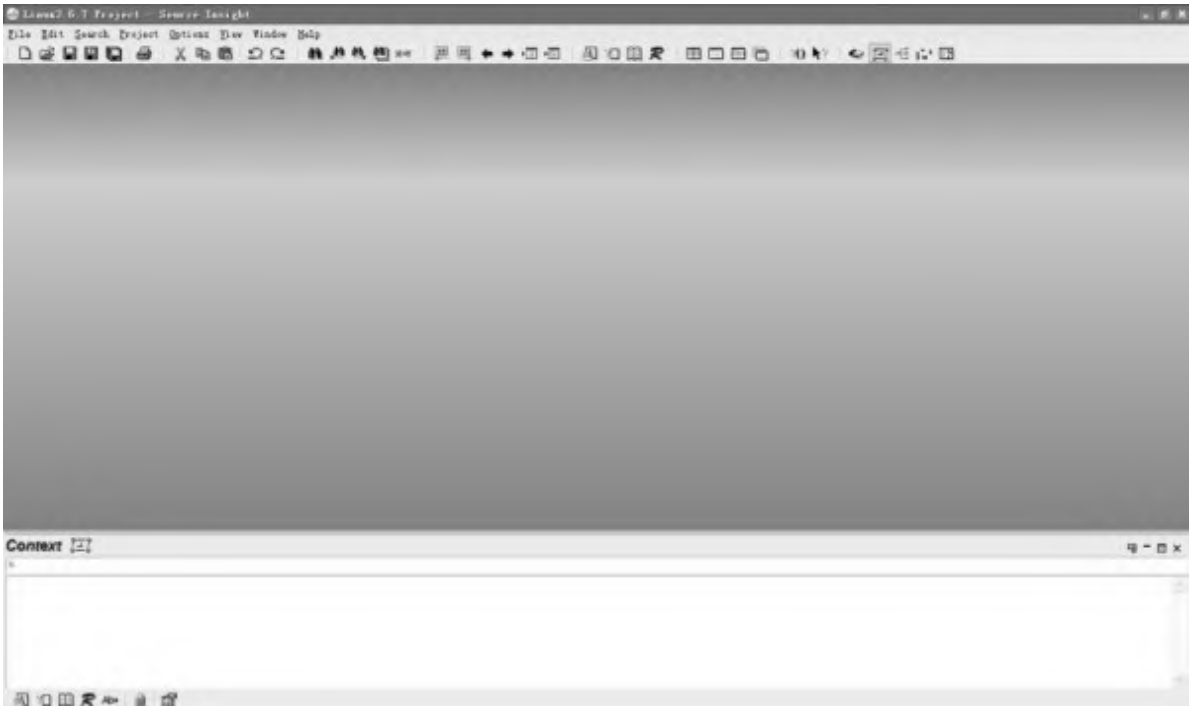


图1.8 打开Source Insight界面

由于Source Insight中C语言文件中所默认支持的只有.c和.h文件，因此需要增添支持其他后缀名的文件。点击“Options”菜单下的“Document Options”子菜单，将弹出如图1.9所示的对话框。

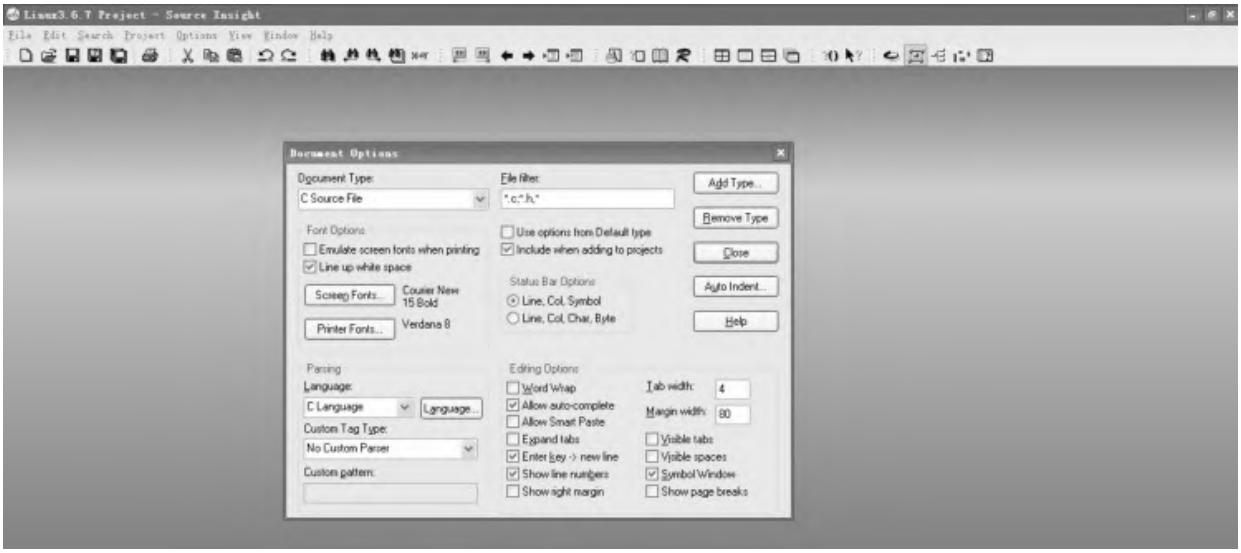


图1.9 添加支持其他后缀名文件

在“File filter”下面添加“*”，表示支持任何后缀名文件。

单击“Project”菜单下的“New Project”子菜单，弹出如图1.10 所示的对话框。

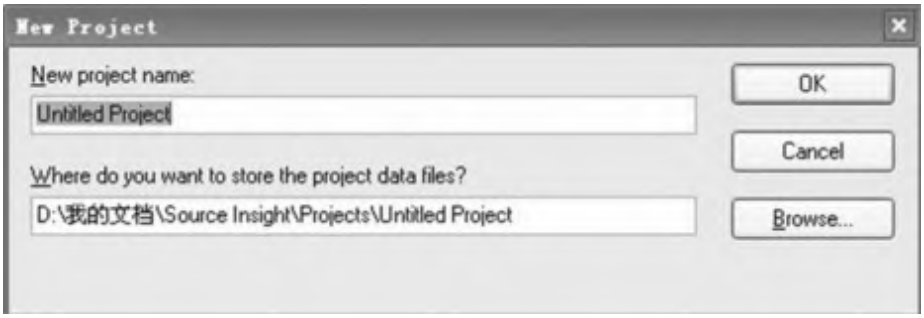


图1.10 新建工程

单击“Browse…”按钮可以更改工程存放路径，在“New project name”文本框下面输入新建工程的名字。完成之后单击“OK”按钮，进入如图1.11所示的对话框。

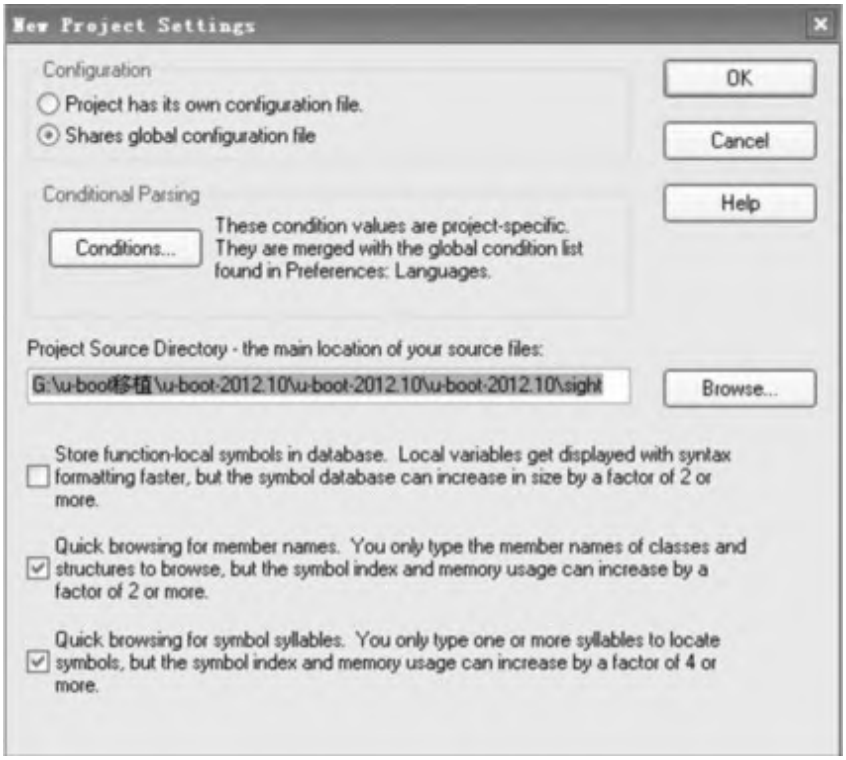


图1.11 源码存放位置

确定源码存放位置之后，单击“OK”按钮，进入添加源码界面，如图1.12所示。

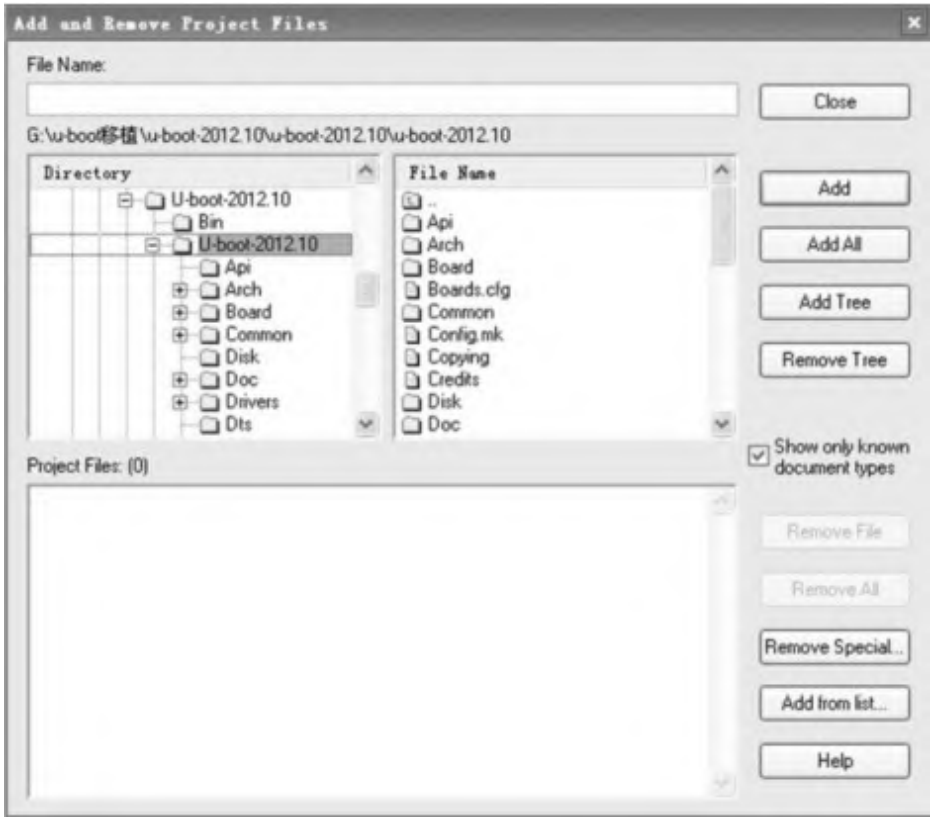


图1.12 给工程添加源码

将所需要添加的源码——添加入工程。如果需要将所有U-Boot-2012.10工程文件全部添加进入工程，则单击左侧的U-Boot-2012.10目录，在单击“Add All”按钮，则全部添加，进入如图1.13所示的界面。

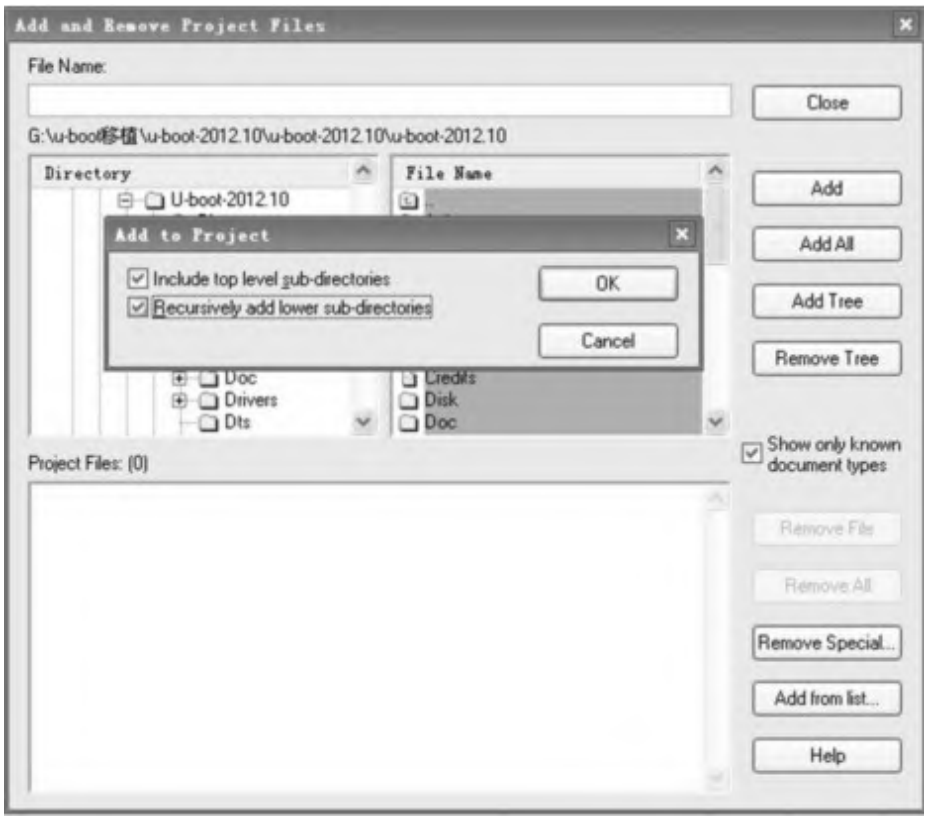


图1.13 添加文件

完成之后，单击“Project”菜单下的“Synchronize Files…”命令，同步所有工程文件，如图1.14所示。

图1.14 同步文件

单击“OK”按钮开始同步文件，这样工程中的各个变量、函数之间的关系就可以快速查阅了。同步完成之后便可进入阅读和编写工程文件，如图1.15所示。

图1.15 工程文件阅读和编写界面

第2章 U-Boot-2013.04分析与移植

2.1 BootLoader概述

从最终用户的角度看，BootLoader（即启动代码）是处理器复位后进入操作系统之前执行的一段代码，用以完成由硬件启动到操作系统启动的过渡，为操作系统的运行提供基本的环境，如关闭看门狗、初始化时钟和配置存储器等。启动代码的最终目的是引导操作系统的启动，但从开发人员的角度看，为了开发和调试的方便，还会增加串口控制、以太网络等功能。

嵌入式系统与应用密切结合，它具有很强的专用性。实际系统的需求往往千差万别， BootLoader代码与CPU的类型、应用系统的配置及使用的操作系统等因素密切相关，这就注定了不可能有完全通用的BootLoader，实际运用时必须根据具体情况对启动代码进行移植。

本文所写的内容都是基于表2.1所示配置的单板（board）：NAND芯片K9GAG08U0D共4096块，每一块包含128 页，每页由4096 字节的数据区和218 字节的空闲区组成。两片64 MB×16 bit的Mobile DDR 芯片K4X1G163PC，组合构成共256 MB 的内存。尽管每个人持有的单板配置各异，但分析、移植的原理相通。

表2.1 开发板配置

类　别	型　号	规　格
CPU	S3C6410	—
NAND Flash	K9GAG08U0D	2GB
DRAM	K4X1G163PC	128MB*2
Ethernet	DM9000A	—
LCD	WXCAT43	4.3 寸

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章01课（Linux常用软件及BootLoader介绍）。

2.2 U-Boot初步分析

U-Boot，全称为Universal Boot Loader（通用bootloader），是遵循GPL 条款的开放源码项目，由德国DENX小组开发和维护。U-Boot能够非常容易地被移植到多种嵌入式CPU中，支持多种嵌入式操作系统内核的引导。不少U-Boot源码就是Linux内核源码的简化，特别是一些设备的驱动程序，使得它在引导 Linux Kernel 方面尤为出色。本文选用当前最新版本的 U-Boot，结合S3C6410处理器自身的特点，分析和探讨它的移植要点。笔者认为移植工作必须在充分理解源码的组织方式、处理器特点、单板外围器件原理等基础上进行。充分利用源码已经实现的功能，最小限度地破坏源码的结构。

2.2.1 源码结构

截止本书编写日期，U-Boot的最新版本为u-boot-2013.04-rc1.tar.bz2，U-Boot所有版本的下载地址为：http://ftp.denx.de/pub/u-boot/。rc（Release Candidate）表示正式发行候选版，1 代表版本号，rc1 即候选版的第一版。rc 版本发布于软件的正式定稿之前，期间不会再加入新的功能或模块，这一版本

的发布主要是为了让开源社区经验丰富的开发者率先试用以及时反馈和修正源码中存在的错误、漏洞，这个阶段过后就会发布相对稳定的正式版。笔者在移植该版本的时候，发现几处较为明显的错误，并通过U-Boot的邮件列表反馈了这些错误信息。这些错误的位置以及修正方法将会在移植的时候逐一介绍。

读者阅读本书时，u-boot-2013.04正式版本可能已经发布，甚至有更新版本的源码发布。其实U-boot版本间的差别并不是很大，较新版本仅仅在前面版本的基础上，增加或者修改了一些驱动程序，对源码的结构稍微的调整，但核心内容不会做太大的改变。但相比2010-03以前的版本，U-Boot后面的版本的源码组织结构作了较大的调整，使得其源码目录、编译形式与Linux Kernel源码愈加相似。

liqiang@ubuntu:~/work/forbook$ tar -jxvf u-boot-2013.04-rc1.tar.bz2 -C ./

输入 tar 命令，解压源码到当前文件夹。进入 u-boot-2013.04-rc1 目录，其中有个文件名为“readme”的帮助文档，通读readme文件，我们能够大体上了解U-Boot获得和寻求帮助的途径，源码目录的组织结构，工程配置、编译的方法等。U-Boot顶层目录有很多子目录，下面介绍一些主要的目录。

（1）arch：对应不同构架的CPU，子目录的名字就是所支持的CPU构架的名称，如arch目录下含有arm、avr32 以及x86等，这些目录可以继续细分。例如arm下含有cpu 、include 和lib等目录，其中cpu用于进一步区分不同体系的arm内核。

（2）common：该目录存在的是一些公共的通用代码文件，包括用于实现各种公共命令的cmd_*.c、env_*.c 文件以及一些通用函数。它们独立于处理器的体系结构，直接跟用户打交道，是用户与设备驱动之间沟通的纽带，也是U-Boot的精髓所在。

（3）drivers：该目录存放的是各类外设的驱动程序，如mmc、serial和net等。

（4）fs：支持文件系统。

（5）net：该目录里面的文件实现了各种网络协议。

（6）nand_spl：实现U-Boot从NAND Flash 中启动的设备驱动。

（7）include：一些头文件和单板配置文件，所有单板配置的文件位于include/configs目录下。

（8）tools：常用工具，包括用于制作uImage的mkimage工具。

（9）Makefile：控制着整个工程的编译。

2.2.2 建立模板

u-boot-2013-04-rc1中没有对S3C6410处理器相关单板的支持，但支持带有S3C6400处理器的SMDK6400单板。如果我们要自己编写所有的启动代码，工作量很大且很容易出错。S3C6400和S3C6410 是三星公司推出的S3C64xx 系列的处理器，都是基于16/32-bit RISC内核的低成本、低功耗、高性能微处理器解决方案。它们的功能基本相同，硬件管脚兼容。如果用现有的SMDK6400作为模板，就能够帮助我们迅速地建立起一个大致的框架，后期再根据二者的异同点填充、修改框架的内容。

事实上，就算U-Boot源码中支持S3C6410相关的单板，由于嵌入式系统应用的场合不一样、需求不同，因此单板之间的配置不可能完全一致，移植U-boot的工作也必须根据实际情况进行。在本节内容中，将会带领大家一步一步地建立一个能够通过编译最小模板，修正一些源码自身的错误，更多功能的移植再分章节具体阐述。

1. 修改顶层Makefile

liqiang@ubuntu:~/work/forbook$ cd u-boot-2013.04-rc1/

输入cd命令进入源码目录。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ vim Makefile

用vim文本编辑器打开顶层Makefile文件。

```
  792 smdk6400_noUSB_config \
  793 smdk6400_config :    unconfig
  794     @mkdir -p $(obj)include
$(obj)board/samsung/smdk6400
  795     @mkdir -p $(obj)nand_spl/board/samsung/smdk6400
  796     @echo "#define CONFIG_NAND_U_BOOT" >
$(obj)include/config.h
  797     @echo "CONFIG_NAND_U_BOOT = y" >>
$(obj)include/config.mk
  798     @if [ -z "$(findstring
smdk6400_noUSB_config,$@)" ]; then
        \
  799         echo "RAM_TEXT = 0x57e00000" >>
$(obj)board/samsung/smdk
        6400/config.tmp;\
  800     else
        \
  801         echo "RAM_TEXT = 0xc7e00000" >>
$(obj)board/samsung/smdk
        6400/config.tmp;\
  802     fi
  803     @$(MKCONFIG) smdk6400 arm arm1176 smdk6400
samsung s3c64xx
```

```
804    @echo "CONFIG_NAND_U_BOOT = y" >>
  $(obj)include/config.mk
```

将这段内容复制，紧跟在这段内容的最后一行粘贴，然后修改成如下内容（6400改成6410）：

```
806 smdk6410_noUSB_config \
807 smdk6410_config :    unconfig
808    @mkdir -p $(obj)include
$(obj)board/samsung/smdk6410
809    @mkdir -p $(obj)nand_spl/board/samsung/smdk6410
810    @echo "#define CONFIG_NAND_U_BOOT" >
$(obj)include/config.h
811    @echo "CONFIG_NAND_U_BOOT = y" >>
$(obj)include/config.mk
812    @if [ -z "$(findstring smdk6410_noUSB_config,$@)"
]; then
        \
813        echo "RAM_TEXT = 0x57e00000" >>
$(obj)board/samsung/smdk
        6410/config.tmp;\
      814    else
        \
      815        echo "RAM_TEXT = 0xc7e00000" >>
  $(obj)board/samsung/smdk
        6410/config.tmp;\
      816fi
      817    @$(MKCONFIG) smdk6410 arm arm1176 smdk6410
  samsung s3c64xx
```

```
818     @echo "CONFIG_NAND_U_BOOT = y" >>
$(obj)include/config.mk
```

2. 创建SMDK6410单板信息

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ mkdir board/samsung/smdk6410

在board/samsung目录下创建smdk6410目录。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ cp board/samsung/smdk6400/*board/samsung/smdk6410

将board/samsung/smdk6400目录包含的所有文件复制到board/samsung/smdk6410目录中。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ cd board/samsung/smdk6410/

进入board/samsung/smdk6410目录。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1/board/samsung/smdk6410$ mvsmdk6400_nand_spl.c smdk6410_nand_spl.c

把smdk6400_nand_spl.c文件重命名为smdk6410_nand_spl.c。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1/board/samsung/smdk6410$ mvsmdk6400.c smdk6410.c

同样把smdk6400.c文件重命名为smdk6410.c。

```
95 int checkboard(void)
96 {
97    printf("Board:  SMDK6400\n");
98    return 0;
99 }
```

打开smdk6410.c文件，把以上单板信息打印函数修改为：

```
95 int checkboard(void)
```

```
96 {
97    printf("Board:  SMDK6410\n");
98    return 0;
99 }
```

打开当前目录下的Makefile 文件，将COBJS-y := smdk6400.o 修改为smdk6410.o。

3. 建立nand_spl

nand_spl是U-Boot 专门为NAND Flash 启动而设计的，其实现原理后文将会详细阐述，现在只是简单地建立模板。

```
liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ mkdir
nand_spl/board/samsung/smdk6410
liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ cp
nand_spl/board/samsung/
smdk6400/*nand_spl/board/samsung/smdk6410/
```

与上述创建单板方法类似，创建nand_spl/board/samsung/smdk6410/目录，并把nand_spl/board/samsung/smdk6400目录中的所有文件拷贝到创建的目录下。

进入 nand_spl/board/samsung/smdk6410/目录，打开当前目录下的 Makefile 文件，将所有的“6400”字符串修改为“6410”，例如smdk6400_nand_spl.o修改为smdk6410_nand_spl.o。

4. 创建s3c6410.h头文件

```
liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ cp
arch/arm/include/asm/arch-s3c64xx/ s3c6400.h
arch/arm/include/asm/arch-s3c64xx/s3c6410.h
```

以头文件s3c6400.h为蓝本，先简单地建立S3C6410处理器的寄存器头文件s3c6410.h。

820 #define DMC1_MEM_CFG 0x00010012 /* burst 4, 13-bit row, 10-bit col */

821 #define DMC1_MEM_CFG2 0xB45

822 #define DMC1_CHIP0_CFG 0x150F8 /* 0x5000_0000~0x57ff_ffff (128 MiB) */

本书所用单板外接的DRAM大小为256MB，而源码默认支持的128MB，因此需要对其进行修改。打开s3c6400.h文件，将以上内容修改为：

820 #define DMC1_MEM_CFG 0x0001001a /* burst 4, 14-bit row, 10-bit col */

821 #define DMC1_MEM_CFG2 0xB45

822 #define DMC1_CHIP0_CFG 0x150F0 /* 0x5000_0000~0x5fff_ffff (256MiB) */

5. 修改处理器Makefile

liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ vim arch/arm//cpu/arm1176/s3c64xx/Makefile

打开s3c640xx目录中的Makefile文件，增加对S3C6410处理器的支持。

33 COBJS-$(CONFIG_S3C6400) += cpu_init.o speed.o

34 COBJS-$(CONFIG_S3C6410) += cpu_init.o speed.o # add here !

6. 创建SMDK6410顶层配置文件

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1$ cp include/configs/smdk6400.hinclude/configs/smdk6410.h

以 SMDK6400 的顶层配置文件 smdk6400.h 为蓝本创建 SMDK6410 单板的顶层配置文件smdk6410.h，并进行初步修改。

```
    /*
    * High Level Configuration Options
    * (easy to change)
    */
    #define CONFIG_S3C6400    1    /* in a SAMSUNG S3C6400
SoC  */
    #define CONFIG_S3C64XX    1    /* in a SAMSUNG S3C64XX
Family  */
    #define CONFIG_SMDK6400    1    /* on a SAMSUNG SMDK6400
Board */
```

将以上内容修改为SMDK6410的配置选项。

```
    /*
    * High Level Configuration Options
    * (easy to change)
    */
    #define CONFIG_S3C6410    1    /* in a SAMSUNG S3C6410
SoC  */
    #define CONFIG_S3C64XX    1    /* in a SAMSUNG S3C64XX
Family  */
    #define CONFIG_SMDK6410    1    /* on a SAMSUNG SMDK6410
Board  */
```

修改监控命令提示符，提示符可以根据个人的喜好修改。

```
    131#define CONFIG_SYS_PROMPT      "SMDK6400 # "  /*
Monitor Command Prompt  */
    131#define CONFIG_SYS_PROMPT      "lq@u-boot#"  /*
Monitor Command Prompt  */
```

修改识别字符串。

196 #define CONFIG_IDENT_STRING " for SMDK6400"

196 #define CONFIG_IDENT_STRING " for SMDK6410"

修改DRAM的大小。

166 #define PHYS_SDRAM_1_SIZE   0x08000000 /* 128 MB in Bank #1 */

166 #define PHYS_SDRAM_1_SIZE   0x10000000 /* 256 MB in Bank #1 */

7. 其他修改

进入board/samsung/smdk6410/目录，该目录下的文件仅仅与单板相关，直接修改不会影响其他单板的编译。打开smdk6410.c和lowlevel_init.S文件。

将这两个文件中的#include <asm/arch/s3c6400.h>改为#include <asm/arch/s3c6410.h>，所有的宏CONFIG_S3C6400修改为CONFIG_S3C6410。

cpu_init.S (arch\arm\cpu\arm1176\s3c64xx):#include <asm/arch/s3c6400.h>

reset.S (arch\arm\cpu\arm1176\s3c64xx):#include <asm/arch/s3c6400.h>

speed.c (arch\arm\cpu\arm1176\s3c64xx):#include <asm/arch/s3c6400.h>

timer.c (arch\arm\cpu\arm1176\s3c64xx):#include <asm/arch/s3c6400.h>

s3c64xx-hcd.c (drivers\usb\host):#include <asm/arch/s3c6400.h>

s3c64xx.c (drivers\mtd\nand):#include <asm/arch/s3c6400.h>

s3c64xx.c (drivers\serial):#include <asm/arch/s3c6400.h>

上面这些文件并不是SMDK6410单板独有的文件，如果直接将s3c6400.h改为s3c6410.h，会破坏SMDK6400的源码结构，而编译SMDK6410单板时必须包含s3c6410.h头文件，可以利用如

下预处理命令解决这个问题。

```
#ifdef CONFIG_S3C6400
#include <asm/arch/s3c6400.h>
#else
#include <asm/arch/s3c6410.h>
#endif
```

打开arch/arm/cpu/arm1176/s3c64xx/speed.c文件，修改打印的CPU类型。

```
139 #ifdef CONFIG_S3C6400
140    printf("\nCPU:  S3C6400@%luMHz\n", get_ARMCLK() / 1000000);
141 #else
142    printf("\nCPU:  S3C6410@%luMHz\n", get_ARMCLK() / 1000000);
143 #endif
```

增加顶层控制宏。进入 common.h （ include ）文件，用同样的方式修改 ohci-hcd.c（drivers/usb/host）。

```
642 #if defined(CONFIG_S3C24X0) || \
643  defined(CONFIG_LH7A40X) || \
644  defined(CONFIG_S3C6400) || \
645  defined(CONFIG_EP93XX)
```

添加CONFIG_S3C6410宏后修改为：

```
642 #if defined(CONFIG_S3C24X0) || \
643  defined(CONFIG_LH7A40X) || \
```

```
644  defined(CONFIG_S3C6400) || \
645  defined(CONFIG_S3C6410) || \
646  defined(CONFIG_EP93XX)
```

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第二章 02 课（U-Boot-2013.04搭建适合OK6410模板）。

2.2.3 编译源码

U-Boot 支持将编译生成的文件与源码文件分开放置，可以通过两种方式指定生成文件的目录。

（1）在命令行参数添加中添加“O =”。

```
liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ make O=build
```

（2）给环境参数变量BUILD_DIR赋值，这个值就是我们期望中间文件存放的位置。

```
liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ export BUILD_DIR=./build liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ make
```

为了保持源代码目录的干净，推荐用以上方式将编译生成的文件输出到一个外部目录。如果没有指定生成文件的目录，则默认为源码顶层目录。

```
liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ make O=../buildsmdk6410_config
liqiang@liqiang-virtual-machine:~/work/forbook/u-boot-2013.04-rc1$ make O=../build
```

输入命令行，编译后提示错误，打印出以下错误信息。

arm-linux-ld:/home/liqiang/work/forbook/build/u-boot.lds:19: syntax error

make: *** [/home/liqiang/work/forbook/build/u-boot] 错误 1

事实上，这是源码出现的第一个bug，u-boot.lds链接脚本的语法有误。u-boot.lds是在编译的时候临时生成的链接脚本，它生成的依据之一是 u-boot-nand.lds 链接文件，该文的位置为board/samsung/smdk6410。打开u-boot-nand.lds，发现内存4的倍数对齐的描述与书写有误，必须大写。

```
51    . = align(4);
52    .u_boot_list : {
53        #include <u-boot.lst>
54    }
55
56    . = align(4);
```

修改为：

```
51    . = ALIGN(4);
52    .u_boot_list : {
53        #include <u-boot.lst>
54
}
55
56    . = ALIGN (4);
```

继续输入make O=../build 编译，出现错误提示信息：

start.o: In function `cpu_init_crit':

/home/liqiang/work/forbook/build/nand_spl/board/samsung/smdk6410/start.S:227:undefined reference to '_main'

make[1]: *** [/home/liqiang/work/forbook/build/nand_spl/u-boot-spl] 错误 1

make[1]:正在离开目录 '/home/liqiang/work/forbook/u-boot-2013.04-rc1/nand_spl/board/samsung/smdk6410'

make: *** [nand_spl] 错误 2

事实上，这是源码出现的第二个bug，修改方法如下：

打开Makefile（nand_spl/board/samsung/smdk6410），添加crt0.S（arch/arm/lib/）文件编译。

```
40 SOBJS = start.o cpu_init.o lowlevel_init.o crt0.o
...
69 $(obj)start.S:
70    @rm -f $@
71    @ln -s $(TOPDIR)/arch/arm/cpu/arm1176/start.S $@
72 $(obj)crt0.S:
73    @rm -f $@
74    @ln -s $(TOPDIR)/arch/arm/lib/crt0.S $@
```

继续编译又出现下面的错误提示信息：

/home/liqiang/work/forbook/build/nand_spl/board/samsung/smdk6410/crt0.S:153:undefined reference to `coloured_LED_init'

/home/liqiang/work/forbook/build/nand_spl/board/samsung/smdk6410/crt0.S:154:undefined reference to `red_led_on'

make[1]: *** [/home/liqiang/work/forbook/build/nand_spl/u-boot-spl] 错误 1

make[1]:正在离开目录 '/home/liqiang/work/forbook/u-boot-2013.04-rc1/nand_spl/board/samsung/smdk6410'

make: *** [nand_spl] 错误 2

这是源码的第三个bug，打开arch/arm/lib/crt0.S文件，增加条件编译。

```
152 #ifndef CONFIG_NAND_SPL
153     bl coloured_LED_init
154     bl red_led_on
155 #endif
```

终于编译过程顺利通过，一个简单的框架搭建完成。用ls命令列出build目录中的所有文件，内容如下所示。

```
api   examples   net     u-boot.bin
arch   fs     post     u-boot.lds
board   include   System.map   u-boot.map
commoninclude2   test     u-boot-nand.bin
disk   lib     tools     u-boot.srec
drivers nand_spl u-boot
```

第一个bug是链接脚本语法有误，很容易理解。第二、第三个bug出现的原因与nand_spl机制有关，将会在后文详细介绍。尽管目前已经到了U-Boot编译生成的u-boot.bin和u-boot-nand.bin二进制文件，单板并没有配置可以直接存储和运行程序的NOR Flash，到目前为止我们依然无法验证移植是否成功。为了加深对U-Boot移植要点的理解，本书移植U-Boot不借助第三方已经移植好的BootLoader烧写程序，所有驱动程序自行编写。事实上，要实现把编译好的代码在单板上运行试验必须利用2.3.2小节的SD卡启动方法。在此之前我们先分析一下U-Boot的启动流程，为后续内容打好基础。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章03课（初步编译U-Boot-2013.04）。

2.2.4 启动分析

在2.1节中，我们已经知道BootLoader的实现依赖于处理器的体系结构。为了移植的方便，大多数BootLoader可以分为两个阶段stage1和stage2。依赖于处理器体系结构的代码，比如 CPU 初始化，一般都放在 stage1 阶段，通常多用汇编语言来实现，stage1 必须是位置无关码。stage2通常用 C 语言来实现，这样可以实现给复杂的功能，而且代码会具有更好的可读性和可移植性。U-Boot也不例外，第一阶段主要使用汇编语言编写，程序的入口在start.s中。stage1在运行时，有可能不在其运行地址，这时不能使用静态变量，必须利用位置无关码进行编程。

U-Boot在stage1阶段经常会出现CONFIG_NAND_SPL和CONFIG_SPL_BUILD两个宏，用于控制程序的条件编译。在编译生成 u-boot.bin 时，它们为假。去掉一些无关紧要的过程和条件编译判定无效的代码段，我们通过分析u-boot.bin的生成过程来分析U-Boot的启动流程。

1．程序入口

```
.globl _start
_start: b  reset
```

.globl：如果一个符号没有用.globl声明，就表示这个符号不会被链接器用到。

b：是跳转指令，ARM的跳转指令可以从当前指令向前或者向后的32MB的地址空间跳转（相对跳转指令），是一种位置无关码，这类跳转指令有以下4种：

（1）B：跳转指令。

（2）BL：带返回的跳转指令。

（3）BX：带状态切换的跳转指令。

（4）BLX：带返回和状态切换的跳转指令。

```
_start: b  reset
```

而这句跳转时，PC寄存器的值将不会保存到LR寄存器中。

2．设置ARM工作模式

```
reset:
  /*
    * set the cpu to SVC32 mode
    */
  mrs r0, cpsr
  bic r0, r0, #0x3f
  orr r0, r0, #0xd3
  msr cpsr, r0
```

ARM微处理器支持如下7种运行模式，分别为：

（1）用户模式（usr）：ARM处理器正常的程序执行状态。

（2）快速中断模式（fiq）：用于高速数据传输或通道处理。

（3）外部中断模式（irq）：用于通用的中断处理。

（4）管理模式（svc）：操作系统使用的保护模式。

（5）数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。

（6）系统模式（sys）：运行具有特权的操作系统任务。

（7）定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

cpsr为当前程序状态寄存器，它包含了条件标志位、中断禁止位、当前处理器模式标志以及其他的一些控制和状态位。cpsr可以在任何处理器模式下被访问。cpsr的格式如图2.1所示。

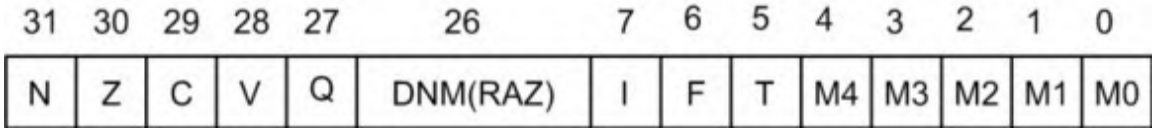


图2.1 CPSR寄存器

N、Z、C、V这四位统称为条件标志位。

cpsr的第8位统称为控制位。

返回上面代码分析，mrs指令是读状态寄存器指令，如下所示：

```
mrs r0, cpsr
```

这行代码的含义是将cpsr状态寄存器读取，保存到r0中。

bic指令是位清除指令，如下所示：

```
bic r0, r0, #0x3f
```

这行代码的作用是将r0的低6位清零。

orr指令是或运算，如下所示：

```
orr r0, r0, #0xd3
```

将r0与1101 0011 进行或运算，由于之前进行了位清零，那么此时r0 的低8 位为：1101 0011。

msr指令是写状态寄存器指令，如下所示：

```
msr cpsr, r0
```

将r0数据写入cpsr程序状态寄存器。同样cpsr 的低8 位即为：1101 0011。那么这8位的含义如下。

（1）第7位，即为I位，当I=1时禁止IRQ中断。

（2）第6位，即为F位，当F=1时禁止FIQ中断。

（3）第5位，即为T位，当T=1时执行ARM指令；当T=0时执行Thumb指令。

（4）低5 位，即为M[4:0]，当M[4:0] = 0x13，即为管理模式。

接着进入cpu_init_crit，即cpu初始化阶段。

3. caches初始化

```
mov r0, #0
mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
```

mov指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。

mov r0, #0

这行代码即表示将0这个立即数加载到r0寄存器中。

mcr 指令将 ARM 处理器的寄存器中的数据传递到协处理器的寄存器中。如果协处理器不能成功地执行该操作，将产生未定义的指令异常中断。

mcr p15, 0, r0, c7, c7, 0

指令从ARM寄存器中将数据传送到协处理器p15的寄存器中，其中r0为ARM寄存器，存放源操作数；c7和c7为协处理器寄存器，为目标寄存器；p15和r0之间的0为操作码1；最后的0为操作码2。

上面这行代码的作用是向c7写入0，使ICache与DCache无效。

mcr p15, 0, r0, c8, c7, 0

而这行代码的作用是向c8写入0，使TLB失效。

4. MMU初始化

```
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002300  @ clear bits 13, 9:8 (--V- --RS)
    bic r0, r0, #0x00000087  @ clear bits 7, 2:0 (B--- -CAM)
    orr r0, r0, #0x00000002  @ set bit 2 (A) Align
    orr r0, r0, #0x00001000  @ set bit 12 (I) I-Cache
  • • • • • •
  mmu_disable:
    mcr p15, 0, r0, c1, c0, 0
```

mrc 指令将协处理器寄存器中的数值传送到 ARM 处理器的寄存器中。如果协处理器不能成功地执行该操作，将产生未定义的指令异常中断。

mrc p15, 0, r0, c1, c0, 0

指令将协处理器p15寄存器中的数据传送到ARM寄存器中。其中，r0为ARM寄存器，是目标寄存器；c1和c0为协处理器寄存器，存放源操作数；p15和r0之间的0是操作码1；最后0是操作码2。

上面这行代码就读出寄存器数据到r0。在剖析这段代码之前，先要了解p15的c1 寄存器格式，如图2.2所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		V	I			R	S	B					C	A	M

图2.2 p15的c1寄存器

V：表示异常向量表所在的位置，其值为0表示异常向量在0x00000000；其值为1表示异常向量在0xFFFF0000。

I：其值为0表示关闭Icaches；其值为1表示开启Icaches。

R、S：用来与页表中的描述符一起确定内存的访问权限。

B：其值为0表示CPU为小字节序；其值为1表示CPU为大字节序。

C：其值为0表示关闭DCaches；其值为1表示开启Dcaches。

A：其值为0表示数据访问时不进行地址对齐检查；其值为1表示数据访问时进行地址对齐检查。

M：其值为0表示关闭MMU；其值为1表示开启MMU。

到这里，再逐句代码分析。

bic r0, r0, #0x00002300

2300 即为0010 0011 0000 0000，即是将r0 的第13、9、8 位清零。

bic r0, r0, #0x00000087

0087 即为0000 0000 1000 0111，即是将r0 的第7、2、1、0 位清零。

orr r0, r0, #0x00000002

5. 外设的基地址初始化

```
#ifdef CONFIG_PERIPORT_REMAP
  /* Peri port setup */
  ldr r0, =CONFIG_PERIPORT_BASE
  orr r0, r0, #CONFIG_PERIPORT_SIZE
  mcr p15,0,r0,c15,c2,4
#endif
```

arm11 把内存（memory）区间和外设（peripheral）区间地址分开，在 CPU 初始化的时候，需要通过协处理器指令CP15告诉CPU外设寄存器的地址范围。如果没有这样做，CPU默认为内存访问，也就无法访问到外设区间的寄存器。

6. 调用lowlevel_init函数

接下来是执行带返回跳转指令“bl lowlevel_init”。调用 lowlevel_init 函数（位于 board\samsung\smdk6410\lowlevel_init.s）。lowlevel_init 函数的工作是进行与单板相关的初始化工作，故名思议，这个初始化仅仅是最低限度（lowlevel）的，包括 led 灯配置（便于观察现象）、关闭看门狗、设置中断、配置系统时钟、初始化串口、初始化内存和初始化唤醒复位。

（1）配置led

```
ldr r0, =ELFIN_GPIO_BASE
ldr r1, =0x55540000
str r1, [r0, #GPNCON_OFFSET]
ldr r1, =0x55555555
str r1, [r0, #GPNPUD_OFFSET]
ldr r1, =0xf000
str r1, [r0, #GPNDAT_OFFSET]
```

这里应该改成与s3c6410相适应的配置，单板使用GPM0-GPM3管脚驱动led。根据s3c6410用户手册中的端口M控制寄存器章节可以对程序作出如下修改。

```
/* LED on only #8 */
ldr r0, =ELFIN_GPIO_BASE
ldr r1, =0x00111111
str r1, [r0, #GPMCON_OFFSET]
ldr r1, =0x00000555
str r1, [r0, #GPMPUD_OFFSET]
/* all of LEDs are power on */
ldr r1, =0x000f
str r1, [r0, #GPMDAT_OFFSET]
```

根据需要，LED测试自行修改：

```
/* LED test */
ldr r0, =ELFIN_GPIO_BASE
ldr r1, =0x0003
str r1, [r0, #GPMDAT_OFFSET]
```

（2）关闭看门狗

```
ldr r0, =0x7e000000    @0x7e004000
orr r0, r0, #0x4000
mov r1, #0
str r1, [r0]
```

大多数微处理器都带有看门狗，当看门狗没有被定时清零（喂狗）时，将引起复位，这可防止程序跑飞，也可以防止程序运行时出现死循环。设计者必须清楚看门狗的溢出时间以决定在合适的时候清除看门狗。在内核中通常用于防止出现死循环，U-Boot直接关闭看门狗。

（3）设置中断

```
/* External interrupt pending clear */
    ldr r0, =(ELFIN_GPIO_BASE+EINTPEND_OFFSET)
/*EINTPEND*/
    ldr r1, [r0]
    str r1, [r0]
    ldr r0, =ELFIN_VIC0_BASE_ADDR @0x71200000
    ldr r1, =ELFIN_VIC1_BASE_ADDR @0x71300000
    /* Disable all interrupts (VIC0 and VIC1) */
    mvn r3, #0x0
    str r3, [r0, #oINTMSK]
    str r3, [r1, #oINTMSK]
    /* Set all interrupts as IRQ */
    mov r3, #0x0
    str r3, [r0, #oINTMOD]
    str r3, [r1, #oINTMOD]
    /* Pending Interrupt Clear */
    mov r3, #0x0
    str r3, [r0, #oVECTADDR]
    str r3, [r1, #oVECTADDR]
```

（4）配置系统时钟

S3C6410有3个PLL（锁相环），分别为APLL、MPLL和EPLL。其中APLL产生ACLK，给CPU使用，MPLL产生HCLKX2、HCLK和PCLK，HCLKX2主要提供时钟给DDR使用，最大可以到266MHz。HCLK用作AXI\AHB总线时钟，PCLK用作APB总线时钟。接AXI和AHB总线的外设最大时钟为133MHz，接 APB 总线的外设最大时钟为 66MHz。UART 的时钟可以由MPLL或者EPLL提供。

系统时钟初始化起始于：

```
system_clock_init:
    ldr r0, =ELFIN_CLOCK_POWER_BASE  /* 0x7e00f000 */
```

S3C6400的时钟系统与S3C6410有所差异，其中将

```
/* FOUT of EPLL is 96MHz */
    ldr r1, =0x200203
```

修改成：

```
ldr r1, =0x80200203
```

（5）串口初始化

```
uart_asm_init:
    /* set GPIO to enable UART */
    ldr r0, =ELFIN_GPIO_BASE
ldr r1, =0x220022
str r1, [r0, #GPACON_OFFSET]
mov pc, lr
```

（6）NAND Flash 控制器初始化

```
nand_asm_init:
    ldr r0, =ELFIN_NAND_BASE
    ldr r1, [r0, #NFCONF_OFFSET]
    orr r1, r1, #0x70
    orr r1, r1, #0x7700
    str r1, [r0, #NFCONF_OFFSET]
    ldr r1, [r0, #NFCONT_OFFSET]
    orr r1, r1, #0x07
    str r1, [r0, #NFCONT_OFFSET]
    mov pc, lr
```

简单地对NAND Flash 主机控制器的时间参数初始化。

（7）内存初始化

调用 mem_ctrl_asm_init 函数，跳入到 arch/arm/cpu/arm1176/s3c64xx/ mem_ctrl_asm_init.s 中。系统上电，在利用内存控制器访问外部内存之前，需要进行一系列初始化工作，如图2.3所示。主要做两件事情：配置内存控制器和初始化外部内存设备。配置内存控制器包括时间参数、位宽、片选和ID配置等。初始化外部内存设备，通过操作P1DIRECTCMD寄存器，发出初始化系列：“nop”命令、Prechargeall命令、Autorefresh命令、Autorefresh命令、EMRS命令、MRS命令。

图2.3 内存初始化流程

S3C6410的DRAM控制器是基于ARM PrimeCell CP003 AXI DMC（PL340）的，S3C6410的存储器端口0并不支持DRAM，所以只能选用存储器端口1（DMC1）。S3C6410的DMC1基址ELFIN_DMC1_BASE的值为0x7e00_1000。当DMC1使用32位数据线DRAM时，需要配置MEM_SYS_CFG寄存器，将芯片管脚Xm1DATA[31:16]设置为DMC1的数据域。单板利用

两块64M×16bit的DDR SDRAM芯片K4X1G163PC组合成一块大小为64M×32bit的芯片，此时，MEM_SYS_CFG[7]必须清零。

DDR时间参数根据K4X1G163PC手册得到，并定义在s3c6410.h头文件中，利用宏 NS_TO_CLK（t）将时间参数转化成时钟周期，再写入相应的寄存器中。一块K4X1G163PC行地址为A0 - A13，列地址为A0 - A9，BANK地址为B0-B1。寻址范围为128MB。特别注意的是，片选寄存器 DMC1_CHIP0_CFG 的值：P1_chip_0_cfg[16] = 1，选择Bank-Row-Column 组织结构。地址匹配值为 0x50，地址屏蔽位0xF0，屏蔽了总线的高八位，因此寻址范围为0x5xxxx_xxxx（0x5000_0000～0x5ff_ffff 即256 MB）。

（8）唤醒复位初始化

```
/* Wakeup support. Don't know if it's going to be used,
untested. */
    ldr r0, =(ELFIN_CLOCK_POWER_BASE + RST_STAT_OFFSET)
    ldr r1, [r0]
    bic r1, r1, #0xfffffff7
    cmp r1, #0x8
    beq wakeup_reset
```

7. 调用_main函数

_main函数的实现代码位于arch/arm/lib/crt0.S文件中，用于建立C语言运行环境。crt0.S文件存放在arm处理器的lib库目录下，从文件的存放位置我们可以知道：_main函数和CPU的构架有关，而与单板的配置无关，即它支持所有的arm单板。编译生成u-boot.bin二进制文件时，用于条件编译的CONFIG_NAND_SPL和CONFIG_SPL_BUILD宏为假。_main函数是stage1和stage2 的过渡，它是一个汇编函数，但成分比较复杂：_main 函数多次调用 C 语言函数（例如board_init_f、board_init_r等）和汇编函数（如重定位函数relocate_code）其中，

board_init_f函数和board_init_r函数的实现代码均在arch/arm/lib/board.c文件中。

（1）声明外部变量

```
.globl board_init_r
.globl __bss_start
.globl __bss_end__
```

声明外部函数board_init_r，外部变量__bss_start和__bss_end__。

（2）为调用board_init_f函数建立运行环境

```
 .global _main
 _main:
   ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
   bic sp, sp, #7  /* 8-byte alignment for ABI
compliance */
   sub sp, #GD_SIZE /* allocate one GD above SP */
   bic sp, sp, #7  /* 8-byte alignment for ABI
compliance */
   mov r8, sp    /* GD is above SP */
   mov r0, #0
```

如图2.4所示，建立运行环境包括初始化堆栈指针sp和预留一个内存空间存储gd_t类型的数据结GD，gd指向这个结构体的首地址。gd_t是关键字typedef为global data 数据结构定义的新名字，定义的原型位于文件 include/asm-generic/global_data.h 中，其成员主要是系统初始化的参数，如程序清单2.1所示。

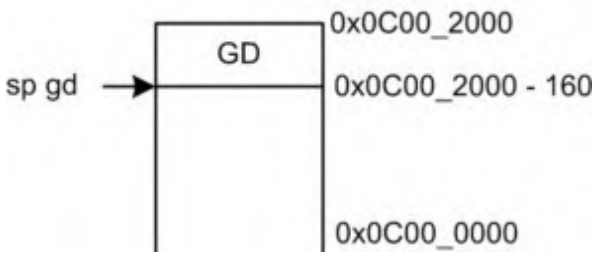


图2.4 建立运行环境

程序清单2.1 global_data 结构

```
  typedef struct global_data {
    bd_t *bd;
    unsigned long flags;
    unsigned long baudrate;
    unsigned long cpu_clk;/* CPU clock in Hz!    */
    unsigned long bus_clk;
    /* We cannot bracket this with CONFIG_PCI due to
mpc5xxx */
    unsigned long pci_clk;
    unsigned long mem_clk;
  #if defined(CONFIG_LCD) || defined(CONFIG_VIDEO)
    unsigned long fb_base;/* Base address of framebuffer
mem */
  #endif
  #if defined(CONFIG_POST) || defined(CONFIG_LOGBUFFER)
    unsigned long post_log_word;  /* Record POST
activities */
    unsigned long post_log_res; /* success of POST test
*/
    unsigned long post_init_f_time;  /* When post_init_f
started */
```

```
  #endif
  #ifdef CONFIG_BOARD_TYPES
    unsigned long board_type;
  #endif
    unsigned long have_console;  /* serial_init() was
called */
  #ifdef CONFIG_PRE_CONSOLE_BUFFER
    unsigned long precon_buf_idx;  /* Pre-Console buffer
index */
  #endif
  #ifdef CONFIG_MODEM_SUPPORT
    unsigned long do_mdm_init;
    unsigned long be_quiet;
  #endif
    unsigned long env_addr;  /* Address  of Environment
struct */
    unsigned long env_valid;  /* Checksum of Environment
valid? */
    /* TODO: is this the same as relocaddr, or something
else? */
    unsigned long dest_addr;  /* Post-relocation address
of U-Boot */
    unsigned long dest_addr_sp;
    unsigned long ram_top;/* Top address of RAM used by
U-Boot */
    unsigned long relocaddr;  /* Start address of U-Boot
in RAM */
```

```
    phys_size_t ram_size; /* RAM size */
    unsigned long mon_len;/* monitor len */
    unsigned long irq_sp;  /* irq stack pointer */
    unsigned long start_addr_sp;  /*
start_addr_stackpointer */
    unsigned long reloc_off;
    struct global_data *new_gd;  /* relocated global data
*/
    const void *fdt_blob; /* Our device tree, NULL if
none */
    void **jt;    /* jump table */
    char env_buf[32]; /* buffer for getenv() before
reloc. */
    struct arch_global_data arch;  /* architecture-
specific data */
} gd_t;
```

在一个源码文件中，访问 gd 结构体前需用宏定义 DECLARE_GLOBAL_DATA_PTR 进行声明，这个宏定义在文件arch/arm/include/asm/global_data.h中。

```
#define DECLARE_GLOBAL_DATA_PTR  register volatile gd_t
*gd asm ("r8")
```

register是C语言中的一个关键字，除了一些特殊的场合，如要求变量高速地被调用，它一般很少被使用。如果一个变量被register修饰，就意味着该变量是一个寄存器变量，变量的值存放在寄存器中。当然，这里的寄存器指的是CPU的内核寄存器，它独立于内存没有地址，所以无法对寄存器变量进行取地址运算。

DECLARE_GLOBAL_DATA_PTR定义了一个gd_t结构体指针变量gd，asm（"r8"）指定了 gd 值的存放位置 r8。volatile 是为了防止变量被编译器优化，要求每次都要去重新读取变量的值。事实上，U-Boot中的这段代码存在一定的缺陷。

在文件include/configs/sdmk6410.h中，CONFIG_SYS_INIT_SP_ADDR的计算过程如下：

```
#define CONFIG_SYS_IRAM_BASE  0x0c000000  /* Internal SRAM base address */
#define CONFIG_SYS_IRAM_SIZE  0x2000  /* 8 KB of internal SRAM memory */
#define CONFIG_SYS_IRAM_END  (CONFIG_SYS_IRAM_BASE + CONFIG_SYS_IRAM_SIZE)
#define CONFIG_SYS_INIT_SP_ADDR (CONFIG_SYS_IRAM_END - GENERATED_GBL_DATA_SIZE)
```

其中，GENERATED_GBL_DATA_SIZE在编译时，会自动生成

build/include/generated/generic-asm-offsets.h

```
#define GENERATED_GBL_DATA_SIZE (160) /* (sizeof(struct global_data) + 15) & ~15 */
```

由注释可知，宏定义CONFIG_SYS_INIT_SP_ADDR已经为gd在SRAM的顶部预留了160字节的空间，因此没必要再将sp指针下调。当然，这样做也并不会影响正常的启动流程，但是偏离了设计者的本意，我们只需要在crt0.S文件中，将下面部分代码段注释掉即可。

```
bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
sub sp, #GD_SIZE /* allocate one GD above SP */
```

（3）调用board_init_f函数

```
bl board_init_f
```

实际上，board_init_f()函数是 U-Boot 执行的第一个 C 语言函数：void board_init_f（ulongbootflag），这个函数位于arch/arm/lib目录下的board.c文件中。

void board_init_f()函数的主要工作是：清空 gd 指向的结构体、逐步填充结构体，执行init_fnc_ptr 函数指针数组中的各个初始化函数和划分内存区域等。结构体成员的初始化贯穿于board_init_f函数的整个过程，多数情况下成员的值是根据顶层配置文件的宏确定的。

```
/* Pointer is writable since we allocated a register for it */
gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);
```

gd_t是一个结构体类型，其定义在arch/arm/include/asm目录下的global_data.h文件中，前面已经详细分析过。

```
memset((void *)gd, 0, sizeof(gd_t));
```

将gd所指向的结构体内的所有变量清零，长度为：sizeof（gd_t）。清空之后，在board_init_f()函数后面有很多代码是对gd所指向的结构体的成员进行重新赋值。

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

在U-Boot中定义了一个init_sequence函数指针数组：

```
  init_fnc_t *init_sequence[] = {
    arch_cpu_init,    /* basic arch cpu dependent setup */
```

```
    mark_bootstage,
#ifdef CONFIG_OF_CONTROL
    fdtdec_check_fdt,
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
    board_early_init_f,
#endif
    timer_init,    /* initialize timer */
#ifdef CONFIG_BOARD_POSTCLK_INIT
    board_postclk_init,
#endif
#ifdef CONFIG_FSL_ESDHC
    get_clocks,
#endif
    env_init,    /* initialize environment */
    init_baudrate,    /* initialze baudrate settings */
    serial_init,    /* serial communications setup */
    console_init_f,    /* stage 1 init of console */
    display_banner,    /* say that we are here */
#if defined(CONFIG_DISPLAY_CPUINFO)
    print_cpuinfo,    /* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,    /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) ||
defined(CONFIG_SOFT_I2C)
```

```
    init_func_i2c,
  #endif
    dram_init,    /* configure available RAM banks */
    NULL,
  };
```

函数的类型为init_fnc_t，init_fnc_t也是一个新定义的数据类型，这个数据类型是传入参数为空，返回值为有符号整形的函数，函数用于初始化工作。如下：

```
typedef int (init_fnc_t) (void);
```

board_init_f函数使用一个for循环语句来逐一执行数组中的初始化函数，如果初始化函数返回值不为0，程序调用hang函数挂起，不再继续往下运行。

```
addr = CONFIG_SYS_SDRAM_BASE + gd->ram_size;
```

这行代码告诉我们SDRAM的末位物理地址为0x5800 0000，即SDRAM的空间分布为0x5000 0000～0x57FF FFFF。说明SDRAM 一共有128MB的空间。

接下来的代码程序就是对这128MB内存进行划分。

```
  #ifdef CONFIG_PRAM
    /*
      * reserve protected RAM
      */
    reg = getenv_ulong("pram", 10, CONFIG_PRAM);
    addr -= (reg << 10);/* size is in kB */
    debug("Reserving %ldk for protected RAM at %08lx\n",
reg, addr);
  #endif /* CONFIG_PRAM */
```

```
    #if !(defined(CONFIG_SYS_ICACHE_OFF) &&
  defined(CONFIG_SYS_DCACHE_OFF))
      /* reserve TLB table */
      addr -= (4096 * 4);
      /* round down to next 64 kB limit */
      addr &= ~(0x10000 - 1);
      gd->tlb_addr = addr;
      debug("TLB table at: %08lx\n", addr);
    #endif
      /* round down to next 4 kB limit */
      addr &= ~(4096 - 1);
      debug("Top of RAM usable for U-Boot at: %08lx\n",
    addr);
```

这里告诉我们是将SDRAM的最后64KB（addr &= ～（0x10000 - 1））分配给TLB，所分配的地址为：0x57FF 0000～0x57FF FFFF。

```
    #ifdef CONFIG_LCD
    #ifdef CONFIG_FB_ADDR
      gd->fb_base = CONFIG_FB_ADDR;
    #else
      /* reserve memory for LCD display (always full pages)
    */
      addr = lcd_setmem(addr);
      gd->fb_base = addr;
    #endif /* CONFIG_FB_ADDR */
    #endif /* CONFIG_LCD */
      /*
        * reserve memory for U-Boot code, data & bss
```

```
        * round down to next 4 kB limit
        */
       addr -= gd->mon_len;
       addr &= ~(4096 - 1);
       debug("Reserving %ldk for U-Boot at: %08lx\n", gd-
    >mon_len >> 10, addr);
```

这段代码是在SDRAM中从后往前给u-boot分配BSS、数据段、代码段，分配地址为：0x57F7 5000～0x57FE FFFF。

```
      /*
        * reserve memory for malloc() arena
        */
       addr_sp = addr - TOTAL_MALLOC_LEN;
       debug("Reserving %dk for malloc() at: %08lx\n",
        TOTAL_MALLOC_LEN >> 10, addr_sp);
```

从后往前紧挨着代码段开辟了一块malloc 空间，给予的地址为：0x57E6 D000～0x57E7 4FFF。

```
      /*
        * (permanently) allocate a Board Info struct
        * and a permanent copy of the "global" data
        */
       addr_sp -= sizeof (bd_t);
       bd = (bd_t *) addr_sp;
       gd->bd = bd;
       debug("Reserving %zu Bytes for Board Info at: %08lx\n",
        sizeof (bd_t), addr_sp);
```

为bd 结构体分配空间，地址为：0x57E6 CFD8～0x57E6 CFFF。

```
       addr_sp -= sizeof (gd_t);
```

```
    id = (gd_t *) addr_sp;
    debug("Reserving %zu Bytes for Global Data at:
  %08lx\n",
      sizeof (gd_t), addr_sp);
```

这是给gd 结构体分配空间，地址为：0x57E6 CF60～0x57E6 CFD7。

```
      /* setup stackpointer for exeptions */
      gd->irq_sp = addr_sp;
    #ifdef CONFIG_USE_IRQ
      addr_sp -=
  (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ);
      debug("Reserving %zu Bytes for IRQ stack at:
  %08lx\n",
        CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ,
    addr_sp);
    #endif
      /* leave 3 words for abort-stack*/
      addr_sp -= 12;
      /* 8-byte alignment for ABI compliance */
      addr_sp &= ~0x07;
    #else
      addr_sp += 128; /* leave 32 words for abort-stack */
      gd->irq_sp = addr_sp;
    #endif
      debug("New Stack Pointer is: %08lx\n", addr_sp);
```

分配异常中断空间，地址：0x57E6 CF50～0x57E6 CF5F。

综合上面SDRAM的分配，那么内存分配即为如图1.25所示。SDRAM的空间大小为256MB。

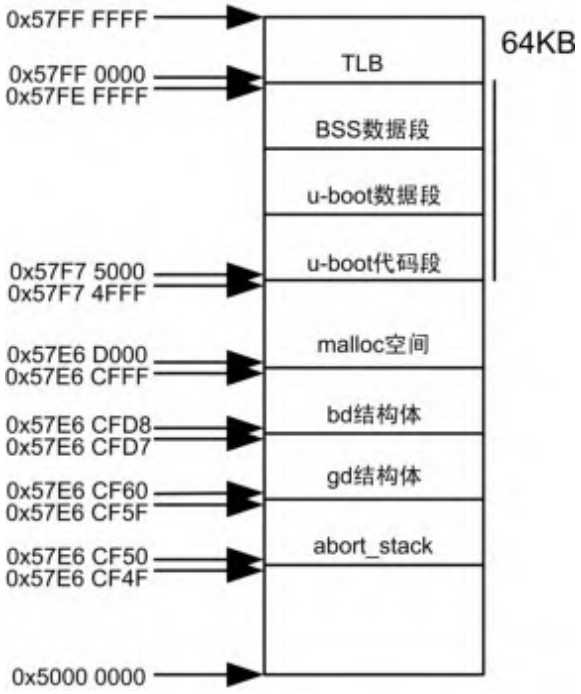


图2.5 SDRAM内存划分图

其中在smdk6410.h中，有这么一个宏定义：

#define CONFIG_SYS_SDRAM_BASE 0x50000000

这说明SDRAM 的起始地址是：0x5000 0000。

完成gd结构体的初始化和内存的划分之后，执行board_init_f()函数的最后一行代码：

relocate_code(addr_sp, id, addr);

这行代码的意思很明显是要跳回到start.S中，跳回start.S中紧接着的是下面这一段代码。

```
    .globl relocate_code
  relocate_code:
    mov r4, r0 /* save addr_sp */
    mov r5, r1 /* save addr of gd */
    mov r6, r2 /* save addr of destination */
```

将relocate_code()带回来的3个参数分别装入r4、r5、r6寄存器中。

但是注意到，relocate_code这个函数的声明是在commom.h中，如下所示：

```
void relocate_code (ulong, gd_t *, ulong) __attribute__ ((noreturn));
```

relocate_code函数的3个参数分别为栈顶地址、数据ID（即全局结构gd）在SDRAM中的起始地址和在SDRAM中存储U-Boot的起始地址。

在start.S中接着进行的是设置堆栈指针，如下：

```
  /* Set up the stack                    */
  stack_setup:
    mov sp, r4
    adr r0, _start
    cmp r0, r6
    moveq  r9, #0    /* no relocation. relocation
offset(r9) = 0 */
    beq clear_bss    /* skip relocation */
    mov r1, r6      /* r1 <- scratch for copy_loop */
    ldr r3, _bss_start_ofs
    add r2, r0, r3    /* r2 <- source end address    */
  copy_loop:
    ldmia  r0!, {r9-r10}  /* copy from source address
[r0]  */
    stmia  r1!, {r9-r10}  /* copy to  target address
[r1]  */
    cmp r0, r2      /* until source end address [r2]  */
    blo copy_loop
```

r4是刚刚传回来的堆栈指针，那么将r4给sp，设定堆栈指针。

r6是在SDRAM中存储u-boot的起始地址，将r0和r6进行比较。如果此时的u-boot已经在SDRAM 中了，则执行beq clear_bss指令，跳转到clear_bss 处；如果不是，在NandFlash中，则要将u-boot复制到SDRAM中。

```
clear_bss:
#ifndef CONFIG_SPL_BUILD
    ldr r0, _bss_start_ofs
    ldr r1, _bss_end_ofs
    mov r4, r6      /* reloc addr */
    add r0, r0, r4
    add r1, r1, r4
    mov r2, #0x00000000    /* clear        */
  clbss_l:cmp  r0, r1      /* clear loop... */
    bhs clbss_e      /* if reached end of bss, exit */
    str r2, [r0]
    add r0, r0, #4
    b  clbss_l
  clbss_e:
  #ifndef CONFIG_NAND_SPL
    bl coloured_LED_init
    bl red_led_on
  #endif
  #endif
```

上面这段代码是对BSS进行清零操作。

```
  /*
```

```
    * We are done. Do not return, instead branch to
second part of board
    * initialization, now running from RAM.
    */
  #ifdef CONFIG_NAND_SPL
    ldr  pc, _nand_boot
  _nand_boot: .word nand_boot
  #else
    ldr r0, _board_init_r_ofs
    adr r1, _start
    add lr, r0, r1
    add  lr, lr, r9
    /* setup parameters for board_init_r */
    mov r0, r5    /* gd_t */
    mov r1, r6    /* dest_addr */
    /* jump to it ... */
    mov pc, lr
  _board_init_r_ofs:
    .word board_init_r - _start
  #endif
```

上面这段代码如果是 NAND 启动的话，那么就设置 SP 后跳到nand_boot()函数里面进行复制代码到SDRAM，然后跳到U-Boot在SDRAM的起始地址开始运行。但是由于CONFIG_NAND_SPL没有宏定义，所以执行else。在进入board_init_r之前，给了两个参数：r5、r6。

r5：数据ID（即全局结构gd）在SDRAM中的起始地址。

r6：在SDRAM中存储U-Boot的起始地址。

board_init_r()函数同样是位于arch/arm/lib目录下的board.c文件中，并且是紧跟在board_init_f()函数后面。

board_init_r()函数的主要操作是：给标志位赋值、清空malloc空间、初始化NAND Flash、初始化外设（I2C、LCD、VIDEO、KEYBOARD、USB、JTAG等）、跳转表初始化、中断初始化和中断使能等。这里希望读者能触类旁通，完成这个函数的分析。

完成之前的操作，board_init_r()函数进入一个for循环，如下所示：

```
/* main_loop() can return to retry autoboot, if so just
run it again. */
for (;;) {
    main_loop();
}
```

main_loop()函数位于/commom目录下的main.c文件中。如下所示：

```
void main_loop (void)
```

main_loop()函数既无入口参数也无返回值。Main_loop()函数的主要实现作用是：

（1）HUSH的相关初始化

```
#ifndef CONFIG_SYS_HUSH_PARSER
    static char lastcommand[CONFIG_SYS_CBSIZE] = { 0, };
    int len;
    int rc = 1;
    int flag;
#endif
……
#ifdef CONFIG_SYS_HUSH_PARSER
```

```
    u_boot_hush_start ();
#endif
#if defined(CONFIG_HUSH_INIT_VAR)
    hush_init_var ();
#endif
```

（2）bootdelay的初始化

```
#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    char *s;
    int bootdelay;
#endif
```

（3）启动次数

```
#ifdef CONFIG_BOOTCOUNT_LIMIT
    bootcount = bootcount_load();
```

上面这行代码的作用是加载保存的启动次数。

```
bootcount++;
```

启动次数加1。

```
bootcount_store(bootcount);
```

更新启动次数。

```
sprintf (bcs_set, "%lu", bootcount);
```

将启动次数通过串口输出。

```
    setenv ("bootcount", bcs_set);
    bcs = getenv ("bootlimit");
    bootlimit = bcs ? simple_strtoul (bcs, NULL, 10) : 0;
#endif /* CONFIG_BOOTCOUNT_LIMIT */
```

这段代码蕴含的东西较多。启动次数限制功能，启动次数限制可以被用户设置一个启动次数，然后保存在Flash存储器的特定位置，当

到达启动次数后，U-Boot无法启动。该功能适合一些商业产品，通过配置不同的License限制用户重新启动系统。

（4）Modem功能

```
#ifdef CONFIG_MODEM_SUPPORT
  debug ("DEBUG: main_loop: do_mdm_init=%d\n",
do_mdm_init);
  if (do_mdm_init) {
    char *str = strdup(getenv("mdm_cmd"));
    setenv ("preboot", str); /* set or delete
definition */
    if (str != NULL)
      free (str);
    mdm_init(); /* wait for modem connection */
  }
#endif /* CONFIG_MODEM_SUPPORT */
```

如果系统中有Modem功能，打开其功能可以接受其他用户通过电话网络的拨号请求。Modem功能通常供一些远程控制的系统使用。

（5）设置U-Boot版本号

```
#ifdef CONFIG_VERSION_VARIABLE
  {
    setenv ("ver", version_string); /* set version
variable */
  }
#endif /* CONFIG_VERSION_VARIABLE */
```

打开动态版本支持功能后，u-boot在启动的时候会显示最新的版本号。

（6）启动tftp功能

```
#if defined(CONFIG_UPDATE_TFTP)
    update_tftp (0UL);
#endif /* CONFIG_UPDATE_TFTP */
```

（7）打印启动菜单

```
#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) :
CONFIG_BOOTDELAY;
    debug ("### main_loop entered: bootdelay=%d\n\n",
bootdelay);
```

在进入主循环之前，如果配置了启动延迟功能，需要等待用户从串口或者网络接口输入。如果用户按下任意键打断，启动流程，会向终端打印出一个启动菜单。

```
#if defined(CONFIG_MENU_SHOW)
    bootdelay = menu_show(bootdelay);
#endif
```

向终端打印出一个启动菜单。

```
# ifdef CONFIG_BOOT_RETRY_TIME
    init_cmd_timeout ();
# endif /* CONFIG_BOOT_RETRY_TIME */
```

初始化命令行超时机制。

```
  #ifdef CONFIG_POST
    if (gd->flags & GD_FLG_POSTFAIL) {
      s = getenv("failbootcmd");
    }
    else
  #endif /* CONFIG_POST */
```

```
#ifdef CONFIG_BOOTCOUNT_LIMIT
  if (bootlimit && (bootcount > bootlimit)) {
    printf ("Warning: Bootlimit (%u) exceeded. Using
altbootcmd.\n",
        (unsigned)bootlimit);
```

检测是否超出启动次数限制。

```
    s = getenv ("altbootcmd");
  }
  else
#endif /* CONFIG_BOOTCOUNT_LIMIT */
    s = getenv ("bootcmd");
```

获取启动命令参数。

main_loop()的主要作用即是U-Boot启动管理。

到此为止，我相信读者应该对U-Boot的启动原理有了大致的了解，在分析启动原理，笔者希望读者要有“刨根问底”的精神，不弄明白誓不罢休。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第二章 04 课、05课（U-Boot-2013.04启动分析）。

2.3 SD/MMC设备移植

S3C6410 是由三星公司生产的 ARM11 应用处理器芯片，广泛用于移动电话和通用应用。市场上，很多公司纷纷推出自己的S3C6410学习开发板，风靡一时。处理器片内没有供用户存储数据的 Flash，用户必须外接存储器存储数据。由表 2.1 可知开发板唯一带有的存储介质

是NAND Flash，如果不经过特殊方式，无法直接将 U-Boot 镜像文件烧写到里面。由于公司间的竞争关系，防止竞争对手的抄袭，很多开发板相关的代码并不开源。以本节涉及的内容为例，被开发板生产商誉为核心技术，商业机密。这对于以研究学习为目的的购买者来说，无疑是巨大的阻碍。本节内容充分结合S3C6410支持SD卡启动的特性，全面阐述利用SD卡烧写、运行嵌入式系统的原理。

2.3.1 IROM启动的概念

在生活中，如果不合理操作计算机，计算机经常会出现无法从硬盘中启动的情况。这时候可以通过设置BIOS选择从其他盘启动，比如启动CD、U盘等。在使用它们启动系统之前，必须将其制作成启动盘，把一个精简的操作系统写入其中。电脑启动时就会识别启动盘，加载存储设备特定扇区的数据至内存，从而启动系统，进行一些修复工作。

同样，如图2.6所示，S3C6410有多种启动模式，分别由XSELNAND，OM[4：0]管脚控制。把OM[4：1]管脚外部电平设置为1111时，选择IROM启动。GPN[15:13]管脚的电平状态用来选择IROM启动时的外部存储设备，如SD/MMC（CH0和CH1）、OneNAND和NAND（数据大小不同的页）。

三星公司在生产 S3C6410 芯片时，在地址为 0x8000_0000 的 IROM 区域固化了一段大小为32KB的代码，称作BL0。处理器上电后，PC指向运行0x8000_0000，运行BL0，这种启动方式称作IROM启动。启动的大体流程如下：

XSELNAND	OM[4:0]	GPN[15:13]	Boot Device	Function	Clock Source
1	0000X	XXX	RESERVED	RESERVED	XXTIpll if OM[0] is 0. XEXTCLK if OM[0] is 1.
1	0001X			RESERVED	
1	0010X			RESERVED	
1	0011X			RESERVED	
X	0100X		SROM(8bit)	-	
X	0101X		SROM(16bit)	-	
0	0110X		OneNAND[1]	Don't use NAND Device	
X	0111X		MODEM	Don't use Xm0CSn2 for SROMC	
X	1111X	000	IROM[2]	SD/MMC(CH0)	
0		001		OneNAND	
1		010		NAND(512Byte, 3-Cycle)	
1		011		NAND(512Byte, 4-Cycle)	
1		100		NAND(2048Byte, 4-Cycle)	
1		101		NAND(2048Byte, 5-Cycle)	
1		110		NAND(4096Byte, 5-Cycle)	
X		111		SD/MMC(CH1)	

图2.6 S3C6410启动设备描述表

（1）运行BL0进行一些初始化工作，如关闭看门狗，初始化TCM、系统时钟、堆栈等。

（2）根据GPN[15:13]管脚的电平状态，判断选定的存储设备的类型，初始化存储设备和它对应的控制器。从存储设备（SD/MMC/OneNand/Nand）的特定区域读取8KB的程序到SteppingStone中运行，被拷贝的这段代码称Bootloader1（BL1）。

（3）BL1是用户自行编写的代码，必须简短精悍，运行与位置无关。BL1一般简单地重新初始化系统，开辟更广阔的内存空间，并将更加完善的Bootloader2（BL2）拷贝到SDRAM中。

（4）跳转到SDRAM中的BL2，继续运行，BL2功能更加强大，把存储设备中的内核和文件系统加载到SDRAM中，从而启动系统。

S3C6410在0x0C00_0000至0x0C005FFF的地址空间内定义了三类内存区域，IRAM、D-TCM0和D-TCM1。IRAM用于加载运行BL1。当选定SD/MMC作为IROM启动的存储设备时，D-TCM0保存了SD/MMC设备被IROM

代码检测到的一些信息，如当前使用的SD/MMC控制器的基地址、SD/MMC卡的类别、设备的扇区总数等。它们被定义为三个全局变量存放，其中扇区总数的存放地址为0x0C00_3FFC，如表2.2所示。

表2.2 IROM启动内存映射地址

类　型	地　址	用　途	大　小
IRAM	0x0C00_0000-0x0C00_1FFF	Stepping Stone （BL1）	8KB
D-TCM0	0x0C00_2000-0x0C00_21FF 0x0C00_2200-0x0C00_2FFF 0x0C00_3000-0x0C00_3FFF	密钥（512B） 保留（3.5KB） 堆区，保存全局变量（4KB）	8KB
D-TCM1	0x0C00_4000-0x0C00_4018 0x0C00_4019-0x0C00_5FFF	存储设备拷贝函数指针（24B） 栈区	8KB

BL1的主要工作是从存储设备中，拷贝更完善的BL2至DRAM，并且BL1大小不能超过8K。如果需要用户自行编写函数实现拷贝功能，开发难度很大。事实上，S3C6410已经在IROM中固化了6个用于从不同外部存储设备拷贝数据到SDRAM中的函数，如表2.3所示，这些函数的指针存放在D-TCM1的前24字节（每个指针变量占4字节）。用户根据需要调用即可，有效地降低了开发难度。

表2.3 设备拷贝函数

函数指针地址	函数参数及返回值	描　　述
0x0C00_4000	int NF8_ReadPage（uint32 blcok, uint32 page, uint8 *buffer） blcok：块起始地址 page：需要拷贝的页数 buffer：目标地址 返回值：0 失败 1 成功	支持 512KB 每页 8 位硬件 ECC 校验
0x0C00_4004	int NF8_ReadPage_Adv（uint32 blcok, uint32 page, uint8 *buffer） blcok：块起始地址 page：需要拷贝的页数 buffer：目标地址 返回值：0 失败 1 成功	支持 2KB 每页 支持 4KB 每页 8 位硬件 ECC 校验
0x0C00_4008	bool CopyMMCtoMem（int channel, uint32StartBlkAddress， uint16 blockSize, uint32*memoryPtr, bool with_init） channel：无效，取决于 GPN15, GPN14 and GPN13 管脚 StartBlkAddress：扇区起始地址 blockSize：需要拷贝的扇区数 memoryPtr：目标地址 with_init: ：是否需要重新初始化 返回值：0 失败 1 成功	支持 SD/MMC 卡 支持 SDHC 卡
0x0C00_400C	boolONENAND_ReadPage（uint32 Controller,uint32 uBlkAddr，uint8 uPageAddr, uint32* aData） Controller: OneNAND 控制器编号，固定为 0 uBlkAddr：块地址 uPageAddr：页地址 aData：目标地址 返回值：0 失败 1 成功	-
0x0C00_4010	bool ONENAND_ReadPage_4burst（uint32 Controller,uint32 uBlkAddr，uint8 uPageAddr, uint32* aData） Controller: OneNAND 控制器编号，固定为 0 uBlkAddr：块地址 uPageAddr：页地址 aData：目标地址 返回值：0 失败 1 成功	-

续表

函数指针地址	函数参数及返回值	描　述
0x0C00_4014	bool ONENAND_ReadPage_8burst（uint32 Controller,uint32 uBlkAddr , uint8 uPageAddr, uint32* aData） Controller: OneNAND 控制器编号，固定为 0 uBlkAddr：块地址 uPageAddr：页地址 aData：目标地址 返回值：0 失败 1 成功	-

以CopyMMCtoMem函数为例，可以通过以下形式调用该函数。

```
#define CopyMMCtoMem(a,b,c,d,e)  (((int(*)(int, uint, ushort, uint *, int))(*((uint*)(0x0C004000 + 0x8))))(a,b,c,d,e))
```

为了更方便地阐释IROM-SD/MMC的启动原理，本书约定从IROM、以SD/MMC为存储设备的启动方式为SD卡启动。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第二章 06 课（SD卡启动U-Boot原理）。

2.3.2 实现SD卡启动

S3C6410控制器兼容MMC4.2协议和SD2.0协议，支持目前市场上流行的SD/MMC卡。通过拨码开关，将OM[4:1]设置为1111，GPN[15:13]设置为000，系统从SD/MMC（CH0）启动。在上节内容提到，运行BL0时，程序从存储设备的特定区域读取由用户编写的BL1至SteppingStone中。

SD卡启动时，BL1数据的存放位置必须是如图2.7所示的区域，ENV和BL2存放位置和大小没有明确限定。通过一定的手段，跳过文件系统的管理，它们被直接写入相应的扇区，因此写入数据后，设备容量大小看起来没有任何改变，如果数据写在靠前的扇区很容易破坏用户文件系统。为了最小限度地不破坏用户文件系统，开发时请按照图示方法规划数据。

图2.7 启动设备分区

在移植U-Boot时，并不需要自己编写BL1，将u-boot.bin前8KB的代码作为BL1，我们只需修改少量源码使BL1具备数据拷贝功能、支持SD卡启动。然后把整个u-boot.bin文件作为BL2，写至BL2区域。这样，实现U-Boot的SD卡启动变得十分简单。BL2和ENV存放区域的大小通常需要预留部分空间。

（1）在arch/arm/cpu/arm1176/s3c64xx/目录下，新建s3c6410_sdboot.c源码文件，内容如程序清单2.2所示。CopyMMCtoMem函数的原型见表2.3。

程序清单2.2 sd 卡启动的实现函数

```
/*
* Samsung S3C6410 SD/MMC Device Boot
* Author: LiQiang E-mail: jxustlq@163.com
* Date: 2013/03/25
* Licensed under the GPL-2 or later.
*/
#include <common.h>
#include <asm/io.h>
#ifdef CONFIG_MMC_CHANNEL
#define MMC_CHANNEL CONFIG_MMC_CHANNEL
#else
```

```
    #define MMC_CHANNEL 0
    #endif
    #define ELFIN_HSMMC_BASE (0x7C200000 +
MMC_CHANNEL*0x10000)
    #define HM_CONTROL4 (ELFIN_HSMMC_BASE+0x8C)
    #define globalBlockSizeHide *((volatile unsigned int*)
(0x0C004000-0x4))
    #define globalSDHCInfo      *((volatile unsigned int*)
(0x0C004000-0x8))
    #define LAST_BLKPOS (globalBlockSizeHide - 2)
    #define BLK_SIZE 512
    #define BL1_SIZE (1024*8)
    #define BL2_SIZE (300*1024)
    #if 0
    #define ENV_SIZE CONFIG_ENV_SIZE
    #else
    #define ENV_SIZE (16*1024)
    #endif
    #define BL1_BLKCNT(BL1_SIZE/BLK_SIZE)
    #define BL2_BLKCNT (BL2_SIZE/BLK_SIZE)
    #define ENV_BLKCNT (ENV_SIZE/BLK_SIZE)
    #define STARTBLKADDR (LAST_BLKPOS - BL1_BLKCNT -
BL2_BLKCNT - ENV_BLKCNT)
    #define DESTADDR CONFIG_SYS_PHY_UBOOT_BASE
    #define CopyMMCtoMem(a,b,c,d,e)(((int(*)(int, uint,
ushort, uint *, int)) \
    (*((uint *)(0x0C004000 + 0x8))))(a,b,c,d,e))
```

```
int BootCopyMMCtoMem()
{
  writel(readl(HM_CONTROL4) | (0x3 << 16),
HM_CONTROL4);
  return CopyMMCtoMem(0, STARTBLKADDR, BL2_BLKCNT,
(uint *)DESTADDR, 0);
}
```

STARTBLKADDR是拷贝扇区的起始地址，它的计算方法在代码中已经很容易看出，这里不再赘述。其中，SD卡总扇区数由BL0检测得出并存放在D-TCM0区域。BL2_BLKCNT是需要拷贝的扇区数目（BL2_SIZE/BLK_SIZE计算得到），由于我们直接以u-boot.bin作为BL2区域，因此BL2_SIZE不能小于u-boot.bin的实际大小。u-boot.bin的实际大小为250KB左右，考虑到后面可能会增加一些驱动，设置BL2_SIZE 的值为 300KB。DESTADDR 是拷贝到内存中的目标地址，直接将它设为U-Boot的基地址，也就是编译时的运行地址。

（2）在顶层配置文件smdk6410.h增加#define CONFIG_BOOT_SD宏用于控制SD启动控制。

```
259 /* Boot configuration (define only one of next 3) */
260 #define CONFIG_BOOT_SD /*added by liqiang */
261 #define CONFIG_BOOT_NAND
```

（3）修改Makefile（arch/arm/cpu/arm1176/s3c64xx），增加对源码的编译。

```
33 COBJS-$(CONFIG_S3C6400) += cpu_init.o speed.o
34 COBJS-$(CONFIG_S3C6410) += cpu_init.o speed.o
35 COBJS-$(CONFIG_BOOT_SD) += s3c6410_sdboot.o /*added by
liqiang */
36 COBJS-y += timer.o init.o
```

（4）在start.S添加对BootCopyMMCtoMem函数的调用（BL1）。

```
225    bl  lowlevel_init    /* go setup pll,mux,memory */
226
227 #if defined(CONFIG_BOOT_SD) &&
!defined(CONFIG_NAND_SPL)
228    ldr sp, =CONFIG_SYS_INIT_SP_ADDR  /*liqiang
2013/3/25 add*/
229    bl BootCopyMMCtoMem
230    cmp r0, #0
231 copyerror:
232    beq copyerror
233    ldr pc, =_main
234 #else
235    bl    _main
236 #endif
```

在编译生成 u-boot.bin 阶段预处理命令结果为真，首先简单地初始化堆栈指针，供函数BootCopyMMCtoMem使用。调用BootCopyMMCtoMem函数，根据ATPCS规则，函数的返回值存放在寄存器R0中，检验代码拷贝是否成功，如果不成功进入死循环；如果成功，跳转到_main函数的运行地址处继续执行。

（5）由于在 start.S 文件中调用了 lowlevel_init 函数，为了确保该实现函数的代码段处于在u-boot.bin的前8KB区域，必须修改链接脚本board/samsung/smdk6410/u-boot-nand.lds。

```
35    .text  :
36    {
37    arch/arm/cpu/arm1176/start.o  (.text)
```

```
    38      board/samsung/smdk6410/libsmdk6410.o
(.text)  /*added by liqiang */
    39        *(.text)
    40     }
```

1. 烧写SD卡方法

经过前面内容的讲述，我们已经清楚了 SD 卡启动的原理。但是到目前为止，还没有将镜像文件载入单板的内存中运行，SD 卡启动需要一个能够将数据烧写至指定扇区的工具。为了方便读者，笔者在Windows平台编写了一个SD卡烧写工具SD_Writer.exe。在使用前，必须确保SD卡为 FAT32 文件系统格式。为了防止不可预知的错误损坏数据，请将重要的数据事先备份。SD_Writer.exe实现了将u-boot.bin按照图2.7的分布示意图写入扇区内。SD_Writer界面如图2.8所示，主要分为四个栏目。

图2.8 SD_Writer软件界面

（1）磁盘信息：如果PC上插有SD卡，首次打开软件，软件会自动检测磁盘信息；如果中途插拔SD卡，需要单击“检测”按钮重新检测

信息。这些信息包括可移动磁盘的盘符、SD卡类型（SD/SDHC）和磁盘扇区的大小。软件把大于2GB的SD卡处理为SDHC卡，如果与实际不符，请自行在下拉框中选择。

（2）文件信息：单击“打开”按钮在 PC 文件系统中得到所要烧写镜像文件的路径，并且在文本框中显示出文件的大小（精确至 KB）。

（3）镜像文件设置：该栏目是为用户预留的接口，BL2_BLKCNT 和 ENV_BLKCNT 的值必须与程序清单2.2中配置的值一致，否则会导致SD卡启动失败。默认值分别为600和32。

（4）单击“开始”按钮，开始烧写，成功后提示“烧写成功”。

综上所示，处理器上电，运行BL0 自动将BL1 拷贝至Stepping Stone，PC 指向0xC00_0000运行u-boot.bin的前段代码：首先执行start.S文件中的程序，当执行到BootCopyMMCtoMem处时，PC跳转到s3c6410.c文件执行BootCopyMMCtoMem函数，BootCopyMMCtoMem函数调用CopyMMCtoMem函数将大小为BL2_SIZE的BL2（u-boot.bin）拷贝至U-Boot的运行地址处，然后PC指向_main函数的运行地址，如图2.9所示。

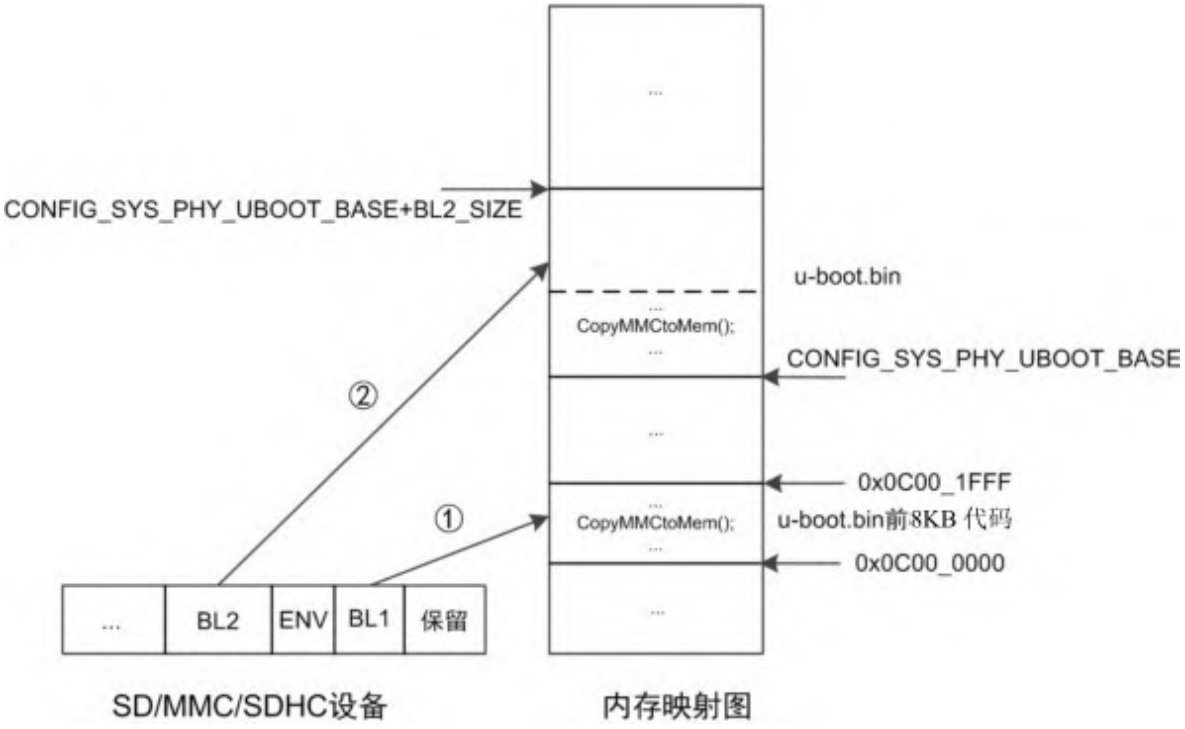


图2.9 SD卡启动流程图

2. 验证效果

设置单板的拨码开关，选择 SD 卡启动，将 SD 卡插入卡槽内，按单板上的复位按键。串口打印信息如下：

U-Boot 2013.04-rc1 (Mar 26 2013 - 00:33:46) for SMDK6410

CPU: S3C6410@533MHz

Fclk = 533MHz, Hclk = 133MHz, Pclk = 66MHz (ASYNC Mode)

Board: SMDK6410

DRAM: 256 MiB

WARNING: Caches not enabled

Flash: *** failed ***

ERROR ### Please RESET the board

串口能够打印出信息，说明PC顺利跳转到_main函数的运行地址，完成了从SD卡拷贝数据到内存和代码重定位工作。同时，系统被挂起

不再运行，根据打印信息可以看出，挂起是由于Flash初始化工作失败导致的。其实，单板并没有配置可以直接存储和执行程序的 Flash 存储器。arch/arm/lib目录下的board.c的board_init_r函数对Flash初始化代码如下所示。

```
565 #if !defined(CONFIG_SYS_NO_FLASH)
566    puts("Flash: ");
567
568    flash_size = flash_init();
569    if (flash_size > 0) {
570 # ifdef CONFIG_SYS_FLASH_CHECKSUM
571        print_size(flash_size, "");
572        /*
574         *
575         * NOTE: Maybe we should add some
WATCHDOG_RESET()? XXX
576         */
577        if (getenv_yesno("flashchecksum") == 1) {
578          printf("  CRC: %08X", crc32(0,
579              (const unsigned char *)
CONFIG_SYS_FLASH_BASE,
580              flash_size));
581        }
582        putc('\n');
583 # else  /* !CONFIG_SYS_FLASH_CHECKSUM */
584        print_size(flash_size, "\n");
585 # endif /* CONFIG_SYS_FLASH_CHECKSUM */
586    } else {
```

```
587         puts(failed);
588         hang();
589     }
590 #endif
```

当程序检测到flash_size不大于0时，调用puts函数打印出failed指针指向的字符串（*** failed ***），并且调用hang函数进入死循环。只需要简单地修改第587行和第588行程序就能解决这个问题。

```
587       puts("0 Mib\n");
588     //  hang();
```

重新编译U-Boot，将生成的u-boot.bin烧写至SD卡。启动时，串口重复输出大量的“raise: Signal # 8 caught”信息。事实上，对于后期版本的U-Boot，这是一个长期存在bug，DENX 小组同时也注意到这类 bug。为了代码的可读性，DENX 小组并没有在源码中修正，而是鼓励大家直接去修改timer.c文件，并且有专门的数据结构供修改使用。

在程序清单2.1的global_data 结构体中，它有一个struct arch_global_data arch 成员。arch是跟体系相关的全局变量，这些变量用在与 CPU 体系相关的 clock.c、timeer.c 文件中。structarch_global_data结构的定义如下：

```
struct arch_global_data {
  #if defined(CONFIG_FSL_ESDHC)
    u32 sdhc_clk;
  #endif
  #ifdef CONFIG_AT91FAMILY
    /* "static data" needed by at91's clock.c */
    unsigned long cpu_clk_rate_hz;
    unsigned long main_clk_rate_hz;
```

```
        unsigned long mck_rate_hz;
        unsigned long plla_rate_hz;
        unsigned long pllb_rate_hz;
        unsigned long at91_pllb_usb_init;
      #endif
        /* "static data" needed by most of timer.c on ARM
    platforms */
        unsigned long timer_rate_hz;
        unsigned long tbu;
        unsigned long tbl;
        unsigned long lastinc;
        unsigned long long timer_reset_value;
      #ifdef CONFIG_IXP425
        unsigned long timestamp;
      #endif
      #if !(defined(CONFIG_SYS_ICACHE_OFF) &&
  defined(CONFIG_SYS_DCACHE_OFF))
        unsigned long tlb_addr;
        unsigned long tlb_size;
      #endif
      };
```

修改的方法很简单：timr.c中用gd->arch.timer_rate_hz替代timer_load_val，用gd->arch.timer_reset_value替代timestamp，用gd->arch.lastinc替代lastdec。去掉timestamp、lastdec和timer_load_val变量的声明，并在文件的顶部位置加上gd全局变量的声明DECLARE_GLOBAL_DATA_PTR。修改后，串口打印信息为：

U-Boot 2013.04-rc1 (Mar 28 2013 - 00:11:52) for SMDK6410

```
CPU:  S3C6410@533MHz
      Fclk = 533MHz, Hclk = 133MHz, Pclk = 66MHz (ASYNC
Mode)
Board:  SMDK6410
DRAM:  256 MiB
WARNING: Caches not enabled
Flash: 0 Mib
NAND:  No oob scheme defined for oobsize 218
2048 MiB
*** Warning - bad CRC, using default environment
In:  serial
Out:  serial
Err:  serial
Net:  CS8900-0
Hit any key to stop autoboot:  0
lq@u-boot #
```

至此，尽管很容易从上面的打印信息判断出当前的移植工作并不完善，但还是成功地从 SD卡中启动了U-Boot。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第二章 07 课、08课、09课、12课（SD卡启动U-Boot-2013.04移植）。

2.3.3 SD/MMC驱动移植

多媒体记忆卡（Multimedia Card，MMC 卡）是一种快闪记忆卡标准，它由闪迪（SanDisk）和西门子于1997年联手推出。SD卡体积小巧，广泛应用在数码产品上，由日本的松下公司、东芝公司和SanDisk

公司共同开发的一种全新的存储卡产品。SD 卡在外形上同MultiMedia Card卡保持一致，SD卡接口向下兼容MMC卡，访问SD卡的协议及部分命令也适用于MMC。SDHC是“High Capacity SD Memory Card”的缩写，即“高容量SD 存储卡”。随着数据存储海量、高速特性的出现，人们对存储设备提出了更高的要求。SD 卡中使用的 FAT16 文件系统所支持的最大容量仅为2GB，并不能完全满足用户的要求。为了解决SD设备日趋增大的容量要求，2006年5 月SD 协会发布了最新版的SD 2.0 的系统规范，在其中就规定SDHC 是符合新的规范、且容量大于 2GB小于等于 32GB的 SD卡。由于历史的原因，在 U-Boot中，这些记忆卡统称为MMC设备。

U-Boot对MMC设备的支持已经相当完善，驱动的实现代码位于drivers\mmc，在顶层配置文件中添加CONFIG_GENERIC_MMC宏使mmc.c文件编译进工程。如图2.10所示，U-Boot对MMC设备支持分为几个层次：主机控制器、驱动通用层、SD/MMC应用层、FAT文件系统、文件系统应用层和MMC环境变量。其中，主机控制器层的实现依赖于具体的平台，通用层依赖于主机控制器层，其他各层均建立在驱动通用层之上。用户移植SD/MMC程序，只需根据具体平台编写少量主机控制器层的程序，这种分层方式有效地提高了代码的可重用性和可移植性。

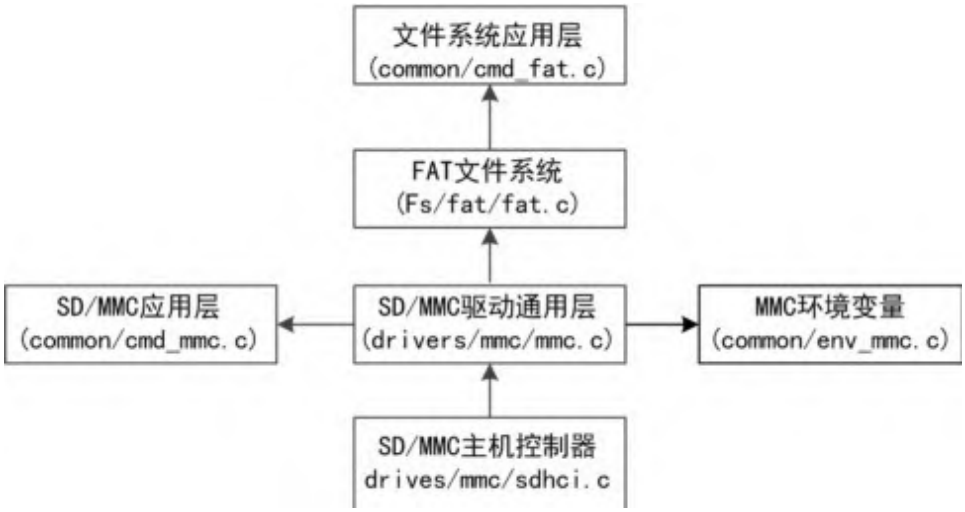


图2.10 SD/MMC驱动分层

1. 驱动通用层

通用层的实现代码在mmc.c（drivers/mmc/mmc.c）中，该层提取了SD/MMC驱动的一些通用操作，例如存储卡的识别、设置、读写。如程序清单2.3 所示，驱动程序用struct mmc 结构来描述一个MMC设备，结构体成员的功能见注释部分。

程序清单2.3 MMC 设备描述符

```
struct mmc {
/* 链表头部指针*/
  struct list_head link;
/*设备名称*/
  char name[32];
/*设备的私有数据指针*/
  void *priv;
/* 主机能够提供的电压*/
  uint voltages;
/* mmc 设备版本信息 */
  uint version;
/* 初始化标志，用于程序判定设备是否已经初始化过 */
  uint has_init;
/* 主机能够提供的最小频率 */
  uint f_min;
/* 主机能够提供的最大频率 */
uint f_max;
/ *SDHC 卡标志 */
int high_capacity;
/* 当前存储卡数据线的宽度*/
  uint bus_width;
```

```
    /* 主机控制器配置的时钟频率，除了主机控制器层，其他代码层修改这个值没有意义*/
      uint clock;
    /* mmc 总线特性*/
      uint card_caps;
    /* 主机控制器特性 * /
      uint host_caps;
      uint ocr;       /* OCR寄存器内容 */
      uint scr[2];       /* SCR寄存器内容*/
      uint csd[4];/* CSD 寄存器内容*/
      uint cid[4];/* CID 寄存器内容*/
      ushort rca;/* RCA 寄存器内容*/
    /* 分区配置 */
      char part_config;
    /* 分区数目 */
      char part_num;
    /* 数据发送速率 */
      uint tran_speed;
    /* 块读取的长度 */
      uint read_bl_len;
    /* 块写入的长度 */
      uint write_bl_len;
    /* 每组擦除的大小 */
      uint erase_grp_size;
    /* 存储卡的特性 */
      u64 capacity;
    /* 块设备结构体*/
```

```
    block_dev_desc_t block_dev;
  /* 接口函数：用以向mmc 设备发送命令，功能代码在主机驱动层实现 */
    int (*send_cmd)(struct mmc *mmc,
      struct mmc_cmd *cmd, struct mmc_data *data);
  /* 接口函数：用于配置主机控制器，例如设置总线宽度、时钟频率等*/
    void (*set_ios)(struct mmc *mmc);
  /* 接口函数：用于初始化主机控制器 */
    int (*init)(struct mmc *mmc);
  /* 接口函数：用于报告mmc 设备的连接状态 */
  int (*getcd)(struct mmc *mmc);
  /* 在multiblock 模式下，能够处理的块的最大数目 */
    uint b_max;
  };
```

struct mmc 结构抽象了MMC 设备驱动的共性,它描述了MMC 设备的电压范围、频率范围、容量大小、寄存器等信息，定义了与主机控制器层的接口函数send_cmd、set_ios、init和getcd以及块设备相关的信息。

struct mmc 描述符里面含有struct list_head link成员，当有多个SD/MMC 设备注册时，它们连接在一起形成一条mmc设备链表。在构造链表数据结构时，U-Boot借鉴了Linux内核的方法，在include/linux/list.h 定义了一个struct list_head 结构。

```
struct list_head {
    struct list_head *next, *prev;
};
```

struct list_head 这个结构比较特殊，它内部没有任何的处理数据，只是起到连接链表的作用。对于它当前所在的这个结点来说，next指向下一个节点，prev指向上一个节点。这个结构经常作为成员与其他数据类型一起组成一个新的结构体，如struct mmc结构体的成员struct list_head link。通常，我们通过指向struc list_head的指针来获取它所在结构体的地址，从而操作该结构体的其他数据。换句话说，利用结构体成员的地址计算出结构体的地址，实现这个功能需要一定的技巧，令人高兴的是list.t文件中已经为我们解决了这个问题。另外，list.h中还提供了更多的链表操作函数，例如添加节点、删除节点、遍历链表和对链表整体翻转等。

```
static struct list_head mmc_devices;
static int cur_dev_num = -1;
```

mmc.c中定义了两个全局变量，mmc_devices和cur_dev_num。mmc_devices指向整个MMC设备链表。cur_dev_num的初始值为-1，主要作用是为MMC设备分配一个不小于0的编号，它的值同时就是当前已注册设备的编号。

驱动通用层提供了一个 MMC 设备注册函数 mmc_register，这个函数被主机控制器层调用，用于将MMC设备添加到MMC设备链表里。mmc_register的代码如下。

```
int mmc_register(struct mmc *mmc)
  {
    /* Setup the universal parts of the block interface
just once */
    mmc->block_dev.if_type = IF_TYPE_MMC;
    mmc->block_dev.dev = cur_dev_num++;
    mmc->block_dev.removable = 1;
    mmc->block_dev.block_read = mmc_bread;
```

```
    mmc->block_dev.block_write = mmc_bwrite;
    mmc->block_dev.block_erase = mmc_berase;
    if (!mmc->b_max)
      mmc->b_max = CONFIG_SYS_MMC_MAX_BLK_COUNT;
    INIT_LIST_HEAD (&mmc->link);
    list_add_tail (&mmc->link, &mmc_devices);
    return 0;
  }
```

mmc_register函数为MMC设备填充了块设备处理信息、分配了一个设备编号和将struct mmc结构添加到链表中。当应用层需要操作 MMC 设备时，首先必须传入设备编号，调用find_mmc_device。

```
  struct mmc *find_mmc_device(int dev_num)
  {
    struct mmc *m;
    struct list_head *entry;
    list_for_each(entry, &mmc_devices) {
      m = list_entry(entry, struct mmc, link);
      if (m->block_dev.dev == dev_num)
        return m;
    }
    printf("MMC Device %d not found\n", dev_num);
    return NULL;
  }
```

find_mmc_device函数根据传入的设备编码dev_num，遍历链表。如果找到对应的设备编号返回该设备的struct mmc 结构体，否则返回NULL。mmc.h 中还定义了两个结构体用于参数传递：命令描述符mmc_cmd和数据描述符mmc_data。

```
struct mmc_cmd {
  ushort cmdidx;/* 命令索引*/
  uint resp_type; /* 回应模式*/
  uint cmdarg; /* 命令参数*/
  uint response[4]; /*回应数值*/
  uint flags; /* 命令其他标志*/
};
struct mmc_data {
  union { /*共用体当用于读取数据时变量为dest，否则为
src*/
    char *dest;
    const char *src; /* src buffers don't get written
to */
  };
  uint flags; /*标志读数据(MMC_DATA_READ)还是写数据
(MMC_DATA_WRITE)*/
  uint blocks;/*传输数据的块数*/
  uint blocksize;/*每块数据的大小*/
};
```

2. 主机控制器

S3C6410 处理器有三通道 SD/MMC Host 控制器，控制器的首地址分别为 0x7C20_0000、0x7C30_0000和0x7C40_0000，如图2.11所示，单板使用通道0，SD卡模式。

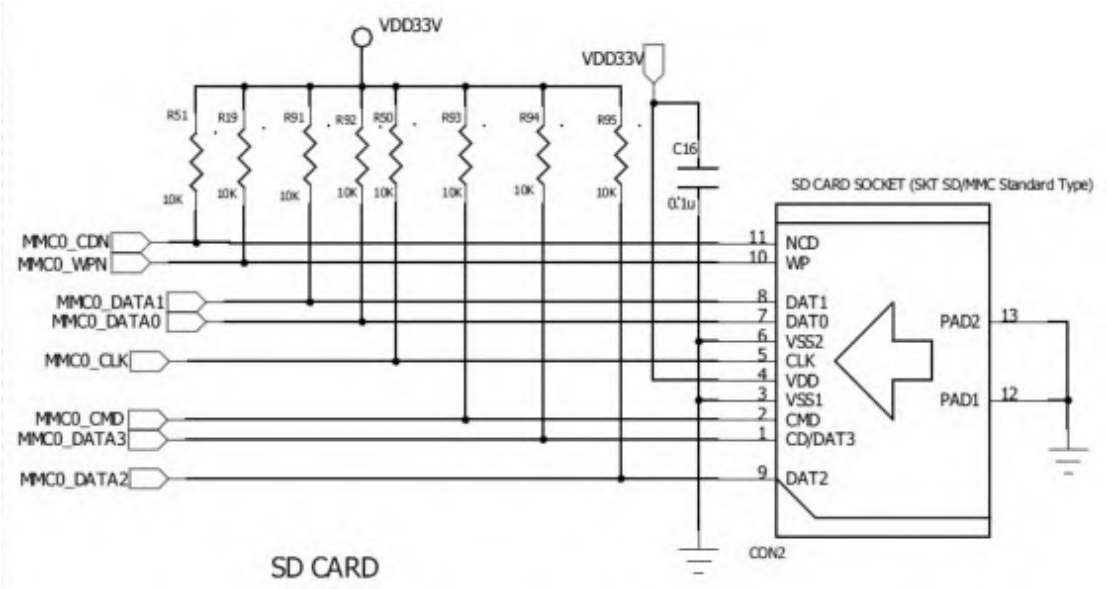


图2.11 SD卡连接原理图

SD/MMC驱动通用层实现了对MMC 设备常用的操作，这些操作最终会调用struct mmc 的接口函数。以mmc_bread块设备读函数为例，它的调用过程是：

```
mmc_bread
    mmc_read_blocks
      mmc_send_cmd
        mmc->send_cmd(mmc, cmd, data)
```

通用层并不关心 send_cmd 接口函数是如何实现的，它只是简单地把参数传递给接口函数，并且假设接口函数能够实现自己的操作。主机控制器必须实现这些函数功能，保证函数实现的可靠性。市场上，有众多生产MMC设备的厂家，如闪迪、东芝、金士顿等。我们知道，尽管产品出自不同的制造商，但是它们生产时都必须遵循相同的规范协议。事实上，不仅MMC设备有一套相同的规范协议，SD/MMC 主机控制器也有一套协议来规范 SD/MMC 主机控制器的设计， S3C6410处理器的SD/MMC控制器遵循SDA主机标准规范。U-Boot在drivers/mmc/sdhci.c文件中，已经为遵循SDA 主机标准规范的处理器完成了主机控制器需要做的工作，实现了struct mmc所有的接口函数

和填充了部分数据。要想把 sdhci.c 添加到工程中必须在顶层配置文件中添加CONFIG_SDHCI宏。U-Boot利用struct sdhci_host 结构来描述sdhci_host 主机，用户根据处理器初始化这个结构，然后调用add_sdhci函数就能加入sdhci总线驱动。

```
struct sdhci_host {
    char *name;
    void *ioaddr;
    unsigned int quirks;
    unsigned int host_caps;
    unsigned int version;
    unsigned int clock;
    struct mmc *mmc;
    const struct sdhci_ops *ops;
    int index;
    void (*set_control_reg)(struct sdhci_host *host);
    void (*set_clock)(int dev_index, unsigned int div);
    uint voltages;
};
```

SDA 主机标准规范定义了一套寄存器组，这些寄存器组的首地址由用户赋给 sdhci_host 的ioaddr成员。sdhci.c根据ioaddr的值和寄存器的偏移地址就能访问相应的寄存器。在drivers/mmc目录下新建一个C源码文件，命名为s3c6410_sdhci.c，文件的内容如下。

```
/*
* Samsung S3C6410 SD/MMC driver
* Based on sdhci.c
* Author: LiQiang E-mail: jxustlq@163.com
* Licensed under the GPL-2 or later.
```

```
 */
 #include <common.h>
 #include <malloc.h>
 #include <sdhci.h>
 #include <asm/arch/s3c6410.h>
   #ifdef  CONFIG_MMC_CHANNEL
   #define MMC_CHANNEL CONFIG_MMC_CHANNEL
   #else
   #define  MMC_CHANNEL 0
   #endif
   #define ELFIN_HSMMC_BASE  0x7C200000
   #define MMC_REGS_BASE    (ELFIN_HSMMC_BASE +
0x100000*MMC_CHANNEL)
   static void sdhc_set_gpio()
   {
     u32 reg;
   #if (MMC_CHANNEL == 0)
     reg = readl(GPGCON) & 0xf0000000;
     writel(reg | 0x02222222, GPGCON);
     reg = readl(GPGPUD) & 0xfffff000;
     writel(reg, GPGPUD);
   #elif (MMC_CHANNEL == 1)
     writel(0x00222222, GPHCON0);
     writel(0x00000000, GPHCON1);
     reg = readl(GPHPUD) & 0xfffff000;
     writel(reg, GPHPUD);
   #else
```

```
    printf("#####err: SDMMC channel is not defined!\n");
  #endif
  }
  int s3c_sdhci_init(u32 regbase, u32 max_clk, u32
min_clk, u32 quirks)
  {
    struct sdhci_host *host = NULL;
    host = (struct sdhci_host *)malloc(sizeof(struct
  sdhci_host));
    if (!host) {
      printf("#####err: sdhci_host malloc fails!\n");
      return 1;
    }
    sdhc_set_gpio();
    host->name = "Samsung  Host Controller";
    host->ioaddr = (void *)regbase;
    host->quirks = quirks;
    host->voltages = MMC_VDD_32_33 | MMC_VDD_33_34 |
  MMC_VDD_165_195;
    if (quirks & SDHCI_QUIRK_REG32_RW)
      host->version = sdhci_readl(host,
    SDHCI_HOST_VERSION - 2) >> 16;
    else
      host->version = sdhci_readw(host,
    SDHCI_HOST_VERSION);
    host->host_caps = MMC_MODE_HC;
    add_sdhci(host, max_clk, min_clk);
```

```
  return 0;
}
int board_mmc_init(bd_t *bis)
{
  return s3c_sdhci_init(MMC_REGS_BASE, 52000000,
400000, 0);
}
```

在当前目录下的Makfile文件中添加下面一行内容：

```
51 COBJS-$(CONFIG_S3C6410_SDHCI) += s3c6410_sdhci.o
```

在drivers/mmc目录下打开sdhci.c文件：

```
165    else if (cmd->resp_type & MMC_RSP_BUSY) {
166        flags = SDHCI_CMD_RESP_SHORT_BUSY;
167    //  mask |= SDHCI_INT_DATA_END; /*liqiang*/
168    } else
169        flags = SDHCI_CMD_RESP_SHORT;
```

注释掉第167行内容，这是源码的一个bug，如果不注释这条语句会导致数据读写超时。

在顶层配置文件smdk6410.h中加入CONFIG_S3C6410_SDHCI、CONFIG_MMC_SDMA宏，使sdhci.c利用DMA传输数据。

在本节MMC驱动移植中需向顶层配置文件smdk6410.h添加MMC驱动宏定义如下：

```
/*support for mmc*/
#define CONFIG_MMC
#define CONFIG_GENERIC_MMC
#define CONFIG_CMD_MMC
#define CONFIG_SDHCI
#define CONFIG_MMC_SDMA
```

#define CONFIG_S3C6410_SDHCI

当加入CONFIG_MMC_SDMA宏时，sdhci.c文件中定义DMA每次传输的大小。

#define SDHCI_DEFAULT_BOUNDARY_SIZE (512 * 1024)

因此，在sdhci.c 文件中的add_sdhci函数将struct mmc 的b_max赋值为1024（块）。

```
490    mmc->b_max = 1024; /*liqiang*/
491    sdhci_reset(host, SDHCI_RESET_ALL);
492    mmc_register(mmc);
```

3. SD/MMC应用层

common/cmd_mmc.c调用驱动通用层的函数，实现了以下mmc命令以用于交互。在顶层文件中添加#define CONFIG_CMD_MMC 宏将使U-Boot支持mmc 命令。工程编译后，将镜像文件烧写至SD卡，启动系统。输入mmcinfo命令，串口打印出如下信息：

```
lq@u-boot #mmcinfo
Device: Samsung Host Controller
Manufacturer ID: 2
OEM: 544d
Name: SA02G
Tran Speed: 50000000
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 1.8 GiB
Bus Width: 4-bit
lq@u-boot #
```

除了mmcinfo，U-Boot还支持其他mmc命令，如下所示：

mmc read addr blk# cnt

mmc write addr blk# cnt

mmc erase blk# cnt

mmc rescan

mmc part - lists available partition on current mmc device

mmc dev [dev] [part] - show or set current mmc device [partition]

mmc list - lists available devices

以mmc read命令为例，addr 是读取数据在内存中的目的地址，blk是读取的扇区起始地址， cnt是需要读取的扇区数，输入的数值均为16进制。

lq@u-boot #mmc read 50000000 0 10

MMC read: dev # 0, block # 0, count 16 ... 16 blocks read: OK

从0扇区地址处读取16扇区数据至内存的0x5000_0000地址。笔者同时测试了8GB的SDHC卡，测试的结果一致。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章11课（MMC驱动移植）。

4 . FAT文件系统

common/cmd_fat.c文件提供了FAT文件系统的一些功能命令，例如fatls、fatinfo和fatload等。在顶层配置文件中添加CONFIG_CMD_FAT宏，把cmd_fat.c文件编译进内核。事实上，在很多存储设备上都能够使用 FAT 文件系统，比如 sata、mmc 等。U-Boot 利用块设备描述符 block_dev_desc_t 提供一个通用接口，屏蔽了与 FAT 文件无关的存储器介质差异。在配置文件中添加CONFIG_MMC 宏，选择MMC 设备作为块设备。

输入fatinfo命令查看fat文件系统的信息。

```
lq@u-boot #fatinfo mmc 0
Interface: MMC
  Device 0: Vendor: Man 000002 Snr 10007200 Rev: 9.12
Prod: SA02G
    Type: Removable Hard Disk
      Capacity: 1876.0 MB = 1.8 GB (3842048 x 512)
Filesystem: FAT32 "NO NAME"
```

输入fatls命令查看MMC设备中的文件信息。

```
lq@u-boot #fatls mmc 0
  58072 arm linux.docx
    279 o@.txt
      .trash-1000/
    5352 .txt
  267272 u-boot.bin
4 file(s), 1 dir(s)
```

输入fatload命令可以将MMC设备中的文件加载至内存当中。

```
lq@u-boot #fatload mmc 0 50000000 u-boot.bin[U1]
reading u-boot.bin
267272 bytes read in 29 ms (8.8 MiB/s)
```

fatload命令在后文一键烧写系统中将会用到，它的使用方法：

```
lq@u-boot #? fatload
fatload - load binary file from a dos filesystem
Usage:
fatload <interface> [<dev[:part]>] <addr><filename>
[bytes [pos]]
```

```
    - Load binary file 'filename' from 'dev' on
'interface'
    to address 'addr' from dos filesystem.
    'pos' gives the file position to start loading from.
    If 'pos' is omitted, 0 is used. 'pos' requires
'bytes'.
    'bytes' gives the size to load. If 'bytes' is 0 or
omitted,
    the load stops on end of file.
    All numeric parameters are assumed to be hex.
```

2.3.4 环境变量

仔细查看U-Boot启动时的打印信息，有如下一行信息：

*** Warning - bad CRC, using default environment

后半句警告使用的是默认的环境变量，原因是环境变量没有保存到存储介质中。使用saveenv命令可以将环境变量保存到存储介质中。环境变量用于指定U-Boot运行的一些参数，为用户预留一定的可配置性。输入printenv命令可以打印出所有的环境变量。

```
lq@u-boot #printenv
baudrate=115200
bootargs=console=ttySAC,115200
bootcmd=nand read 0xc0018000 0x60000 0x1c0000; bootm
0xc0018000
bootdelay=3
ethact=CS8900-0
stderr=serial
```

stdin=serial

stdout=serial

打印出来的环境变量包括波特率、启动参数、启动命令行、延时时间、网卡、标准出错、标准输入和标准输出。到目前为止，并没有移植NAND Flash 和网卡启动，它们的环境变量是默认的。保存环境变量的存储介质多种多样，打开common/Makefile文件，可以看到以下内容。

```
COBJS-$(CONFIG_ENV_IS_IN_DATAFLASH) += env_dataflash.o
COBJS-$(CONFIG_ENV_IS_IN_EEPROM) += env_eeprom.o
XCOBJS-$(CONFIG_ENV_IS_EMBEDDED) += env_embedded.o
COBJS-$(CONFIG_ENV_IS_IN_EEPROM) += env_embedded.o
XCOBJS-$(CONFIG_ENV_IS_IN_FLASH) += env_embedded.o
COBJS-$(CONFIG_ENV_IS_IN_NVRAM) += env_embedded.o
COBJS-$(CONFIG_ENV_IS_IN_FLASH) += env_flash.o
COBJS-$(CONFIG_ENV_IS_IN_MMC) += env_mmc.o
COBJS-$(CONFIG_ENV_IS_IN_FAT) += env_fat.o
COBJS-$(CONFIG_ENV_IS_IN_NAND) += env_nand.o
COBJS-$(CONFIG_ENV_IS_IN_NVRAM) += env_nvram.o
COBJS-$(CONFIG_ENV_IS_IN_ONENAND) += env_onenand.o
COBJS-$(CONFIG_ENV_IS_IN_SPI_FLASH) += env_sf.o
COBJS-$(CONFIG_ENV_IS_IN_REMOTE) += env_remote.o
COBJS-$(CONFIG_ENV_IS_NOWHERE) += env_nowhere.o
```

env_eeprom.c实现环境变量存储在EEPROM中，CONFIG_ENV_IS_IN_EEPROM宏用于配置这种方式，其他方式类似。common/env_common.c中是环境变量存储的通用接口函数，它们隐藏了介质间的差异。我们现在唯一能够使用的存储介质是MMC设备，只能将环境变量保存到MMC设备里面。

```
288 #ifdef CONFIG_BOOT_SD
289 #define CONFIG_ENV_IS_IN_MMC
290 #define CONFIG_SYS_MMC_ENV_DEV 0
291 #else
292 #define CONFIG_ENV_IS_IN_NAND
293 #endif
```

修改顶层配置文件，增加对环境变量存放到MMC设备中的宏，选择设备号为0。

CONFIG_ENV_OFFSET和CONFIG_ENV_SIZE用于确定环境变量在存储介质中的偏移地址和环境变量的空间大小。

```
lq@u-boot #setenv bootdelay 1
lq@u-boot #saveenv
Saving Environment to MMC...
Writing to MMC(0)... done
lq@u-boot #printenv
...
bootdelay=1
...
Environment size: 145/16380 bytes
lq@u-boot #
```

使用setenv命令重新设定启动延时时间，把延时时间从3秒改为1秒。然后保存环境变量，提示写入编号为0的MMC设备成功。再次打印环境变量，发现延时时间已经成功被修改。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章12课（FAT文件系统）。

2.4 U-Boot命令实现

当U-Boot成功启动后，用户可以用它的内部命令进行人机交互：查看目标系统的信息、设置环境变量等。本节内容介绍U-Boot命令的使用方法和实现原理，最后通过新增一个“hello”命令加1等对U-Boot命令实现的理解。

2.4.1 命令概述

U-Boot命令为用户开发提供一些交互功能，本身已经带有几十个常用的命令，实现了环境变量设置、存储器操作、数据操作和系统引导等。在等待用户输入的状态下输入“help”或“?”，查看当前U-Boot中支持的所有命令的名称和简要说明。

```
lq@u-boot #
?    - alias for 'help'
base  - print or set address offset
bdinfo  - print Board Info structure
...
usbboot - boot from USB device
version - print monitor, compiler and linker version
```

类似Linux操作系统的man命令的用法，U-Boot也提供了在线帮助功能，“help”或“?”后面添加所要查询命令的名称，就能得到该命令的帮助信息。

```
lq@u-boot #? bdinfo
bdinfo - print Board Info structure
Usage:
bdinfo
```

```
lq@u-boot #help bdinfo
bdinfo - print Board Info structure
Usage:
bdinfo
```

这些命令对于一些普通的交互操作基本能够胜任，但在嵌入式开发过程中如有特殊要求，需要添加新的命令以增加新的功能。U-Boot命令巧妙的实现机制能够让用户方便地建立一个新的命令，本文先分析U-Boot命令的实现原理，再以一个具体的实例演示命令增加的过程和方法。

2.4.2 实现原理

在启动分析的时候，我们已经知道程序最终执行common/main.c中的main_loop。在此之前都是进行一些初始化工作，U-Boot的main_loop函数相当于main主函数。main_loop函数的结构很复杂，它所做的工作与具体的平台无关，主要目的是处理用户输入的命令和引导内核启动。main_loop 函数的调用关系错综复杂，而且掺杂关系复杂的条件编译，我们抓住与命令实现密切相关的操作来分析命令的实现原理。命令实现的大致流程如图2.12所示。

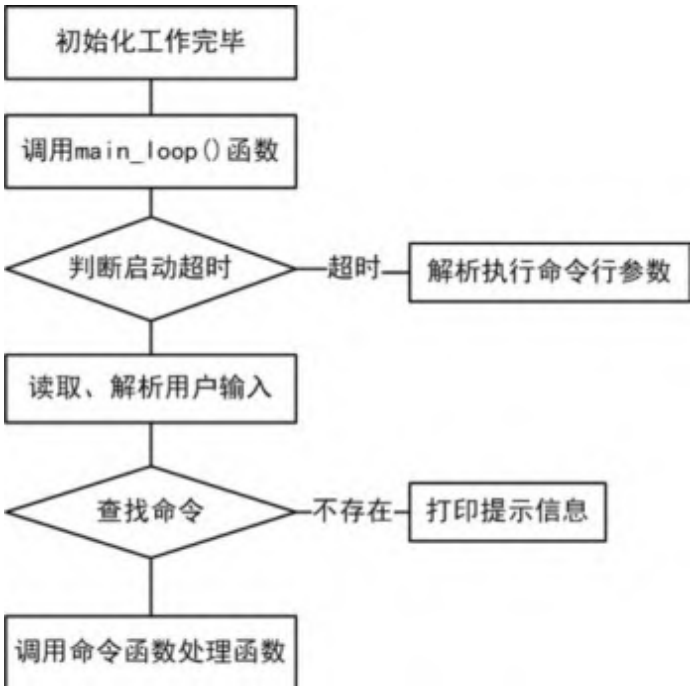


图2.12 命令实现流程

1．启动延时

如果配置了启动延迟功能，U-Boot等待用户从控制台（一般为串口）输入字符，等待的时间由顶层配置文件中的宏定义 CONFIG_BOOTDELAY 决定。在此期间，只要用户按下任意按键就会中断等待，进入命令行输入模式。如果没有配置启动功能或者启动延时超过了设置的时间， U-Boot 运行启动命令行参数，启动命令参数在顶层配置文件中，由 CONFIG_BOOTCOMMAND宏定义。

2．读取命令行输入

命令行输入模式实际上是一个死循环，循环体简化后如下所示：

```
for (;;) {
  len = readline (CONFIG_SYS_PROMPT);
  flag = 0;/* assume no special flags for now */
  if (len > 0)
    strcpy (lastcommand, console_buffer);
  else if (len == 0)
```

```
            flag |= CMD_FLAG_REPEAT;
        if (len == -1)
            puts ("<INTERRUPT>\n");
        else
            rc = run_command(lastcommand, flag);
        if (rc <= 0) {
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
```

每次循环调用readline函数从控制台读取命令行，并且读取到的字符存储在console_buffer缓冲区中。console_buffer缓冲区的长度在顶层文件中通过CONFIG_SYS_CBSIZE宏定义。当该函数在接收到一个回车键时认定为命令行输入结束，返回命令行长度len。如果len大于0，将存储在缓冲区的命令行拷贝至静态数组lastcommand中，flag设置为0。如果len等于0，即readline函数仅仅接收到一个回车键，即直接返回，flag设置为CMD_FLAG_REPEAT，lastcommand数组存放的数据不变。flag 用于标志是否重复上次操作，每个命令都有一个repeatable标志，当命令的该标志为1时，此时，命令能够重复操作。把lastcommand和flag作为run_command函数的参数，进而调用run_command函数。

从 run_command 函数是否会返回的角度看，U-Boot 的命令分为两类。一类是函数返回数值rc，rc小于等于0，则传入的命令行参数有误，命令无效，此时把lastcommand数组清零，不再执行重复操作。另外一类是不再返回，一去不再复返，例如bootm、go等命令，这类用于启动内核，将CPU的管理权从U-Boot交付给内核，完成自己启动内核的终极使命。

3. 解析命令行

传入的 lastcommand 参数仅仅是 readline 函数读取到用户输入的字符，接下来最主要的工作是解析命令行。首先判断传入的lastcommand参数是否为空，如果是返回-1，否则继续往下解析。截取函数的关键代码如下，str指针指向lastcommand区域。

```
while (*str) {
  for (inquotes = 0, sep = str; *sep; sep++) {
    if ((*sep=='\'') &&
      (*(sep-1) != '\\'))
        inquotes=!inquotes;
    if (!inquotes &&
      (*sep == ';') && /* separator*/
      ( sep != str) && /* past string start */
      (*(sep-1) != '\\'))  /* and NOT escaped  */
        break;
  }
```

U-Boot允许命令行存在多个命令，命令间用“;”或者“\;”字符分割。

```
token = str;
if (*sep) {
  str = sep + 1;/* start of command for next pass */
  *sep = '\0';
}
else
  str = sep; /* no more commands for next pass */
/* Extract arguments */
if ((argc = parse_line (finaltoken, argv)) == 0) {
```

```
        rc = -1; /* no command at all */
        continue;
      }
      if (cmd_process(flag, argc, argv, &repeatable, NULL))
        rc = -1;
```

首先解析一个命令，token指向待解析命令的地址。parse_line函数分离出命令的各个参数，分别存放在argv中，参数的数目为argc，接着调用common/command.c文件中的cmd_process函数处理解析得到的命令。值得注意的是，命令的第一个参数是命令的名称。当前命令处理完毕后， token指向命令行中的下一个命令，直到所有的命令都处理完毕。

4. 命令处理

main.c中的代码实现了将一个命令的所有参数分离存放在argv数组中，参数的数目为argc，完成了读取命令行和解析命令行的工作。命令的处理由common/command.c文件中的函数完成。U-Boot在include/command.h中定义了一个非常重要的cmd_tbl_s结构体，它在命令的实现方面起着至关重要的作用。

```
    struct cmd_tbl_s {
      char    *name;    /* 命令名称        */
      int     maxargs; /* 命令的最大参数  */
      int     repeatable;  /* 是否可重复（按回车键是否会重复执行）
      */
      int     (*cmd)(struct cmd_tbl_s *, int, int, char * const []);  /* 命令响应函数*/
      char    *usage;    /* 简短的用法说明  */
    #ifdef  CONFIG_SYS_LONGHELP
```

```
    char    *help;    /* 较详细的帮助 */
  #endif
  #ifdef CONFIG_AUTO_COMPLETE
    /* 响应自动补全参数*/
    int    (*complete)
(intargc,char*constargv[],charlast_char,intmaxv,char*cmdv
[]);
  #endif
  };
```

cmd_tbl_s结构体包含的成员变量：命令名称、最大参数个数、可重复性、命令响应函数、用法、帮助和命令补全函数，每个命令都由这个结构体来描述。当输入“help”或者“?”会打印出所有的命令和它的usage，输入“help”或者“?”和命令名称时，会打印出help信息。

添加一个命令时，利用宏U_BOOT_CMD定义一个新的cmd_tbl_s结构体，并对这个结构体初始化和定义结构体的属性。例如，在文件common/cmd_bdinfo.c中：

```
U_BOOT_CMD(
    bdinfo, 1, 1, do_bdinfo,
    "print Board Info structure",
    ""
);
```

增加了一个命令，它的名称为bdinfo，最大参数数目为1，可重复，响应函数是do_bdinfo， usage 为“print Board Info structure”，没有帮助信息。U_BOOT_CMD 宏在include/command.h中定义，当不配置命令补全时，它最终被展开为：

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
```

cmd_tbl_t __u_boot_cmd_##name __attribute__((unused, section(".u_boot_cmd"), aligned(4))) = {#name, maxargs, rep, cmd, usage, help}

其中，“##”与“#”是预编译操作符，“##”表示字符串连接，“#”表示后面紧接着的是一个字符串。cmd_tbl_t就是struct cmd_tbl_s，用于__u_boot_cmd_##name 结构体。__attribute__定义了结构体的属性，将结构体放在.u_boot_cmd段中。简单的说，就是利用U_BOOT_CMD定义structcmd_tbl_s结构体变量，并把类变量都放在一个段中。在链接脚本中指定了.u_boot_cmd段的起始地址和结束地址，又已知每个struct cmd_tbl_s结构体占用内存空间的大小，这样就很方便地遍历所有的struct cmd_tbl_s 结构体。这种巧妙的方式充分利用了链接器的功能特点，避免了花费大量的精力，去维护和更新命令结构体表。

```
cmdtp = find_cmd(argv[0]);
if (cmdtp == NULL) {
    printf("Unknown command '%s' - try 'help'\n",
argv[0]);
    return 1;
}
```

cmd_process函数首先调用find_cmd函数根据传入的参数，在.u_boot_cmd段区域查找命令，如果没有找到对应的命令，打印出提示信息并返回。如果找到则返回命令结构体 cmdtp，再检查传入参数的合法性，最后通过cmd_call函数调用命令响应函数（cmdtp->cmd）（cmdtp, flag, argc,argv）。

2.4.3 新增命令

为了更好地理解U-Boot的实现原理，我们建立一个简单的hello命令，它没有任何实质性的作用。在common目录下新建一个cmd_hello.c文件，加入以下代码。

```
/*
  *  Built a simple command
  *  Author: LiQiang  E-mail: jxustlq@163.com
  *  date: 2013/04/07
  *  Licensed under the GPL-2 or later.
  */
#include <common.h>
#include <command.h>
static int do_hello(cmd_tbl_t *cmdtp, int flag, int
 argc, char * const argv[])
{
  int i = 0;
  for(i = 0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 0;
}
U_BOOT_CMD(hello, 10, 1, do_hello,
  "just for test\n",
  "\n"
  "  hello command---\n"
  "    -- Built a simple command");
```

修改common目录下的Makefile文件，增加对cmd_hello.c的编译。do_hello()是hello命令的响应函数，打印出所有的参数。输入“help”或“?”可以查看hello命令的简要说明。

```
go   - start application at address 'addr'
hello  - just for test
help   - print command description/usage
```

测试下 hello命令的效果：

```
lq@u-boot # hello U-Boot world
argv[0] = hello
argv[1] = U-Boot
argv[2] = world
```

分别打印出命令的3个参数hello、U-Boot和world。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章13课（U-Boot命令）。

2.5 NAND Flash设备移植

在嵌入式系统开发中，Flash作为一种安全、快速的存储体，具有体积小、功耗低、容量大、成本低、掉电数据不丢失等一系列优点，成为嵌入式系统中数据和程序使用最广泛的存储介质。Flash根据结构的不同可以将其分成NOR Flash和NAND Flash 两种类型。NOR Flash带有SRAM接口，将地址线和数据线分离，具有EIP（Execute In Place）特性，应用程序可以直接在存储器内部运行。而NAND Flash 是利用I/O口串行读取数据，总线分时复用，相对NOR Flash 使用起来比较复杂。但是，NOR Flash 和NAND Flash 相比较，NOR Flash 有布线多、成本高、存储容量小、擦写速度慢等缺点。通常在设计嵌入式系统时，利用小容量的 NOR Flash，如 2MB [1]，存储启动程序和内核，大容量的NAND Flash 存储文件系统数据。本文使用的单板没有配备

NOR Flash，因此必须结合S3C6410 支持NAND Flash 启动的特点，实现系统程序和数据的存储。

2.5.1 NAND Flash的结构

当前广泛使用的NAND Flash 根据存储单元的结构分为两类：SLC（Single Layer Cell，单层单元）和MLC（Multi-Level Cell，多层单元）。SLC的特点是成本高、容量小、速度快，可以存取多达10万次，而MLC的特点是容量大、成本低，但是速度慢只能存取约1万次。由于MLC可以在一个单元中有2bit数据，这样同样大小的晶圆就可以存放更多的数据，也就是成本相同的情况下，容量可以做的更大，这也是同样容量大小时，MLC价格比SLC低很多的原因。

NAND Flash 的读写都以页为单位进行，每页由数据区和OOB 区两部分组成。主存区存储数据，OOB 区通常用来存储 ECC 信息和一些文件系统信息。写 NAND Flash 时，每一位只能从 1变为0，不能从0 变成1。写之前必须擦除NAND Flash（全部变为1），擦除的最小单位是块。不同型号的NAND Flash 对应的页、块的大小和数目不一样，以单板配置的K9GAG08U0D 芯片为例。如图2.13所示，K9GAG08U0D一共有4096块，每块有128页，每页由4K的数据区和218KB的OOB区组成，总的大小为17256Mbits。

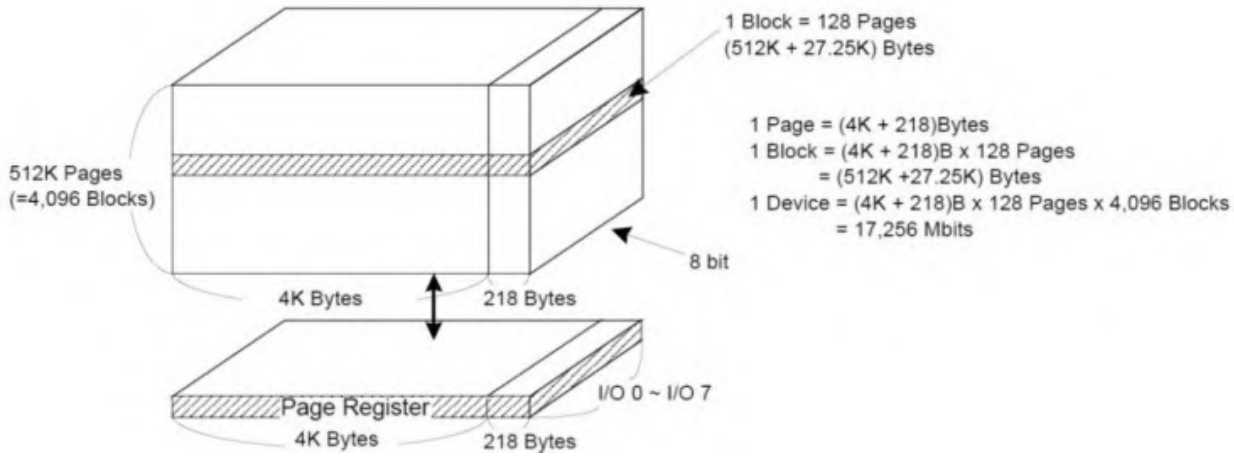


图2.13 NAND Flash存储结构

1．功能引脚介绍

如图2.14所示，K9GAG08U0D芯片共有48个引脚，其中大多数引脚是悬空（NC）状态。该芯片只有8位的I/O口，地址、数据和命令线分时复用，分时复用的方法是通过不同的控制线发出不同的信号，从而实现不同的操作。

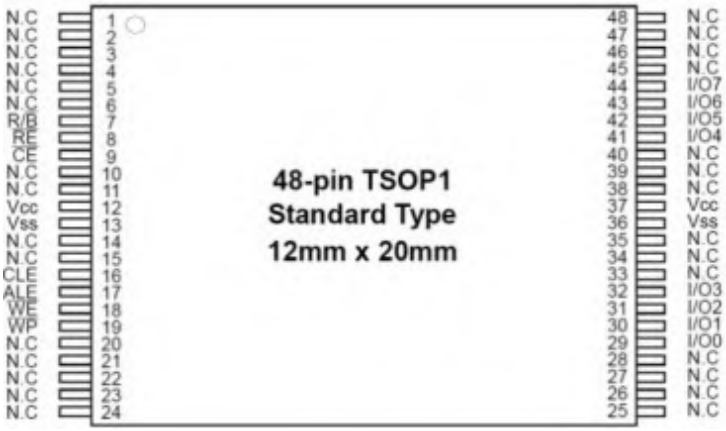


图2.14 K9GAG08U0D引脚图

芯片各功能引脚的描述如下。

ALE：Address Latch Enable，地址锁存使能，高电平时表示I/O端口输入的数据为地址，当WE#信号的上升沿出现时，端口上的数据被写入芯片内部的地址寄存器。

CLE：Command Latch Enable，命令锁存使能，高电平时表示I/O端口输入的数据为命令，当WE#信号的上升沿出现时，端口上的数据被写入芯片内部的命令寄存器。

CE#：Chip Enable，芯片选择使能，低电平时选通NAND Flash。

RE#：Read Enable，读使能，当该管脚为低电平时，对NAND Flash 进行读操作；对NAND Flash写操作时，该管脚必须为高电平。

WE#：Write Enable，写使能，当该管脚为低电平时，对NAND Flash 进行写操作；对NANDFlash读操作时，该管脚必须为高电平。

WP#：Write Protect，写保护，当该管脚为低电平时，写保护有效，此时不能对芯片进行写入或者擦除操作。

R/B#：Read/Busy，准备/忙碌标志，当该管脚为高电平时表示芯片处于空闲状态，否则处于忙碌状态。

I/O0-I/O7：Data Input/Output，数据的输入输出口。当芯片被选中，或者输入有效时，I/O 口为高阻态。

Vcc 是电源，Vss 是地。

2. 常用操作

NAND Flash 的基本操作有：复位操作、读芯片ID 操作、读状态操作、读页操作、写页操作和块擦除操作等。由 CLE、CE#、RE#、WE#、WP#信号管脚协调配合，将 I/O 端口输入的数据锁存到NAND Flash 相应的指令寄存器、地址寄存器或数据寄存器；或者通过I/O 端口把状态寄存器、数据寄存器中的数据从NAND Flash 发送出去。在进行对NAND Flash 操作前必须检测R/B#管脚，判断NAND Flash 是否处于空闲状态。

2.5.2 控制器的特性

NAND Flash 控制器为操作NAND Flash 存储设备提供了一个简单的接口，它负责产生命令、地址和数据时序，用户只需访问控制器模块的特殊功能寄存器（SFR）即可。

如图2.15 所示，S3C6410 处理器NAND Flash 控制器包括以下特性：

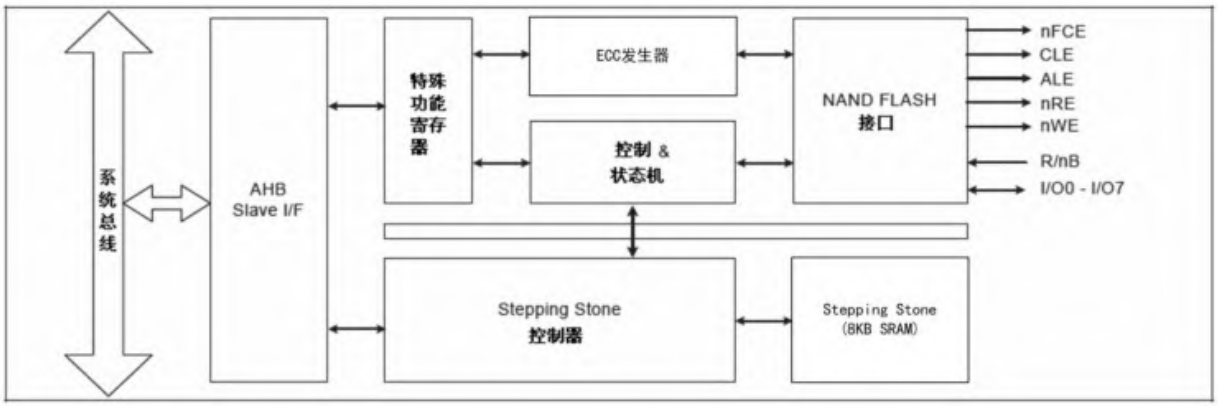


图2.15 NandFlash控制器功能框图

（1）支持自动NAND Flash 启动模式，能将NAND Flash 存储器的8KB 数据拷贝至SteppingStone中。

（2）用户可以通过软件直接访问控制器的寄存器，对存储器进行读写、擦除等操作。

（3）8位存储器总线接口。

（4）硬件生成ECC校验码。

（5）同时支持SLC和MLC 类型的NAND Flash，1位ECC用于SLC NAND Flash，4 位、8位ECC用于MLC NAND Flash。

为了兼容不同操作频率的NAND Flash，控制器发出时序的时间参数能够配置。

目前市场上，相对SDRAM和NAND Flash，NOR Flash的价格比较昂贵。为了降低开发成本，结合S3C6410 处理器支持从NAND Flash 中启动的特点，单板使用NAND Flash 存储数据，程序在SDRAM中运行。

1. NAND Flash 启动

我们已经知道IROM启动模式时，有很多类型的存储器可以作为启动设备，包括NAND Flash在内。与我们约定SD卡启动定义（以IROM为启动设备、SD/MMC为存储设备的启动方式）不同，就算选择NAND Flash 作为IROM启动方式的存储设备，也不能称为NAND Flash 启动。在图 2.6 中，当 XSELNAND 管脚为 1，OM[4:0]为 0000X、0001X、

0010X 或 0011X 均为保留（RESERVED）状态。事实上，在S3C6400处理器用户手册中也有一张启动设备描述表。

如图2.16所示，当XSELNAND管脚为1，OM[4:0]为0000X、0001X、0010X或0011X时，启动设备为NAND Flash。S3C6410 处理器向下兼容S3C6400 处理器，兼容的内容应该包括启动模式。实际上S3C6410确实支持NandFlash启动，S3C6410启动设备描述表中却没有显示的原因，可能是相对NAND Flash 启动方式，三星更加推崇IROM启动。

XSELNAND	OM[4:0]	Boot Device	Function	Clock Source
1	0000X	NAND	Small page, AddrCycle=3	XXTIpll if OM[0] is 0. XEXTCLK if OM[0] is 1.
1	0001X		Small Page, AddrCycle=4	
1	0010X		Large Page, AddrCycle=4	
1	0011X		Large Page, AddrCycle=5	
X	0100X	SROM(8-bit)	-	
X	0101X	SROM(16-bit)	-	
0	0110X	OneNAND	Don't use NAND Device.	
X	0111X	MODEM	Don't use Xm0CSn2 for SROMC.	
1: NAND 0: OneNAND	1111X	Internal ROM	-	

图2.16 S3C6400启动设备描述表

图2.17 所示是NAND Flash 启动原理框图，当设置为NAND Flash 启动时，NAND Flash 控制器能够将NAND Flash 的8KB 数据自动导入至SRAM。NAND Flash 启动支持地址周期为3 的小页、地址周期为4的小页、地址周期为4的大页和地址周期为5的大页。小页指的是每页512字节，大页指的是每页2KB。单板使用的K9GAG08U0D芯片，每页数据区的大小为4KB。XSELNAND管脚设置为1，OM[4:0]设置为0011X。启动时NAND Flash 控制器只能将前4 页、每页的前2KB数据导入到SRAM，这一点对移植U-Boot支持NAND Flash 启动至关重要。

2．硬件ECC校验

虽然NAND Flash 已经具有较高的可靠性，但受其制作工艺和使用环境的限制，在使用过程中仍可能出现位翻转的现象。位翻转，顾名

思义就是存储的数据有比特位出错。这种错误出现的概率较低，对于普通的用户数据，例如图片、音频等，出错产生的影响不是很大。但在一些关键的场合，例如启动代码、内核代码等，若错误发生在某个重要的数据上，导致的后果往往是灾难性的，而且一旦出错，往往难以查找和修改。为了嵌入式系统数据存储整体的稳定，需要为所设计的存储系统添加必要的纠错和检测，常用的有奇偶校验、循环冗余校验和ECC（Error Checking and Correcting，错误检查和纠正）校验等。

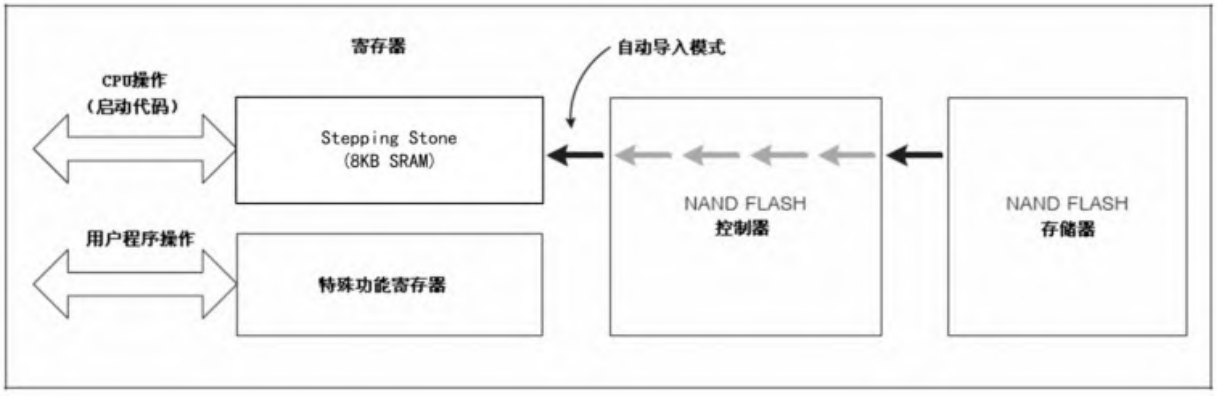


图2.17 NANDFlash启动原理框图

在NAND Flash 处理中，一般使用ECC校验。ECC能纠正单比特错误和检测双比特错误。当数据被写入时，产生的 ECC 码同时被保存下来。当重新读回刚才存储的数据时，用保存下来的ECC 码和读数据时产生的 ECC 码做比较。如果两个代码不相同，他们则会被解码，以确定数据中的哪一位是不正确的。传统的ECC校验算法一般采用软件实现，这样设计虽然复杂度低、通用性高，但每次读写数据均需要进行ECC校验操作，大量的校验操作带来庞大的软件开销，直接导致存储器的读写性能下降。

S3C6410支持硬件ECC校验，在硬件水平上集成了校验模块。1位ECC用于SLC NAND Flash， 4 位、8 位ECC用于MLC NAND Flash。在介绍S3C6410 的硬件ECC校验之前，首先查看一下NAND Flash 控制器配

置寄存器NFCONF，NFCONF 寄存器的内存映射地址为0x7020_0000，各位段如图2.18所示。

NFCONF	Bit	Description	Initial State
Reserved			
MLCClkCtrl	[30]	Clock control for 4-bit ECC & 8-bit ECC engine.(Hidden Spec.) 0: Recommended when system clock is more than 66MHz. 1: Recommended when system clock is less than 66MHz	0
Reserved	[29:26]	Reserved	0000
MsgLength	[25]	Message (Data) length for 4/8 bit ECC 0: 512-byte 1: 24-byte	0
ECCType	[24:23]	This bit indicates what kind of ECC should be used. 00: 1-bit ECC 10: 4-bit ECC 01 : 8-bit ECC Note. Don't confuse the value of 4-bit ECC and 8-bit ECC.	H/W Set (CfgBootEcc)
Reserved	[22:15]	Reserved	000000000
TACLS	[14:12]	CLE & ALE duration setting value (0~7) Duration = HCLK x TACLS	001
Reserved	[11]	Reserved	0
TWRPH0	[10:8]	TWRPH0 duration setting value (0~7) Duration = HCLK x (TWRPH0 + 1)	000
Reserved	[7]	Reserved	0
TWRPH1	[6:4]	TWRPH1 duration setting value (0~7) Duration = HCLK x (TWRPH1 + 1)	000
Reserved			
Reserved	[2]	Reserved. Must be written 1	1
Reserved			
Reserved	[0]	Reserved. Must be written 0.	0

图2.18 NFCONF各位段描述表

以8位硬件ECC为例，它分为ECC校验码编码和译码两个阶段。在使用8位硬件ECC校验模式之前必须配置相关的寄存器：NFCONF寄存器MsgLength位段设置为0（每次操作的数据单位为512字节），ECCType位段设置为01（8位ECC）。当MsgLength位段设置为1时，操作数据的单位为24字节，它的原理跟操作数据的单位为512字节相同，本文以512字节的操作单位为例子。

在每次编码、译码（通过操作控制器的寄存器读、写NAND Flash）之前，都必须配置NFCONT寄存器的一些位段，控制ECC模块的工作状态：设置MainECCLock位段为0，使ECC模块处于未锁定状态；设置InitMECC位段为1，初始化ECC模块。注意初始化ECC模块时，必须处于未锁定状态，即MainECCLock为0。操作完成后设置MainECCLock位段为1，使其处于锁定状态。

编码阶段是往NAND Flash 写入数据后，ECC模块产生校验码的阶段。写入数据时，首先配置ECC模块，再发送512字节的数据，发送完成后ECC发生器开始工作。检测NFSTAT寄存器的ECCEncDone位段可以判定ECC模块工作状态，如果它为1表示编码完成。此时，ECC校验码编码完成、并将结果输出到NF8MECC0、NFMECC1、NF8MECC2和NF8MECC3这4个寄存寄存器，最后我们锁定ECC模块。如果使用的NAND Flash 每页数据区的大小刚好为512字节，直接将这4 个寄存器中的数据写入NAND Flash 的OOB 区，但是如果使用的NAND Flash 每页数据区大于512字节，一般先将这4个寄存器中的数据保存到内存的缓冲区中，再重复上述操作，等到写完一页数据后，再把所缓冲区中的数据一起写入NAND Flash 的OOB 区。对于编码的规则和寄存器输出数据的含义，是NAND Flash 控制器设计者关心的问题，对于普通用户没有必要去深入研究。

译码阶段是读取NAND Flash 中的数据后，再发送编码阶段产生的校验码，NAND Flash 控制器模块译码查找错误的阶段。读取数据时，首先配置ECC模块，再读取发送512字节的数据，然后发送OOB区对应这512字节的ECC校验码（编码阶段写入的，需要把它从OOB区读取出来），发送完成后锁定ECC模块。一旦ECC校验模块接收到校验码，就开始检测是否存在错误。检测NFSTAT寄存器的ECCDecDone位段可以判定ECC模块的工作状态，如果它为1表示译码完成，并将错误信息输出到 NF8ECCERR0、NF8ECCERR01、NF8ECCERR02 和 NFMLC8BITPT0、

NFMLC8BITPT1 寄存器中。编写驱动时，我们可以根据这些信息就能判定NAND Flash 是否有错，如果有错且错误类型属于支持范围内就修改错误。如果使用的 NAND Flash 每页数据区大于 512字节，重复译码阶段的过程，直到读完一页数据。详见S3C6410处理器用户手册和后文的驱动代码。

由此，我们还可以得出结论，S3C6410处理器的NAND Flash 控制器支持每页数据区大小为512字节（或者24字节）整数倍的NAND Flash。编写不同型号NAND Flash 的驱动程序时，仅仅是单位操作重复的次数不一致，本质上没有任何区别。

2.5.3 NAND Flash驱动移植

在Linux系统中，为了建立Flash对上层的统一接口，屏蔽底层硬件的差异，利用MTD（Memory Technology Device，内存技术设备）访问memory设备（比如NAND flash、NOR Flash）。MTD 的主要目的是为了使新的memory设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。Linux系统中MTD可以分为4层：设备节点、MTD设备层、MTD原始设备层和Flash硬件驱动。U-Boot对NAND Flash 驱动的支持，与Linux内核中MTD的Flash硬件驱动和MTD原始设备层类似，所有源代码在drivers/mtd子目录下。

Flash硬件驱动层：Flash硬件驱动层对应的是不同硬件的驱动程序，它负责驱动具体的硬件。例如：NAND Flash 的驱动程序在/drivers/mtd/nand目录下。MTD原始设备层：在原始设备层中，各种内存设备抽象化为原始设备，原始设备层的实现依赖于Flash硬件驱动层。

1．重要的数据结构

MTD 使用struct mtd_info 结构体描述MTD 原始设备，该结构体包含了MTD 原始设备操作函数的指针和各种基本数据。结构体在include/mtd/mtd.h文件中定义如下：

```
struct mtd_info {
  u_char type;/* 设备的类型 */
  u_int32_t flags; /* 设备的性能描述 */
  uint64_t size;/* 设备的容量大小 */
  u_int32_t erasesize; /* 擦除单位的字节数 */
  u_int32_t writesize; /* 写操作单位的字节数 */
  u_int32_t oobsize; /* 每页空闲区的字节数*/
  u_int32_t oobavail; /* 文件系统可用的字节数 */
  const char *name; /* 设备名称 */
  struct nand_ecclayout *ecclayout; /*NAND Flash ECC 布局结构体*/
...
  /* 一些操作函数指针 */
  int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
  int (*point) (struct mtd_info *mtd, loff_t from, size_t len,
    size_t *retlen, void **virt, phys_addr_t *phys);
  void (*unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
  int (*read) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
  int (*write) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, constu_char *buf);
```

```
    int (*panic_write) (struct mtd_info *mtd, loff_t to,
size_t len, size_t *retlen, const u_char *buf);
    int (*read_oob) (struct mtd_info *mtd, loff_t from,
      struct mtd_oob_ops *ops);
    int (*write_oob) (struct mtd_info *mtd, loff_t to,
      struct mtd_oob_ops *ops);
    ....
    int (*block_isbad) (struct mtd_info *mtd, loff_t
ofs);
    int (*block_markbad) (struct mtd_info *mtd, loff_t
ofs);
    struct mtd_ecc_stats ecc_stats;
    int subpage_sft;
    void *priv; /* 私有数据指针 */
    struct module *owner;
    int usecount;
    int (*get_device) (struct mtd_info *mtd);
    void (*put_device) (struct mtd_info *mtd);
};
```

struct mtd_info 结构体的变量在drivers/mtd/nand/nand.c文件中定义如下：

```
nand_info_t nand_info[CONFIG_SYS_MAX_NAND_DEVICE];
```

在 include/nand.h 中，为 struct mtd_info 定义一个别名 typedef struct mtd_info nand_info_t。
CONFIG_SYS_MAX_NAND_DEVICE 表示单板中配备NAND Flash 的物理数目，通常情况为1。在整个MTD构架中，mtd_info结构在参数的传递中起着至关重要的作用。

mtd_info 的 ecclayout 指针变量用于描述 OOB 区的布局，它的数据类型为 structnand_ecclayout，结构体在include/nand.h文件中定义如下：

```
struct nand_ecclayout {
  uint32_t eccbytes;
  uint32_t eccpos[128];
  uint32_t oobavail;
  struct nand_oobfree oobfree[MTD_MAX_OOBFREE_ENTRIES];
};
struct nand_oobfree {
  uint32_t offset;
  uint32_t length;
};
```

nand_ecclayout的eccbytes字段表示ECC校验码的字节数。eccpos字段是一个数组，用于确定存放ECC校验码的位置。oobavail字段表示可用OOB区的字节数，用户不必对它赋值，它能够通过nand_ecclayout的其他3个字段计算得到。oobfree字段用于显示可用OOB区域的相对偏移地址和长度，用作存储文件系统的信息。通常情况下，OOB 区的前两个字节为坏块标记。以S3C6410 硬件ECC为例，当写NAND Flash 时，NAND Flash 控制器自动生成ECC校验码，我们把它写入OOB 区中eccpos指示的位置。当读NAND Flash 时，我们从OOB 区中eccpos 指示的位置把ECC校验码读出，传递给NAND Flash 控制器，NAND Flash 控制器进行出错校验，并提示出错信息。根据出错信息，就能改正 NAND Flash 出现的数据错误。

另外一个重要的结构体是struct nand_chip，它的定义在 include/linux/mtd/nand.h 文件中。

```
struct nand_chip {
```

```
    void __iomem *IO_ADDR_R;
    void __iomem *IO_ADDR_W;
    uint8_t (*read_byte)(struct mtd_info *mtd);
    u16 (*read_word)(struct mtd_info *mtd);
    void (*write_buf)(struct mtd_info *mtd, const uint8_t
*buf, int len);
    void (*read_buf)(struct mtd_info *mtd, uint8_t *buf,
int len);
    int (*verify_buf)(struct mtd_info *mtd, const uint8_t
*buf, int len);
    void (*select_chip)(struct mtd_info *mtd, int chip);
    int (*block_bad)(struct mtd_info *mtd, loff_t ofs,
int getchip);
    int (*block_markbad)(struct mtd_info *mtd, loff_t
ofs);
    void (*cmd_ctrl)(struct mtd_info *mtd, int dat,
unsigned int ctrl);
    int (*init_size)(struct mtd_info *mtd, struct
nand_chip *this,
      u8 *id_data);
    int (*dev_ready)(struct mtd_info *mtd);
    void (*cmdfunc)(struct mtd_info *mtd, unsigned
command, int column,
      int page_addr);
    int(*waitfunc)(struct mtd_info *mtd, struct nand_chip
*this);
    void (*erase_cmd)(struct mtd_info *mtd, int page);
```

```
    int (*scan_bbt)(struct mtd_info *mtd);
    int (*errstat)(struct mtd_info *mtd, struct nand_chip
*this, int state, int status, int page);
    int (*write_page)(struct mtd_info *mtd, struct
nand_chip *chip,
      const uint8_t *buf, int page, int cached, int raw);
    int chip_delay;
    unsigned int options;
    int page_shift;
    int phys_erase_shift;
    int bbt_erase_shift;
    int chip_shift;
    int numchips;
    uint64_t chipsize;
    int pagemask;
    int pagebuf;
    int subpagesize;
    uint8_t cellinfo;
    int badblockpos;
    int badblockbits;
    int onfi_version;
  #ifdef CONFIG_SYS_NAND_ONFI_DETECTION
    struct nand_onfi_params onfi_params;
  #endif
    int state;
    uint8_t *oob_poi;
    struct nand_hw_control *controller;
```

```
    struct nand_ecclayout *ecclayout;
    struct nand_ecc_ctrl ecc;
    struct nand_buffers *buffers;
    struct nand_hw_control hwcontrol;
    struct mtd_oob_ops ops;
    uint8_t *bbt;
    struct nand_bbt_descr *bbt_td;
    struct nand_bbt_descr *bbt_md;
    struct nand_bbt_descr *badblock_pattern;
    void *priv;
};
```

在初始化时，mtd_info的priv字段指向nand_chip结构体。

2. 初始化分析

NAND Flash设备的初始化过程分为NAND Flash控制器初始化和NAND Flash结构体成员初始化两个过程。NAND Flash 控制器初始化程序是 lowlevel_init.S 文件的 nand_asm_init 函数。函数的功能是设置S3C6410 处理器NFCONF寄存器的3 个位段：TACLS[14:12]、TWRPH0[10:8] 和TWRPH1 [6:4]，这3 个位段用于配置NAND Flash 的时序。初始化时把它们默认为最大值。

如图2.19所示是TACLS位段为1、TWRPH0位段为0、TWRPH1位段为0时，CLE和ALE的时序图。

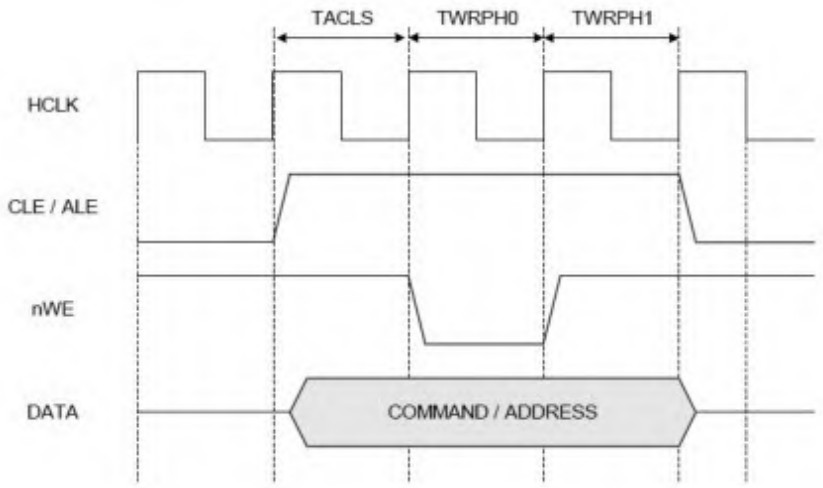


图2.19 CLE和ALE的时序图

如图2.20所示是TWRPH0位段为0、TWRPH1位段为0时，WE#和RE#的时序图。

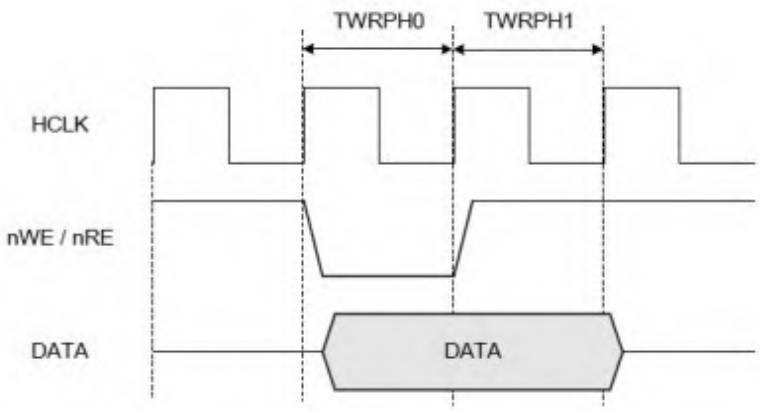


图2.20 WE#和RE#的时序图

在S3C6410处理器用户参考手册中，并没有详细说明这3个位段的的具体含义，但通过上面两个时序图（用户参考手册中提供的）可以看出，TACLS表示CLE/ALE有效到WE#有效之间的持续时间t_1，TWRPH0表示WE#有效的持续时间t_2，TWRPH1为WE#无效到CLE/ALE无效之间的持续时间t_3。这些时间都是以HCLK的周期为单位的：t_1等于HCLK周期乘以TACLS，t_2等于HCLK周期乘以（TWRPH0+1），t_3等于HCLK周期乘以（TWRPH1+1）。

查阅K9GAG08U0D的数据手册，我们可以发现手册中有很多时序图。

如图2.21所示，我们可以发现图中t_{CLS} 或t_{CS} 与t_1 相对应，t_{CLH} 与t_3 相对应，t_{wp} 与t_2 相对应。K9GAG08U0D 用户手册给出了t_{CLS}（与t_{CS} 大小相等）、t_{CLH} 和t_{wp} 的最小时间，分别为20ns、5ns和15ns。t_1 不能小于20ns，t_2 不能小于5ns，t_3 不能小于15ns。在时钟初始化时，HCLK被设置为133MHz，周期大约为7.52ns。

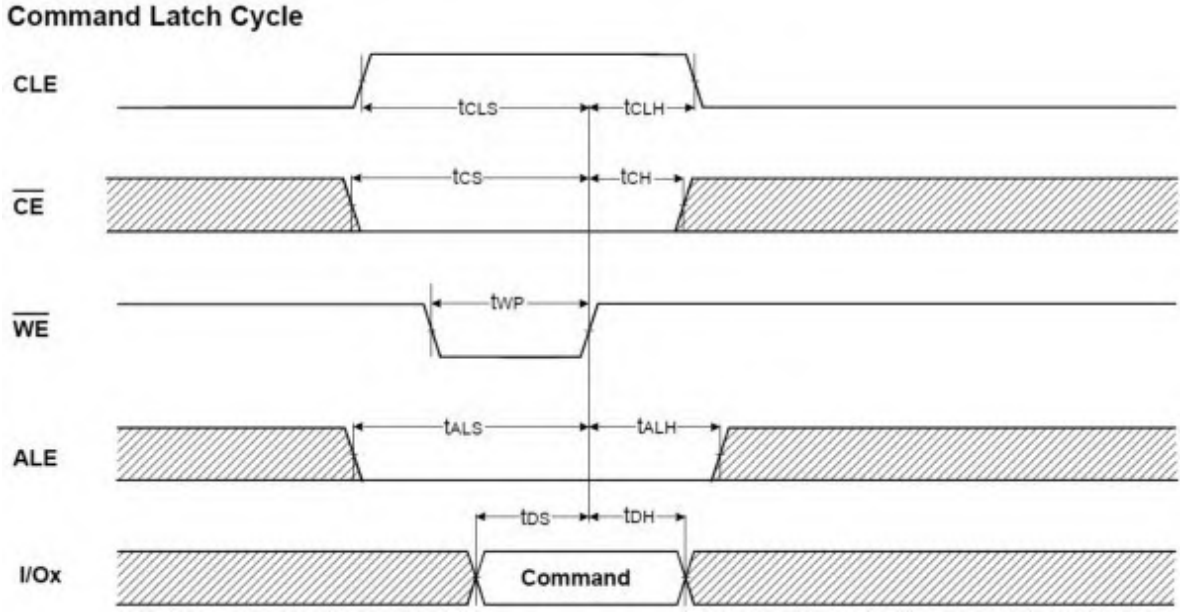


图2.21 命令锁存时序图

为了提高 NAND Flash 存取的效率，根据上述条件计算出合适的TACLS、TWRPH0 和TWRPH1的值。计算结果为TACLS等于3，TWRPH0等于0,，TWRPH1等于1。nand_asm_init()修改如下：

```
270 nand_asm_init:
271     ldr  r0, =ELFIN_NAND_BASE
272     ldr  r1, [r0, #NFCONF_OFFSET]
273 #     orr  r1, r1, #0x70  /*changed by liqiang */
274 #     orr  r1, r1, #0x7700
275     orr  r1, r1, #10  /*TWRPH1 = 1*/
```

```
276     orr  r1, r1, #0x300  /*TWRPH0 = 0, TACLS = 3*/
277     str  r1, [r0, #NFCONF_OFFSET]
278
279     ldr  r1, [r0, #NFCONT_OFFSET]
280     orr  r1, r1, #0x07
281     str  r1, [r0, #NFCONT_OFFSET]
282
283     mov  pc, lr
```

与其他外围设备的初始化一样，NAND Flash 初始化函数在board_init_r函数中被调用。函数

间具体的调用关系和程序所在的文件如下所示：

```
board_init_r()  arch/arm/lib/board.c
  nand_init()    drivers/mtd/nand/nand.c
    nand_init_chip()        drivers/mtd/nand/nand.c
      board_nand_init()       drivers/mtd/nand/xxx.c
      nand_scan()         drivers/mtd/nand/nand_base.c
        nand_scan_ident()  drivers/mtd/nand/nand_base.c
        nand_scan_tail()  drivers/mtd/nand/nand_base.c
      nand_register()    drivers/mtd/nand/nand.c
```

nand_init 函数用一个 for 循环语句调用 nand_init_chip 函数，循环调用的次数由宏定义CONFIG_SYS_MAX_NAND_DEVICE 决定。nand_init_chip 函数开始正式初始化NAND flash，分board_nand_init函数和nand_scan函数两个步骤进行。board_nand_init函数在MTD源码中没有定义，在include/nand.h头文件中只有一个如下的函数原型声明。

```
extern int board_nand_init(struct nand_chip *nand);
```

board_nand_init函数的主要功能是完成一些与底层硬件操作相关的初始化，属于MTD的Flash硬件驱动层。我们知道，底层硬件操作大多是对处理器寄存器的操作，与具体NAND Flash 芯片型号及主机控制器相关，MTD 不可能提供通用函数来完成这些操作。因此，用户必须自行编写board_nand_init函数来完成这些操作的初始化。编写board_nand_init函数是移植MTD驱动的核心工作。nand_scan()中完成的工作多与通用的接口函数有关，它完成了核心结构体如 mtd_info、nand_chip中绝大多数字段的赋值，处在mtd原始设备层，在移植时很少需要修改。由于nand_scan函数以及延伸调用比较复杂，本文不再赘述，感兴趣的读者可以阅读源码，自行分析。

3. 修改配置文件

NAND Flash 相关配置位于include/configs/smdk6410.h 文件中。[U2]

```
/* NAND chip page size      */
#define CONFIG_SYS_NAND_PAGE_SIZE    2048
/* NAND chip block size    */
#define CONFIG_SYS_NAND_BLOCK_SIZE  (128 * 1024)
/* NAND chip page per block count  */
#define CONFIG_SYS_NAND_PAGE_COUNT  64
/* Location of the bad-block label */
#define CONFIG_SYS_NAND_BAD_BLOCK_POS  0
/* Extra address cycle for > 128MiB */
#define CONFIG_SYS_NAND_5_ADDR_CYCLE
```

对比图2.13单板使用NAND Flash 的页大小为4096字节，块的大小为128*4096 字节，每块包含128页。修改如下：

```
/* NAND chip page size      */
/*changed by liqiang 2013/04/16*/
```

```
#define CONFIG_SYS_NAND_PAGE_SIZE    4096
/* NAND chip block size    */
#define CONFIG_SYS_NAND_BLOCK_SIZE  (128 * 4096)
/* NAND chip page per block count  */
#define CONFIG_SYS_NAND_PAGE_COUNT  128
/* Location of the bad-block label */
#define CONFIG_SYS_NAND_BAD_BLOCK_POS  0
/* Extra address cycle for > 128MiB */
#define CONFIG_SYS_NAND_5_ADDR_CYCLE
```

4. 芯片类型

在drivers/mtd/nand/nand_ids.c文件里定义了两个只读属性的全局结构体数组：nand_flash_ids[]和nand_manuf_ids[]。nand_flash_ids是芯片类型的结构体数组，结构体的原型如下。

```
struct nand_flash_dev {
    char *name;
    int id;
    unsigned long pagesize;
    unsigned long chipsize;
    unsigned long erasesize;
    unsigned long options;
};
```

nand_flash_dev的name 字段用于描述NAND Flash 的类型，id 字段是设备号，pagesize 字段是NAND Flash 每页数据区的大小等。nand_manuf_ids 是芯片生产厂商信息结构体数组，结构体的原型如下。

```
struct nand_manufacturers {
    int id;
```

```
    char *name;
};
```

nand_manufacturers的id字段是厂商的序号，name字段为厂商的名称。

```
113    /* 16 Gigabit */
114    {"NAND 2GiB 1,8V 8-bit",    0xA5, 0, 2048, 0, LP_OPTIONS},
115    {"NAND 2GiB 3,3V 8-bit",    0xD5, 0, 2048, 0, LP_OPTIONS},
0xD5, 4096, 2048, 4096*128, LP_OPTIONS},116    {"NAND 2GiB 3,3V 8-bit",/*added by liqiang*/
117    {"NAND 2GiB 1,8V 16-bit",    0xB5, 0, 2048, 0, LP_OPTIONS16},
118    {"NAND 2GiB 3,3V 16-bit",    0xC5, 0, 2048, 0, LP_OPTIONS16},
119
```

nand_scan()被执行时，驱动程序会通过NAND Flash 的id命令，读取芯片的信息，用读取到的信息完善几个重要的结构体。

5. board_nand_init()实现

U-Boot中，已经为S3C6400处理器实现了4位ECC硬件校验，实现代码在drivers/mtd/nand/s3c64xx.c文件中。S3C6410处理器支持8位硬件ECC校验，驱动程序根据官方提供的实例代码编写。读者可到网盘中下载，文件名为 s3c64xx.c，覆盖 U-Boot 的文件即可。

s3c64xx.c代码量较大，不能也没有必要逐一分析，实现的函数都是用于填充nand_chip结构体的。下面是board_nand_init()的程序片段：

```
int board_nand_init(struct nand_chip *nand)
```

```
  {
    static int chip_n;
    if (chip_n >= MAX_CHIPS)
      return -ENODEV;
    NFCONT_REG = (NFCONT_REG & ~NFCONT_WP) |
NFCONT_ENABLE | 0x6;
    nand->IO_ADDR_R    = (void __iomem *)NFDATA;
    nand->IO_ADDR_W    = (void __iomem *)NFDATA;
    nand->cmd_ctrl    = s3c_nand_hwcontrol;
    nand->dev_ready    = s3c_nand_device_ready;
    nand->select_chip = s3c_nand_select_chip;
    nand->options  = 0;
  #ifdef CONFIG_NAND_SPL
    nand->read_byte    = nand_read_byte;
    nand->write_buf    = nand_write_buf;
    nand->read_buf    = nand_read_buf;
  #endif
  #ifdef CONFIG_SYS_S3C_NAND_HWECC
    nand->ecc.hwctl    =
s3c_nand_enable_hwecc_8bit;  //liqiang
    nand->ecc.calculate  =
s3c_nand_calculate_ecc_8bit;  //liqiang
    nand->ecc.correct =
s3c_nand_correct_data_8bit;  //liqiang
    nand->ecc.read_page  = s3c_nand_read_page_8bit;
    nand->ecc.write_page  = s3c_nand_write_page_8bit;
    nand->ecc.mode    = NAND_ECC_HW;
```

```
    nand->ecc.size    = 512;  //liqiang
    nand->ecc.bytes    = 13;  //liqiang
    nand->ecc.layout  = &s3c_nand_oob_mlc_128_8bit;
  #else
    nand->ecc.mode    = NAND_ECC_SOFT;
  #endif /* ! CONFIG_SYS_S3C_NAND_HWECC */
    nand->priv    = nand_cs + chip_n++;
    return 0;
  }
```

下面简要概述board_nand_init()的操作流程：

（1）硬件初始化，设置NFCONT_REG寄存器。

（2）R/W 数据寄存器访问接口初始化，nand->IO_ADDR_R 和nand->IO_ADDR_W 均为NFDATA。

（3）控制函数 nand->cmd_ctrl、R/B 状态读取函数 nand->dev_ready 初始化，它们没有在nand_scan()初始化， MTD 也没有提供通用函数，我们必须自行编写。实现的函数分别s3c_nand_hwcontrol()和s3c_nand_device_ready()。

（4）片选函数nand->select_chip初始化，这个函数如果在board_nand_init()中没有初始化，它将会在nand_scan()中赋予一个默认的函数。但是，为了提供完整的功能，通常由用户自行编写，并在board_nand_init()中初始化。

（5）功能参数nand->options 初始化，当它为0 时，表示NAND Flash 位宽为8。

（6）nand->ecc结构体初始化，当选择硬件ECC校验时，所有的功能函数都必须根据处理器的特点编写。当选择软件ECC校验时，MTD提供通用的ECC功能函数。

6. NAND Flash 命令

与NAND Flash 相关的命令有：

```
nand - NAND sub-system
Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
  read/write 'size' bytes starting at offset 'off'
  to/from memory address 'addr', skipping bad blocks.
nand read.raw - addr off|partition [count]
nand write.raw - addr off|partition [count]
  Use read.raw/write.raw to avoid ECC and access the flash as-is.
nand erase[.spread] [clean] off size - erase 'size' bytes from offset 'off'
  With '.spread', erase enough for given file size, otherwise,
  'size' includes skipped bad blocks.
nand erase.part [clean] partition - erase entire mtd partition'
nand erase.chip [clean] - erase entire chip'
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub [-y] off size | scrub.part partition | scrub.chip
  really clean NAND erasing bad blocks (UNSAFE)
```

nand markbad off [...] - mark bad block(s) at offset (UNSAFE)

nand biterr off - make a bit error at offset (UNSAFE)

命令的实现代码在common/cmd_nand.c文件中。其中，nand info命令用于输出NAND Flash的信息。

```
lq@u-boot #nand info
Device 0: nand0, sector size 512 KiB
    Page size  4096 b
    OOB size    218 b
    Erase size 524288 b
lq@u-boot #
```

其他命令的用法到使用的时候再介绍。我们需要添加一个烧写 u-boot-nand.bin 的命令，使 u-boot-nand.bin 在K9GAG08U0D NAND Flash的芯片的布局如图2.22所示。打开 common/cmd_nand.c ，在文件的第 666 行开始（即#endif……#ifdef CONFIG_CMD_NAND_YAFFS 之间）添加以下代码即可。

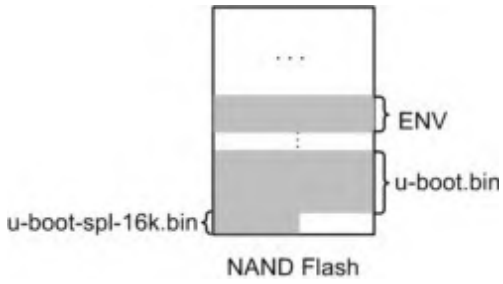


图2.22 u-boot-nand.bin布局

```
} else if (!read && s != NULL && (!strcmp(s, ".uboot")) && nand->writesize == 4096) {rwsize=4096;
    nand_write(nand, off, &rwsize, (u_char *)addr);
    off+=4096;
    addr+=2048;
```

```
nand_write(nand, off, &rwsize, (u_char *)addr);
off+=4096;
addr+=2048;
nand_write(nand, off, &rwsize, (u_char *)addr);
off+=4096;
addr+=2048;
nand_write(nand, off, &rwsize, (u_char *)addr);
off+=4096;
addr+=2048;
rwsize= CONFIG_SYS_NAND_U_BOOT_SIZE - 8*1024;
ret = nand_write(nand, off, &rwsize, (u_char *)addr);
```

2.5.4 nand spl启动原理

u-boot-nand.bin文件是由两个独立的文件连接而成的，u-boot-spl-16k.bin文件和u-boot.bin文件。u-boot.bin文件的生成过程在U-Boot启动分析的时候已经详细地介绍过了，在这个小节里，我们通过追踪u-boot-spl-16k.bin文件的生成过程来理清U-Boot的nand_spl启动的实现原理。在顶层Makefile文件中，有如下内容。

```
602 nand_spl:    $(TIMESTAMP_FILE) $(VERSION_FILE) depend
603         $(MAKE) -C nand_spl/board/$(BOARDDIR) all
604
605 $(obj)u-boot-nand.bin:  nand_spl $(obj)u-boot.bin
606         cat $(obj)nand_spl/u-boot-spl-16k.bin $(obj)u-boot.bin >
        $(obj)u-boot-nand.bin
```

第605行，u-boot-nand.bin的生成依赖于nand_spl和u-boot.bin，而nand_spl的动作是执行nand_spl/board/$（BOARDDIR）目录下的Makefile 文件。在移植的前期，我们已经针对S3C6410单板建立了一个nand_spl目录，即nand_spl/board/samsung/smdk6410。当这两个依赖满足的时候，通过cat命令将nand_spl/u-boot-spl-16k.bin和u-boot.bin中的数据内容，按如图2.23所示的顺序，输出至u-boot-nand.bin文件。可以想象一下，NAND Flash 启动时，控制器将位于前端的u-boot-spl-16k.bin拷贝至SRAM中，然后执行u-boot-spl-16k.bin。u-boot-spl-16k.bin的功能与SD卡启动的BL0类似，最主要是初始化DRAM，将更加完善的u-boot.bin拷贝到它的运行地址，然后跳到该地址处执行。

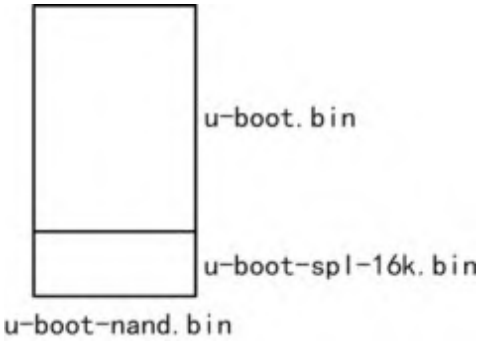


图2.23 u-boot-nand.bin构成

1. Makefile文件

nand_spl/board/samsung/smdk6410目录下的Makefile文件用于指导nand_spl的编译，它是独立于u-boot.bin的，打开该文件。

27 CONFIG_NAND_SPL = y

编译时，增加CONFIG_NAND_SPL宏定义。

29 include $(TOPDIR)/config.mk

包含当前目录下的config.mk文件，config.mk文件的核心内容如下：

-> PAD_TO = CONFIG_SYS_TEXT_BASE + 4096

PAD_TO := $(shell expr $$[$(CONFIG_SYS_TEXT_BASE) + 4096])

它的主要工作是定义了一个PAD_TO变量，大小为CONFIG_SYS_TEXT_BASE + 4096。注意，编译nand_spl时CONFIG_SYS_TEXT_BASE为0。S3C6400处理器的SRAM只有大约4KB（4096字节），而S3C6410处理器的SRAM大小约为8KB，因此将4096修改为8192。

40 SOBJS = start.o cpu_init.o lowlevel_init.o crt0.o

41 COBJS = nand_boot.o nand_ecc.o s3c64xx.o smdk6410_nand_spl.o nand_base.o

nand_spl工程所有源码的中间文件。

```
52 $(nandobj)u-boot-spl-16k.bin: $(nandobj)u-boot-spl
53    $(OBJCOPY) ${OBJCFLAGS} --pad-to=$(PAD_TO) -O binary $< $@
54
55 $(nandobj)u-boot-spl.bin:    $(nandobj)u-boot-spl
56    $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
57
58 $(nandobj)u-boot-spl:  $(OBJS) $(nandobj)u-boot.lds
59    cd $(LNDIR) && $(LD) $(LDFLAGS) $(__OBJS) \
60        -Map $(nandobj)u-boot-spl.map \
61        -o $(nandobj)u-boot-spl
```

从上面的Makefile片段中，已知u-boot-spl-16k.bin的编译依赖于u-boot-spl，而u-boot-spl是所有中间文件和链接脚本作用的结果。第53行和第55行，都是当u-boot-spl依赖满足时，从u-boot-spl文件中提取出处理器可执行二进制文件。不同之处是：第53行增加了--pad-to=$（PAD_TO）参数，该参数的作用是在文件末尾填充数据直到

文件的大小为PAD_TO，如图2.24所示。事实上，这样做就可以得到大小固定的二进制文件，至于填充的内容我们并不需要关心。

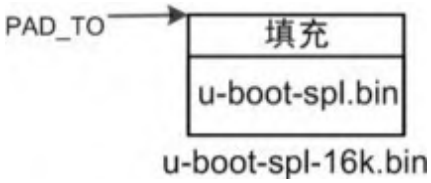


图2.24 u-boot-spl-16k.bin

```
69 $(obj)start.S:
  70    @rm -f $@
  71    @ln -s $(TOPDIR)/arch/arm/cpu/arm1176/start.S
$@
  72 $(obj)crt0.S:
  73    @rm -f $@
  74    @ln -s $(TOPDIR)/arch/arm/lib/crt0.S $@
 ....
 104 $(obj)nand_base.c:
 105    @rm -f $@
 106    @ln -s $(TOPDIR)/drivers/mtd/nand/nand_base.c $@
```

源码文件的相对地址。第72行是我们在建立最小模板时添加的，添加的原因是因为start.S文件调用了crt0.S文件中的函数。进入build/nand_spl目录（如果没有将编译生成的文件和源码文件分开存放，直接进入nand_spl目录），输入ls命令查看当前目录下的文件的信息。

```
liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1-
sdboot/build/nand_spl$ ls  -l
total 124
drwxrwxr-x 3 liqiang liqiang  4096 Mar 27 22:59 board
```

-rwxrwxr-x 1 liqiang liqiang 109898 Apr 17 20:44 u-boot-spl

-rwxrwxr-x 1 liqiang liqiang 4200 Apr 17 20:44 u-boot-spl-16k.bin

-rwxrwxr-x 1 liqiang liqiang 4200 Apr 17 20:44 u-boot-spl.bin

-rw-rw-r-- 1 liqiang liqiang 19537 Apr 17 20:44 u-boot-spl.map

-rw-rw-r-- 1 liqiang liqiang 922 Mar 27 22:59 u-boot.lds

发现u-boot-spl.bin和u-boot-spl-16k.bin都是4200个字节，大小相同，并没有和我们分析时的预期结果一致。事实上，这也是源码的一个bug。config.mk中的以下shell语句有误。

PAD_TO := $(shell expr $$[$(CONFIG_SYS_TEXT_BASE) + 8192])

修改为：

PAD_TO := $(shell expr $(CONFIG_SYS_TEXT_BASE) + 8192)

再次查看当前目录的文件信息。

drwxrwxr-x 3 liqiang liqiang 4096 Apr 21 10:52 ./

drwxrwxr-x 18 liqiang liqiang 4096 Apr 21 10:52 ../

drwxrwxr-x 3 liqiang liqiang 4096 Apr 21 10:49 board/

-rwxrwxr-x 1 liqiang liqiang 109898 Apr 21 10:52 u-boot-spl*

-rwxrwxr-x 1 liqiang liqiang 8192 Apr 21 10:52 u-boot-spl-16k.bin*

-rwxrwxr-x 1 liqiang liqiang 4200 Apr 21 10:52 u-boot-spl.bin*

-rw-rw-r-- 1 liqiang liqiang 19537 Apr 21 10:52 u-boot-spl.map

-rw-rw-r-- 1 liqiang liqiang 922 Apr 21 10:50 u-boot.lds、

u-boot-spl-16k.bin 的大小为 8KB，为了验证 u-boot-nand.bin 的组成，在 build 目录下查看u-boot-nand.bin和u-boot.bin的文件信息。

liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1-sdboot/build$ ls -l u-boot.binu-boot-nand.bin

-rw-rw-r-- 1 liqiang liqiang 279076 Apr 21 10:52 u-boot-nand.bin

-rw-rw-r-- 1 liqiang liqiang 270884 Apr 21 10:52 u-boot.bin

u-boot-nand.bin的文件大小为279076，u-boot.bin的文件大小为270884，二者相差8192，在大小上也符合u-boot-nand.bin由u-boot-spl-16k.bin和u-boot.bin链接而成。

2. u-boot-spl-16k.bin的组织

在Makefile文件中，我们已经知道u-boot-spl-16k.bin所有的源码文件位置，这些源码文件编译生成的中间目标文件在nand_spl/board/samsung/smdk6410/u-boot.lds链接脚本的指示下，被链接器组织在一起。u-boot.lds内容如下：

```
30 SECTIONS
31 {
32     . = 0x00000000;
33
34     . = ALIGN(4);
35     .text   :
```

```
36    {
37    start.o    (.text)
38    cpu_init.o  (.text)
39    nand_boot.o  (.text)
40
41      *(.text)
42    }
43
44    . = ALIGN(4);
45    .rodata : { *
(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }
46
47    . = ALIGN(4);
48    .data : { *(.data) }
49
50    . = ALIGN(4);
51    .got : { *(.got) }
52
53
54    . = ALIGN(4);
55    .u_boot_list : {
56    #include <u-boot.lst>
57    }
58
59    . = ALIGN(4);
60
61    .rel.dyn : {
```

```
62    __rel_dyn_start = .;
63    *(.rel*)
64    __rel_dyn_end = .;
65    }
66
67    .dynsym : {
68    __dynsym_start = .;
69    *(.dynsym)
70    }
71
72    _end = .;
73
74    .bss __rel_dyn_start (OVERLAY) : {
75    __bss_start = .;
76    *(.bss)
77    . = ALIGN(4);
78    __bss_end__ = .;
79    }
80 }
```

3. start.S

在u-boot启动分析时，主要是针对u-boot.bin 详细探讨了u-boot的启动过程。从nand_spl的Makefile文件和u-boot.lds链接脚本知道，与u-boot.bin相同的是u-boot-spl-16k.bin包含了Start.S和cpu_init.S 等文件中的程序段。不同的是编译生成 u-boot.bin 时，CONFIG_NAND_SPL 和CONFIG_SPL_BUILD 两个宏没有定义，而编译生成u-boot-spl-16k.bin 时，CONFIG_NAND_SPL被定义。所以，尽管编译相同的源码文件，由于预处理器的作用，源码编译的结果也不一致。

注意CONFIG_SPL_BUILD是在spl/Makefile文件的第18行定义的，实际上，nand_spl跟spl/Makefile文件无关，CONFIG_SPL_BUILD没有定义。去除预处理命令编译和无关紧要的代码，start.S整理为：

```
.globl _start
_start: b  reset
... //全局变量
reset:
    mrs r0, cpsr
    bic r0, r0, #0x3f
    orr r0, r0, #0xd3
    msr cpsr, r0
  cpu_init_crit:
  #ifdef CONFIG_PERIPORT_REMAP
    /* Peri port setup */
    ldr r0, =CONFIG_PERIPORT_BASE
    orr r0, r0, #CONFIG_PERIPORT_SIZE
    mcr p15,0,r0,c15,c2,4
  #endif
    bl  lowlevel_init  /* go setup pll,mux,memory */
    bl  _main
    .globl  relocate_code
  relocate_code:
    mov r4, r0  /* save addr_sp */
    mov r5, r1  /* save addr of gd */
    mov r6, r2  /* save addr of destination */
    adr r0, _start
    cmp r0, r6
```

```
    moveq  r9, #0    /* no relocation. relocation
offset(r9) = 0 */
    beq relocate_done  /* skip relocation */
    mov r1, r6      /* r1 <- scratch for copy_loop */
    ldr r3, _bss_start_ofs
    add r2, r0, r3    /* r2 <- source end address    */
  copy_loop:
    ldmia  r0!, {r9-r10}  /* copy from source address
[r0]  */
    stmia  r1!, {r9-r10}  /* copy to  target address
[r1]  */
    cmp r0, r2      /* until source end address [r2]  */
    blo copy_loop
  relocate_done:
    bx  lr
```

相比生成u-boot.bin阶段的start.S，此时的start.S的有效代码段精简了许多，它对CPU仅仅做了最低限度的初始化，然后调用lowlevel_init作一些板级相关的初始化，最后调用crt0.S文件中的_main函数。

进入_main 函数，调用的一个函数为 board_init_f()，它是一个 C 函数，函数的实现代码在board/samsung/smdk6410/smdk6410_nand_spl.c文件中，如下所示：

```
31 #include <common.h>
32
33 void board_init_f(unsigned long bootflag)
34 {
```

```
35    relocate_code(CONFIG_SYS_TEXT_BASE -
TOTAL_MALLOC_LEN, NULL,
36          CONFIG_SYS_TEXT_BASE);
37 }
```

relocate_code函数的程序在start.S文件中，然后调用relocate_code()，当它返回的时候，通过b跳转指令再次调用relocate_code()，显然这是U-Boot源码错误之一。事实上，u-boot-spl-16k.bin的运行地址为0，加载地址同样为0，不需要代码重定位，只需要清除bss段。对crt0.S文件简单地修改如下：

```
102 #if defined(CONFIG_NAND_SPL)
103    /* deprecated, use instead CONFIG_SPL_BUILD */
104    ldr  sp, =(CONFIG_SYS_INIT_SP_ADDR)
105    //added by liqiang
106 bss_clean:cmp  r0, r1        /* while not at end of
BSS */
107    strlo  r2, [r0]      /* clear 32-bit BSS word */
108    addlo  r0, r0, #4      /* move to next */
109    blo  bss_clean
110    ldr  pc, =nand_boot
111    //add end
```

这样，执行到_main函数时，首先初始化堆栈指针，然后清除bss段，最后PC跳转到nand_boot函数的运行地址处。

4. nand_boot.c

nand_boot()的实现代码在nand_spl/nand_boot.c文件中，nand_boot.c是实现NAND Flash 启动的数据拷贝实现代码。我们知道，S3C6410处理器SRAM的大小只有8K，空间十分有限，在这有限的空间内必须完成系统的初始化和代码数据的拷贝工作。这两个工作步骤

的分界线就是nand_boot()，进入nand_boot()，意味着系统初始化已经完成，开始将NAND Flash 中的u-boot.bin部分数据拷贝到DRAM。nand_boot.c文件已经很好的完成了数据拷贝功能，我们不需要对该文件做任何修改，但是必须根据单板的实际情况修改相关的配置宏。nand_boot函数的程序如下：

```
void nand_boot(void)
{
    struct nand_chip nand_chip;
    nand_info_t nand_info;
    __attribute__((noreturn)) void (*uboot)(void);
    /*
    * Init board specific nand support
    */
    nand_chip.select_chip = NULL;
    nand_info.priv = &nand_chip;
    nand_chip.IO_ADDR_R = nand_chip.IO_ADDR_W = (void
__iomem *)CONFIG_SYS_NAND_BASE;
    nand_chip.dev_ready = NULL;/* preset to NULL */
    nand_chip.options = 0;
    board_nand_init(&nand_chip);
    if (nand_chip.select_chip)
      nand_chip.select_chip(&nand_info, 0);
    /*
      * Load U-Boot image from NAND into RAM
      */
    nand_load(&nand_info, CONFIG_SYS_NAND_U_BOOT_OFFS,
CONFIG_SYS_NAND_U_BOOT_SIZE,
```

```
        (uchar *)CONFIG_SYS_NAND_U_BOOT_DST);
    #ifdef CONFIG_NAND_ENV_DST
      nand_load(&nand_info, CONFIG_ENV_OFFSET,
  CONFIG_ENV_SIZE,
        (uchar *)CONFIG_NAND_ENV_DST);
    #ifdef CONFIG_ENV_OFFSET_REDUND
      nand_load(&nand_info, CONFIG_ENV_OFFSET_REDUND,
  CONFIG_ENV_SIZE,
        (uchar *)CONFIG_NAND_ENV_DST + CONFIG_ENV_SIZE);
    #endif
    #endif
      if (nand_chip.select_chip)
        nand_chip.select_chip(&nand_info, -1);
      /*
        * Jump to U-Boot image
        */
      uboot = (void *)CONFIG_SYS_NAND_U_BOOT_START;
      (*uboot)();
    }
```

nand_boot()首先定义了 nand_chip 结构体，然后调用drivers/mtd/nand/s3c64xx.c 文件中的board_nand_init函数初始化。nand_load () 是nand_boot.c文件中数据拷贝的实现函数。它的原型为：

```
static int nand_load(struct mtd_info *mtd, unsigned int
offs,
      unsigned int uboot_size, uchar *dst)
```

mtd 为 MTD 设备描述符指针，offs 表示数据在 NAND Flash 中的偏移地址，uboot_size 参数表示数据的大小，dst 表示拷贝的目的地址。数据拷贝分析 NAND Flash 启动原理的时候，我们知道，NAND Flash 每页至多能够自动导入 2KB 数据，因此只能把 u-boot-nand.bin，即u-boot-spl-16k.bin 分四页，每页 2KB 放置在 NAND Flash 起始部位。u-boot-nand.bin 布局在NAND Flash 的布局如图 2.22 所示。nand_boot()数据拷贝分为两个步骤：拷贝 u-boot.bin 镜像文件和环境变量。

```
nand_load(&nand_info, CONFIG_SYS_NAND_U_BOOT_OFFS,
CONFIG_SYS_NAND_U_BOOT_SIZE, (uchar
*)CONFIG_SYS_NAND_U_BOOT_DST);
```

上面的函数用于拷贝 u-boot.bin 镜像文件，CONFIG_SYS_NAND_U_BOOT_OFFS 表示u-boot.bin 在NAND Flash 中的偏移地址，默认值为：

```
#define CONFIG_SYS_NAND_U_BOOT_OFFS (4 * 1024) /* Offset
to RAM U-Boot image */
```

由图2.22可知，u-boot.bin部分在NAND Flash偏移地址为16K，因此我们必须将它修改为（16 * 1024）。CONFIG_SYS_NAND_U_BOOT_SIZE 表示 u-boot.bin 文件的大小范围，它必须大于u-boot.bin 文件的实际大小，CONFIG_SYS_NAND_U_BOOT_DST 是 u-boot.bin 的运行地址。nand_boot()做的工作就是将u-boot.bin拷贝到它的运行地址处。

```
nand_load(&nand_info, CONFIG_ENV_OFFSET, CONFIG_ENV_SIZE,
    (uchar *)CONFIG_NAND_ENV_DST);
```

上面的函数是用来拷贝环境变量的，当U-Boot 从NAND Flash 启动时，环境变量显然保存在NAND Flash 比较合适。所有的数据拷贝完

成后，接下来就是PC 跳转到u-boot.bin 的运行地址执行，通过以下代码实现：

```
uboot = (void *)CONFIG_SYS_NAND_U_BOOT_START;
(*uboot)();
```

2.5.5 nand spl启动实现

nand_spl启动时U-Boot 直接从NAND Flash 中启动，在此之前必须将u-boot-nand.bin 文件烧写至NAND Flash内。要烧写u-boot-nand.bin文件必须先借助于SD卡启动方式：把u-boot-nand.bin放入SD卡中（直接拷贝进去），然后U-Boot从SD卡启动，利用fatload命令将它加载到内存中，最后利用nand write.uboot命令烧写u-boot-nand.bin，拨动拨码开关选择NAND Flash 启动。

1. 编译SD卡启动的u-boot.bin

```
224 #define CONFIG_BOOTCOMMAND  "fatload mmc 0 50008000 u-boot-nand.bin;" \
225                 "nand erase.chip;"\
226                 "nand write.uboot 50008000 0 0"
```

修改启动命令参数为以上内容，U-Boot 指令的流程为：首先用fatload 命令从 SD 卡加载u-boot-nand.bin 至内存的地址为50008000 处，再用nand erase.chip命令擦除整个芯片，最后利用我们前面实现的nand write.uboot命令烧写u-boot-nand.bin 文件。

将编译生成的u-boot.bin用SD_Writer软件烧写到SD卡中。

2. 编译NAND Flash 启动的u-boot-nand.bin

NAND Flash 启动是，u-boot.bin 中不能包含 SD 卡数据拷贝功能段，在配置文件中去除CONFIG_BOOT_SD宏定义即可。

```
273 //#define CONFIG_BOOT_SD
```

该宏是我们自己添加的，用于控制u-boot.bin是否执行SD卡数据拷贝函数和环境变量的保存位置。如果它被定义，环境变量保存在SD卡中，否则保存在NAND Flash 中。

同时将启动命令参数修改为：

```
224 #ifdef CONFIG_BOOT_SD
225 #define CONFIG_BOOTCOMMAND  "fatload mmc 0 50008000 u-boot-nand.bin;" \
226                 "nand erase.chip;"\
227                 "nand write.uboot 50008000 0 0"
228 #else
229 #define CONFIG_BOOTCOMMAND  "fatload mmc 0 50008000 uImage;"\
230                 "bootm 50008000"
231
232 #endif
```

将编译生成的u-boot-nand.bin拷贝至SD卡中。

3. 操作步骤

拨动拨码开关选择SD卡启动，系统启动后，通过串口打印出如下信息：

```
Hit any key to stop autoboot: 0
reading u-boot-nand.bin
279516 bytes read in 32 ms (8.3 MiB/s)
NAND erase.chip: device 0 whole chip
Skipping bad block at 0x30a00000
Erasing at 0x7ff80000 -- 100% complete.
OK
NAND write: device 0 offset 0x0, size 0x0
```

299008 bytes written: OK

lq@u-boot #

拨动拨码开关选择NAND Flash 启动，系统启动后，通过串口打印出如下信息：

U-Boot 2013.04-rc1 (Apr 23 2013 - 10:56:37) for SMDK6410

CPU:S3C6410@533MHz

Fclk = 533MHz, Hclk = 133MHz, Pclk = 66MHz (ASYNC Mode)

Board: SMDK6410

DRAM: 256 MiB

WARNING: Caches not enabled

Flash: 0 Mib

NAND: 2048 MiB

MMC: Samsung Host Controller: 0

*** Warning - bad CRC, using default environment

In:serial

Out: serial

Err: serial

Net: CS8900-0

Hit any key to stop autoboot: 0

reading uImage

** Unable to read file uImage **

Wrong Image Format for bootm command

ERROR: can't get kernel image!

lq@u-boot #

NAND Flash 启动成功，打印出错误信息是因为SD 卡中没有Linux内核uImage 文件。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第二章 14 课、15课、16课（NAND Flash移植）。

2.6 DM9000网卡移植

尽管我们用 SD 卡启动的方式能够很好地实现NAND Flash 中数据的更新，但是操作步骤相对繁琐。在嵌入式系统开发阶段通常会使用网络下载镜像文件、挂载文件系统等。U-Boot对一些网卡已经有比较完善的代码支持，例如CS8900、smc911x和DM9000等，实现代码在drivers/net/目录下。SMDK6410（即SMDK6400）默认使用的是CS8900网卡，单板使用的是DM9000网卡芯片，稍微修改源码，就能添加对DM9000的支持。

2.6.1 修改配置文件

进入smdk6410.h文件，找到有关CS8900驱动的宏定义：

```
#define CONFIG_CS8900      /* we have a CS8900 on-board
*/
#define CONFIG_CS8900_BASE  0x18800300
#define CONFIG_CS8900_BUS16    /* follow the Linux driver
*/
```

将这段代码用预处理语句

```
#if 0
......
#endif
```

注释掉，得到：

```
#if 0
94 #define CONFIG_CS8900        /* we have a CS8900 on-board  */
95 #define CONFIG_CS8900_BASE      0x18800300
96 #define CONFIG_CS8900_BUS16      /* follow the Linux driver    */
97 #endif
```

添加有关DM9000网卡的宏定义，首先定义U-Boot包含DM9000网卡。

```
98 #define CONFIG_DM9000
99 #define CONFIG_DM9000_NO_SROM 1
100
101
102 #define CONFIG_DRIVER_DM9000 1
```

定义S3C6410处理器访问DM9000网卡的基地址、I/O地址、数据地址和位宽。

```
103 #define CONFIG_DM9000_BASE 0x18800300
104 #define DM9000_IO CONFIG_DM9000_BASE
105 #define DM9000_DATA (CONFIG_DM9000_BASE+4)
106 #define CONFIG_DM9000_USE_16BIT
```

定义DM9000的物理地址。

```
108 #define CONFIG_ETHADDR  00:40:5c:26:0a:5b
```

定义子网掩码。

```
109 #define CONFIG_NETMASK  255.255.255.0
```

定义单板硬件平台的IP地址。

```
110 #define CONFIG_IPADDR  172.16.114.2
```

定义服务器的IP地址。

111 #define CONFIG_SERVERIP 172.16.114.150

定义单板硬件平台的网关地址。

112 #define CONFIG_GATEWAYIP 172.16.114.1

IP地址的配置与用户所在实际网络环境有关，以上配置仅供参考，请读者根据自身实际自行修改。

2.6.2 增加驱动代码

打开board/samsung/smdk6410/smdk6410.c文件。

```
126 int board_eth_init(bd_t *bis)
127 {
128     int rc = 0;
129 #ifdef CONFIG_CS8900
130     rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
131 #endif
132 #ifdef CONFIG_DM9000  //added by liqiang
133     rc = dm9000_initialize(bis);
134 #endif          // end
135     return rc;
136 }
```

添加dm9000_initialize()函数。编译后，配合SD 卡启动，将u-boot-nand.bin 烧写至NAND Flash中。启动系统，串口打印信息为：

```
U-Boot 2013.04-rc1 (Apr 23 2013 - 11:26:21) for SMDK6410
CPU:S3C6410@533MHz
    Fclk = 533MHz, Hclk = 133MHz, Pclk = 66MHz (ASYNC
Mode)
Board: SMDK6410
```

DRAM: 256 MiB

WARNING: Caches not enabled

Flash: 0 Mib

NAND: 2048 MiB

MMC: Samsung Host Controller: 0

*** Warning - bad CRC, using default environment

In:serial

Out: serial

Err: serial

Net: dm9000

Hit any key to stop autoboot: 0

lq@u-boot #

我们发现，Net栏目由CS8900-0变为dm9000，再通过ping命令检测dm9000网卡驱动是否移植成功，ping一下Linux服务器。

lq@u-boot #ping 172.16.114.150

dm9000 i/o: 0x18800300, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 00:40:5c:26:0a:5b

operating at 100M full duplex mode

Using dm9000 device

host 172.16.114.150 is alive

lq@u-boot #

172.16.114.150被ping通，dm9000网卡驱动移植成功。

2.6.3 配置TFTP服务器

TFTP（Trivial File Transfer Protocol，简单文件传输协议）是TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。它的网络运输层采用的是UDP协议，不可靠、无连接，适用于小文件传输。TFTP网络传输协议的具体内容在这里就不过多介绍，只讲述如何配置TFTP服务器。

在ubuntu系统中下载tftp所需软件包，建立/tftpboot，并且赋予/tftpboot最大的权限。

```
sudo apt-get install tftp-hpa tftpd-hpa
mkdir /var/tftpboot
chmod 777 /var/tftproot
```

打开vim /etc/default/tftpd-hpa 文件，修改成以下内容：

```
# /etc/default/tftpd-hpa
TFTP_USERNAME="tftp"
#TFTP_DIRECTORY="/var/lib/tftpboot"
TFTP_DIRECTORY="/var/tftpboot"
TFTP_ADDRESS="0.0.0.0:69"
#TFTP_OPTIONS="--secure"
TFTP_OPTIONS="-l-c-s"
```

完成之后重启tftp服务即可。

```
sudo service tftpd-hpa restart
```

将编译U-Boot生成的u-boot-nand.bin拷贝至/var/tftpboot目录。

```
liqiang@ubuntu:~/work/forbook/u-boot-2013.04-rc1-sdboot$ cp build/u-boot-nand.bin/var/tftpboot/
```

在串口中输入tftp命令下载u-boot-nand.bin到0x5000_8000内存地址处。

```
lq@u-boot #tftp 50008000 u-boot-nand.bin
dm9000 i/o: 0x18800300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
Using dm9000 device
TFTP from server 172.16.114.150; our IP address is
172.16.114.2
Filename 'u-boot-nand.bin'.
Load address: 0x50008000
Loading: ####################
    987.3 KiB/s
done
Bytes transferred = 281208 (44a78 hex)
lq@u-boot #
```

擦除NAND Flash 芯片，再烧写u-boot-nand.bin。

```
lq@u-boot #nand erase.chip
NAND erase.chip: device 0 whole chip
Skipping bad block at 0x30a00000
Erasing at 0x7ff80000 -- 100% complete.
OK
lq@u-boot #nand write.uboot 50008000 0 0
NAND write: device 0 offset 0x0, size 0x0
299008 bytes written: OK
```

复位后，启动成功。到目前为止有两种烧写NAND Flash 的方式：SD 卡、TFTP网络。在移植过程中，如果我们修改了U-Boot，需要将镜像文件烧写到NAND Flash。一般情况下，使用TFTP网络下载u-boot-

nand.bin 和烧写NAND Flash 更加方便。但是，如果修改后的U-Boot不能正常工作，很可能网络功能就失效，此时必须使用SD卡辅助烧写。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第二章17课（DM9000网卡移植）。

注　释

[1]. MB的全称是MByte，其含义是兆字节；Mb的全称是Mbit，其含义是兆比特。1Byte=8bit。

第3章 Linux-3.8.3内核移植

3.1 Linux内核简介

Linux 是一种自由和开放源代码的类 UNIX 操作系统，该操作系统内核由 Linus Torvalds 在1991年10月5日首次发布。Linux最初是作为支持英特尔x86架构的个人电脑的一个自由操作系统。目前Linux已经被移植到更多的计算机硬件平台。

1994年3月，代码量达17万的Linux-1.0发布。

1996年6月，Linux-2.0内核发布，此时的Linux可以支持多个处理器。

1997年夏，大片《泰坦尼克号》在制作特效中使用的160台Alpha图形工作站中，有105台采用了Linux操作系统。

2001年1月，Linux-2.4发布，它进一步地提升了SMP系统的扩展性，同时它也集成了很多用于支持桌面系统的特性：USB、PC卡（PCMCIA）的支持，内置的即插即用等功能。

2003年12月，Linux-2.6版内核发布，Linux-2.6在对系统的支持都有很大的变化。可以毫不夸张地说，Linux-2.6是Linux发展的一个里程碑。Linux-2.6支持更多平台，使用新的调度器，进程的切换也更高效。

2011年7月，Linux-3.0版本公布，开启Linux的又一个新时代。

2013年3月14日，Linux-3.8.3发布，这是迄今（2013年3月14日）Linux最新稳定版本。

今天在Linus Torvalds 的带领下，众多开发者共同参与开发和维护Linux 内核。理查德·斯托曼领导的自由软件基金会，继续提供大量支持Linux内核的GNU组件。

Linux的低成本、强大的定制功能以及良好的移植性能，使得Linux在嵌入式系统方面也得到广泛应用。在手机、平板电脑等移动设备方面，Linux得到重要发展，基于Linux内核的操作系统也成为最广泛的操作系统。2010年，基于Linux内核的Android操作系统已经超越诺基亚的Symbian操作系统，成为当今全球最流行的智能手机操作系统。在 2010 年第三季度，销售全球的全部智能手机中使用Android的占据25.5%（所有的基于Linux的手机操作系统在这段时间为27.6%）。

本书 Linux 内核移植就是采用最新稳定版本 Linux-3.8.3 进行的，从 Linux 内核的官方网站https://www.kernel.org/下载Linux-3.8.3内核，如图3.1所示。

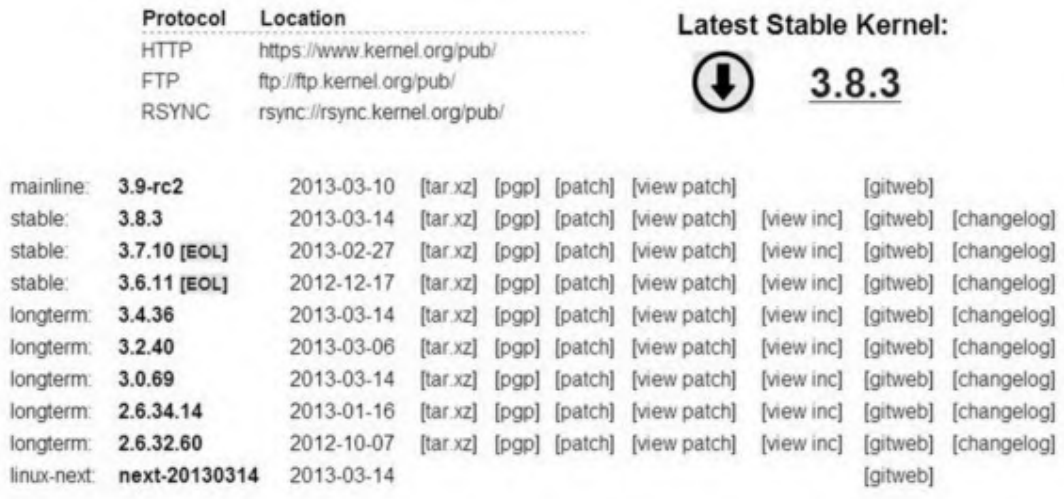


图3.1 kernel官网

下载完成之后可以看到Linux-3.8.3.tar.xz压缩包。我们将Linux-3.8.3内核解压出来，看看这个神秘的内核到底是什么样子。解压缩操作，先将Linux-3.8.3.tar.xz解压成Linux-3.8.3.tar，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux$ xz -d linux-3.8.3.tar.xz

得到Linux-3.8.3.tar之后再一次解压，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux$ tar xvf linux-3.8.3.tar

解压完成之后，得到linux-3.8.3文件夹，进入查看：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ ls

arch Documentation init lib README sound

block drivers ipc MAINTAINERS REPORTING-BUGS tools

COPYING firmware Kbuild Makefile samples usr

CREDITS fs Kconfig mm scripts virt

crypto include kernel net security

这就是linux-3.8.3根目录下的文件，这是一个庞大的代码工程，要想一步一步理清工程的条理，Makefile永远都是你进入这片圣地的地图。打开Makefile文件：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ gedit Makefile

在Makefile的开篇，展现的是linux内核版本信息，如下：

VERSION = 3

PATCHLEVEL = 8

SUBLEVEL = 3

EXTRAVERSION =

NAME = Unicycling Gorilla

在Makefile中有KERNELVERSION，即为内核版本变量。

KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if$(SUBLEVEL),. $(SUBLEVEL)))$(EXTRAVERSION)

通过Makefile的if关键字，很容易知道KERNELVERSION = 3.8.3，NAME 翻译过来是独轮车大猩猩，其实Linux的每个版本都有自己的名字，Linux-3.8.3的名字就为独轮车大猩猩。

了解Linux-3.8.3版本信息之后，正式开启移植过程，希望各位读者做好心理准备，因为这个移植过程是漫长且艰辛的，但是不经历风雨怎能见彩虹呢。笔者站在一个学生的角度去将Linux-3.8.3 版本内核移植到 OK6410，内核所包含的知识是广泛的，所以在本章的移植过程中有些地方不会深究，如果读者感兴趣的话可以自行查阅资料。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第三章 01 课（Linux-3.8.3内核介绍）。

3.2 初步测试内核

内核的移植相对复杂，不可能一步到位，我们只有步步为营，方能步步为赢。本节的目的是修改内核，使得Linux-3.8.3内核适应于OK6410开发平台。外设的移植，在接下来的章节会一步一步完成。

3.2.1 mkimage工具

制作 Linux 内核的压缩镜像文件，需要使用到 mkimage 工具。mkimage 这个工具位于u-boot-2013.04 中的 tools 目录下，它可以用来制作不压缩或者压缩的多种可启动镜像文件。mkimage在制作镜像文件的时候，是在原来的可执行镜像文件的前面加上一个16个byte（0x40）的头，用来记录参数所指定的信息，这样 u-boot 才能识别出制作出来的这个镜像是针对哪一个CPU 体系结构、哪一种 OS、哪种类型、加载到内存中的哪个位置、入口点在内存的哪个位置以及

镜像名是什么等信息。在/u-boot-2013.04/tools目录下执行./mkimage，输出信息如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/u-boot/u-boot-2013.04/u-boot-2013.04/tools$ ./mkimage Usage: ./mkimage -l image

-l ==> list image header information

./mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data_file[:data_file...] image

-A ==> set architecture to 'arch'

-O ==> set operating system to 'os'

-T ==> set image type to 'type'

-C ==> set compression type 'comp'

-a ==> set load address to 'addr' (hex)

-e ==> set entry point to 'ep' (hex)

-n ==> set image name to 'name'

-d ==> use image data from 'datafile'

-x ==> set XIP (execute in place)

./mkimage [-D dtc_options] -f fit-image.its fit-image

./mkimage -V ==> print version information and exit

针对上面的输出信息，-A 指定CPU的体系结构，也就是说，arch的取值可以如下表所示。

表 CPU体系结构

取　值	表示的体系结构	取　值	表示的体系结构
alpha	Alpha	arm	ARM
x86	Intel x86	ia64	IA64
mips	MIPS	mips64	MIPS 64 Bit
ppc	PowerPC	s390	IBM S390
sh	SuperH	sparc	SPARC
sparc64	SPARC 64 Bit	m68k	MC68000

-O 指定操作系统类型，os可以是：openbsd、netbsd、freebsd、4_4bsd、linux、svr4、esix、solaris、irix、sco、dell、ncr、lynxos、vxworks、psos、qnx、u-boot、rtems、artos。

-T 指定镜像类型，type 可以是：standalone、kernel、ramdisk、multi、firmware、script、filesystem。

-C 指定镜像压缩方式，comp 可以是：none（不压缩）、gzip（用gzip 的压缩方式）、bzip2 （用bzip2的压缩方式）。

-a 指定镜像在内存中的加载地址，镜像下载到内存中时，要按照用mkimage 制作镜像时，这个参数所指定的地址值来下载。

-e 指定镜像运行的入口点地址，这个地址就是-a 参数指定的值加上 0x40（因为前面有个mkimage添加的0x40个字节的头）。

-n 指定镜像名。

-d 指定制作镜像的源文件。

将 u-boot-2013.04 下的 tools 这个文件夹下中的 mkimage 工具复制到 ubuntu 系统的/user/bin下，这样可以直接当作操作命令使用。

3.2.2 配置menuconfig

make menuconfig 是基于文本菜单选择的图形化内核配置界面。

打开最顶层的Makefile，有这么两行代码。

```
ARCH ?= $(SUBARCH)
CROSS_COMPILE ?= $(CONFIG_CROSS_COMPILE:"%"=%)
```

ARCH针对何种CPU体系结构，OK6410的cpu是三星公司的S3C6410，为arm，那么这句就得修改成arm。CROSS_COMPILE是编译工具链，和u-boot配置一样。则需修改成：

```
ARCH ?= arm
```

CROSS_COMPILE ?= /usr/local/arm/4.4.1/bin/arm-linux-

进入arch/arm/mach-s3c64xx，有Kconfig文件，Kconfig作用是描述所属目录源文档相关的内核配置菜单，在执行make menuconfig时，将从Kconfig文件中读出菜单。打开Kconfig文件。其中：

S3C6410 machine support

所支持的平台有：

config MACH_ANW6410

config MACH_MINI6410

config MACH_REAL6410

config MACH_SMDK6410

但是没有OK6410，这里就需要修改文件，使得Linux-3.8.3能适合运行在OK6410开发平台的内核，取以上的4种平台中的一种作为基础进行修改，这里就采用MINI6410。

在当前arch/arm/mach-s3c64xx文件下，复制一份mach-mini6410.c并且重命名为mach-ok6410.c。使用操作命令：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$ cpmach-mini6410.c mach-ok6410.c

打开 mach-ok6410.c 文件，将 mini6410 （ MINI6410 ）修改为 ok6410 （ OK6410 ），打开mach-ok6410.c。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$ geditmach-ok6410.c

使用gedit最大的好处是可以很好地进行文本操作，使用替换功能，将mini6410 （MINI6410）替换成为ok6410（OK6410）。

在arch/arm/mach-s3c64xx目录下打开Makefile，找到如下：

obj-$(CONFIG_MACH_MINI6410) += mach-mini6410.o

在其后面添加ok6410的配置：

obj-$(CONFIG_MACH_OK6410)   += mach-ok6410.o

添加这行代码则是告诉编译器要将ok6410.c编译进内核。

回到arch/arm/mach-s3c64xx目录下的Kconfig，打开文件，为OK6410添加配置菜单。在如下：

config MACH_MINI6410

后面添加OK6410的配置：

```
  config MACH_OK6410
    bool "OK6410"
    select CPU_S3C6410
    select SAMSUNG_DEV_ADC
    select S3C_DEV_HSMMC
    select S3C_DEV_HSMMC1
    select S3C_DEV_I2C1
  select SAMSUNG_DEV_IDE
    select S3C_DEV_FB
    select S3C_DEV_RTC
    select SAMSUNG_DEV_TS
    select S3C_DEV_USB_HOST
  #  select S3C_DEV_USB_HSOTG
    select S3C_DEV_WDT
    select SAMSUNG_DEV_KEYPAD
    select SAMSUNG_DEV_PWM
    select HAVE_S3C2410_WATCHDOG if WATCHDOG
    select S3C64XX_SETUP_SDHCI
    select S3C64XX_SETUP_I2C1
    select S3C64XX_SETUP_IDE
    select S3C64XX_SETUP_FB_24BPP
```

select S3C64XX_SETUP_KEYPAD

help

Machine support for the feiling OK6410

添加之后执行make menuconfig 就会有ok6410 选项。

进入arch/arm/tools，打开mach-types文件：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/tools$ gedit mach-types

可以看到如下：

machine_is_xxx CONFIG_xxxx MACH_TYPE_xxx number

mini6410 MACH_MINI6410 MINI6410 2520

mini6410的ID是2520，但是OK6410的ID是1626，这个在u-boot也曾经出现过，这就如每一个人都有自己相对应的ID，如果ID号不匹配，将导致u-boot无法启动内核，在mini6410后面添加如下。

ok6410 MACH_OK6410 OK6410 1626

回到linux3.8.3根文件夹下，配置menuconfig，输入操作命令：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make menuconfig

如果虚拟机之前没有更新过或者没安装过编译linux内核，则会出现如图3.2所示的错误。

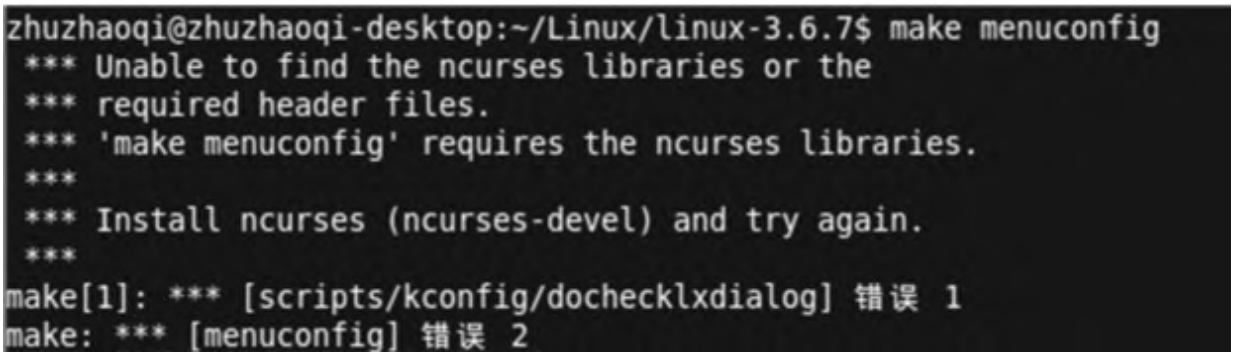


图3.2 make menuconfig报错

从报错的信息可知没有安装ncurses这个库，ncurces是提供字符中断处理，包括面板和菜单。那么接下来进行安装，执行如下操作命令。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$sudo apt-get install libncurses*

为了使得内核在 make xconfig 不出错，也将 build-essential、kernel-package 和两个 QT 库libqt3-headers、libqt3-mt-dev一起安装。

make menuconfig提供一个基于文本的图形界面，它依赖于ncurses5这个包，键盘操作可以修改选项，读者编译内核修改选项推荐用 make menuconfig；make xconfig 需要有 x window system支持，也就是说要在KDE、GNOME等的X桌面环境下才可用，好处是支持鼠标，坏处是X本身占用系统周期，而且X环境容易引起编译器的不稳定。

再一次输入make menuconfig 操作命令：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make menuconfig

如果上面的相关库都安装好了的话，就会出现如图3.3所示界面。

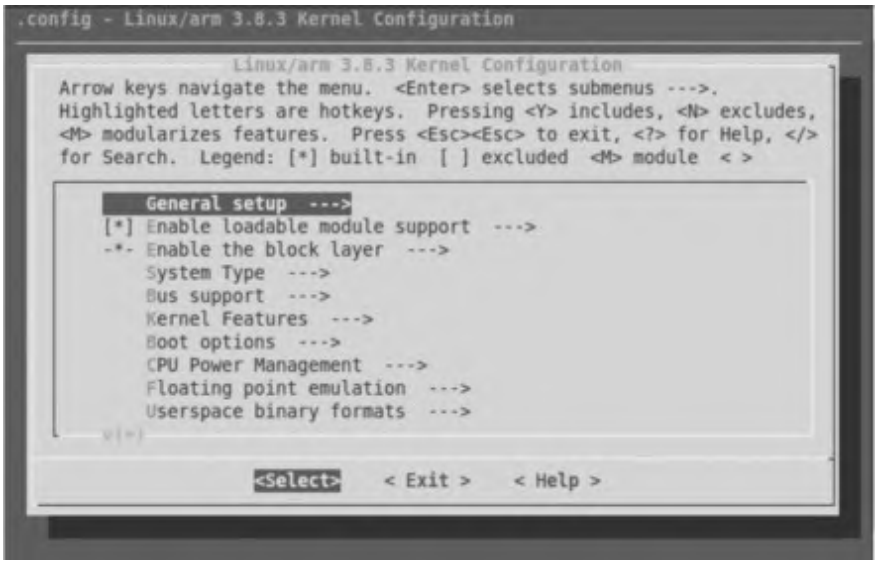


图3.3 make menuconfig界面

在 Linux/arm 3.8.3 的 Kernel Configuration 界面中的倒数第二个是 Load an Alternate Configuration File，是要求输出配置文件的路径。进入linux-3.8.3/arch/arm/configs查看，与OK6410最接近的是s3c6400_defconfig，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/configs$ ls s*

s3c2410_defconfig s5pv210_defconfig socfpga_defconfig spitz_defconfig

s3c6400_defconfig shannon_defconfig spear13xx_defconfig

s5p64x0_defconfig shark_defconfig spear3xx_defconfig

s5pc100_defconfig simpad_defconfig spear6xx_defconfig

选择Load an Alternate Configuration File，输入 arch/arm/configs/s3c6400_defconfig ，如图3.4所示。

图3.4 添加配置文件路径

进入 General Setup，打开 Cross-compiler tool perfix，输入编译工具链路径/usr/local/arm/4.4.1/bin/arm-

linux- ，如图3.5 所示。

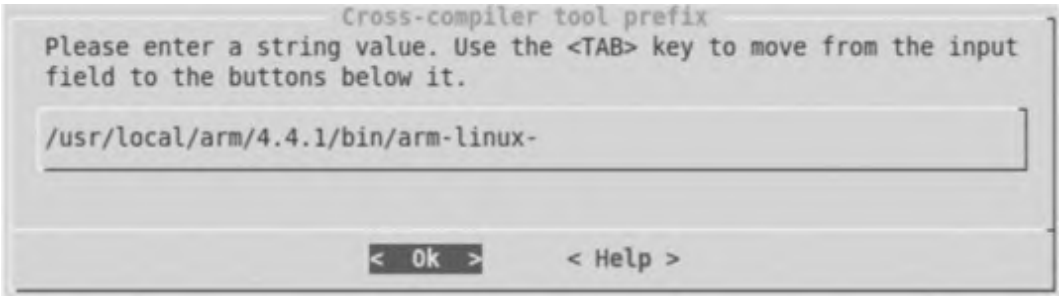


图3.5 编译工具链路径

进入 System Type，设置开发板平台。取消 SMDK6400、A&W6410、SMDK6410、SmartQ5等平台，只选择OK6410选项，如图3.6所示。

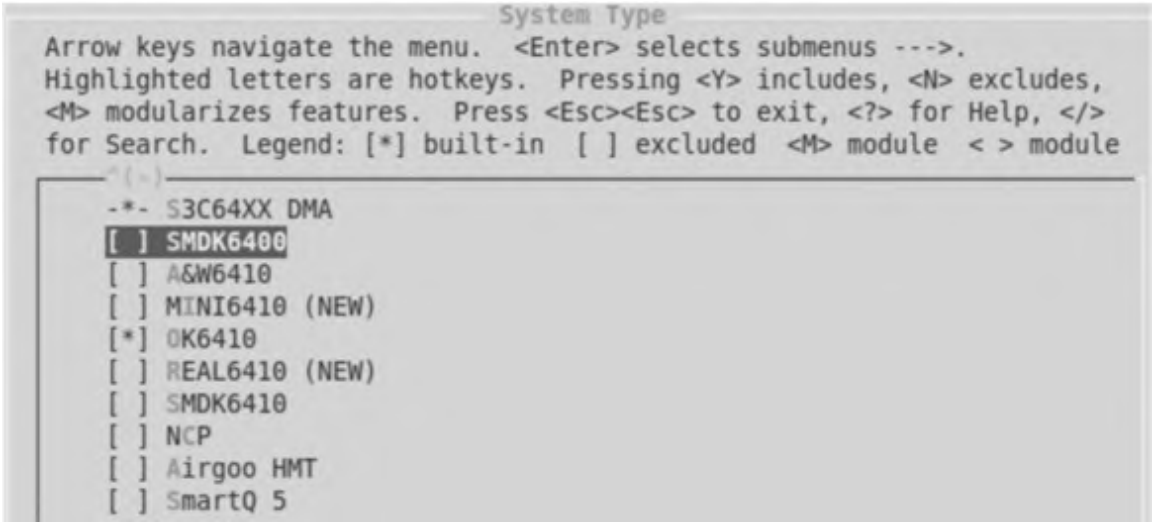


图3.6 选择开发平台

最后进入Save an Alternate Configuration File，保存为.config 然后退出。

完成之后编译内核make uImage，如果之前操作没错的话，编译成功后输出信息如下：

```
……
OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
```

```
Image Name:  Linux-3.8.3
Created:  Fri Mar 15 12:57:11 2013
Image Type:  ARM Linux Kernel Image (uncompressed)
Data Size:  1664080 Bytes = 1625.08 kB = 1.59 MB
Load Address: 50008000
Entry Point:  50008000
  Image arch/arm/boot/uImage is ready
```

进入arch/arm/boot目录下，可以看到uImage和zImage，这两个就是生成的不同的两种内核镜像。如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/boot$ ls
bootp compressed dts Image install.sh Makefile uImage zImage
```

zImage内核镜像下载到开发板之后，可以使用u-boot的go命令进行直接跳转，这个时候内核直接解压启动。但是此时的内核无法挂载文件系统，因为go命令没有将内核需要的相关启动参数从u-boot中传递给内核。传递相关启动参数必须使用u-boot的bootm命令进行跳转，但是u-boot的bootm命令只能处理uImage镜像。uImage相对于zImage在头部多了64个byte，即为0x40。先对内核进行测试，下文会详细讲述这两个地址的关系。

调试的办法常见有两种，其一，通过SD卡烧写内核；其二，通过TFTP下载内核。从调试的方便程度而言，TFTP 是首选，但是两种方法都必须掌握。现在先使用第一种方法进行调试。

将uImage（如果下载指令指定了是zImage，则需把uImage更改名称为zImage）与u-boot.bin放入SD卡一起烧写入NandFlash中，启动之后看看u-boot是否能引导内核。

启动之后通过串口输出显示的信息是：

```
……
NAND read: device 0 offset 0x100000, size 0x500000
  5242880 bytes read: OK
## Booting kernel from Legacy Image at 50008000 ...
  Image Name: Linux-3.8.3
  Image Type: ARM Linux Kernel Image (uncompressed)
  Data Size:1664080 Bytes = 1.6 MiB
  Load Address: 50008000
  Entry Point: 50008000
  Verifying Checksum ... OK
  XIP Kernel Image ... OK
OK
Starting kernel ...
Starting kernel ...
```

串口输出停在这里就不动了，但是这句话同时也是u-boot执行完成之后给出的最后一个信息：

```
Starting kernel ...
```

也就是说，这个时候u-boot就把CPU交给了内核，也就是此时正准备进入内核。导致的原因可能是配置debug串口错误，也有可能是加载内核地址出现问题。那么就逐一分析，找出问题的根源。

首先查看debug串口配置，打开linux-3.8.3根目录下的.config文件，操作如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ gedit .config
```

找到UART的debug串口配置，如下：

```
CONFIG_DEBUG_S3C_UART0=y
# CONFIG_DEBUG_S3C_UART1 is not set
```

CONFIG_DEBUG_S3C_UART2 is not set

CONFIG_DEBUG_LL_UART_NONE is not set

也就是说UART0被设置成为调试串口，用以输出调试信息。

回到linux3.8.3 的根目录下，执行make menuconfig，查看System Type ---> （0） S3C UART to use for low-level messages的默认调试串口是否为0，如果不是则加以修改。

完成之后再次进行编译、调试，如果还是出现同样的错误，则加载内核地址、入口地址等可能出现错误。

回到串口信息输出，对比如下信息：

Booting kernel from Legacy Image at 50008000 ...

……

Load Address: 50008000

Entry Point: 50008000

从上面第一行信息可知启动内核的地址为：50008000。这个启动内核的地址来源于 U-Boot源码中include/configs/目录下的s3c6410.h文件中：

```
#ifdef CONFIG_ENABLE_MMU
#define CONFIG_SYS_MAPPED_RAM_BASE  0xc0000000
#define CONFIG_BOOTCOMMAND  "nand read 0xc0018000 0x60000 0x1c0000;" \
  "bootm 0xc0018000"
#else
#define CONFIG_SYS_MAPPED_RAM_BASE CONFIG_SYS_SDRAM_BASE
#ifdef CONFIG_BOOT_SD
#define CONFIG_BOOTCOMMAND  "fatload mmc 0 50008000 u-boot-nand.bin;" \
```

```
        "nand erase.chip;"\
        "nand write.uboot 50008000 0 0"
    #else
    #define CONFIG_BOOTCOMMAND  "fatload mmc 0 50008000
uImage;"\
        "bootm 50008000"
    #endif
```

Load Address 指的是镜像在内存中的加载地址，为50008000。

Entry Point 指的是镜像运行的入口地址，为50008000。

这三者的关系将在下一节展开详细讲解。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第三章 02 课、03课（初步测试内核）。

3.2.3 加载地址和入口地址

在上一节中，无法启动内核，导致的原因可能是加载地址、入口地址等错误导致的。执行./mkimage之后如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/u-boot/u-boot-2013.04/u-
boot-2013.04/tools$ ./mkimage./mkimage [-x] -A arch -O os -T
type -C comp -a addr -e ep -n name -d
data_file[:data_file...] image
```

其中-a addr 指的就是镜像在内存中的加载地址，镜像下载到内存中时，要按照用mkimage 制作镜像时，这个参数所指定的地址值来下载。

而-e ep 是指定镜像运行的入口点地址。

还有两个概念需要明白，即bootm address 和kernel运行地址。bootm address：通过uboot的bootm命令，从address启动kernel。

kernel运行地址：在具体mach目录中的Makefile.boot中指定，是kernel启动后实际运行的物理地址。

如果bootm address 和Load Address 相等，在这种情况下，bootm不会对uImage header 后的zImage进行memory move的动作，而会直接go 到Entry Point开始执行。因此此时的Entry Point必须设置为Load Address+ 0x40。如果kernel boot过程没有到uncompressing the kernel，就可能是设置不对。它们之间的关系为：boom address == Load Address == Entry Point - 0x40。

如果bootm address 和Load Address 不相等（但需要避免出现memory move 时因为覆盖导致zImage 被破坏的情况）。此种情况下，bootm 会把 uImage header 后的 zImage 文件 move 到Load Address，然后go到entry point开始执行。这段代码在common/cmd_bootm.c 中的bootm_load_os函数中，如下程序所示。由此知道此时的Load Address 必须等于Entry Point。它们之间的关系则为：boom address != Load Address == Entry Point。

```
  case IH_COMP_NONE:
    if (load == blob_start || load == image_start)
{
  printf(" XIP %s ... ", type_name);
  no_overlap = 1;
}
else
{
  printf(" Loading %s ... ", type_name);
  memmove_wd((void *)load, (void *)image_start,
    image_len, CHUNKSZ);
}
```

```
*load_end = load + image_len;
puts("OK\n");
break;
```

zImage 的头部有地址无关的自解压程序，因此刚开始执行的时候，zImage 所在的内存地址（Entry Point）不需要同编译kernel的地址相同。自解压程序会把kernel解压到编译时指定的物理地址，然后开始地址相关代码的执行。在开启MMU之前，kernel都是直接使用物理地址（可参看内核符号映射表System.map）。

通过上面的分析，大概找出了问题的根源，由于bootm address和Load Address都为50008000，属于相等情况，也就是说Entry Point: 50008000 这个地址需要修改，替换成50008040。

找到Load Address 和Entry Point 这两个地址的定义，存在于scripts/makefile.lib 中。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/scripts$ gedit Makefile.lib

打开之后可以找到如下：

```
318  UIMAGE_LOADADDR ?= arch_must_set_this
319  UIMAGE_ENTRYADDR ?= $(UIMAGE_LOADADDR)
```

这里就是说Entry Point等于Load Address，那么应该修改成为Entry Point=Load Address+0x40。在GNU make 中，有sed -e 替换操作，如sed -e "s/..$$/40/"，就是把输出的字符串的最后两个字符删掉，并且用40来补充，也就是说把字符串最后两个字符用40来替换。

那么做如下修改：

```
318  UIMAGE_LOADADDR ?= arch_must_set_this
319  #UIMAGE_ENTRYADDR ?= $(UIMAGE_LOADADDR)
```

320 UIMAGE_ENTRYADDR ?=$(shell echo $(UIMAGE_LOADADDR) |

sed -e "s/..$$/40/")

修改完成之后，回到linux-3.8.3根目录下进行编译，如下操作：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make uImage

如果有如下报错：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make uImage

scripts/kconfig/conf --silentoldconfig Kconfig

*** Error during update of the configuration.

make[2]: *** [silentoldconfig] 错误 1

make[1]: *** [silentoldconfig] 错误 2

make: *** 没有规则可以创建“include/config/kernel.release”需要的目标“include/config/auto.conf”。停止。

那就是权限的问题，要么修改文件权限，要么在root下编译。这样即可：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ sudo make uImage

编译成功之后输出如下信息：

• • • • • •

Image Name: Linux-3.8.3

Created: Sat Mar 16 10:38:47 2013

Image Type: ARM Linux Kernel Image (uncompressed)

Data Size: 1664080 Bytes = 1625.08 kB = 1.59 MB

Load Address: 50008000

Entry Point: 50008040

Image arch/arm/boot/uImage is ready

从串口输出可知，Entry Point=Load Address+0x40，依旧按照SD 烧写方式进行测试，如果bootdelay延时过长，可以修改bootdelay时间，如下操作：

Hit any key to stop autoboot: 0

zzq6410 >>> set bootdelay 3

zzq6410 >>> sav

Saving Environment to NAND...

Erasing Nand...

Erasing at 0x80000 -- 100% complete.

Writing to Nand... done

zzq6410 >>>

重启OK6410开发平台，测试结果如下：

NAND read: device 0 offset 0x100000, size 0x500000

5242880 bytes read: OK

Booting kernel from Legacy Image at 50008000 ...

Image Name: Linux-3.8.3

Image Type: ARM Linux Kernel Image (uncompressed)

Data Size:1664080 Bytes = 1.6 MiB

Load Address: 50008000

Entry Point: 50008040

Verifying Checksum ... OK

XIP Kernel Image ... OK

OK

Starting kernel ...

Starting kernel ...

Uncompressing Linux... done, booting the kernel.

Booting Linux on physical CPU 0x0

Linux version 3.8.3 (zhuzhaoqi@zhuzhaoqi-desktop) (gcc version 4.4.1 (Sourcery G++ Lite2009q3-67)) #1 Fri Mar 15 12:56:52 CST 2013

CPU: ARMv6-compatible processor [410fb766] revision 6 (ARMv7), cr=00c5387d

CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing instruction cache

Machine: OK6410

Memory policy: ECC disabled, Data cache writeback

CPU S3C6410 (id 0x36410101)

S3C24XX Clocks, Copyright 2004 Simtec Electronics

S3C64XX: PLL settings, A=533000000, M=533000000, E=24000000

S3C64XX: HCLK2=266500000, HCLK=133250000, PCLK=66625000

mout_apll: source is fout_apll (1), rate is 533000000

mout_epll: source is epll (1), rate is 24000000

mout_mpll: source is mpll (1), rate is 533000000

usb-bus-host: source is clk_48m (0), rate is 48000000

irda-bus: source is mout_epll (0), rate is 24000000

CPU: found DTCM0 8k @ 00000000, not enabled

CPU: moved DTCM0 8k to fffe8000, enabled

CPU: found DTCM1 8k @ 00000000, not enabled

CPU: moved DTCM1 8k to fffea000, enabled

CPU: found ITCM0 8k @ 00000000, not enabled

CPU: moved ITCM0 8k to fffe0000, enabled

CPU: found ITCM1 8k @ 00000000, not enabled

CPU: moved ITCM1 8k to fffe2000, enabled

Built 1 zonelists in Zone order, mobility grouping on. Total pages: 65024

Kernel command line: root=/dev/mtdblock2 rootfstype=cramfs console=ttySAC0,115200

PID hash table entries: 1024 (order: 0, 4096 bytes)

Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)

Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)

__ex_table already sorted, skipping sort

Memory: 256MB = 256MB total

Memory: 256532k/256532k available, 5612k reserved, 0K highmem

Virtual kernel memory layout:

vector : 0xffff0000 - 0xffff1000 (4 kB)

DTCM : 0xfffe8000 - 0xfffec000 (16 kB)

ITCM : 0xfffe0000 - 0xfffe4000 (16 kB)

fixmap : 0xfff00000 - 0xfffe0000 (896 kB)

vmalloc : 0xd0800000 - 0xff000000 (744 MB)

lowmem : 0xc0000000 - 0xd0000000 (256 MB)

modules : 0xbf000000 - 0xc0000000 (16 MB)

.text : 0xc0008000 - 0xc02bed88 (2780 kB)

.init : 0xc02bf000 - 0xc02da7a4 (110 kB)

.data : 0xc02dc000 - 0xc03076a0 (174 kB)

.bss : 0xc0308000 - 0xc0338ef8 (196 kB)

```
      SLUB: Genslabs=13, HWalign=32, Order=0-3, MinObjects=0,
  CPUs=1, Nodes=1
    NR_IRQS:246
    VIC @f6000000: id 0x00041192, vendor 0x41
    VIC @f6010000: id 0x00041192, vendor 0x41
    sched_clock: 32 bits at 100 Hz, resolution 10000000ns,
wraps every 4294967286ms
    Console: colour dummy device 80x30
    Calibrating delay loop... 353.89 BogoMIPS (lpj=1769472)
    pid_max: default: 32768 minimum: 301
    Mount-cache hash table entries: 512
    CPU: Testing write buffer coherency: ok
    Setting up static identity map for 0x502149a8 -
0x50214a04
    DMA: preallocated 256 KiB pool for atomic coherent
allocations
    OK6410: Option string ok6410=0
    OK6410: selected LCD display is 480x272
    s3c64xx_dma_init: Registering DMA channels
    PL080: IRQ 73, at d0846000, channels 0..8
    PL080: IRQ 74, at d0848000, channels 8..16
    S3C6410: Initialising architecture
    bio: create slab <bio-0> at 0
    usbcore: registered new interface driver usbfs
    usbcore: registered new interface driver hub
    usbcore: registered new device driver usb
    ROMFS MTD (C) 2007 Red Hat, Inc.
```

io scheduler noop registered

io scheduler deadline registered

io scheduler cfq registered (default)

s3c-fb s3c-fb: window 0: fb

Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled

s3c6400-uart.0: ttySAC0 at MMIO 0x7f005000 (irq = 69) is a S3C6400/10

console [ttySAC0] enabled

s3c6400-uart.1: ttySAC1 at MMIO 0x7f005400 (irq = 70) is a S3C6400/10

s3c6400-uart.2: ttySAC2 at MMIO 0x7f005800 (irq = 71) is a S3C6400/10

s3c6400-uart.3: ttySAC3 at MMIO 0x7f005c00 (irq = 72) is a S3C6400/10

brd: module loaded

loop: module loaded

s3c24xx-nand s3c6400-nand: Tacls=4, 30ns Twrph0=8 60ns, Twrph1=6 45ns

s3c24xx-nand s3c6400-nand: System booted from NAND

s3c24xx-nand s3c6400-nand: NAND soft ECC

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd5 (Samsung NAND 2GiB 3,3V 8-bit), 2048MiB,page size: 4096, OOB size: 218

No oob scheme defined for oobsize 218

……

Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b

中间省去了很多信息，因为这些信息暂时是没有太大意义，但是也给出了很多信息，可以很好地和接下来的每一步移植作对比。从串口的输出可以得知，内核是启动了。也就是说，此时u-boot已经成功将相关参数传递给linux3.8.3内核，完成了u-boot到内核的交接。并且内核已经识别了是OK6410开发平台，控制CPU是s3c6410等信息。

当然，读者不仅仅可以通过修改Entry Point使得内核启动，还可以修改启动内核的地址使得bootm address 和 Load Address 不相等，也就是修改 U-Boot 源码中 include/configs/目录下的 s3c6410.h文件中：

```
#ifdef CONFIG_ENABLE_MMU
#define CONFIG_SYS_MAPPED_RAM_BASE 0xc0000000
#define CONFIG_BOOTCOMMAND"nand read 0xc0018000 0x600000x1c0000;\"bootm 0xc0018000"
#else
#define CONFIG_SYS_MAPPED_RAM_BASE CONFIG_SYS_SDRAM_BASE
#define CONFIG_BOOTCOMMAND"nand read 0x50018000 0x100000 0x500000;"\"bootm 0x50018000"
#endif
```

3.2.4 TFTP测试内核

在U-Boot移植章节已经完成了TFTP环境搭建，这里使用TFTP进行烧写内核、调试内核。

将uImage拷贝到tftpboot：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/boot$ cpuImage ../../../../../../../tftpboot/
```

宿主机进入tftp：

```
zhuzhaoqi@zhuzhaoqi-desktop:/tftpboot$ tftp
tftp>
```

启动U-Boot，执行TFTP下载内核命令：

```
zzq6410 >>> tftp 0x50008000 uImage
dm9000 i/o: 0x18000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
Using dm9000 device
TFTP from server 192.168.1.187; our IP address is
192.168.1.100
Filename 'uImage'.
Load address: 0x50008000
Loading:
#############################################################
#############################################################
#############################################################
#############################################################
#############################################################
#############################
#############################################################
##########################
done
Bytes transferred = 2151800 (20d578 hex)
zzq6410 >>>
zzq6410 >>> bootm 0x50008000
```

```
## Booting kernel from Legacy Image at 50008000 ...
   Image Name:   Linux-3.8.3
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2151736 Bytes = 2.1 MiB
   Load Address: 50008000
   Entry Point:  50008040
   Verifying Checksum ... OK
   XIP Kernel Image ... OK
OK
Starting kernel ...
```

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章04课（下载地址和入口地址分析）。

3.2.5 内核启动分析

对于 ARM 处理器，内核启动大体上可以分为两个阶段：与处理器相关的汇编启动阶段和与处理器无关的C代码启动阶段。汇编启动阶段从head.S（arch/arm/kernel/head.S）文件开始，C代码启动阶段从start_kernel函数（init/main.c）开始。当然，经过压缩的内核镜像文件zImage在进入汇编启动阶段前还要运行一段自解压代码（arch/arm/boot/compressed/head.S）。

1. 汇编启动阶段

省略一些无关紧要的过程和编译后不运行的代码，该过程的启动流程如图 3.7 所示。相对早期linux-2.6.38的版本，linux-3.8.3在汇编启动阶段并没有出现__lookup_machine_type，但这并不意味着内核不再检查bootloader传入的machine_arch_type参数（R1），只是将检查机制推迟到了C代码阶段。

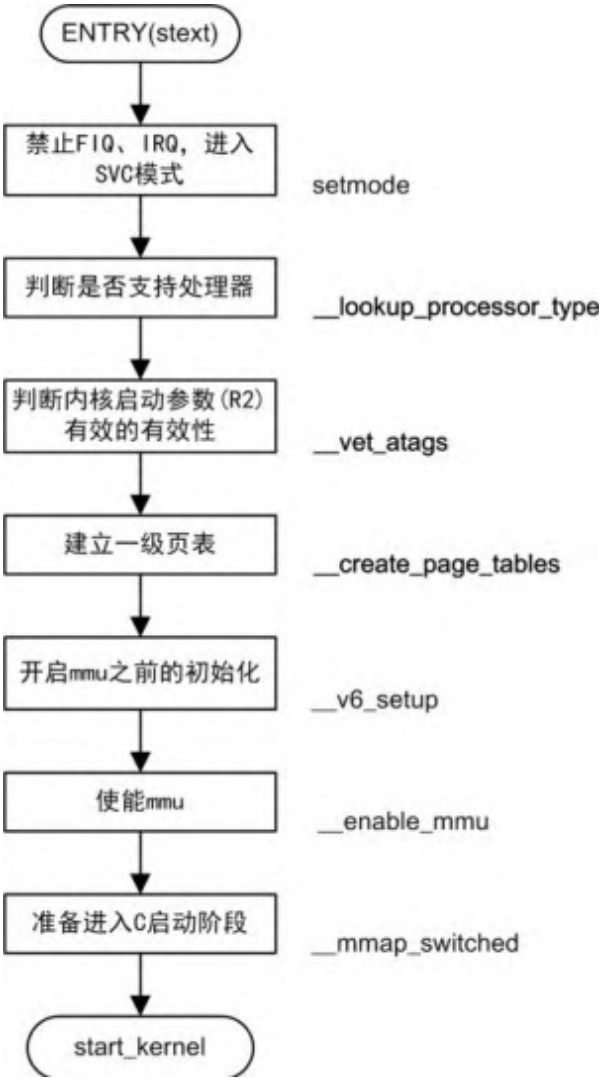


图3.7 内核汇编启动阶段

（1）__lookup_processor_type

__lookup_processor_type函数的具体实现如程序清单3.1所示。

程序清单3.1 查找处理器类型函数

```
__lookup_processor_type:
    adr r3, __lookup_processor_type_data
    ldmia r3, {r4 - 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:  ldmia  r5, {r3, r4}       @ value, mask
  and r4, r4, r9      @ mask wanted bits
  teq r3, r4
  beq 2f
  add r5, r5, #PROC_INFO_SZ    @ sizeof(proc_info_list)
  cmp r5, r6
  blo 1b
  mov r5, #0         @ unknown processor
2:  mov pc, lr
ENDPROC(__lookup_processor_type)
  .align  2
  .type  __lookup_processor_type_data, %object
__lookup_processor_type_data:
  .long  .
  .long  __proc_info_begin
  .long  __proc_info_end
  .size  __lookup_processor_type_data, . -
__lookup_processor_type_data
```

__lookup_processor_type函数的主要功能是将内核支持的所有CPU类型与通过程序实际读取的cpu id 进行查表匹配。如果匹配成功，将匹配到的proc_info_list的基地址存到r5,否则，r5为0，程序将会进入一个死循环。函数传入参数 r9 为程序实际读取的 cpu id，传出参数 r5 为匹配到的proc_info_list 指针的地址。同时为了使 C 语言能够调用这个函数，根据 APCS（ARM 过程调用标准）规则，简单使用以下代码就能包装成一个C语言版本__lookup_processor_type的API函数，函数的原型为struct proc_info_list *lookup_processor_type（unsigned int）。

```
  ENTRY(lookup_processor_type)
    stmfd  sp!, {r4 - r6, r9, lr}
    mov r9, r0
    bl  __lookup_processor_type
    mov r0, r5
    ldmfd  sp!, {r4 - r6, r9, pc}
  ENDPROC(lookup_processor_type)
```

ENTRY和ENDPROC宏的定义如下：

```
#define ENTRY(name)\
    .globl name;\
    name:
#define ENDPROC(name)
```

内核利用一个结构体 proc_info_list 来记录处理器相关的信息，在文件 arch/arm/include/asm/procinfo.h声明了该结构体的类型，如下所示。

```
struct proc_info_list {
    unsigned int    cpu_val;
    unsigned int    cpu_mask;
    unsigned long   __cpu_mm_mmu_flags;  /* used by head.S
*/
    unsigned long   __cpu_io_mmu_flags;  /* used by head.S
*/
    unsigned long   __cpu_flush;    /* used by head.S */
    const char    *arch_name;
    const char    *elf_name;
    unsigned int    elf_hwcap;
    const char    *cpu_name;
```

```
        struct processor  *proc;
        struct cpu_tlb_fns  *tlb;
        struct cpu_user_fns  *user;
        struct cpu_cache_fns  *cache;
    };
```

事实上，在 arch/arm/mm/proc-*.S 这类文件中，程序才真正给内核所支持的 arm 处理器的proc_info_list分配了内存空间，例如linux/arch/arm/mm/proc-v6.S文件用汇编语言定义__v6_proc_info结构体， .section 指示符来指定这些结构体编译到.proc.info 段。.proc.info 的起始地址为__proc_info_begin，终止位置为__proc_info_end，把它们作为全局变量保存在内存中，链接脚本arch/arm/kernel/vmlinux.lds的部分内容参考如下：

```
      .init.proc.info : {
        . = ALIGN(4);
      __proc_info_begin = .;
      *(.proc.info.init)
    __proc_info_end = .;
      }
```

（2）__vet_atags

在启动内核时，bootloader会向内核传递一些参数。通常，bootloader 有两种方法传递参数给内核：一种是旧的参数结构方式（parameter_struct）——主要是2.6版本之前的内核使用的方式；另外一种是现在的内核在用的参数列表（tagged list） 的方式。这些参数主要包括，系统的根设备标志、页面大小、内存的起始地址和大小、当前内核命令参数等。而这些参数是通过struct tag结构体组织，利用指针链接成一个按顺序摆放的参数列表。bootloader 引导内

核启动时，就会把这个列表的首地址放入R2中，传给内核，内核通过这个地址就分析出传入的所有参数。

内核要求参数列表必须存放在RAM物理地址的头16KB位置，并且ATAG_CORE类型的参数需要放置在参数的列表的首位。__vet_atags的功能就是初步分析传入的参数列表，判断的方法也很简单。如果这个列表起始参数是ATAG_CORE类型，则表示这是一个有效的参数列表。如果起始参数不是ATAG_CORE，就认为bootloader没有传递参数给内核或传入的参数不正确。

（3）__create_page_tables

Linux 内核使用页式内存管理，应用程序给出的内存地址是虚拟地址，它需要经过若干级页表一级一级的变换，才变成真正的物理地址。32 位 CPU 的虚拟地址大小从 0x0000_0000 到0xFFFF_FFFF 共4GB。以段（1 MB）的方式建立一级页表，可以将虚拟地址空间分割成4096 个段条目（section entry）。条目也称为“描述符”（Descriptor），每一个段描述符为32 位，因此一级页表占用16KB（0x4000）内存空间。

s3c6410处理器DRAM的地址空间从0x5000_0000开始，上文提到bootloader传递给内核的参数列表存放在RAM物理地址的头16KB位置，页表放置在内核的前16KB，因此内核的偏移地址为32KB（0x8000），由此构成了如图3.8所示的实际内存分布图。

__create_page_tables 函数初始化了一个非常简单的页表，仅映射了使内核能够正常启动的代码空间，更加细致的工作将会在后续阶段完善。流程如图 3.9 所示，获取页表物理地址、清空页表区和建立启动参数页表通过阅读源码很容易理解，不加分析。

图3.8 实际内存分布图

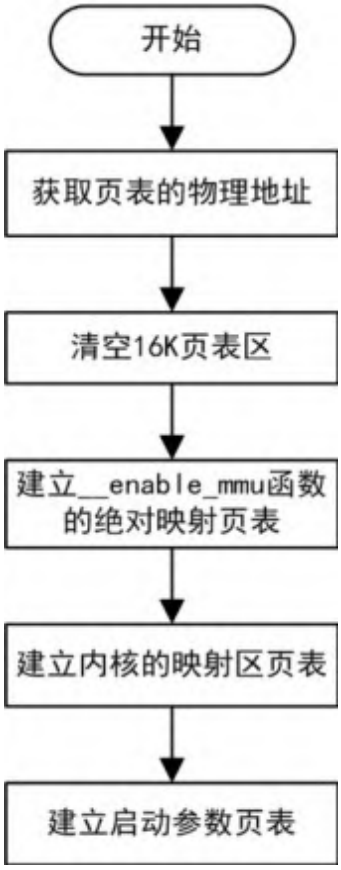


图3.9 初步页表建立流程

__enable_mmu函数使能MMU后，CPU发出的地址是虚拟地址，程序正常运行需要映射得到物理地址，为了保障正常地配置MMU，需要对这段代码1:1的绝对映射，映射范围为__turn_mmu_on至__turn_mmu_on_end。正常使能MMU后，不需要这段特定的映射了，在

后续C代码启动阶段时被paging_init()函数删除。建立__enable_mmu函数区域的页表代码如程序清单3.2所示。

程序清单3.2 __enable_mmu 页表的建立

```
  //r4 =页表物理地址
  //获取段描述符的默认配置flags
    ldr r7, [r10, #PROCINFO_MM_MMUFLAGS]
    adr r0, __turn_mmu_on_loc //得到__turn_mmu_on_loc 的物理地址
    ldmia  r0, {r3, r5, r6}
    sub r0, r0, r3    //计算得到物理地址与虚拟地址的偏差
    add r5, r5, r0    //修正得到__turn_mmu_on的物理地址
    add r6, r6, r0    //修正得到__turn_mmu_on_end的物理地址
    mov r5, r5, lsr #SECTION_SHIFT //1M对齐
    mov r6, r6, lsr #SECTION_SHIFT //1M对齐
  1:  orr r3, r7, r5, lsl #SECTION_SHIFT //生成段描述符：flags + 段基址
    str r3, [r4, r5, lsl #PMD_ORDER]  //设置段描述绝对映射，物理地址等于虚拟地址。每个段描述符占4字节，PMD_ORDER = 2
    cmp r5, r6
    addlo  r5, r5, #1      //下一段，实际上__turn_mmu_on_end-__turn_mmu_on<  1M
    blo 1b
  ..........................
  __turn_mmu_on_loc:
    .long  .        //__turn_mmu_on_loc当前位置的虚拟地址
```

```
    .long  __turn_mmu_on  //__turn_mmu_on的虚拟地址
    .long  __turn_mmu_on_end  //__turn_mmu_on_end的虚拟地址
```

建立内核的映射区页表，分析见程序清单3.3。

程序清单3.3 内核的映射区页表的建立

```
  //r4 =页表物理地址
  mov r3, pc           //r3 = 当前物理地址
    mov r3, r3, lsr #SECTION_SHIFT  //物理地址转化段基址
    orr r3, r7, r3, lsl #SECTION_SHIFT  //段基址 + flags = 段描述符
    //KERNEL_START = 0xC000_8000  SECTION_SHIFT = 20 PMD_ORDER =  2
    //由于arm 的立即数只能是8位表示，所有用两条指令实现了将r3存储到对应的页表项中
      add r0, r4,  #(KERNEL_START & 0xff000000) >> (SECTION_SHIFT - PMD_ORDER)
      str r3, [r0, #((KERNEL_START & 0x00f00000) >> SECTION_SHIFT) << PMD_ORDER]!
      ldr r6, =(KERNEL_END - 1)
      add r0, r0, #1 << PMD_ORDER
      add r6, r4, r6, lsr #(SECTION_SHIFT - PMD_ORDER)  //内核映射页表结束的段基址
  1:  cmp r0, r6
      add r3, r3, #1 << SECTION_SHIFT  //得到段描述符
      strls  r3, [r0], #1 << PMD_ORDER  //设置段描述符
      bls 1b
```

（4）__v6_setup

__v6_setup 函数在 proc-v6.S 文件中，在页表建立起来之后，此函数进行一些使能 MMU 之前的初始化操作。

（5）__enable_mmu

__v6_setup已经为使能 MMU做好了必要的准备，为了保证MMU启动后程序顺利返回，在进入__enable_mmu 函数之前，已经将__mmap_switched 的虚拟地址（链接地址）存储在R13中。

（6）__mmap_switched

程序运行到这里，MMU已经启动，__mmap_switched函数为内核进入C代码阶段做了一些准备工作：复制数据段，清除BSS段，设置堆栈指针，保存processor ID、machine type（bootloader中传入的）、atags pointer 等。最后，终于跳转到start_kernel 函数，进入C 代码启动阶段。

2. C代码启动阶段

在编写不带操作系统的裸机程序时，通常要在汇编阶段完成中断向量表的建立、代码的拷贝、初始化堆栈等工作后，程序跳到main函数中继续执行。start_kernel函数是执行第一个C代码的函数，被看作是内核的main函数。事实上，内核的初始化在这个阶段才真正的开始，前期仅仅给这个阶段创造了最低限度的运行环境。初始化过程从进入start_kernel到启动第一个init进程，期间调用了一系列的函数对内核初始化，输出Linux版本信息、设置与体系结构相关的环境、页表结构初始化、内存初始化、控制台初始化等。如下文所示，过程非常复杂，本文仅选取与内核的移植密切相关的setup_arch()函数进行分析。

```
start_kernel
  ...
  -> setup_arch()
    ->setup_processor()
```

1．得到处理器信息，并通过printk打印

2．初始化cacheid和CPU

->setup_machine_tags()

1．匹配bootloader传入的machine_arch_type参数

2．拷贝tags参数列表到全局变量（从物理地址拷贝到虚拟地址），并检测所有的参数是否有效

3．拷贝启动命令行参数到全局变量（从物理地址拷贝到虚拟地址）

4．返回machine_arch_type对应的machine_desc结构体

->parse_early_param()

早期处理boot_command_line参数

...

->paging_init()

建立各种类型的页表

->console_init()

初始化控制台

...

->rest_init()

启动init进程

（1）setup_processor

内核所支持的每一种处理器，均用一个proc_info_list结构来保存它的信息。结构的真正定义是在汇编代码中，在汇编启动阶段已经确认了内核支持该处理器， setup_processor 调用 lookup_processor_type 查找到存储当前处理器信息的物理地址，并把这些信息拷贝到虚拟地址。汇编启动阶段讲解函数lookup_processor_type时，分析过lookup_processor_type仅仅是一

段包装过的汇编代码。完成查找工作后打印出处理器的信息、初始化CPU。

（2）setup_machine_tags

每种处理器有着自己独特的标识以相互区分，称作CPU ID。CPU ID 固化在处理器的某个寄存器中，内核通过程序读出，自动匹配是否支持该处理器，不需要bootloader告知处理器的类型。以处理器为核心，外围器件根据不同需求配置，例如nandflash、内存、显示屏等。需要一个machinetype来标识不同的单板，bootloader有责任去确认这个值并传给内核。内核尽管已经支持该处理器，但由于不同的配置，必须根据实际情况适当地修改与单板密切相关的mach-*.c文件，修改过程称为移植，将在后面详细介绍。内核支持的所有 machine type 信息（不同单板的信息通过结构体machine_desc组织）存放在一个.arch.info.init段内，内核用bootloader 传入的machine type 与它们逐个匹配，直到找到对应的信息，否则内核进入死循环。

3.3 MTD分区

Memory Technology Device，缩写为 MTD，即内存技术设备，是Linux系统中快闪存储器转换层。创造MTD子系统的主要目的是提供一个介于快闪存储器硬件与上层应用之间的抽象接口。

因为具备以下特性，所以 MTD 设备和硬盘相较之下，处理起来要复杂许多。

（1）具有 eraseblocks 的特征，而不是像硬盘一样使用簇。

（2）eraseblocks （32KiB～128KiB） 跟硬盘的 sector size（512 bytes～1024 bytes）比起来要大很多。

（3）操作上主要分作 3 个动作：从 eraseblock 读取、写入 eraseblock 、还有就是清除 eraseblock 。

（4）坏掉的 eraseblocks 无法隐藏，需要软件加以处理。

（5）eraseblocks 的寿命大约会在 104 到 105 的清除动作之后退出。

进入arch/arm/mach-s3c64xx目录，打开mach-ok6410.c文件。可以看到MTD分区信息如下：

```
static struct mtd_partition ok6410_nand_part[] = {
  [0] = {
    .name  = "uboot",
    .size  = SZ_1M,
    .offset  = 0,
  },
  [1] = {
    .name  = "kernel",
    .size  = SZ_2M,
    .offset  = SZ_1M,
  },
  [2] = {
    .name  = "rootfs",
    .size  = MTDPART_SIZ_FULL,
    .offset  = SZ_1M + SZ_2M,
  },
};
```

linux3.8.3的源码将NandFlash划分为3个分区：前1MB用于存放u-boot，1MB后面的2MB空间之间用于存放内核，3MB之后的空间用来存放虚拟文件系统。rootfs文件系统是基于内存的文件系统，也是虚拟的

文件系统，在系统启动之后，隐藏在真正的根文件系统后面，不能被卸载。

这里需要对NandFlash重新分区，使这个MTD分区适合OK6410开发平台，也能适合当前的u-boot、内核、文件系统以及用户。修改如下：

```
static struct mtd_partition ok6410_nand_part[] = {
  [0] = {
    .name    = "Bootloader",
    .offset  = 0,
    .size    = (1 * SZ_1M),
    .mask_flags = MTD_CAP_NANDFLASH,
  },
  [1] = {
    .name    = "Kernel",
    .offset  = (1 * SZ_1M),
    .size    = (5 * SZ_1M) ,
    .mask_flags = MTD_CAP_NANDFLASH,
  },
  [2] = {
    .name  = "File System",
    .offset = (6 * SZ_1M),
    .size  = (200 * SZ_1M) ,
  },
  [3] = {
    .name  = "User",
    .offset = MTDPART_OFS_APPEND,
    .size  = MTDPART_SIZ_FULL,
```

```
        },
    };
```

将NandFlash划分成了4个MTD分区，其中0～1MB之间的空间用来存放Bootloader，也就是u-boot，1MB～6MB之间的空间用来存放Linux内核，6MB～206MB之间的空间用来存放文件系统，剩下的空间提供给用户使用。

修改完成之后，执行 make uImage 重新生成内核。将 uImage 重命名为 zImage，使用 TFTP调试内核。

```
tftp 0x50008000 zImage
bootm 0x50008000
```

由于还没有添加NandFlash驱动，所以串口输出信息暂时无法看到MTD分区信息。

本节配套视频位于光盘中“嵌入式Linux 开发实用教程视频”目录下第三章05 课（MTD分区）。

3.4 NAND Flash驱动移植

NAND Flash 硬件原理在U-Boot 章节已经讲得很详细了，这里就不再赘述了，下面直接进入Linux 系统的NAND Flash 驱动移植。

NAND Flash 驱动的很多工作由Linux内核完成，只需要稍作修改即可使用。从三星官网下载s3c_nand.c文件，并将其放入drivers/mtd/nand/中。如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/drivers/mtd/nand$ ls s3c*
s3c2410.c  s3c2410.o  s3c_nand.c
```

因为需要将其编译进入内核，那么理所应当就该修改当前目录下（ drivers/mtd/nand/）的Makefile，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/drivers/mtd/nand$ gedit Makefile

在文件任何一个位置添加如下语句：

```
obj-$(CONFIG_MTD_NAND_S3C)    += s3c_nand.o
```

添加NAND Flash的配置选项，这样在make menuconfig的时候便可随意选择是否需要NANDFlash驱动。涉及配置选项，需要修改的文件都是Kconfig。打开当前目录下（drivers/mtd/nand/）的Kconfig，在config MTD_NAND_S3C2410 后面添加NandFlash 选项，如下所示：

```
config MTD_NAND_S3C
  tristate "NAND Flash support for S3C SoC"
  depends on (ARCH_S3C64XX || ARCH_S5P64XX ||
ARCH_S5PC1XX)&& MTD_NAND
  help
    This enables the NAND flash controller on the S3C.
    No board specfic support is done by this driver,
  eachboard
     must advertise a platform_device for the driver
  toattach.
  config MTD_NAND_S3C_DEBUG
    bool "S3C NAND driver debug"
    depends on MTD_NAND_S3C
    help
      Enable debugging of the S3C NAND driver
  config MTD_NAND_S3C_HWECC
    bool "S3C NAND Hardware ECC"
```

```
  depends on MTD_NAND_S3C
  help
    Enable the use of the S3C's internal ECC
generatorwhen
    using NAND. Early versions of the chip have
hadproblems with
    incorrect ECC generation, and if using these, the
defaultof
    software ECC is preferable.
    If you lay down a device with the hardware ECC,
thenyou will
    currently not be able to switch to software, as
thereis no
    implementation for ECC method used by the S3C
```

由于 s3c_nand.c 文件的许多寄存器还没有定义，则需要在 arch/arm/plat-samsung/ include/plat/regs_nand.h文件中添加相关的寄存器定义，如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-
3.8.3/arch/arm/plat-samsung/include/plat$ gedit regs-nand.h
```

在文件的最后添加如下相关定义：

```
#if 1
//zzq-> by 2013-3-16
/* for s3c_nand.c */
#define S3C_NFCONF S3C2410_NFREG(0x00)
#define S3C_NFCONT S3C2410_NFREG(0x04)
#define S3C_NFCMMD S3C2410_NFREG(0x08)
#define S3C_NFADDR S3C2410_NFREG(0x0c)
```

```
#define S3C_NFDATA8 S3C2410_NFREG(0x10)
#define S3C_NFDATA S3C2410_NFREG(0x10)
#define S3C_NFMECCDATA0 S3C2410_NFREG(0x14)
#define S3C_NFMECCDATA1 S3C2410_NFREG(0x18)
#define S3C_NFSECCDATA S3C2410_NFREG(0x1c)
#define S3C_NFSBLK S3C2410_NFREG(0x20)
#define S3C_NFEBLK S3C2410_NFREG(0x24)
#define S3C_NFSTAT S3C2410_NFREG(0x28)
#define S3C_NFMECCERR0 S3C2410_NFREG(0x2c)
#define S3C_NFMECCERR1 S3C2410_NFREG(0x30)
#define S3C_NFMECC0 S3C2410_NFREG(0x34)
#define S3C_NFMECC1 S3C2410_NFREG(0x38)
#define S3C_NFSECC S3C2410_NFREG(0x3c)
#define S3C_NFMLCBITPT S3C2410_NFREG(0x40)
#define S3C_NF8ECCERR0 S3C2410_NFREG(0x44)
#define S3C_NF8ECCERR1 S3C2410_NFREG(0x48)
#define S3C_NF8ECCERR2 S3C2410_NFREG(0x4c)
#define S3C_NFM8ECC0 S3C2410_NFREG(0x50)
#define S3C_NFM8ECC1 S3C2410_NFREG(0x54)
#define S3C_NFM8ECC2 S3C2410_NFREG(0x58)
#define S3C_NFM8ECC3 S3C2410_NFREG(0x5c)
#define S3C_NFMLC8BITPT0 S3C2410_NFREG(0x60)
#define S3C_NFMLC8BITPT1 S3C2410_NFREG(0x64)
#define S3C_NFCONF_NANDBOOT (1<<31)
#define S3C_NFCONF_ECCCLKCON (1<<30)
#define S3C_NFCONF_ECC_MLC (1<<24)
#define S3C_NFCONF_ECC_1BIT (0<<23)
```

```
    #define S3C_NFCONF_ECC_4BIT (2<<23)
    #define S3C_NFCONF_ECC_8BIT (1<<23)
    #define S3C_NFCONF_TACLS(x) ((x)<<12)
    #define S3C_NFCONF_TWRPH0(x) ((x)<<8)
    #define S3C_NFCONF_TWRPH1(x) ((x)<<4)
    #define S3C_NFCONF_ADVFLASH (1<<3)
    #define S3C_NFCONF_PAGESIZE (1<<2)
    #define S3C_NFCONF_ADDRCYCLE (1<<1)
    #define S3C_NFCONF_BUSWIDTH (1<<0)
    #define S3C_NFCONT_ECC_ENC (1<<18)
    #define S3C_NFCONT_LOCKTGHT (1<<17)
    #define S3C_NFCONT_LOCKSOFT (1<<16)
    #define S3C_NFCONT_8BITSTOP (1<<11)
    #define S3C_NFCONT_MECCLOCK (1<<7)
    #define S3C_NFCONT_SECCLOCK (1<<6)
    #define S3C_NFCONT_INITMECC (1<<5)
    #define S3C_NFCONT_INITSECC (1<<4)
    #define S3C_NFCONT_nFCE1 (1<<2)
    #define S3C_NFCONT_nFCE0 (1<<1)
    #define S3C_NFCONT_INITECC (S3C_NFCONT_INITSECC |
S3C_NFCONT_INITMECC)
    #define S3C_NFSTAT_ECCENCDONE (1<<7)
    #define S3C_NFSTAT_ECCDECDONE (1<<6)
    #define S3C_NFSTAT_BUSY (1<<0)
    #define S3C_NFECCERR0_ECCBUSY (1<<31)
    //<-zzq
    #endif
```

这段宏定义中的#if……#endif，是为了更好地添加与注释代码。

与u-boot中的NAND Flash 一样，支持ECC校验。由于OK6410 开发平台使用的NAND Flash芯片是：K9GAG08U0D，那么在nand_base.c文件中添加支持ECC校验，操作如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/drivers/mtd/nand$ gedit nand_base.c

添加如下：

```
static struct nand_ecclayout nand_oob_218_128Bit = {
    .eccbytes = 104,
    .eccpos = {
      24,25,26,27,28,29,30,31,32,33,
      34,35,36,37,38,39,40,41,42,43,
      44,45,46,47,48,49,50,51,52,53,
      54,55,56,57,58,59,60,61,62,63,
      64,65,66,67,68,69,70,71,72,73,
      74,75,76,77,78,79,80,81,82,83,
      84,85,86,87,88,89,90,91,92,93,
      94,95,96,97,98,99,100,101,102,103,
      104,105,106,107,108,109,110,111,112,113,
      114,115,116,117,118,119,120,121,122,123,
      124,125,126,127},
  .oobfree =
  {
    {
      .offset = 2,
      .length = 22
    }
```

```
        }
    };
```

ECC校验位有104 位，即24～127。NAND Flash 制造商规定0～1这2 位为检测坏块位，2～23这22位为用户自定义检测位。

在int nand_scan_tail（struct mtd_info *mtd）函数中添加与之相对应的校验，如下所示：

```
case 218:
chip->ecc.layout = &nand_oob_218_128Bit;
break;
```

NAND Flash 的驱动代码添加完成，在linux-3.8.3/arch/arm/mach-s3c64xx 中添加NAND Flash初始化，如下所示：

```
static void __init ok6410_machine_init(void)
{
……
s3c_device_nand.name = "s3c6410-nand";
    s3c_nand_set_platdata(&ok6410_nand_info);
    s3c_fb_set_platdata(&ok6410_lcd_pdata[features.lcd_index]);
    s3c24xx_ts_set_platdata(NULL);
……
}
```

回到linux-3.8.3内核的根目录下，如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make menuconfig
```

内核配置操作如下所示：

```
Device Drivers -->
```

Memory Technology Device (MTD) support --->

NAND Device Support

进入NAND Device Support 之后，取消NAND Flash support for Samsung S3C SoCs，选择

NAND Flash support for S3C SoC 配置。如下所示：

<> NAND Flash support for Samsung S3C SoCs

<*> NAND Flash support for S3C SoC

[*] S3C NAND driver debug

[*] S3C NAND Hardware ECC

完成配置之后重新编译内核，通过TFTP进行调试，串口输出有一个错误，如下所示：

S3C NAND Driver, (c) 2008 Samsung Electronics

dev_id == 0xd5 select s3c_nand_oob_mlc

****Nandflash:ChipType= MLC ChipName=samsung-K9GAG08U0D**********

S3C NAND Driver is using hardware ECC.

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd5 (Samsung NAND 2GiB 3,3V 8-bit), page size: 4096, OOB size: 218

Driver must set ecc.strength when using hardware ECC

------------[cut here]------------

kernel BUG at drivers/mtd/nand/nand_base.c:3382!

Internal error: Oops - BUG: 0 [#1] ARM

从串口错误信息很容易可以知道错误的根源在 linux3.8.3 内核的 drivers/mtd/nand/nand_base.c:3382中，进入文件，找到根源，如下所示：

if (mtd->writesize >= chip->ecc.size) {

```
if (!chip->ecc.strength) {
pr_warn("Driver must set ecc.strength when using hardware ECC\n");
BUG();
}
break;
}
```

因为在内核配置的时候选择了硬件ECC校验，在执行上面代码的时候进入BUG()，而BUG()语句是一个死循环，内核进去之后无法出来。为了使调试通过，暂时先注释掉BUG()语句，使得内核不进入死循环。

重新编译内核，通过TFTP调试，串口输出信息为：

```
……
****Nandflash:ChipType= MLC ChipName=samsung-K9GAG08U0D**********
S3C NAND Driver is using hardware ECC.
NAND device: Manufacturer ID: 0xec, Chip ID: 0xd5 (Samsung NAND 2GiB 3,3V 8-bit), 2048MiB, page size: 4096, OOB size: 218
Driver must set ecc.strength when using hardware ECC
Creating 4 MTD partitions on "NAND 2GiB 3,3V 8-bit":
0x000000000000-0x000000100000 : "Bootloader"
0x000000100000-0x000000600000 : "Kernel"
0x000000600000-0x00000ce00000 : "File System"
0x00000ce00000-0x000080000000 : "User"
……
```

从串口的输出信息可以看到MTD 分区信息和NAND Flash 的驱动信息，便可知NAND Flash移植没有问题。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章06课（NANDFlash移植）。

3.5 DM9000网卡驱动

DM9000含有自带通用处理器接口、10M/100M物理层和16K的SRAM，DM9000详细介绍和硬件原理参考本书的2.6节U-Boot的DM9000章节。

linux-3.8.3 版本已经有 DM9000 的设备结构，在/linux-3.8.3/arch/arm/mach-s3c64xx/目录下的mach-ok6410.c文件中，如下所示：

```
static struct resource ok6410_dm9k_resource[] = {
  [0] = DEFINE_RES_MEM(S3C64XX_PA_XM0CSN1, 2),
  [1] = DEFINE_RES_MEM(S3C64XX_PA_XM0CSN1 + 4, 2),
  [2] = DEFINE_RES_NAMED(S3C_EINT(7), 1, NULL,
IORESOURCE_IRQ \
    | IORESOURCE_IRQ_HIGHLEVEL),
};
```

ok6410_dm9k_resource[0]和 ok6410_dm9k_resource[1]是访问DM9000 时使用的地址， ok6410_dm9k_resource[2]定义了DM9000使用的中断号。

```
static struct dm9000_plat_data ok6410_dm9k_pdata = {
    .flags = (DM9000_PLATF_16BITONLY |
DM9000_PLATF_NO_EEPROM),
};
```

宏定义在include/linux目录下的dm9000.h中：

```
#define DM9000_PLATF_16BITONLY (0x0002)
......
#define DM9000_PLATF_NO_EEPROM (0x0010)
```

这里定义了访问DM9000时数据宽度是16位。

```
  static struct platform_device ok6410_device_eth = {
    .name     = "dm9000",
    .id       = -1,
    .num_resources   = ARRAY_SIZE(ok6410_dm9k_resource),
    .resource     = ok6410_dm9k_resource,
    .dev       = {
      .platform_data   = &ok6410_dm9k_pdata,
    },
  };
```

ok6410_device_eth结构体给出了DM9000的相关信息。

Linux-3.8.3 版本对 DM9000 网卡的支持已经相当完美了，只需要在内核配置界面中添加DM9000的相应驱动即可。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章07课（DM9000网卡移植）。

3.6 YAFFS2根文件系统

YAFFS 的全称是Yet Another Flash File System，是由Aleph One 公司开发出来的NAND Flash 嵌入式文件系统。在YAFFS 中，最小存储单位为一个page，文件内的数据是存储在固定512 Bytes的page

中，每一个page 亦会有一个对应的16 Bytes 的Spare（OOB,Out-Of-Band）。

YAFFS1和YAFFS2的主要差异还是在于page读写 size的大小，YAFFS2可支持每page 2KB，远高于YAFFS1 的每page 512 Bytes，因此对大容量NAND Flash 更具优势。

3.6.1 使Linux-3.8.3内核支持YAFFS2文件系统

在线下载YAFFS2源码，操作如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ git clone git://www.aleph1.co.uk/yaffs2
```

如果之前没有安装git-core，则会提示先得安装git-core，安装操作如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ sudo apt-get install git-core
```

安装完成之后再次输入下载源码操作，下载过程如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ git clone git://www.aleph1.co.uk/yaffs2
Initialized empty Git repository in /home/zhuzhaoqi/Linux/linux-3.8.3/yaffs2/.git/
remote: Counting objects: 7476, done.
remote: Compressing objects: 100% (4574/4574), done.
remote: Total 7476 (delta 5920), reused 3625 (delta 2818)
Receiving objects: 100% (7476/7476), 3.54 MiB | 5 KiB/s, done.
Resolving deltas: 100% (5920/5920), done.
```

下载之后，在linux-3.8.3根目录下会有yaffs2文件夹，如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ ls

arch drivers Kbuild mm scripts virt

block firmware Kconfig security System.map

COPYING fs kernel net sound

CREDITS include lib README yaffs2

crypto init MAINTAINERS REPORTING-BUGS tools

Documentation ipc Makefile samples usr

下载完成之后，进入yaffs2目录下，有一个patch-ker.sh补丁脚本，这个脚本的功能是给内核添加yaffs2文件系统。yaffs2给linux-3.8.3内核打好补丁，操作如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2$ ./patch-ker.sh c m/home/zhuzhaoqi/Linux/linux-3.8.3

Updating /home/zhuzhaoqi/Linux/linux-3.8.3/fs/Kconfig

Updating /home/zhuzhaoqi/Linux/linux-3.8.3/fs/Makefile

补丁操作完成之后，进入linux-3.8.3/fs目录下，如果有yaffs2文件夹，就说明yaffs2添加进入linux-3.8.3内核的补丁成功。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/fs$ ls

…… yaffs2 ……

接下来进行yaffs2的内核配置，回到linux-3.8.3的根目录下，执行内核配置命令：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make menuconfig

按照如下操作进行配置：

Device Drivers --->

<*> Memory Technology Device (MTD) support --->

<*> Caching block device access to MTD devices

退回到和Device Drivers 一个目录下，完成如下操作配置：

File systems --->

[*] Miscellaneous filesystems --->

<*> yaffs2 file system support

此项yaffs2 file system support 配置选项选择“yes”是为了告诉内核支持yaffs2 文件系统。

完成之后重新编译内核。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make uImage

scripts/kconfig/conf --silentoldconfig Kconfig

CHK include/generated/uapi/linux/version.h

CHK include/generated/utsrelease.h

make[1]: “include/generated/mach-types.h”是最新的

CALL scripts/checksyscalls.sh

CHK include/generated/compile.h

CC fs/yaffs2/yaffs_ecc.o

CC fs/yaffs2/yaffs_vfs.o

CC fs/yaffs2/yaffs_guts.o

CC fs/yaffs2/yaffs_checkptrw.o

CC fs/yaffs2/yaffs_packedtags1.o

CC fs/yaffs2/yaffs_packedtags2.o

CC fs/yaffs2/yaffs_nand.o

CC fs/yaffs2/yaffs_tagscompat.o

CC fs/yaffs2/yaffs_tagsmarshall.o

CC fs/yaffs2/yaffs_mtdif.o

CC fs/yaffs2/yaffs_nameval.o

CC fs/yaffs2/yaffs_attribs.o

```
  CC  fs/yaffs2/yaffs_allocator.o
  CC  fs/yaffs2/yaffs_yaffs1.o
  CC  fs/yaffs2/yaffs_yaffs2.o
  CC  fs/yaffs2/yaffs_bitmap.o
  CC  fs/yaffs2/yaffs_summary.o
  CC  fs/yaffs2/yaffs_verify.o
  LD  fs/yaffs2/yaffs.o
  LD  fs/yaffs2/built-in.o
  LD  fs/built-in.o
  CC  drivers/mtd/mtd_blkdevs.o
  CC  drivers/mtd/mtdblock.o
  LD  drivers/mtd/built-in.o
  LD  drivers/built-in.o
  LINK  vmlinux
  LD  vmlinux.o
  MODPOST vmlinux.o
  GEN  .version
  CHK  include/generated/compile.h
  UPD  include/generated/compile.h
  CC  init/version.o
  LD  init/built-in.o
  KSYM  .tmp_kallsyms1.o
  KSYM  .tmp_kallsyms2.o
  LD  vmlinux
  SORTEX  vmlinux
sort done marker at 2f3658
  SYSMAP  System.map
```

```
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  GZIP  arch/arm/boot/compressed/piggy.gzip
  AS  arch/arm/boot/compressed/piggy.gzip.o
  LD  arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:  Linux-3.8.3
Created:  Sun Mar 17 10:15:12 2013
Image Type:  ARM Linux Kernel Image (uncompressed)
Data Size:  1723152 Bytes = 1682.77 kB = 1.64 MB
Load Address: 50008000
Entry Point:  50008040
  Image arch/arm/boot/uImage is ready
```

从输出信息可以看到，本次内核将yaffs2相关的一些文件编译进去了。通过TFTP测试，串口信息输出如下：

```
……
****Nandflash:ChipType= MLC ChipName=samsung-K9GAG08U0D**********
S3C NAND Driver is using hardware ECC.
NAND device: Manufacturer ID: 0xec, Chip ID: 0xd5 (Samsung NAND 2GiB 3,3V 8-bit), 2048MiB, page size: 4096, OOB size: 218
Driver must set ecc.strength when using hardware ECC
Creating 4 MTD partitions on "NAND 2GiB 3,3V 8-bit":
0x000000000000-0x000000100000 : "Bootloader"
```

```
0x000000100000-0x000000600000 : "Kernel"
0x000000600000-0x00000ce00000 : "File System"
0x00000ce00000-0x000080000000 : "User"
......
VFP support v0.3: implementor 41 architecture 1 part 20 variant b rev 5
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
List of all partitions:
1f00       1024 mtdblock0  (driver?)
1f01       5120 mtdblock1  (driver?)
1f02     204800 mtdblock2  (driver?)
1f03    1886208 mtdblock3  (driver?)
No filesystem could mount root, tried: cramfs
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)
......
```

从串口的输出信息可以知道内核已经支持yaffs2文件系统，下一小节将详细讲解制作根文件系统。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章08课（使内核支持YAFFS2文件系统）。

3.6.2 制作根文件系统

1. 制作mkyaffs2image工具

制作yaffs2根文件系统镜像文件需要使用到mkyaffs2image工具，mkyaffs2image工具的制作是在yaffs2源码中完成的。

进入 yaffs2 源码文件夹中（不是 linux-3.8.3/fs/yaffs2）的 utils 目录下，打开 mkyaff2image.c文件，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/utils$ ls

Makefile mkyaffs2image.c mkyaffsimage.c yutilsenv.h

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/utils$ gedit mkyaffs2image.c

mkyaffs2image.c文件的宏定义如下所示：

```
// Adjust these to match your NAND LAYOUT:
#define chunkSize 2048
#define spareSize 64
#define pagesPerBlock 64
```

但是 OK6410 开发平台使用的 NAND Flash 芯片是 K9GAG08U0D，根据 u-boot 移植部分对K9GAG08U0D详细讲解可知应该将宏定义修改为如下所示：

```
// Adjust these to match your NAND LAYOUT:
#define chunkSize  4096
#define spareSize  218
#define pagesPerBlock 128
```

进入yaffs2源码文件夹中的direct目录下，打开yportenv.h文件，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/direct$ gedit yportenv.h

增加对CONFIG_YAFFS_DEFINES_TYPES的定义，如下所示：

```
#define CONFIG_YAFFS_DEFINES_TYPES
```

保存之后，进入utils目录下执行make，生成mkyaffs2image制作工具。如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/utils$ make

编译完成之后将生成mkyaffs2image，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/utils$ ls mkyaffs2*

mkyaffs2image mkyaffs2image.c mkyaffs2image.o

将其复制到宿主机ubuntu中的/usr/bin目录下。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/yaffs2/utils$ sudo cp mkyaffs2image/usr/bin/

2. 制作根文件系统

制作根文件系统需要使用到BusyBox，BusyBox 是一个遵循GPL，以自由软件形式发布的应用程序。由于可执行文件尺寸小，并使用 Linux 内核，使得它非常适合使用于嵌入式系统。此外，由于BusyBox功能强大，因此它也被称为Linux工具里的瑞士军刀。

在http://www.busybox.net/中下载busybox1.20.2源码，再解压缩：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox$ tar -xjf busybox-1.20.2.tar.bz2

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox$ ls

busybox-1.20.2 busybox-1.20.2.tar.bz2

打开根目录下的Makefile文件，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ gedit Makefile

与Linux内核一样，开篇就定义了BusyBox的版本信息，如下所示：

VERSION = 1

PATCHLEVEL = 20

SUBLEVEL = 2

EXTRAVERSION =

NAME = Unnamed

而这个版本信息同样赋给了KERNELVERSION，如下所示：

KERNELVERSION =
$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)

由于BusyBox也需要进行编译，则需要添加CPU的类型和交叉编译链的路径，如下所示：

CROSS_COMPILE ?=

……

ARCH ?= $(SUBARCH)

添加之后为：

CROSS_COMPILE ?=/usr/local/arm/4.4.1/bin/arm-linux-

……

ARCH ?= arm

完成之后进行BusyBox的配置，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ make menuconfig

配置选项操作如下所示：

Busybox Settings --->

Build Options --->

() Cross Compiler prefix

选择Cross Compiler prefix之后，添加交叉编译链的路径：/usr/local/arm/4.4.1/bin/arm-linux-。添加之后如下：

(/usr/local/arm/4.4.1/bin/arm-linux-) Cross Compiler prefix

返回到和Build Options同一目录下，操作如下所示：

General Configuration --->

[*] Don't use /usr

配置完成之后，进行编译，操作如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ make

SPLIT include/autoconf.h -> include/config/*

GEN include/bbconfigopts.h

HOSTCC applets/usage

……

编译成功之后生产busybox文件，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ ls busybox

busybox

安装busybox，操作命令如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ make install

将默认安装在busybox-1.20.2目录下的_install文件夹下，进入_install文件夹，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ ls

bin linuxrc sbin

可以看到，已经生产bin、sbin目录和linuxrc文件。

为了根文件系统可以正常运行，现需添加其他目录，创建一个脚本 creat_yaffs2.sh 用于在_install建立根文件系统的其他目录。如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ vim creat_yaffs2.sh

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ ls

bin creat_yaffs2.sh linuxrc sbin

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ gedit creat_yaffs2.sh

脚本如下所示：

```
#!/bin/sh
echo "=== zhuzhaoqi ==="
echo "=== jxlgzzq@163.com ==="
echo "=== Create yaffs2 other ..... ==="
mkdir root dev etc bin sbin mnt sys proc lib home tmp var usr
mkdir usr/sbin usr/bin usr/lib usr/modules usr/etc
mkdir mnt/usb mnt/nfs mnt/etc mnt/etc/init.d
mkdir lib/modules
chmod 777 tmp
sudo mknod -m 600 dev/console c 5 1
sudo mknod -m 666 dev/null c 1 3
echo "=== Created ==="
```

保存之后执行脚本，如下所示：

shzhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$creat_yaffs2.sh

```
=== zhuzhaoqi ===
=== jxlgzzq@163.com ===
=== Create yaffs2 other ..... ===
[sudo] password for zhuzhaoqi:
=== Created ===
```

这信息说明已经创建完毕，查看busybox-1.20.2/_install目录：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ ls

bin dev home linuxrc proc sbin tmp var

creat_yaffs2.sh etc lib mnt root sys usr

逐个对_install目录下的文件进行分析。

（1）/bin目录

本目录下存放着系统管理用户和一般用户都能使用的基本命令，进入bin目录下，使用ls便可看到这些常用的命令，如：cat、chmod、cp、echo、ip、ls、mv、mkdir、vi 等。/bin 目录下的命令在挂接其他文件系统之前就能使用，所以/bin目录必须和根文件系统存放在同一个分区中。

（2）/dev目录

本目录下存放的是设备文件。设备文件有两种：字符设备和块设备。在Linux设备驱动中，加载驱动模块使用mknod创建设备文件，如下所示：

[YJR@zhuzhaoqi /]# mknod /dev/led c 240 0

通过“/dev/led”便可操作led驱动，c是表示字符驱动，240是主设备号，0是次设备号。进入/dev目录下，查看现有的设备文件：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/dev$ ll

总用量 8

drwxr-xr-x 2 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 ./

drwxr-xr-x 15 zhuzhaoqi zhuzhaoqi 4096 2013-03-17 16:33 ../

crw------- 1 root root 5, 1 2013-03-17 16:33 console

crw-rw-rw- 1 root root 1, 3 2013-03-17 16:33 null

（3）/etc目录

本目录下存放着很多配置文件。系统中的部分程序都需要这些配置文件。

a）profile文件

新建profile文件，为系统设定环境变量。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ vim profile

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ geditprofile

内容如下：

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

b）fstab文件

本文件用来指明当执行“mount -a”时，需要挂接的文件系统。新建fstab 文件。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ vim fstabzhuzhaoqi@zhuzhaoqi-

desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ gedit fstab

内容如下：

proc /proc proc defaults 0 0

none /tmp ramfs defaults 0 0

none /var ramfs defaults 0 0

mdev /dev ramfs defaults 0 0

sysfs /sys sysfs defaults 0 0

c）inittab文件

本文件是init进程的配置文件。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ vim inittab

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$geditinittab

内容如下：

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

::askfirst:-/bin/sh

::ctrlaltdel:/bin/umount -a -r

::shutdown:/bin umount -a -r

::shutdown:/sbin/swapoff -a

d）init.d子目录

建立init.d文件夹，并在init.d文件夹下面创建rcS文件。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ mkdir init.d

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc$ cd init.d/

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc/init.d$ vim rcS

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc/init.d$gedit rcS

rcS文件内容如下：

```
#! /bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin:
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel
#
# Trap CTRL-C &c only in this shell so we can interrupt subprocesses.
#
trap ":" INT QUIT TSTP
/bin/hostname hcm
/bin/mount -n -t proc none /proc
/bin/mount -n -t sysfs none /sys
/bin/mount -n -t usbfs none /proc/bus/usb
/bin/mount -t ramfs none /dev
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s
/bin/hotplug
# mounting file system specified in /etc/fstab
mkdir -p /dev/pts
mkdir -p /dev/shm
```

```
/bin/mount -n -t devpts none /dev/pts -o mode=0622
/bin/mount -n -t tmpfs tmpfs /dev/shm
/bin/mount -n -t ramfs none /tmp
/bin/mount -n -t ramfs none /var
mkdir -p /var/empty
mkdir -p /var/log
mkdir -p /var/lock
mkdir -p /var/run
mkdir -p /var/tmp
/sbin/hwclock -s -f /dev/rtc
syslogd
/etc/rc.d/init.d/netd start
echo " " > /dev/tty1
echo "Starting networking..." > /dev/tty1
[U2]echo "************************************"
echo " Welcome to zzq Root! "
echo " "
echo " Name : zhuzhaoqi "
echo " E-mali : jxlgzzq@163.com"
echo "************************************"
mkdir /mnt/disk
mount -t yaffs2 /dev/mtdblock3 /mnt/disk
mount -t vfat /dev/mmcblk0p1 /home/
mount -t yaffs2 /dev/mtdblock3 /mnt/
cd /mnt/
tar zxvf /home/urbetter-rootfs-qt-2.2.0.tgz
sync
```

```
cd /
umount /mnt/
umount /home/
/sbin/ifconfig lo 127.0.0.1
chmod +x etc/init.d/ifconfig-eth0
/etc/init.d/ifconfig-eth0
/bin/qtopia &
echo " " > /dev/tty1
echo "Starting Qtopia, please waiting..." > /dev/tty1
echo " "
echo "Starting Qtopia, please waiting..."
```

为了确保一般用户可以执行，给予rcS最大权限。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/etc/init.d$chmod 777 rcS

（4）/home目录

本目录是用户目录。

（5）/lib目录

本目录是标准程序设计库，又叫动态链接共享库。当然，加载模块也是放在这个目录下。

（6）linuxrc文件

本文件是系统启动时在内存盘执行的启动脚本，用于加载模块驱动等，运行在init进程启动之前，它退出后内核才会调用init进程读取inittab中的设置。查看linuxrc，如下所示：

lrwxrwxrwx 1 zhuzhaoqi zhuzhaoqi 11 2013-03-17 15:34 linuxrc -> bin/busybox*

（7）/mnt目录

/mnt目录是系统管理员临时安装（mount）文件系统的安装点。程序并不自动支持安装到/mnt 。/mnt 下面可以分为许多子目录，例如/mnt/dosa 可能是使用 msdos 文件系统的软驱，而/mnt/exta 可能是使用ext2文件系统的软驱，/mnt/cdrom光驱等。

（8）/proc目录

本目录是常作为proc文件系统的挂接点。同时系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口。

（9）/root目录

超级用户（通常所说的root用户）的主目录。

（10）/sbin目录

本目录下存放着基本的系统命令，即只有管理员能够使用的命令，如：chroot、fdisk、insmod、reboot、poweroff等，这些命令用于启动系统、修复系统等。/sbin目录和/bin目录相似，在挂接其他文件系统之前就可以使用/sbin，所以/sbin目录和根文件系统也是存在于一个分区中。

（11）/sys目录

本目录作为虚拟文件系统（sysfs，类似于/proc，一个procfs），它存储且允许修改连接到系统的设备，然而许多传统UNIX和类UNIX操作系统使用/sys作为内核代码树的符号链接。

（12）/tmp目录

本目录下存放着临时文件，在系统重启时目录中文件不会被保留。

（13）/usr目录

用于存储只读用户数据的第二层次；包含绝大多数的用户工具和应用程序。在之前创建脚本creat_yaffs2.sh中，已经为/usr创建了子目录，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/usr$ ls

bin etc lib modules sbin

a）/bin子目录

多数日常应用程序存放的位置。如果/usr被放在单独的分区中，Linux的单用户模式不能访问/usr/bin，所以对系统至关重要的程序不应放在此文件夹中。

b）/etc子目录

除了/etc目录下之外的配置文件，比如网络配置等。新建mdev.conf空文件和init文件。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/usr/etc$ vim init

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/usr/etc$ gedit init

内容如下：

#!/bin/sh

ifconfig eth0 192.168.1.0 up

ifconfig lo 127.0.0.1

同时给予init最高权限，使得所有用户可以执行。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/usr/etc$ chmod777 init

c）/lib子目录

本子目录存放着系统的库文件。

d）/sbin子目录

本子目录存放着在单用户模式中不用的系统管理程序。

（14）/var目录

本目录存放着一些数据文件，如系统日志等。

为/lib目录添加相应工具链中的库文件，操作如下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install$ cp/usr/local/arm/4.4.1/arm-none-linux-gnueabi/libc/lib/*so* lib/

拷贝完成之后可以查看工具链的库文件。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2/_install/lib$ ls

用mkyaffs2image工具制作根文件系统：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ mkyaffs2image _installrootfs.yaffs

制作完成之后将生成rootfs.yaffs2：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/BusyBox/busybox-1.20.2$ ls -l rootfs.yaffs2

-rw------- 1 zhuzhaoqi zhuzhaoqi 14356992 2013-04-16 21:45 rootfs.yaffs2

3. 内核配置

制作根文件系统之后，对内核进行相关的基本配置。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make menuconfig

配置操作如下：

Device Drivers --->

Generic Driver Options --->

[*] Maintain a devtmpfs filesystem to mount at /dev

[*] Automount devtmpfs at /dev, after the kernel mounted···

回到Device Drivers 同一目录下，进行如下配置操作：

File systems --->

<> Second extended fs support

<> Ext3 journalling file system support

同时在同一目录下进行如下操作：

[*] Miscellaneous filesystems --->

<*> BeOS file system (BeFS) support(read only) (EXPERIMENTAL)

[ ] Debug BeFS

回到Device Drivers、File systems同一目录下，进行如下配置操作：

Boot options --->

进入之后输入启动参数：noinitrd root=/dev/mtdblock2 rootfstype=yaffs2 init=/linuxrc console=ttySAC0,115200。

(0) Compressed ROM boot loader BSS address

(noinitrd root=/dev/mtdblock2 rootfstype=yaffs2 init=/linuxrc console=ttySAC0,115200)

完成上述操作之后保存退出。

编译内核，将新生成的内核及文件系统烧写入OK6410开发平台，运行，即可启动。

3.6.3 NFS文件系统挂载

NFS 即为网络文件系统，全称为Network File System，一种是用于分散式文件系统的协定。

NFS的功能是通过网络让不同的机器、不同的操作系统能够彼此分享个别的数据，使得应用程序能在客户端通过网络访问位于服务器磁盘中的数据，是在UNIX系统间实现磁盘文件共享的一种方法。

在ubuntu宿主机上配置NFS

之前没有使用过NFS的宿主机，是没有安装NFS服务器。首先安装NFS服务器。

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo apt-get install portmap

……

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo apt-get install nfs-kernel-server

在当前目录下建立rootfs目录，以作为ubuntu宿主机和OK6410开发平台共享目录。

zhuzhaoqi@zhuzhaoqi-desktop:~$ mkdir rootfs

同时给予rootfs最大权限。

zhuzhaoqi@zhuzhaoqi-desktop:~$ chmod 777 rootfs

安装完成NFS服务器之后为NFS挂载的文件系统添加目录途径：

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo vim /etc/exports

在/etc/exports文件的末尾添加如下代码，使得文件系统能在宿主机和OK6410开发平台之间共享。

/home/zhuzhaoqi/rootfs *(rw,sync,no_root_squash)

/home/zhuzhaoqi/rootfs：ubuntu宿主机与OK6410开发平台的共享目录；

*：代表允许所有的网络段访问这个共享目录；

rw：代表具有可读可写的权限；

sync：资料同步写入内存和硬盘；

no_root_squash：是NFS客户端分享目录使用者的权限，如果客户端使用的是root用户，那么对于该共享目录而言，该客户端就具有root权限。

添加完成之后保存退出，使得/etc/exports文件生效。

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo exportfs -rv

接着启动端口映射：

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo /etc/init.d/portmap start

或：zhuzhaoqi@zhuzhaoqi-desktop:~$sudo service portmap start

最后启动NFS服务器：

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo/etc/init.d/nfs-kernel-server restart

启动NFS服务器也可使用如下命令：

zhuzhaoqi@zhuzhaoqi-desktop:~$sudo service nfs-kernel-server restart

由于上面的操作都得在root用户下方有权限执行，所以要加sudo。此时NFS搭建完成。将busybox/_install目录下的所有文件（即根文件系统）复制到/rootfs目录下（即为NFS共享目录）。

启动OK6410开发板，停留在U-Boot中设置启动参数：

set bootargs root=/dev/nfs console=ttySAC0,115200 nfsroot=192.168.1.187:/ home/ zhuzhaoqi/rootfs ip=192.168.1.100:192.168.1.187:192.168.1.1:255.255.255.0::eth0:off

对上面的启动参数进行进一步分析。

nfsroot=192.168.1.187:/home/zhuzhaoqi/rootfs

这是NFS文件系统存放在ubuntu中的地址，192.168.1.187是ubuntu的IP，这个依据ubuntu的IP地址而不同。

在u-boot启动时刻，按下任意键，修改OK6410开发平台的相关IP参数：

OK6410的IP:

zzq6410 >>> set ipaddr 192.168.1.100

OK6410的网关：

```
zzq6410 >>> set gatewayip 192.168.1.1
```

ubuntu服务地址：

```
zzq6410 >>> set serverip 192.168.1.187
```

OK6410的子网掩码：

```
zzq6410 >>> set netmask 255.255.255.0
```

设置完成之后，重新启动开发板。

```
  U-Boot 2012.10 (Dec 17 2012 - 09:33:49) for OK6410
  Author :  zhuzhaoqi
  E-mail : jxlgzzq@163.com
  CPU:  S3C6410@533MHz
    Fclk = 533MHz, Hclk = 133MHz, Pclk = 66MHz (ASYNC
Mode)
  Board:  OK6410
  DRAM:  256 MiB
  NAND:  2048 MiB
  In:  serial
  Out:  serial
  Err:  serial
  Net:  dm9000
  Unknown command 'mtdparts' - try 'help'
  Hit any key to stop autoboot:  0
  NAND read: device 0 offset 0x100000, size 0x500000
    5242880 bytes read: OK
  ## Booting kernel from Legacy Image at 50008000 ...
    Image Name:  Linux-3.8.3
    Image Type:  ARM Linux Kernel Image (uncompressed)
```

```
   Data Size:  2151736 Bytes = 2.1 MiB
   Load Address: 50008000
   Entry Point:  50008040
   Verifying Checksum ... OK
   XIP Kernel Image ... OK
OK
Starting kernel ...
Starting kernel ...
Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0x0
Linux version 3.8.3 (zhuzhaoqi@zhuzhaoqi-desktop) (gcc
version 4.4.1 (Sourcery G++ Lite2009q3-67) ) #5 Mon Apr 1
22:46:56 CST 2013
CPU: ARMv6-compatible processor [410fb766] revision 6
(ARMv7), cr=00c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing
instruction cache
Machine: OK6410
Memory policy: ECC disabled, Data cache writeback
CPU S3C6410 (id 0x36410101)
.....
Kernel    command    line:
console=ttySAC0,115200root=/dev/nfsnfsroot=192.168.1.187:/hom
e/zhuzhaoqi/rootfs
    ip=192.168.1.100:192.168.1.187:192.168.1.1:255.255.255.
0::eth0:off
   PID hash table entries: 1024 (order: 0, 4096 bytes)
```

Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)

Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)

__ex_table already sorted, skipping sort

Memory: 256MB = 256MB total

Memory: 255404k/255404k available, 6740k reserved, 0K highmem

Virtual kernel memory layout:

vector : 0xffff0000 - 0xffff1000 (4 kB)

DTCM: 0xfffe8000 - 0xfffec000 (16 kB)

ITCM: 0xfffe0000 - 0xfffe4000 (16 kB)

fixmap : 0xfff00000 - 0xfffe0000 (896 kB)

vmalloc : 0xd0800000 - 0xff000000 (744 MB)

lowmem : 0xc0000000 - 0xd0000000 (256 MB)

modules : 0xbf000000 - 0xc0000000 (16 MB)

.text : 0xc0008000 - 0xc03c0e98 (3812 kB)

.init : 0xc03c1000 - 0xc03e0c8c (128 kB)

.data : 0xc03e2000 - 0xc0419dc0 (224 kB)

.bss : 0xc041a000 - 0xc04529ac (227 kB)

SLUB: Genslabs=13, HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1

NR_IRQS:246

VIC @f6000000: id 0x00041192, vendor 0x41

VIC @f6010000: id 0x00041192, vendor 0x41

sched_clock: 32 bits at 100 Hz, resolution 10000000ns, wraps every 4294967286ms

Console: colour dummy device 80x30

Calibrating delay loop... 353.89 BogoMIPS (lpj=1769472)

pid_max: default: 32768 minimum: 301

Mount-cache hash table entries: 512

CPU: Testing write buffer coherency: ok

Setting up static identity map for 0x502c91c8 - 0x502c9224

NET: Registered protocol family 16

DMA: preallocated 256 KiB pool for atomic coherent allocations

OK6410: Option string ok6410=0

OK6410: selected LCD display is 480x272

s3c64xx_dma_init: Registering DMA channels

PL080: IRQ 73, at d0846000, channels 0..8

PL080: IRQ 74, at d0848000, channels 8..16

S3C6410: Initialising architecture

bio: create slab <bio-0> at 0

usbcore: registered new interface driver usbfs

usbcore: registered new interface driver hub

usbcore: registered new device driver usb

NET: Registered protocol family 2

TCP established hash table entries: 2048 (order: 2, 16384 bytes)

TCP bind hash table entries: 2048 (order: 3, 40960 bytes)

TCP: Hash tables configured (established 2048 bind 2048)

TCP: reno registered

```
    UDP hash table entries: 256 (order: 1, 12288 bytes)
    UDP-Lite hash table entries: 256 (order: 1, 12288 bytes)
    NET: Registered protocol family 1
    RPC: Registered named UNIX socket transport module.
    RPC: Registered udp transport module.
    RPC: Registered tcp transport module.
    RPC: Registered tcp NFSv4.1 backchannel transport module.
    NFS: Registering the id_resolver key type
    Key type id_resolver registered
    Key type id_legacy registered
    Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
    ROMFS MTD (C) 2007 Red Hat, Inc.
    io scheduler noop registered
    io scheduler deadline registered
    io scheduler cfq registered (default)
    Console: switching to colour frame buffer device 60x34
    s3c-fb s3c-fb: window 0: fb
    Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
    s3c6400-uart.0: ttySAC0 at MMIO 0x7f005000 (irq = 69) is
a S3C6400/10
    console [ttySAC0] enabled
    s3c6400-uart.1: ttySAC1 at MMIO 0x7f005400 (irq = 70) is
a S3C6400/10
    s3c6400-uart.2: ttySAC2 at MMIO 0x7f005800 (irq = 71) is
a S3C6400/10
    s3c6400-uart.3: ttySAC3 at MMIO 0x7f005c00 (irq = 72) is
a S3C6400/10
```

brd: module loaded

loop: module loaded

dm9000 Ethernet Driver, V1.31

dm9000 dm9000: eth%d: Invalid ethernet MAC address. Please set using ifconfig

eth0: dm9000a at d08de000,d08e0004 IRQ 108 MAC: ca:56:76:fc:fc:d6 (random)

ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver

s3c2410-ohci s3c2410-ohci: S3C24XX OHCI

s3c2410-ohci s3c2410-ohci: new USB bus registered, assigned bus number 1

s3c2410-ohci s3c2410-ohci: irq 79, io mem 0x74300000

s3c2410-ohci s3c2410-ohci: init err (00000000 0000)

s3c2410-ohci s3c2410-ohci: can't start s3c24xx

s3c2410-ohci s3c2410-ohci: startup error -75

s3c2410-ohci s3c2410-ohci: USB bus 1 deregistered

s3c2410-ohci: probe of s3c2410-ohci failed with error -75

mousedev: PS/2 mouse device common for all mice

i2c /dev entries driver

sdhci: Secure Digital Host Controller Interface driver

sdhci: Copyright(c) Pierre Ossman

s3c-sdhci s3c-sdhci.0: clock source 0: mmc_busclk.0 (133250000 Hz)

s3c-sdhci s3c-sdhci.0: clock source 2: mmc_busclk.2 (24000000 Hz)

mmc0: SDHCI controller on samsung-hsmmc [s3c-sdhci.0] using ADMA

s3c-sdhci s3c-sdhci.1: clock source 0: mmc_busclk.0 (133250000 Hz)

s3c-sdhci s3c-sdhci.1: clock source 2: mmc_busclk.2 (24000000 Hz)

mmc0: mmc_rescan_try_freq: trying to init card at 400000 Hz

mmc0: mmc_rescan_try_freq: trying to init card at 300000 Hz

mmc1: SDHCI controller on samsung-hsmmc [s3c-sdhci.1] using ADMA

mmc0: mmc_rescan_try_freq: trying to init card at 200000 Hz

usbcore: registered new interface driver usbhid

usbhid: USB HID core driver

TCP: cubic registered

NET: Registered protocol family 17

Key type dns_resolver registered

VFP support v0.3: implementor 41 architecture 1 part 20 variant b rev 5

drivers/rtc/hctosys.c: unable to open rtc device (rtc0)

mmc0: mmc_rescan_try_freq: trying to init card at 100000 Hz

dm9000 dm9000 eth0: link down

mmc1: mmc_rescan_try_freq: trying to init card at 400000 Hz

mmc1: mmc_rescan_try_freq: trying to init card at 300000 Hz

```
    mmc1: mmc_rescan_try_freq: trying to init card at 200000
Hz
    mmc1: mmc_rescan_try_freq: trying to init card at 100000
Hz
    IP-Config: Complete:
        device=eth0, hwaddr=ca:56:76:fc:fc:d6,
    ipaddr=192.168.1.100, mask=255.255.255.0,gw=192.168.1.1
        host=192.168.1.100, domain=, nis-domain=(none)
        bootserver=192.168.1.187, rootserver=192.168.1.187,
    rootpath=dm9000 dm9000 eth0: link up, 100Mbps, full-
    duplex, lpa 0x4DE1
    VFS: Mounted root (nfs filesystem) on device 0:9.
    Freeing init memory: 124K
    mount: mounting none on /proc/bus/usb failed: No such
file or directory
    hwclock: can't open '/dev/rtc': No such file or directory
    /etc/rc.d/init.d/netd: line 16: /usr/sbin/inetd: not
found
    mkdir: cannot create directory '/mnt/disk': File exists
    zhuzhaoqi >>> Starting Qtopia4.4.3, please waiting...
    Please press Enter to activate this console.
    [YJR@zhuzhaoqi]\# ls
    Applications  dev    linuxrc    root    tmp
    Packages  empty    mnt    sbin    udisk
    Settings  etc    opt    sdcard    usr
    bin    lib    proc    sys    var
    [YJR@zhuzhaoqi]\#
```

进入字符界面可知已经通过NFS挂载文件系统成功。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章09课（制作YAFFS2文件系统）。

3.7 LCD驱动移植

OK6410的标准配置LCD型号为WXCAT43，4.3寸电阻触摸屏，分辨率为480*272。WXCAT43型号LCD属于TFT，即薄膜晶体管。WXCAT43的屏幕响应时间小于80ms，其画面色彩饱和度、真实效果和对比度都有较高优势，广泛应用于手机、MP4等移动产品中。

OK6410所标配的LCD已经集成为一个模块，实现了WXCAT43与单板之间接口的转换。单板上与WXCAT43型号LCD模块接口如图3.10所示。

1、2号引脚为3.3V电源。

3～10号这8个引脚为RED数据线；12～19号这8个引脚为GREEN数据线；21～28号这8个引脚为BLUE数据线。

目前大多数LCD显示器都是采用RGB颜色模式，LCD屏幕上的所有颜色都由这红色、绿色、蓝色3种色光按照不同的比例混合而成的。一组红绿蓝颜色就是一个最小的显示单位。屏幕上的任何一个颜色都可以由一组RGB值来记录和表达。前面已经提到WXCAT43的分辨率为480*272，那也就是屏幕每一行有480个像素点、每一列有272个像素点，整个屏幕的像素点有130560个。这里的每一个像素点都是一组RGB值。

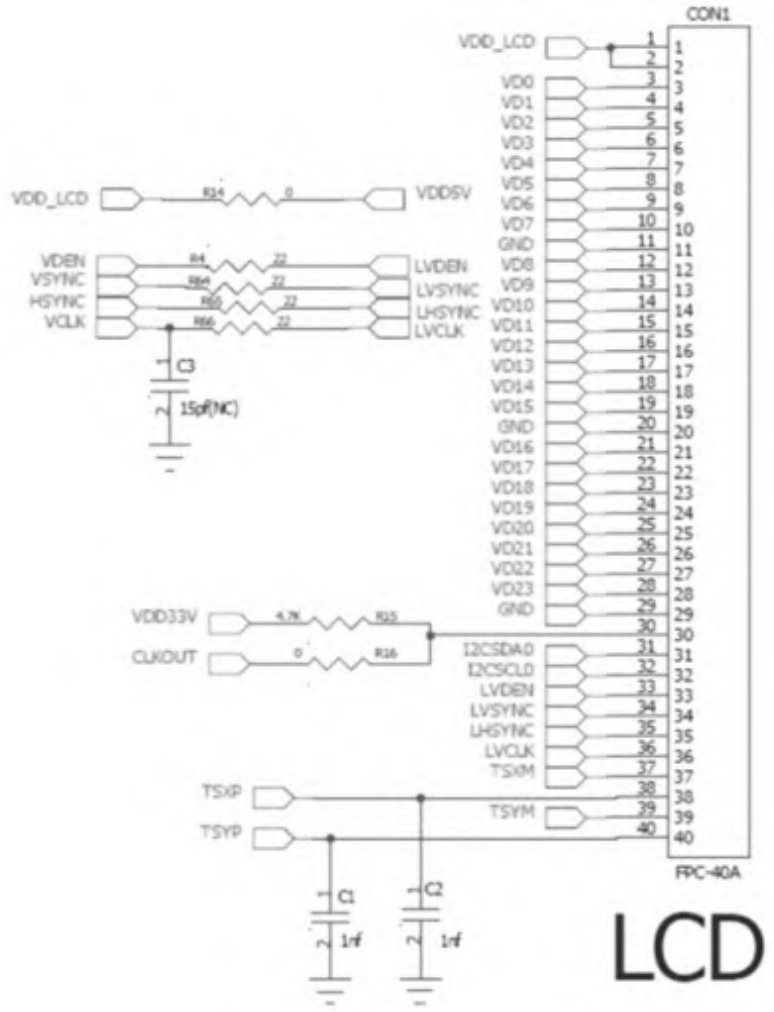


图3.10 LCD模块接口原理图

RED的值用0到255表示红色由浅到深，同理GREEN和BLUE也是使用0到255表示绿色和蓝色的由浅到深。那么一组RGB值即由（R,G,B）构成。如单板给3～10的值为1111 1111，则此刻RED的值255。GREEN和BLUE同理。

31、32号引脚为I2C总线控制。

33、34、35、36号这4个引脚为LCD图像使能频率等控制。其中LVDEN引脚为像素点使能，像素点是否显示在屏幕上在于这个引脚是否使能；LVSYNC引脚为垂直同步信号，该信号有效一次，则刷新一帧像素点；LHSYNC引脚为行同步信号，该信号有效一次，刷新一行像素点；LVCLK为刷新频率。

37、38、39、40号这4个引脚是“四线触摸”控制信号线，都是ADC采样。电阻式触摸屏传感器将矩形区域中触摸点（X,Y）的物理位置转换为代表X坐标和Y坐标的电压。电阻式触摸屏可以用四线产生屏幕偏置电压，同时读回触摸点的电压用以确定触摸点位置。

初步了解LCD原理之后，进入LCD驱动移植阶段，移植分为显示和触摸两部分。

3.7.1 LCD显示驱动

在Linux内核中已经有相关的LCD驱动程序。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$ geditmach-ok6410.c

在mach-ok6410.c文件中有：

```
static struct s3c_fb_pd_win ok6410_lcd_type0_fb_win = {
    .max_bpp = 32,
    .default_bpp = 16,
    .xres    = 480,
    .yres    = 272,
};
static struct fb_videomode ok6410_lcd_type0_timing = {
    /* 4.3" 480x272 */
    .left_margin = 3,
    .right_margin= 2,
    .upper_margin= 1,
    .lower_margin= 1,
    .hsync_len  = 40,
    .vsync_len  = 1,
```

```
    .xres     = 480,
    .yres     = 272,
  };
```

标配4.3 寸触摸屏分辨率为480*272，所以s3c_fb_pd_win ok6410_lcd_type0_fb_win 结构体参数无需修改。

fb_videomode ok6410_lcd_type0_timing 结构体中的相关参数声明如下：

```
__u32 pixclock;/*像素时钟*/
__u32 left_margin; /*行切换，从同步到绘图之间的延迟*/
__u32 right_margin; /*行切换，从绘图到同步之间的延迟*/
__u32 upper_margin; /*帧切换，从同步到绘图之间的延迟*/
__u32 lower_margin; /*帧切换，从绘图到同步之间的延迟*/
__u32 hsync_len;/*水平同步的长度*/
__u32 vsync_len;/*垂直同步的长度*/
```

HBP（horizontal back porch）：表示从水平同步信号开始到一行的有效数据开始之间的VCLK的个数，对应驱动中的left_margin；

HFP（horizontal front porth）：表示一行的有效数据结束到下一个水平同步信号开始之间的VCLK的个数，对应驱动中的right_margin；

VBP（vertical back porch）：表示在一帧图像开始时，垂直同步信号以后的无效的行数，对应驱动中的upper_margin；

VFB（vertical front porch）：表示在一帧图像结束后，垂直同步信号以前的无效的行数，对应驱动中的lower_margin；

HSPW（horizontal sync pulse width）：表示水平同步信号的宽度，用VCLK计算，对应驱动中的hsync_len；

VSPW（vertical sync pulse width）：表示垂直同步脉冲的宽度，用行数计算，对应驱动中的vsync_len。

由于启动OK6410开发平台会发现，LCD屏幕画面向上移动了。LCD常见的几种问题有以下几个。

（1）刷新频率不正常

现象：屏幕像流动瀑布一样有明显向下刷新的光条。

原因：LCD的时钟频率设置不对。

（2）整体颜色出现反色

现象：原本应该为红色却变成蓝色，原本为蓝色却变成红色。

原因：三原色RGB数据中R、G、B排列顺序有误。

（3）图像整体水平偏移

现象：LCD画面整体左右偏移，与下一个图像之间出现一个黑色竖条纹。

原因：行同步信号的时序参数不对，需要调整。

（4）图像整体上下偏移

现象：LCD画面整体上下偏移，与下一个图像之间出现一个黑色横条纹。

原因：帧同步信号的时序参数不对，需要调整。

（5）图像只显示在上半部分

现象：LCD画面只显示上半部分，但是下半部分未显示。

原因：显示缓冲区长度设置有误，造成只显示部分数据。

（6）显示缓冲区数据错误

现象：LCD屏幕出现随机的竖条纹。

原因：显示缓冲区是乱码，或者显存数据没有及时更新。

对fb_videomode ok6410_lcd_type0_timing 结构体成员进行修改，如下所示：

```
static struct fb_videomode ok6410_lcd_type0_timing = {
  /* 4.3" 480x272 */
#if 0
```

```
    .left_margin  = 3,
    .right_margin  = 2,
    .upper_margin  = 1,
    .lower_margin  = 1,
    .hsync_len  = 40,
    .vsync_len  = 1,
    .xres    = 480,
    .yres    = 272,
  #endif
    .left_margin  = 3,
    .right_margin  = 5,
    .upper_margin  = 3,
    .lower_margin  = 3,
    .hsync_len  = 42,
    .vsync_len  = 12,
    .xres    = 480,
    .yres    = 272,
};
```

同样在mach-ok6410.c添加支持LCD驱动结构体，如下所示：

```
  static struct map_desc ok6410_iodesc[] = {
    {
      /* LCD support */
      .virtual  = (unsigned long)S3C_VA_LCD,
      .pfn    = __phys_to_pfn(S3C_PA_FB),
      .length  = SZ_16K,
      .type    = MT_DEVICE,
    },
```

```
};
```

由于S3C_VA_LCD没有宏定义，在arch/arm/plat-samsung/include/plat/map-base.h中添加：

```
#define S3C_VA_LCD  S3C_ADDR(0x01100000)  /* LCD */
```

在mach-ok6410.c 的static void __init ok6410_map_io（void）函数中添加LCD 初始化：

```
static void __init ok6410_map_io(void)
  {
  ……
  #if 0
    s3c64xx_init_io(NULL, 0);
  #endif
    s3c64xx_init_io(ok6410_iodesc,
ARRAY_SIZE(ok6410_iodesc));
  ……
  }
```

在/linux-3.8.3/drivers/video/目录下添加samsung文件夹，samsung目录下有：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-
3.8.3/drivers/video/samsung$ ls
built-in.o    s3cfb_AT056.o  s3cfb_fimd4x.c
s3cfb.o      s3cfb_VGA800.os3cfb_WXCAT43.o
Kconfig      s3cfb_AT070.c  s3cfb_fimd4x.o
s3cfb_spi.c    s3cfb_WXCAT35.cs3cfb_XGA1024.c
Makefile      s3cfb_AT070.o  s3cfb_fimd5x.c
s3cfb_spi.o    s3cfb_WXCAT35.os3cfb_XGA1024.o
```

s3cfb_AT056.c s3cfb.c s3cfb.h s3cfb_VGA800.c s3cfb_WXCAT43.c

将regs-lcd.h和regs-fb.h拷贝到/linux-3.8.3/arch/arm/mach-s3c64xx/include/mach目录下，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx/include/mach$ ls regs-fb.h regs-lcd.h

regs-fb.h regs-lcd.h

将regs-fb-v4.h、regs-fb.h文件拷贝到/linux-3.8.3/arch/arm/plat-samsung/include/plat目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/plat-samsung/include/plat$lsregs-fb-v4.h

regs-fb-v4.h

在/linux-3.8.3/drivers/video/目录下的Kconfig添加：

source "drivers/video/samsung/Kconfig"

并且在/linux-3.8.3/drivers/video/目录下添加：

obj-$(CONFIG_FB_S3C_EXT) += samsung/

如果此时编译的话，编译信息输出会提示drivers/video/samsung/s3cfb_fimd4x.c 中的s3c6410_pm_do_save函数和s3c6410_pm_do_restore函数隐式声明。做如下修改：

```
int s3cfb_suspend(struct platform_device *dev, pm_message_t state)
{
……
#if 0
s3c6410_pm_do_save(s3c_lcd_save, ARRAY_SIZE(s3c_lcd_save));
```

```
    #endif
    s3c_pm_do_save(s3c_lcd_save, ARRAY_SIZE(s3c_lcd_save));
    ……
    }
    int s3cfb_resume(struct platform_device *dev)
    {
    ……
    #if 0
    s3c6410_pm_do_restore(s3c_lcd_save,
ARRAY_SIZE(s3c_lcd_save));
    #endif
    s3c_pm_do_restore(s3c_lcd_save,
ARRAY_SIZE(s3c_lcd_save));
    ……
    }
```

修改完成之后执行make menuconfig，进行LCD 显示内核配置。

```
  Device Drivers --->
    Graphics support --->
      <*> Support for frame buffer devices --->
        <> Samsung S3C framebuffer support
  Device Drivers --->
    Graphics support --->
      <*> S3C Framebuffer Support (eXtended)
        Select LCD Type (4.3 inch 480x272 TFT LCD) --->
          (X) 4.3 inch 480x272 TFT LCD
  Device Drivers --->
    Graphics support --->
```

```
        <*> Advanced options for S3C Framebuffer
          Select BPP(Bits Per Pixel) (16 BPP) --->
            (X) 16 BPP
    Device Drivers --->
      Graphics support --->
        <*> Advanced options for S3C Framebuffer
        [*]Enable Double Buffering
    Device Drivers --->
      Graphics support --->
    Console display driver support --->
      <*> Framebuffer Console support
    [] Map the console to the primary display device
```

这样已经将LCD屏幕的显示移植完成，接下来需要完成LCD屏幕的触摸移植。

3.7.2 LCD触摸驱动

打开mach-ok6410.c：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$ vimmach-ok6410.c

在mach-ok6410.c中添加ts.h头文件：

#include <mach/ts.h>

将dev-ts.c文件拷贝至/linux-3.8.3/arch/arm/mach-s3c64xx/目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx$ ll dev-ts.c

-rwxr-xr-x 1 zhuzhaoqi zhuzhaoqi 1552 2013-04-14 00:14 dev-ts.c*

同时在dev-ts.c文件中添加：

#include <linux/gfp.h>

将ts.h文件拷贝至/linux-3.8.3/arch/arm/mach-s3c64xx/include/mach目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/mach-s3c64xx/include/mach$ ll ts.h

-rwxr-xr-x 1 zhuzhaoqi zhuzhaoqi 916 2013-04-14 00:15 ts.h*

在/linux-3.8.3/arch/arm/mach-s3c64xx/目录下的Makefile文件中添加：

obj-$(CONFIG_TOUCHSCREEN_S3C)   += dev-ts.o

进入mach-ok6410.c文件添加触摸初始化数据：

```
static struct s3c_ts_mach_info s3c_ts_platform __initdata = {
        .delay      = 10000,
        .presc      = 49,
        .oversampling_shift  = 2,
        .resol_bit    = 12,
        .s3c_adc_con    = ADC_TYPE_2,
};
```

并且在mach-ok6410.c文件中添加初始化函数：

```
static void __init ok6410_machine_init(void)
{
……
#if 0
```

```
        s3c24xx_ts_set_platdata(NULL);
    #endif
        s3c_ts_set_platdata(&s3c_ts_platform);
    ……
    }
```

将s3c-ts.c文件拷贝至/linux-3.8.3/drivers/input/touchscreen/目录下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/drivers/input/touchscreen$    lss3c-ts.c
s3c-ts.c
```

在/linux-3.8.3/drivers/input/touchscreen/目录下的Makefile添加：

```
obj-$(CONFIG_TOUCHSCREEN_S3C)    += s3c-ts.o
```

在/linux-3.8.3/arch/arm/plat-samsung/include/plat/目录下添加 s3c-ts.c 文件所需的 ADC 部分控制寄存器的宏定义：

```
/*----------------- Common definitions for S3C ----------
------*/
/* The following definitions will be applied to S3C24XX,
S3C64XX, S5PC1XX.
*/
/*----------------------------------------------------------
------*/
#define S3C_ADCREG(x)         (x)
#define S3C_ADCCON          S3C_ADCREG(0x00)
#define S3C_ADCTSC          S3C_ADCREG(0x04)
#define S3C_ADCDLY          S3C_ADCREG(0x08)
#define S3C_ADCDAT0          S3C_ADCREG(0x0C)
```

```
#define S3C_ADCDAT1             S3C_ADCREG(0x10)
#define S3C_ADCUPDN             S3C_ADCREG(0x14)
#define S3C_ADCCLRINT           S3C_ADCREG(0x18)
#define S3C_ADCMUX             S3C_ADCREG(0x1C)
#define S3C_ADCCLRWK           S3C_ADCREG(0x20)
/* ADCCON Register Bits */
#define S3C_ADCCON_RESSEL_10BIT     (0x0<<16)
#define S3C_ADCCON_RESSEL_12BIT     (0x1<<16)
#define S3C_ADCCON_ECFLG        (1<<15)
#define S3C_ADCCON_PRSCEN        (1<<14)
#define S3C_ADCCON_PRSCVL(x)        (((x)&0xFF)<<6)
#define S3C_ADCCON_PRSCVLMASK     (0xFF<<6)
#define S3C_ADCCON_SELMUX(x)        (((x)&0x7)<<3)
#define S3C_ADCCON_SELMUX_1(x)     (((x)&0xF)<<0)
#define S3C_ADCCON_MUXMASK        (0x7<<3)
#define S3C_ADCCON_RESSEL_10BIT_1     (0x0<<3)
#define S3C_ADCCON_RESSEL_12BIT_1     (0x1<<3)
#define S3C_ADCCON_STDBM        (1<<2)
#define S3C_ADCCON_READ_START     (1<<1)
#define S3C_ADCCON_ENABLE_START     (1<<0)
#define S3C_ADCCON_STARTMASK        (0x3<<0)
/* ADCTSC Register Bits */
#define S3C_ADCTSC_UD_SEN       (1<<8)
#define S3C_ADCTSC_YM_SEN       (1<<7)
#define S3C_ADCTSC_YP_SEN       (1<<6)
#define S3C_ADCTSC_XM_SEN       (1<<5)
#define S3C_ADCTSC_XP_SEN       (1<<4)
```

```
#define S3C_ADCTSC_PULL_UP_DISABLE  (1<<3)
#define S3C_ADCTSC_AUTO_PST      (1<<2)
#define S3C_ADCTSC_XY_PST(x)      (((x)&0x3)<<0)
/* ADCDAT0 Bits */
#define S3C_ADCDAT0_UPDOWN      (1<<15)
#define S3C_ADCDAT0_AUTO_PST      (1<<14)
#define S3C_ADCDAT0_XY_PST      (0x3<<12)
#define S3C_ADCDAT0_XPDATA_MASK    (0x03FF)
#define S3C_ADCDAT0_XPDATA_MASK_12BIT  (0x0FFF)
/* ADCDAT1 Bits */
#define S3C_ADCDAT1_UPDOWN      (1<<15)
#define S3C_ADCDAT1_AUTO_PST      (1<<14)
#define S3C_ADCDAT1_XY_PST      (0x3<<12)
#define S3C_ADCDAT1_YPDATA_MASK    (0x03FF)
#define S3C_ADCDAT1_YPDATA_MASK_12BIT  (0x0FFF)
```

在/Linux/linux-3.8.3/include/linux目录下的interrupt.h文件添加：

```
#define IRQF_SAMPLE_RANDOM  0x00000040
```

并在/linux-3.8.3/drivers/input/touchscreen/目录下的Kconfig添加LCD触摸配置：

```
config TOUCHSCREEN_S3C
    tristate "S3C touchscreen driver"
    depends on ARCH_S3C2410 || ARCH_S3C64XX ||
ARCH_S5P64XX || ARCH_S5PC1XX
    default y
    help
```

Say Y here to enable the driver for the touchscreen on the
S3C SMDK board.
If unsure, say N.
To compile this driver as a module, choose M here: the
module will be called s3c_ts.

修改完成，执行make menuconfig，进行LCD 触摸配置：

```
Device Drivers --->
  Input device support --->
    [*] Touchscreens --->
      <*> S3C touchscreen driver
System Type --->
  [*] ADC common driver support
Device Drivers --->
  Input device support --->
    <*> Event interface
```

LCD 的显示和触摸配置完成之后执行make uImage 命令：

```
……
  LDvmlinux.o
arch/arm/plat-samsung/built-in.o:(.data+0x878): multiple definition of 's3c_device_ts'
arch/arm/mach-s3c64xx/built-in.o:(.data+0x34e0): first defined here
……
```

出现多重定义错误，则在/linux-3.8.3/arch/arm/plat-samsung目录下的 devs.c 注释掉s3c_device_ts即可：

//--->zzq

#undef CONFIG_SAMSUNG_DEV_TS

//<---zzq

#ifdef CONFIG_SAMSUNG_DEV_TS

如果还有错误，则可根据错误追溯源头进行修改。

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3$ make uImage

……

Image Name: Linux-3.8.3

Created: Sun Apr 14 01:47:28 2013

Image Type: ARM Linux Kernel Image (uncompressed)

Data Size: 2156336 Bytes = 2105.80 kB = 2.06 MB

Load Address: 50008000

Entry Point: 50008040

Image arch/arm/boot/uImage is ready

将生成的uImage拷贝至tftp目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/Linux/linux-3.8.3/arch/arm/boot$ cp uImage /tftpboot/

使用tftp进行烧写、调试内核。

zhuzhaoqi@zhuzhaoqi-desktop:/tftpboot$ ls

uImage

zhuzhaoqi@zhuzhaoqi-desktop:/tftpboot$ tftp

tftp>

启动ok6410开发平台，停在U-Boot烧写内核：

zzq6410 >>> tftp 0x50008000 uImage

dm9000 i/o: 0x18000300, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 00:40:5c:26:0a:5b

operating at 100M full duplex mode

Using dm9000 device

TFTP from server 192.168.1.187; our IP address is 192.168.1.100

Filename 'uImage'.

Load address: 0x50008000

Loading:

##
##
###
##
############################
##
############################
##
############################

done

Bytes transferred = 2156400 (20e770 hex)

烧写完成之后启动内核：

zzq6410 >>>bootm 0x50008000

Booting kernel from Legacy Image at 50008000 ...

Image Name: Linux-3.8.3

Image Type: ARM Linux Kernel Image (uncompressed)

Data Size:2156336 Bytes = 2.1 MiB

Load Address: 50008000

Entry Point: 50008040

```
    Verifying Checksum ... OK
    XIP Kernel Image ... OK
OK
Starting kernel ...
Starting kernel ...
Uncompressing Linux... done, booting the kernel.
......
S3C_LCD clock got enabled :: 133.250 Mhz
LCD TYPE :: LTE480WV will be initialized
......
S3C Touchscreen driver, (c) 2008 Samsung Electronics
S3C TouchScreen got loaded successfully : 12 bits
input: S3C TouchScreen as /devices/virtual/input/input0
......
```

从串口的信息输出可知LCD的显示和触摸驱动成功。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第三章10课（LCD移植）。

第4章 Linux设备驱动程序设计

4.1 设备驱动概述

Linux系统将设备分成3种基本类型：字符设备、块设备、网络接口。

（1）字符设备

字符设备是一个能够像字节流一样被访问的设备，字符终端（/dev/console）和串口（/dev/ttys0）就是两个字符设备。字符设备可以通过文件系统节点来访问，比如/dev/tty1和/dev/lp0等。这些设备文件和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个只能顺序访问的数据通道。

（2）块设备

块设备和字符设备相类似，块设备也是通过/dev目录下的文件系统节点来进行访问的。在大多数UNIX系统中，进行I/O操作时块设备每次只能传输一个或多个完整的块；在Linux系统中，应用程序可以像字符设备一样地读写块设备，允许一次传递任意多字节的数据。块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核及驱动程序之间的接口。

（3）网络接口

任何网络事务都是经过一个网络接口形成的，即一个能够和其他主机交换数据的设备。网络接口由内核中的网络子系统驱动，负责发送和接收数据包，但它不需要了解每项事物如何映射到实际传送的数据包。

4.2 字符设备驱动

Linux 操作系统将所有的设备都会看成是文件，因此当我们需要访问设备时，都是通过操作文件的方式进行访问。对字符设备的读写是以字节为单位进行的。

对字符设备驱动程序的学习过程，主要以两个具有代表性且在OK6410开发平台可实践性的字符驱动展开分析，分别为LED驱动程序、ADC驱动程序。

4.2.1 LED驱动程序设计

为了展现LED的裸板程序和基于Linux系统的LED驱动程序的区别与减少难度梯度，在写LED驱动程序之前很有必要先看一下LED的裸板程序是怎样设计的。

1．LED裸板程序

OK6410开发平台中有4个LED灯，原理图如图4.1所示。

图4.1 LED原理图

从图4.1中可知，4个LED采用的是共阳极连接方式， GPM0～GPM3分别控制着 LED1～LED4。而 GPMCON 寄存器地址为：0x7F008820；GPMDAT寄存器地址为：0x7F008824。那么GPM中3个寄存器宏定义为：

```
/*=========================================================
========
**  基地址的定义
=========================================================
======*/
#define  AHB_BASE  (0x7F000000)
/*********************************************************
********
**  GPX的地址定义
*********************************************************
*******/
#define  GPX_BASE  (AHB_BASE+0x08000)
……
/*********************************************************
********
**    GPM寄存器地址定义
```

```
    ************************************************************
*******/
    #define  GPMCON    (*(volatile unsigned long *)(GPX_BASE
+ 0x0820))
    #define  GPMDAT    (*(volatile unsigned long *)(GPX_BASE
+ 0x0824))
    #define  GPMPUD    (*(volatile unsigned long *)(GPX_BASE
+ 0x0828))
```

将GPM0～GPM3设置为输出功能：

```
    /* GPM0,1,2,3 设为输出引脚 */
    /*
    ** 每一个GPXCON 的引脚有 4 位二进制进行控制
    **  0000-输入  0001-输出
    */
    GPMCON = 0x1111;
```

点亮LED1，则是让GPM3～GPM0输出：1110。

```
    GPMDAT = 0x0e;
```

点亮LED3，则是让GPM3～GPM0输出：1011。

```
    GPMDAT = 0x0b;
```

2. LED驱动程序

有了LED裸板程序的基础，那么移植到Linux系统LED驱动设备程序的难度也不会很大了。但是在Linux中，特别注意《s3c6410用户手册》提供的GPM寄存器地址不能直接用于Linux中。

在一般情况下，Linux系统中，进程的4GB（2^{32} ）内存空间被划分成为两个部分：用户空间（3G）和内核空间（1GB），大小分别为0～3GB和3～4GB，如图4.2所示。

在 3～4GB 之间的内核空间中，从低地址到高地址依次为：系统物理内存映射区、VMALLOC_OFFSET、vmalloc 用来分配物理地址非连续的内存空间、8KB 隔离带、高端内存永久映射区、高端内存固定映射区。

在通常情况下，进程只能访问用户空间的虚拟地址，不能访问内核空间。

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。而内核空间是由内核负责映射的，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，内核的虚拟空间独立于其他程序。

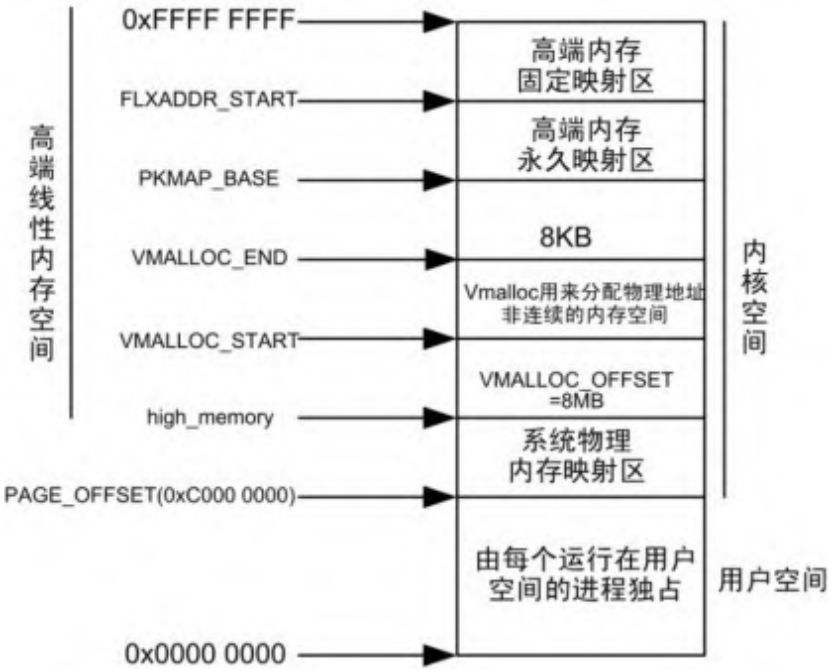


图4.2 Linux内存空间

在内核中，访问I/O内存之前，我们只有I/O内存的物理地址，这样是无法通过软件直接访问的，需要首先用ioremap()函数将设备所处的物理地址映射到内核虚拟地址空间（3GB～4GB）。然后才能根据映射所得到的内核虚拟地址范围，通过访问指令访问这些I/O内存资源。

一般来说，在系统运行时，外设的I/O内存资源的物理地址是已知的，由硬件的设计决定。但是CPU通常并没有为这些已知的外设I/O内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问I/O内存资源，而必须将它们映射到核心虚拟地址空间内（通过页表），然后才能根据映射所得到的核心虚拟地址范围，通过访内指令访问这些I/O内存资源。Linux在io.h头文件中声明了函数ioremap()，用来将I/O内存资源的物理地址映射到核心虚拟地址空间（3GB～4GB）中，如下所示：

```
void * ioremap(unsigned long phys_addr, unsigned long size,
unsigned long flags);
```

iounmap函数用于取消ioremap()所做的映射，如下所示：

```
void iounmap(void * addr);
```

到这里应该明白，像GPMCON（0x7F00 8820）这个物理地址是不能直接操控的，必须通过映射到内核的虚拟地址中，才能进行操作。

现在开始设计第一个LED驱动程序。

字符驱动程序所要包含的头文件主要位于 include/linux及/arch/arm/mach-s3c64xx/include/mach目录下，如下LED驱动程序所包含的头文件：

```
/*
* head file
*/
//moudle.h 包含了大量加载模块需要的函数和符号的定义
#include <linux/module.h>
//kernel.h以便使用printk()等函数
#include <linux/kernel.h>
//fs.h 包含常用的数据结构，如struct file 等
```

```
#include <linux/fs.h>
//uaccess.h 包含copy_to_user()、copy_from_user()等函数
#include <linux/uaccess.h>
//io.h 包含inl()、outl()、readl()、writel()等I/O 操作函数
#include <linux/io.h>
#include <linux/miscdevice.h>
#include <linux/pci.h>
//init.h来指定你的初始化和清理函数，例如：
module_init(init_function)、module_exit(cleanup_function)
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/gpio.h>
//irq.h中断与并发请求事件
#include <asm/irq.h>
//下面这些头文件是I/O口在内核的虚拟映射地址，涉及I/O口的操作所必须包含的
//#include <mach/gpio.h>
#include <mach/regs-gpio.h>
#include <plat/gpio-cfg.h>
#include <mach/hardware.h>
#include <mach/map.h>
```

上面所列出的头文件即是本次LED驱动程序所需要包含的头文件。

```
#define DEVICE_NAME  "led"
#define LED_MAJOR    240        /*主设备号*/
```

这是LED驱动程序的驱动名称和主设备号。

设备节点位于/dev目录下，如下所示，例举出了ubuntu系统/dev/vcs*的设备节点：

```
zhuzhaoqi@zhuzhaoqi-desktop:～$ ls -l /dev/vcs*
……
crw-rw---- 1 root tty 7, 7 2013-04-09 20:56 /dev/vcs7
crw-rw---- 1 root tty 7, 128 2013-04-09 20:56 /dev/vcsa
……
```

/dev/vcs7设备节点的主设备号为：7，次设备号为：7；/dev/vcsa设备节点的主设备号为：7，次设备号为：128。

```
#define LED_ON    0
#define LED_OFF    1
```

这是LED灯打开或者关闭的宏定义，由于OK6410开发平台的4个LED是共阳连接，所以输出1即为熄灭LED，输出0为点亮LED。

字符驱动程序中实现了open、close、read、write等系统调用。

open函数指针的声明位于fs.h的file_operations结构体中，如下所示：

```
struct file_operations {
    ……
    int (*open) (struct inode *, struct file *);
    ……
};
```

open函数指针的回调函数led_open()完成的任务是设置GPM的输出模式。

```
  static int led_open(struct inode *inode,struct file
*file)
    {
      unsigned int i;
```

```
        /*设置GPM0～GPM3为输出模式*/
        for (i = 0; i < 4; i++)
        {
          s3c_gpio_cfgpin(S3C64XX_GPM(i),S3C_GPIO_OUTPUT);
      printk("The GPMCON %x is %x
  \n",i,s3c_gpio_getcfg(S3C64XX_GPM(i)) );
        }
        printk("Led open... \n");
        return 0;
      }
```

s3c_gpio_cfgpin()函数原型位于gpio-cfg.h中，如下：

```
extern int s3c_gpio_cfgpin(unsigned int pin, unsigned int to);
```

内核对这个函数是这样注释的：s3c_gpio_cfgpin()函数用于改变引脚的 GPIO 功能。参数pin是GPIO的引脚名称，参数to是需要将GPIO这个引脚设置成为的功能。

GPIO的名称在arch/arm/mach-s3c6400/include/mach/gpio.h进行了宏定义：

```
/* S3C64XX GPIO number definitions. */
#define S3C64XX_GPA(_nr) (S3C64XX_GPIO_A_START + (_nr))
#define S3C64XX_GPB(_nr) (S3C64XX_GPIO_B_START + (_nr))
#define S3C64XX_GPC(_nr) (S3C64XX_GPIO_C_START + (_nr))
#define S3C64XX_GPD(_nr) (S3C64XX_GPIO_D_START + (_nr))
#define S3C64XX_GPE(_nr) (S3C64XX_GPIO_E_START + (_nr))
#define S3C64XX_GPF(_nr) (S3C64XX_GPIO_F_START + (_nr))
#define S3C64XX_GPG(_nr) (S3C64XX_GPIO_G_START + (_nr))
#define S3C64XX_GPH(_nr) (S3C64XX_GPIO_H_START + (_nr))
```

```
#define S3C64XX_GPI(_nr) (S3C64XX_GPIO_I_START + (_nr))
#define S3C64XX_GPJ(_nr) (S3C64XX_GPIO_J_START + (_nr))
#define S3C64XX_GPK(_nr) (S3C64XX_GPIO_K_START + (_nr))
#define S3C64XX_GPL(_nr) (S3C64XX_GPIO_L_START + (_nr))
#define S3C64XX_GPM(_nr) (S3C64XX_GPIO_M_START + (_nr))
#define S3C64XX_GPN(_nr) (S3C64XX_GPIO_N_START + (_nr))
#define S3C64XX_GPO(_nr) (S3C64XX_GPIO_O_START + (_nr))
#define S3C64XX_GPP(_nr) (S3C64XX_GPIO_P_START + (_nr))
#define S3C64XX_GPQ(_nr) (S3C64XX_GPIO_Q_START + (_nr))
```

S3C64XX_GPIO_M_START的定义如下：

```
enum s3c_gpio_number {
    S3C64XX_GPIO_A_START = 0,
    S3C64XX_GPIO_B_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_A),
    S3C64XX_GPIO_C_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_B),
    S3C64XX_GPIO_D_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_C),
    S3C64XX_GPIO_E_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_D),
    S3C64XX_GPIO_F_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_E),
    S3C64XX_GPIO_G_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_F),
    S3C64XX_GPIO_H_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_G),
```

```
    S3C64XX_GPIO_I_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_H),
    S3C64XX_GPIO_J_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_I),
    S3C64XX_GPIO_K_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_J),
    S3C64XX_GPIO_L_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_K),
    S3C64XX_GPIO_M_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_L),
    S3C64XX_GPIO_N_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_M),
    S3C64XX_GPIO_O_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_N),
    S3C64XX_GPIO_P_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_O),
    S3C64XX_GPIO_Q_START =
S3C64XX_GPIO_NEXT(S3C64XX_GPIO_P),
};
```

S3C64XX_GPIO_NEXT的定义：

```
#define S3C64XX_GPIO_NEXT(__gpio) \
    ((__gpio##_START) + (__gpio##_NR) +
CONFIG_S3C_GPIO_SPACE + 1)
```

宏定义一层一层很多，但是通过这个设置，可以很方便地选择想要的任何一个GPIO口进行操作。

GPIO功能设置位于gpio-cfg.h中：

```
#define S3C_GPIO_SPECIAL_MARK (0xfffffff0)
```

```
#define S3C_GPIO_SPECIAL(x)  (S3C_GPIO_SPECIAL_MARK | (x))
/* Defines for generic pin configurations */
#define S3C_GPIO_INPUT(S3C_GPIO_SPECIAL(0))
#define S3C_GPIO_OUTPUT  (S3C_GPIO_SPECIAL(1))
#define S3C_GPIO_SFN(x)  (S3C_GPIO_SPECIAL(x))
```

通过上面的宏定义可知，GPIO 的引脚功能有输入、输出，和你想要的任何可以实现的功能设置，S3C_GPIO_SFN（x）这个函数即是通过设定x的值，实现任何存在功能的设置。如果要设置GPM0～GPM3为输出功能，则：

```
for (i = 0; i < 4; i++) {
s3c_gpio_cfgpin(S3C64XX_GPM(i),S3C_GPIO_OUTPUT);
}
```

通过这样的操作，设置就显得比较简洁实用。

```
s3c_gpio_getcfg(S3C64XX_GPM(i))
```

这行代码的作用是获取GMP（argv）的当前值。这个函数的原型在include/linux/gpio.h中：

```
static inline void gpio_get_value(unsigned int gpio)
{
    __gpio_get_value(gpio);
}
```

完成端口模式设定，接下来的程序是完成LED操作。在fs.h的file_operations结构体中，有unlocked_ioctl函数指针的声明，如下所示：

```
struct file_operations {
……
```

```
    long (*unlocked_ioctl) (struct file *,unsigned
  int,unsigned long);
  ……
  };
```

unlocked_ioctl函数指针所要回调的led_ioctl()函数即是需要实现应用层对LED1～LED4的控制操作。

```
    static long led_ioctl ( struct file *file, unsigned int
  cmd, \
      unsigned long argv )
    {
      if (argv > 4) {
        return -EINVAL;
      }
    printk("LED ioctl... \n");
    /* 获取应用层的操作 */
      switch(cmd) {
    /* 如果是点亮LED(argv) */
      case LED_ON:
        gpio_set_value(S3C64XX_GPM(argv),0);
        printk("LED ON \n");
    printk( "S3C64XX_GPM(i) =
  %x\n",gpio_get_value(S3C64XX_GPM(argv)) );
      return 0;
    /* 如果是熄灭LED(argv) */
      case LED_OFF:
        gpio_set_value(S3C64XX_GPM(argv),1);
        printk("LED OFF \n");
```

```
        printk( "S3C64XX_GPM(i) = %x
    \n",gpio_get_value(S3C64XX_GPM(argv)) );
        return 0;
      default:
        return -EINVAL;
      }
    }
```

本函数调用了GPIO端口值设定函数。

```
gpio_set_value(S3C64XX_GPM(argv),1);
```

这是设定GMP（argv）输出为1。函数的原型位于include/linux/gpio.h中：

```
static inline void gpio_set_value(unsigned int gpio, int value)
{
    __gpio_set_value(gpio, value);
}
```

release 函数指针所要回调的函数led_release ()函数：

```
static int led_release(struct inode *inode,struct file *file)
{
    printk("zhuzhaoqi >>> s3c6410_led release \n");
    return 0;
}
```

这是驱动程序的核心控制，各个函数指针所对应的回调函数：

```
struct file_operations led_fops = {
    .owner    = THIS_MODULE,
    .open     = led_open,
```

```
    .unlocked_ioctl = led_ioctl,
    .release    = led_release,
};
```

由于 Linux3.8.3 内核中没有 ioctl 函数指针，取而代之的是unlocked_ioctl 函数指针实现对led_ioctl()函数的回调。

驱动程序的加载分为静态加载和动态加载，将驱动程序编译进内核称为静态加载，将驱动程序编译成模块，使用时再加载称为动态加载。动态加载模块的扩展名为：.ko，使用 insmod 命令进行加载，使用rmmod命令进行卸载。

```
static int __init led_init(void)
{
  int rc;
  printk("LEDinit... \n");
  rc = register_chrdev(LED_MAJOR,"led",&led_fops);
  if (rc < 0)
  {
    printk("register %s char dev error\n","led");
    return -1;
  }
  printk("OK!\n");
  return 0;
}
```

_init修饰词对内核是一种暗示，表明该初始化函数仅仅在初始化期间使用，在模块装载之后，模块装载器就会将初始化函数释放掉，这样就能将初始化函数所占用的内存释放出来以作他用。

当使用insmod命令加载LED驱动模块时，led_init()初始化函数将被调用，向内核注册LED驱动程序。

```
static void __exit led_exit(void)
{
    unregister_chrdev(LED_MAJOR,"led");
    printk("LED exit...\n");
}
```

_exit这个修饰词告诉内核这个退出函数仅仅用于模块卸载，并且仅仅能在模块卸载或者系统关闭时被调用。

当使用rmmod 命令卸载LED驱动模块时，led_exit ()清除函数将被调用，向内核注册LED驱动程序。

```
module_init(led_init);
module_exit(led_exit);
```

module_init 和 module_exit 是强制性使用的，这个宏会在模块的目标代码中增加一个特殊的段，用于说明函数所在位置。如果没有这个宏，则初始化函数和退出函数永远不会被调用。

```
MODULE_LICENSE("GPL");
```

如果没有声明LICENSE，模块被加载时，会给出处理内核被污染（kernel taint）的警告。如果在zzq_led.c中没有许可证（LICENSE），则会给出如下提示：

```
[YJR@zhuzhaoqi 3.8.3]# insmod zzq_led.ko
zzq_led: module license 'unspecified' taints kernel.
Disabling lock debugging due to kernel taint
```

Linux遵循GNU通用公共许可证（GPL），GPL是由自由软件基金会为GNU项目设计，它允许任何人对其重新发布甚至销售。

当然，也许程序还会有驱动程序的作者和描述信息：

```
MODULE_AUTHOR("zhuzhaoqi jxlgzzq@163.com");
MODULE_DESCRIPTION("OK6410(S3C6410) LED Driver");
```

完成驱动程序的设计之后，将 zzq_led.c 驱动程序放置于/drivers/char 目录下，打开 Makefile文件：

zhuzhaoqi@zhuzhaoqi-desktop:～/Linux/linux-3.8.3/drivers/char$ gedit Makefile

在Makefile中添加LED驱动：

obj-m += zzq_led.o

回到内核的根目录执行make modules 命令生成LED 驱动模块：

zhuzhaoqi@zhuzhaoqi-desktop:～/Linux/linux-3.8.3$ make modules

……

CC [M] drivers/char/zzq_led.o

……

编译完成之后在/drivers/char 目录下会生成 zzq_led.ko 模块，将其拷贝到文件系统下面的/lib/modules/3.8.3（如果没有3.8.3目录，则建立）目录下。

加载LED驱动模块：

[YJR@zhuzhaoqi]\# cd lib/module/3.8.3/

[YJR@zhuzhaoqi]\# ls

zzq_led.ko

[YJR@zhuzhaoqi]\# insmod zzq_led.ko

LED init...

OK!

根据信息输出可知加载zzq_led.ko驱动模块成功。通过lsmod查看加载模块：

[YJR@zhuzhaoqi]\# lsmod

zzq_led 1548 0 - Live 0xbf000000

在/dev目录下建立设备文件，进行如下操作：

[YJR@zhuzhaoqi]\# mknod /dev/led c 240 0

是否建立成功，可以查看/dev下的节点得知：

[YJR@zhuzhaoqi]\# ls /dev/l*

/dev/led /dev/log /dev/loop-control

说明LED设备文件已经成功建立。

3. LED应用程序

驱动程序需要应用程序对其操控。程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define  LED_ON  0
#define  LED_OFF  1
/*
* LED 操作说明信息输出
*/
  void usage(char *exename)
  {
    printf("How to use: \n");
    printf("  %s <LED Number><on/off> \n", exename);
    printf("  LED Number = 1, 2, 3 or 4 \n");
  }
  /*
  * 应用程序主函数
```

```
*/
int main(int argc, char *argv[])
{
  unsigned int led_number;
  if (argc != 3) {
    goto err;
  }
  int fd = open("/dev/led",2,0777);
  if (fd < 0) {
    printf("Can't open /dev/led \n");
    return -1;
  }
  printf("open /dev/led ok ... \n");
  led_number = strtoul(argv[1], 0, 0) - 1;
  if (led_number > 3) {
    goto err;
}
  /* LED ON */
  if (!strcmp(argv[2], "on")) {
    ioctl(fd, LED_ON, led_number);
  }
  /* LED OFF */
  else if (!strcmp(argv[2], "off")) {
    ioctl(fd, LED_OFF, led_number);
  }
  else {
    goto err;
```

```
    }
    close(fd);
    return 0;
  err:
    if (fd > 0) {
      close(fd);
    }
    usage(argv[0]);
    return -1;
  }
```

在main()函数中，涉及了open()函数，其原型如下：

```
int open( const char * pathname,int flags, mode_t mode);
```

当然，很多open函数中的入口参数也只有2个，原型如下：

```
int open( const char * pathname, int flags);
```

第一个参数pathname是一个指向将要打开的设备文件途径的字符串。

第二个参数flags是打开文件所能使用的旗标，常用的几种旗标有：

O_RDONLY：以只读方式打开文件

O_WRONLY：以只写方式打开文件

O_RDWR：以可读写方式打开文件

上述3种常用的旗标是互斥使用，但可与其他的旗标进行或运算符组合。

第3个参数mode是使用该文件的权限。比如777、755等。

通过这个应用程序实现对LED驱动程序的控制，为了更加方便快捷地编译这个应用程序，为其写一个Makefile文件，如下所示：

```
#交叉编译链安装路径
```

```
CC = /usr/local/arm/4.4.1/bin/arm-linux-gcc
zzq_led_app:zzq_led_app.o
  $(CC) -o zzq_led_appzzq_led_app.o
zzq_led_app.o:zzq_led_app.c
  $(CC) -c zzq_led_app.c
clean :
  rm zzq_led_app.o zzq_led_app
```

执行Makefile之后会生成zzq_led_app可执行应用文件，如下：

```
zhuzhaoqi@zhuzhaoqi-desktop:～/LDD/linux-3.8.3/zzq_led$ make
/usr/local/arm/4.4.1/bin/arm-linux-gcc -c zzq_led_app.c
/usr/local/arm/4.4.1/bin/arm-linux-gcc -o zzq_led_app zzq_led_app.o
zhuzhaoqi@zhuzhaoqi-desktop:～/LDD/linux-3.8.3/zzq_led$ ls
Makefile zzq_led_app zzq_led_app.c zzq_led_app.o zzq_led.c
```

将生成的zzq_led_app可执行应用文件拷贝到根文件系统的/usr/bin目录下，执行应用文件，如下操作：

```
[YJR@zhuzhaoqi]\# ./zzq_led_app
How to use:
    ./zzq_led_app <LED Number><on/off>
    LED Number = 1, 2, 3 or 4
```

根据信息提示可以进行对LED驱动程序的控制，点亮LED1，则如下：

```
[YJR@zhuzhaoqi]\# ./zzq_led_app 1 on
The GPMCON 0 is fffffff1
```

```
The GPMCON 1 is fffffff1
The GPMCON 2 is fffffff1
The GPMCON 3 is fffffff1
zhuzhaoqi >>> LED open...
LED ioctl...
LED ON
S3C64XX_GPM(i) = 0
LED release...
open /dev/led ok ...
```

此时可以看到LED1点亮。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第四章01课（字符设备驱动之LED）。

4.2.2 ADC驱动程序设计

A/D转换即是将模拟量转换为数字量，在物联网迅速发展的今天，作为物联网的感知前端传感器也随之迅速更新，压力、温度、湿度等众多模拟信号的处理都需要涉及A/D转换，因此A/D驱动程序在学习嵌入式中占据着重要地位。

1. S3C6410的ADC控制寄存器简介

S3C6410控制芯片自带有4路独立专用A/D转换通道，如图4.3所示。

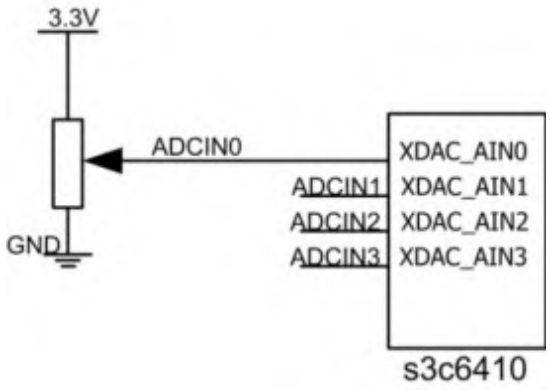


图4.3 A/D转换连接图

通过三星公司提供的《S3C6410 用户手册》可知，ADCCON 为ADC控制寄存器，地址为：0x7E00 B0000。ADCCON 的复位值为：0x3FC4，即为：0011 1111 1100 0100。

```
#define S3C_ADCREG(x)        (x)
#define S3C_ADCCON       S3C_ADCREG(0x00)
```

ADCCON控制寄存器具有16位，每一位都能通过赋值来实现其相对应的功能。

ADCCON[0]：ENABLE_START，A/D 转换开始启用。如果READ_START启用，这个值是无效的。ENABLE_START = 0，无行动；ENABLE_START = 1，A/D 转换开始和该位被清理后开启。ADCCON[0]的复位值为0，即复位之后默认为无行动。

```
#define S3C_ADCCON_NO_ENABLE_START     (0<<0)
#define S3C_ADCCON_ENABLE_START  (1<<0)
```

ADCCON[1]：READ_START，A/D 转换开始读取。READ_START = 0，禁用开始读操作；READ_START = 1，启动开始读操作。ADCCON[1]的复位值为0，禁用开始读操作。

```
#define S3C_ADCCON_NO_READ_START     (0<<1)
#define S3C_ADCCON_READ_START     (1<<1)
```

ADCCON[2]：STDBM，待机模式选择。STDBM = 0，正常运作模式；STDBM = 1，待机模式。ADCCON[2]的复位值为1，待机模式。

#define S3C_ADCCON_RUN (0<<2)

#define S3C_ADCCON_STDBM   (1<<2)

ADCCON[5:3]：SEL_MUX，模拟输入通道选择。SEL_MUX = 000，AIN0；SEL_MUX = 001， AIN1；SEL_MUX = 010，AIN2；SEL_MUX = 011，AIN3；SEL_MUX = 100，YM；SEL_MUX = 101，YP；SEL_MUX = 110，XM；SEL_MUX = 111，XP。ADCCON[5:3]的复位值为000，选用AIN0通道。

#define S3C_ADCCON_RESSEL_10BIT_1 (0x0<<3)

#define S3C_ADCCON_RESSEL_12BIT_1 (0x1<<3)

#define S3C_ADCCON_MUXMASK (0x7<<3)

#define S3C_ADCCON_SELMUX(x)   (((x)&0x7)<<3) //任意通道的选择

ADCCON[13:6]：PRSCVL，ADC 预定标器值0xFF。数据值：5～255。ADCCON[13:6]的复位值为1111 1111，即为0xFF。

#define S3C_ADCCON_PRSCVL(x)   (((x)&0xFF)<<6) // 任意值设定

#define S3C_ADCCON_PRSCVLMASK   (0xFF<<6) //复位值

ADCCON[14]：PRSCEN，ADC预定标器启动。PRSCEN = 0，禁用；PRSCEN = 0，启用。ADCCON[14]的复位值为0，禁用ADC预定标器。

#define S3C_ADCCON_NO_PRSCEN   (0<<14)

#define S3C_ADCCON_PRSCEN   (1<<14)

ADCCON[15]：ECFLG，转换的结束标记（只读）。ECFLG = 0，A/D转换的过程中；ECFLG = 1，A/D 转换结束。ADCCON[15]的复位值为0，A/D 转换的过程中。

#define S3C_ADCCON_ECFLG_ING   (0<<15)

#define S3C_ADCCON_ECFLG   (1<<15)

ADCDAT0 寄存器为ADC 的数据转换寄存器。地址为：0x7E00B00C。

ADCDAT0[9:0]：XPDATA，X 坐标的数据转换（包括正常的ADC 的转换数据值）。数据值：0x000～0x3FF。

ADCDAT0[11:10]：保留。当启用12位AD时作为转换数据值使用。

```
#define S3C_ADCDAT0_XPDATA_MASK   (0x03FF)
#define S3C_ADCDAT0_XPDATA_MASK_12BIT   (0x0FFF)
```

上面所介绍的是专用A/D转换通道常用寄存器，LCD触摸屏A/D转换有另外的A/D通道。

2. ADC驱动程序

A/D 转化驱动由于也属于字符设备驱动，所以其程序设计流程和LED 驱动大体一致。在linux-3.8.3/drivers/char 目录下新建zzqadc.c 驱动文件，当然也可写好之后再拷贝到 linux-3.8.3/drivers/char目录下。

```
zhuzhaoqi@zhuzhaoqi-desktop:～/Linux/linux-3.8.3/drivers/char$ vim zzqadc.c
```

头文件是必不可少的，A/D驱动程序所要包含的头文件如下所示：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/input.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/serio.h>
#include <linux/delay.h>
#include <linux/clk.h>
#include <linux/sched.h>
```

```
#include <linux/cdev.h>
#include <linux/miscdevice.h>
#include <asm/io.h>
#include <asm/irq.h>
#include <asm/uaccess.h>
#include <mach/map.h>
#include <mach/regs-clock.h>
#include <mach/regs-gpio.h>
#include <plat/regs-adc.h>
```

与LED驱动程序所包含的头文件相比较，多了ADC专用的头文件，如regs-adc.h，这个头文件位于linux-3.8.3/arch/arm/plat-samsung/include/plat目录下。

```
static void __iomem *base_addr;
static struct clk *adc_clock;
#define __ADCREG(name) (*(unsigned long int *)(base_addr + name))
```

自从linux-2.6.9版本开始便把_iomem加入内核，_iomem是表示指向一个I/O的内存空间。将_iomem加入linux，主要是考虑到驱动程序的通用性。由于不同的CPU体系结构对I/O空间的表示可能不同，但是当使用_iomem时，就会忽略对变量的检查，因为_iomem使用的是void。

```
#define  S3C_ADCREG(x)  (x)
#define  S3C_ADCCON    S3C_ADCREG(0x00)
#define  S3C_ADCDAT0    S3C_ADCREG(0x0C)
/* ADC contrl */
#define  ADCCON      _ADCREG(S3C_ADCCON)
/* read the ADdata */
```

```
#define  ADCDAT0      _ADCREG(S3C_ADCCON)
```

声明ADC控制寄存器的地址。

```
/* The set of ADCCON */
#define  S3C_ADCCON_ENABLE_START    (1 << 0)
#define  S3C_ADCCON_READ_START      (1 << 1)
#define  S3C_ADCCON_RUN         (0 << 2)
#define  S3C_ADCCON_STDBM         (1 << 2)
#define  S3C_ADCCON_SELMUX(x)       ( ((x)&0x7) << 3 )
#define  S3C_ADCCON_PRSCVL(x)       ( ((x)&0xFF) << 6 )
#define  S3C_ADCCON_PRSCEN         (1 << 14)
#define  S3C_ADCCON_ECFLG         (1 << 15)
/* The set of ADCDAT0 */
#define  S3C_ADCDAT0_XPDATA_MASK    (0x03FF)
#define  S3C_ADCDAT0_XPDATA_MASK_12BIT  (0x0FFF)
```

根据上一小节对ADCCON和ADCDAT0的介绍，可以很容易写出上面的宏定义。

在使用ADC之前，先得对ADC进行初始化设置，由于OK6410开发平台自带的A/D电压采样电路选用的是 AIN0 通道，则这里需要对 AIN0 进行初始化。初始化阶段需要完成的事情为：A/D 转换开始和该位被清理后开启、正常运作模式、模拟输入通道选择AIN0、ADC 预定标器值0xFF、ADC预定标器启动。

```
/*
  * AIN0 init
  */
static int adc_init(void)
{
```

```
    ADCCON = S3C_ADCCON_PRSCEN | S3C_ADCCON_PRSCVL(0xFF)
| \
    S3C_ADCCON_SELMUX(0x00) | S3C_ADCCON_RUN;
    ADCCON |=S3C_ADCCON_ENABLE_START;
      return 0;
    }
```

open函数指针的实现函数adc_open()：

```
    /*
      * open dev
      */
    static int adc_open(struct inode *inode, struct file
*filp)
    {
      adc_init();
      return 0;
    }
```

release函数指针的实现函数adc_release()：

```
    /*
      * release dev
      */
    static int adc_release(struct inode *inode,struct file
*filp)
    {
      return 0;
    }
```

read()函数指针的实现函数adc_read()，这个函数的作用是读取ADC采样数据。

```
    /*
      * adc_read
      */
    static ssize_t adc_read(struct file *filp, char __user
*buff,
    size_t size, loff_t *ppos)
    {
      ADCCON |= S3C_ADCCON_READ_START;
      /* check the adc Enabled ,The [0] is low*/
      while(ADCCON & 0x01);
      /* check adc change end */
      while(!(ADCCON & 0x8000));
      /* return the data of adc */
      return (ADCDAT0 & S3C_ADCDAT0_XPDATA_MASK);
    }
```

ADC驱动程序的核心控制部分：

```
    static struct file_operations dev_fops =
    {
      .owner = THIS_MODULE,
      .open= adc_open,
      .release = adc_release,
      .read= adc_read,
    };
    static struct miscdevice misc =
    {
      .minor = MISC_DYNAMIC_MINOR,
      .name = “zzqadc “,
```

```
    .fops = &dev_fops,
  };
```

加载insmod驱动程序，如下所示：

```
  static int __init dev_init()
  {
    int ret;
    /* Address Mapping */
    base_addr = ioremap(SAMSUNG_PA_ADC,0X20);
    if(base_addr == NULL)
    {
      printk(KERN_ERR"failed to remap \n");
      return -ENOMEM;
    }
    /* Enabld acd clock */
    adc_clock = clk_get(NULL,"adc");
    if(!adc_clock)
    {
      printk(KERN_ERR"failed to get adc clock \n");
      return -ENOENT;
    }
    clk_enable(adc_clock);
    ret = misc_register(&misc);
    printk("dev_init return ret: %d \n", ret);
    return ret;
  }
```

加载 insmod 驱动程序，这里使用到了 ioremap()函数。在内核驱动程序的初始化阶段，通过ioremap()函数将物理地址映射到内核虚

拟空间；在驱动程序的 mmap 系统调用中，使用remap_page_range()函数将该块ROM映射到用户虚拟空间。这样内核空间和用户空间都能访问这段被映射后的虚拟地址。

ioremap()宏定义在asm/io.h内：

#define ioremap(cookie,size)
__ioremap(cookie,size,0)

_ioremap函数原型为（arm/mm/ioremap.c）：

void _iomem * _ioremap(unsigned long phys_addr, size_t size, unsigned longflags);

phys_addr：要映射的起始的I/O地址；

size：要映射的空间的大小；

flags：要映射的I/O空间和权限有关的标志。

该函数返回映射后的内核虚拟地址（3GB～4GB），接着便可以通过读写该返回的内核虚拟地址去访问之这段I/O内存资源。

base_addr = ioremap(SAMSUNG_PA_ADC,0X20);

这行代码即是将SAMSUNG_PA_ADC（0x7E00 B000）映射到内核，返回内核的虚拟地址给base_addr。

clk_get（NULL,"adc"）可以获得adc时钟，每一个外设都有自己的工作频率，PRSCVL是A/D转换器时钟的预分频功能时A/D 时钟的计算公式，A/D 时钟 = PCLK / （PRSCVL+1）。

注意：AD时钟最大为2.5MHz并且应该小于PCLK的1/5。

adc_clock = clk_get(NULL,"adc");

即为获取adc的工作时钟频率。

ret = misc_register(&misc);

创建杂项设备节点。这里使用到了杂项设备，杂项设备也是在嵌入式系统中用得比较多的一种设备驱动。在 Linux 内核的 include/linux 目录下有 miscdevice.h 文件，要把自己定义的

miscdevice 从设备定义到这里。其实是因为这些字符设备不符合预先确定的字符设备范畴，所有这些设备采用主编号10，一起归于misc device，其实misc_register就是用主标号10调用register_chrdev()的。也就是说，misc设备其实也就是特殊的字符设备，可自动生成设备节点。

卸载rmmod驱动程序：

```
static void __exit dev_exit()
{
  iounmap(base_addr);
  /* disable ths adc clock */
  if(adc_clock)
  {
    clk_disable(adc_clock);
    clk_put(adc_clock);
    adc_clock = NULL;
  }
  misc_deregister(&misc);
}
```

许可证声明、作者信息、调用加载和卸载程序：

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zhuzhaoqi jxlgzzq@163.com");
module_init(dev_init);
module_exit(dev_exit);
```

在/linux-3.8.3/drivers/char目录下的Makefile中添加：

```
obj-m            += zzqadc.o
```

回到/linux-3.8.3根目录下：

```
/home/zhuzhaoqi/Linux/linux-3.8.3# make modules
```

将/linux-3.8.3/drivers/char目录下生成的zzqadc.ko拷贝到文件系统的/lib/module/3.8.3目录中。

3. ADC应用程序

ADC应用程序也是相对简单，打开设备驱动文件之后进行数据读取即可。

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
  int fp,adc_data,i;
  fp = open("/dev/zzqadc",O_RDWR);
  if (fp < 0)
  {
    printf("open failed! \n");
  }
  printf("opened ... \n");
  for ( ; ; i++)
  {
    adc_data = read(fp,NULL,0);
    printf("Begin the NO. %d test... \n",i);
    printf("adc_data = %d \n",adc_data);
    printf("The Value = %f V \n" , ( (float)adc_data )*
3.3 / 1024);
    printf("End the NO. %d test ..... \n \n",i);
    sleep(1);
  }
```

```
    close(fp);
    return 0;
  }
```

由于本次使用的A/D转换是10位，则数据转换值即为1024，而OK6410的参考电压是3.3V，则A/D 采集数据和电压之间的转换公式为：（float）adc_data ）* 3.3 / 1024。

为ADC应用程序编写Makefile：

```
CC = /usr/local/arm/4.4.1/bin/arm-linux-gcc
zzqadcapp:zzqadcapp.o
    $(CC) -o zzqadcapp zzqadcapp.o
zzqadcapp.o:zzqadcapp.c
    $(CC) -c zzqadcapp.c
clean :
    rm zzqadcapp.o zzqadcapp
```

将生成的zzqadcapp应用文件拷贝到文件系统/usr/bin文件夹下。

加载zzqadc.ko设备：

```
[YJR@zhuzhaoqi 3.8.3]# insmod zzqadc.ko
dev_init return ret: 0
[YJR@zhuzhaoqi]\# ls -l /dev/zzqadc
crw-rw----  1 root  root  10,  60 Jan  1 08:00
/dev/zzqadc
```

在/dev目录下存在zzqadc设备节点，则说明ADC驱动加载成功。

执行ADC应用程序，电压采样如下所示：

```
[YJR@zhuzhaoqi]\# ./zzqadcapp
opened ...
……
Begin the NO. 10 test...
```

```
adc_data = 962
The Value = 3.100195 V
End the NO. 10 test ......
......
```

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第四章02课（字符设备驱动之ADC）。

4.3 块设备驱动

块设备和字符设备从字面上理解最主要的区别在于读写的基本单元不同，块设备的读写基本单元为数据块，数据的输入输出都是通过一个缓冲区来完成的。而字符设备不带有缓冲，直接与实际的设备相连而进行操作，读写的基本单元为字符。从实现的角度来看，块设备和字符设备是两种不同的机制，字符设备的read、write的API直接到字符设备层，但是块设备相对复杂，是先到文件系统层，然后再由文件系统层发起读写请求。

数据块指的是固定大小的数据，这个值的大小由内核来决定。一般而言，数据块的大小通常是 4096 Bytes，但是大小并不是恒定不变的，而是可以根据体系结构和所使用的文件系统进行改变。与数据块相对应的是扇区，它是由底层硬件决定大小的一个块。内核所处理的设备扇区大小是 512 Bytes，无论何时内核为用户提供一个扇区编号，该扇区的大小都是 512 Bytes。但是如果要使用不同的硬件扇区大小，用户必须对内核的扇区数做相应的修改。

4.3.1 块设备操作

1. file_operations结构体

和字符设备驱动中的 file_operations 结构体类似，块设备驱动中也有一个 block_device_operations结构体，它的声明位于/include/linux目录下的fs.h文件中，它是对块操作的集合。

```
struct block_device_operations{
int(*open)(struct inode *, struct file*); //打开设备
int(*release)(struct inode *, struct file*); //关闭设备
//实现ioctl系统调用
int(*ioctl)(struct inode *, struct file *, unsigned, unsigned long);
long(*unlocked_ioctl)(struct file *, unsigned, unsigned long);
long(*compat_ioctl)(struct file *, unsigned, unsigned long);
int(*direct_access)(struct block_device *, sector_t, unsigned long*);
//调用该函数用以检查用户是否更换了驱动器的介质
int(*media_changed)(struct gendisk*);
int(*revalidate_disk)(struct gendisk*); //当介质被更换时，调用该函数做出响应
int(*getgeo)(struct block_device *, struct hd_geometry*);//获取驱动器信息
struct module *owner; //指向拥有这个结构体模块的指针，通常被初始化为THIS_MODULE
};
```

与字符驱动不同的是在这个结构体中缺少了read()和write()函数，那是因为在块设备的I/O子系统中，这些操作都是由request函数

进行处理的。

request函数的原型如下：

```
void request(request_queue_t *queue);
```

当内核需要驱动程序处理读取、写入以及其他对设备的操作时，便会调用request函数。

2. gendisk结构体

gendisk结构体的定义位于/include/linux目录下的genhd.h文件中，如下所示。

```
  struct gendisk {
  /*
  *这3个成员的定义依次是：主设备号、第一个次设备号，次设备号。一个驱动中至少有一个次设备号，
  *如果驱动器是一个可被分区，那么每一个分区都将分配一个次设号。
  */
    int major;
    int first_minor;
    int minors;
  //这个数组用以存储驱动设备的名字
    char disk_name[DISK_NAME_LEN];
  char *(*devnode)(struct gendisk *gd, umode_t *mode);
    unsigned int events;
    unsigned int async_events;
    struct disk_part_tbl __rcu *part_tbl;
    struct hd_struct part0;
  //这个结构体用以设置驱动中的各种设备操作
    const struct block_device_operations *fops;
```

```
    //Linux内核使用这个结构体为设备管理I/O请求，具体详解见request_queue结构
    struct request_queue *queue;
      void *private_data;
      int flags;
      struct device *driverfs_dev;
      struct kobject *slave_dir;
      struct timer_rand_state *random;
      atomic_t sync_io;
      struct disk_events *ev;
    #ifdef CONFIG_BLK_DEV_INTEGRITY
      struct blk_integrity *integrity;
    #endif
      int node_id;
    };
```

gendisk结构体是动态分配，但是驱动程序自己不能动态分配该结构，而是通过调用alloc_disk()函数进行动态分配。

```
struct gendisk *alloc_disk(int minors);
```

其中minors是该磁盘使用的次设备号。

但是分配了 gendisk 结构并不意味着该磁盘就对系统可用，使用之前的初始化结构体并且调用add_disk()函数。

```
//初始化结构体
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
//添加分区
void add_disk(struct gendisk *gd)
```

如果不再需要这个磁盘，则对其进行卸载。

//删除分区

Void del_gendisk(struct gendisk *gd)

void blk_cleanup_queue(struct request_queue *q)

3. bio结构体

bio结构体的定义位于/include/linux目录下的linux_blk_types.h文件中。

```
struct bio {
  //需要传输的第一个(512 bytes)扇区
sector_t  bi_sector;
  struct bio    *bi_next;
  struct block_device  *bi_bdev;
  unsigned long  bi_flags;
  unsigned long  bi_rw;
  unsigned short    bi_vcnt;
  unsigned short    bi_idx;
  //BIO中所包含的物理段数目
  unsigned int    bi_phys_segments;
//所传输的数据大小(以byte为单位)
  unsigned int    bi_size;
  unsigned int    bi_seg_front_size;
  unsigned int    bi_seg_back_size;
  unsigned int    bi_max_vecs;
  atomic_t  bi_cnt;
  struct bio_vec      *bi_io_vec;
  bio_end_io_t    *bi_end_io;
  void      *bi_private;
#ifdef CONFIG_BLK_CGROUP
```

```
    struct io_context *bi_ioc;
    struct cgroup_subsys_state *bi_css;
  #endif
  #if defined(CONFIG_BLK_DEV_INTEGRITY)
    struct bio_integrity_payload *bi_integrity;
  #endif
    bio_destructor_t  *bi_destructor;
    struct bio_vec    bi_inline_vecs[0];
  };
```

bio 结构体包含了驱动程序执行请求的所有信息，既描述了磁盘的位置，又描述了内存的位置，是上层内核与下层驱动的连接纽带。

bio结构体的核心在于：

```
struct bio_vec      *bi_io_vec;
```

而bio_vec结构体的声明为：

```
struct bio_vec {
struct page    *bv_page;  /*数据段所在的页*/
unsigned short  bv_len;  /*数据段的长度*/
unsigned short bv_offset; /*数据段页内偏移*/
};
```

结构bio_vec代表了内存中的一个数据段，数据段用页、偏移和长度来描述。bio_vec结构体和bio结构体之间的关系如图4.4所示。

从图4.4可知，当I/O请求被转换到bio结构体之后，它将被单独的物理内存页所销毁。

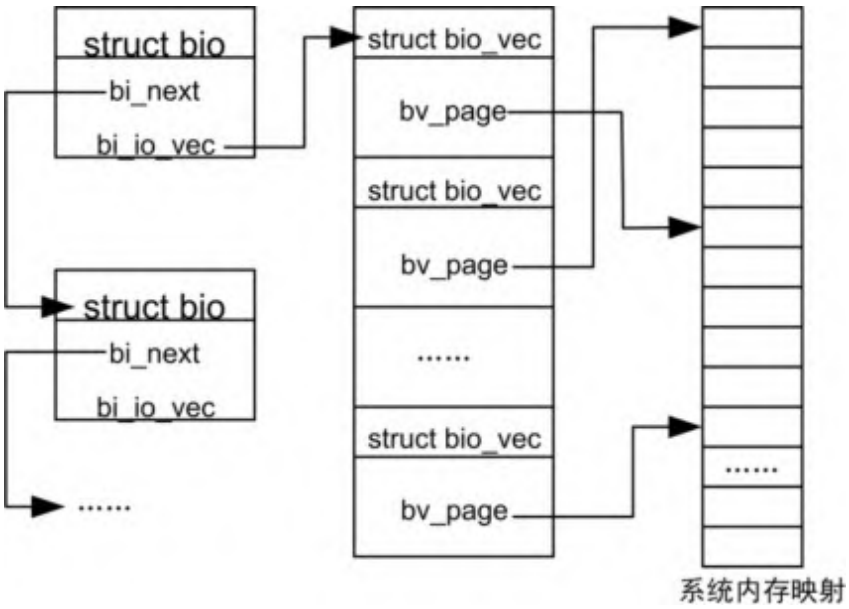


图4.4 bio_vec结构体和bio结构体之间的关系

4. requeset结构体

request结构体代表了挂起的I/O请求，每个请求用一个结构request实例描述，存放在请求队列链表中，由电梯算法进行排序，每个请求包含一个或多个结构bio实例。requeest结构体声明位于/include/linux目录下的blkdev.h文件中。

```
struct request {
  struct list_head queuelist;
  struct call_single_data csd;
  struct request_queue *q;
  unsigned int cmd_flags;
  enum rq_cmd_type_bits cmd_type;
  unsigned long atomic_flags;
  int cpu;
  unsigned int __data_len;
  sector_t __sector;
  struct bio *bio;
```

```
  struct bio *biotail;
  struct hlist_node hash;
  union {
    struct rb_node rb_node;
    void *completion_data;
  };
  union {
    struct {
      struct io_cq    *icq;
      void      *priv[2];
    } elv;
    struct {
      unsigned int    seq;
      struct list_head  list;
      rq_end_io_fn    *saved_end_io;
    } flush;
  };
  struct gendisk *rq_disk;
  struct hd_struct *part;
  unsigned long start_time;
#ifdef CONFIG_BLK_CGROUP
  struct request_list *rl;
  unsigned long long start_time_ns;
  unsigned long long io_start_time_ns;
#endif
  unsigned short nr_phys_segments;
#if defined(CONFIG_BLK_DEV_INTEGRITY)
```

```
    unsigned short nr_integrity_segments;
  #endif
    unsigned short ioprio;
    int ref_count;
    void *special;
    char *buffer;
    int tag;
    int errors;
    unsigned char __cmd[BLK_MAX_CDB];
    unsigned char *cmd;
    unsigned short cmd_len;
    unsigned int extra_len;
    unsigned int sense_len;
    unsigned int resid_len;
    void *sense;
    unsigned long deadline;
    struct list_head timeout_list;
    unsigned int timeout;
    int retries;
    rq_end_io_fn *end_io;
    void *end_io_data;
    struct request *next_rq;
};
```

5. request_queue结构体

每个块设备都有一个请求队列，每个请求队列单独执行I/O调度。请求队列是由请求结构实例链接成的双向链表，链表以及整个队列的信息用request_queue结构体描述，称为请求队列对象结构或请求队列

结构。request_queue结构体声明位于/include/linux目录下的blkdev.h文件中。

```
struct request_queue {
    struct list_head  queue_head;
    struct request    *last_merge;
    struct elevator_queue *elevator;
    int      nr_rqs[2];
    int      nr_rqs_elvpriv;
    struct request_list  root_rl;
    request_fn_proc    *request_fn;
    make_request_fn    *make_request_fn;
    prep_rq_fn    *prep_rq_fn;
    unprep_rq_fn    *unprep_rq_fn;
    merge_bvec_fn    *merge_bvec_fn;
    softirq_done_fn    *softirq_done_fn;
    rq_timed_out_fn    *rq_timed_out_fn;
    dma_drain_needed_fn  *dma_drain_needed;
    lld_busy_fn    *lld_busy_fn;
    sector_t  end_sector;
    struct request    *boundary_rq;
    struct delayed_work  delay_work;
    struct backing_dev_info  backing_dev_info;
    void      *queuedata;
    unsigned long  queue_flags;
    int      id;
    gfp_t      bounce_gfp;
    spinlock_t    __queue_lock;
```

```
  spinlock_t    *queue_lock;
  struct kobject kobj;
  unsigned long  nr_requests; /* Max # of requests */
  unsigned int    nr_congestion_on;
  unsigned int    nr_congestion_off;
  unsigned int    nr_batching;
  unsigned int    dma_drain_size;
  void      *dma_drain_buffer;
  unsigned int    dma_pad_mask;
  unsigned int    dma_alignment;
  struct blk_queue_tag  *queue_tags;
  struct list_head  tag_busy_list;
  unsigned int    nr_sorted;
  unsigned int    in_flight[2];
  unsigned int    rq_timeout;
  struct timer_list timeout;
  struct list_head  timeout_list;
  struct list_head  icq_list;
#ifdef CONFIG_BLK_CGROUP
  DECLARE_BITMAP    (blkcg_pols, BLKCG_MAX_POLS);
  struct blkcg_gq    *root_blkg;
  struct list_head  blkg_list;
#endif
  struct queue_limits  limits;
  unsigned int    sg_timeout;
  unsigned int    sg_reserved_size;
  int      node;
```

```
#ifdef CONFIG_BLK_DEV_IO_TRACE
  struct blk_trace  *blk_trace;
#endif
  unsigned int    flush_flags;
  unsigned int    flush_not_queueable:1;
  unsigned int    flush_queue_delayed:1;
  unsigned int    flush_pending_idx:1;
  unsigned int    flush_running_idx:1;
  unsigned long  flush_pending_since;
  struct list_head  flush_queue[2];
  struct list_head  flush_data_in_flight;
  struct request    flush_rq;
  struct mutex    sysfs_lock;
  int      bypass_depth;
#if defined(CONFIG_BLK_DEV_BSG)
  bsg_job_fn    *bsg_job_fn;
  int      bsg_job_size;
  struct bsg_class_device bsg_dev;
#endif
#ifdef CONFIG_BLK_CGROUP
  struct list_head  all_q_node;
#endif
#ifdef CONFIG_BLK_DEV_THROTTLING
  struct throtl_data *td;
#endif
};
```

4.3.2 块设备驱动程序

块设备是一种抽象的设备驱动，它看不到、摸不着。由于这本书是针对嵌入式Linux初学者，因此笔者使用最简单、最通俗易懂的一个块设备驱动程序向各位读者展现如何开辟、挂载、使用一个分区。或许这个程序有很多不合理之处，但却是初学者容易接受的。

```
/*
 * 头文件
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/blkdev.h>
/* 主设备号，COMPAQ_SMART2_MAJOR = 72 */
#define ZZQ_BLKDEV_DEVICEMAJOR  COMPAQ_SMART2_MAJOR
/* 设备名称 */
#define ZZQ_BLKDEV_DISKNAME    "zzqdisk"
#define ZZQ_BLKDEV_SIZE    (1 * 1024 * 1024)
/* 次设备号 */
#define ZZQ_MIJORS        1
/* 使用数组储块设备数据 */
/*
 * 这个数组应该是最忌讳的，1MB的全局变量
*/
unsigned char zzq_blkdev_data[ZZQ_BLKDEV_SIZE];
/* 定义一个指向请求队列的结构体指针 */
static struct request_queue *zzq_blkdev_queue;
/* 定义一个指向独立分区(磁盘)的结构体指针 */
```

```
static struct gendisk    *zzq_blkdev_disk;
/*
  * 请求队列的操作
  */
static void zzq_blkdev_do_request(struct request_queue
*q)
{
  struct request *req;
  /*
    * blk_rq_pos()      : the current sector 当前扇区
    * blk_rq_bytes()    : bytes left in the entire
request
    * blk_rq_cur_bytes()  : bytes left in the current
segment
    * blk_rq_err_bytes()  : bytes left till the next
error boundary
    * blk_rq_sectors()  : sectors left in the entire
request
    * blk_rq_cur_sectors() : sectors left in the
current segment
    */
    /*
    * 从请求队列中取出一个请求(可能是请求中的一段),
    * 如果不是为空的话
    */
  while((req = blk_fetch_request(q)) != NULL)
  {
```

```
    if(((blk_rq_pos(req) + blk_rq_sectors(req)) << 9) >
ZZQ_BLKDEV_SIZE)
    {
      printk(KERN_ERR ZZQ_BLKDEV_DISKNAME "bad request
  :block = %llu,count = %u\n", (u64)blk_rq_pos(req),
  blk_rq_sectors(req));
      /*
        * 结束一个队列请求，第二个参数表示请求处理结果
        * 成功的话设定为1，失败的话设定为0 或者错误号
        */
          __blk_end_request_all(req, -EIO);
      continue;
    }
      /*
    * rq_data_dir()函数返回该请求的方向:读还是写
      */
    switch( rq_data_dir(req))
    {
      /* 如果是读 */
      case READ:
        printk(KERN_ALERT "read\n");
          /* 把块设备的数据装入队列缓冲区 */
        memcpy(req->buffer, zzq_blkdev_data +
    (blk_rq_pos(req) << 9), blk_rq_sectors(req) << 9);
        /* 请求结束 */
          __blk_end_request_all(req, 1);
        break;
```

```
        /* 如果是写 */
      case WRITE:
        printk(KERN_ALERT "write\n");
          /* 把缓冲区的数据写入块设备 */
        memcpy(zzq_blkdev_data + (blk_rq_pos(req) <<
    9), req->buffer, blk_rq_sectors(req) << 9);
        /* 请求结束 */
          __blk_end_request_all(req, 1);
        break;
      default:
      printk(KERN_ALERT "this should not happen");
      break;
    }
  }
}
/*
  * 块设备操作的集合
  */
struct block_device_operations zzq_blkdev_fops = {
  .owner = THIS_MODULE,
};
/*
  * 块设备的初始化
  */
static int __init zzq_blkdev_init(void )
{
  int ret;
```

```
  printk(KERN_ALERT ZZQ_BLKDEV_DISKNAME "init! \n");
    /* 初始化请求队列 */
    zzq_blkdev_queue =
blk_init_queue(zzq_blkdev_do_request, NULL);
    if(!zzq_blkdev_queue)
    {
      ret = -ENOMEM;
      goto err_init_queue;
    }
      /* 为独立分区开辟一个空间 */
    zzq_blkdev_disk = alloc_disk(ZZQ_MIJORS);
    if(!zzq_blkdev_disk)
    {
      ret = -ENOMEM;
      goto err_alloc_disk;
    }
      /*
      * 以下是初始化分区结构体成员
        */
      /* 设备名称 */
      strcpy(zzq_blkdev_disk->disk_name,
  ZZQ_BLKDEV_DISKNAME);
        /* 主设备号 */
      zzq_blkdev_disk->major = ZZQ_BLKDEV_DEVICEMAJOR;
        /* 调用块设备操作集合 */
      zzq_blkdev_disk->fops = &zzq_blkdev_fops;
        /* 初始化设备的请求队列 */
```

```
    zzq_blkdev_disk->queue = zzq_blkdev_queue;
      /*
        * 给分区分配空间
      *
      * 由于块设备的大小使用扇区作为基本单元，
      * 扇区的默认大小是512byte，也就是向右移动9 位
        */
    set_capacity(zzq_blkdev_disk, ZZQ_BLKDEV_SIZE >>
9);
      /* 注册分区 */
    add_disk(zzq_blkdev_disk);
    return 0;
  err_alloc_disk:
    blk_cleanup_queue(zzq_blkdev_queue);
  err_init_queue:
    return ret;
  }
  /*
    * 块设备卸载
    */
  static void zzq_blkdev_exit(void)
  {
  printk(KERN_ALERT"exit zzqblkdev! \n");
    /* 释放删除分区 add_disk() */
    del_gendisk(zzq_blkdev_disk);
      /* blk_init_queue() */
    blk_cleanup_queue(zzq_blkdev_queue);
```

```
}
module_init(zzq_blkdev_init);
module_exit(zzq_blkdev_exit);
MODULE_LICENSE("GPL");
```

这个程序实现的是开辟一个1MB的独立分区（磁盘），我们可以对这个分区进行读写和挂载等操作。

这里面有一个数组，使用了 1MB 的全局变量空间，这是很忌讳的。但是为了程序的通俗易懂性，笔者还是这样做了。要写好一个漂亮的块设备驱动程序，程序员必须要有深厚的C语言功底和数据结构知识。如果要去掉这个 1MB 的全局变量空间，这里内存的申请可以使用基树，或者也可以使用红黑树、哈希表等。

在这里基树是首选，笔者也希望各位读者能使用基树优化这个驱动程序。内核提供了一个基树库，代码在/lib/目录下的 radix-tree.c 文件中。基树是一种空间换时间的数据结构，通过空间的冗余减少时间上的消耗。我们使用如图4.5所示来描述基树算法。

如图4.5所示，元素空间总共为256，但元素个数不固定。那么如果用数组存储，好处是插入查找只用一次操作，但是存储空间需要256，空间换取时间，但是在嵌入式中，内存是宝贵的。如果用链表存储，存储空间节省了，但是极限情况下查找操作次数等于元素的个数，时间换取空间，但是时间同样是宝贵的。能不能有一种算法，可以兼顾时间和空间呢，有，基树。采用一棵高度为2的基树，第一级最多16个冗余结构，代表元素前四位的索引。第二级代表元素后四位的索引。那么只要两级查找就可以找到特定的元素，而且只有少量的冗余数据。图中假设只有一个元素10001000，那么只有树的第一级有元素，而且树的第二级只有1000个节点有子节点，其他节点都不必分配空间。这样既可以快速定位查找，也减少了冗余数据。基树很适合存储稀疏的数据，内核中文件的页cache就是采用的基树。

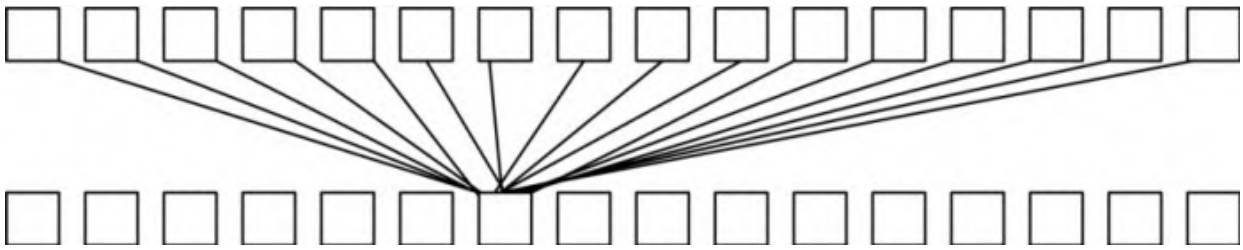


图4.5 基树查找算法

将这个块设备驱动程序zzqblkdev.c放入/drivers/block目录下，修改Makefile，将其编译成独立模块。

```
obj-m          += zzqblkdev.o
```

编译：

```
  zhuzhaoqi@zhuzhaoqi-desktop:～/Linux/linux-3.8.3$ make
modules
    ......
      CC  drivers/block/zzqblkdev.mod.o
      LD [M]  drivers/block/zzqblkdev.ko
    ......
```

将生成的zzqblkdev.ko模块放入OK6410的根文件系统中，挂载：

```
[YJR@zhuzhaoqi]\# insmod zzqblkdev.ko
[YJR@zhuzhaoqi]\# lsmod
zzqblkdev 1049808 0 - Live 0xbf000000
```

可知挂载zzqblkdev模块成功，并且分配给这个分区的大小即为1MB的空间。再看看/dev下面的设备节点。

```
[YJR@zhuzhaoqi]\# ls -l /dev/zzqdisk
brw-rw----  1 root  root  72,  0 Jan  1 08:11
/dev/zzqdisk
```

主设备号为72，次设备号为1。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第四章03课（块驱动）。

第5章 Qt-4.8.4移植

5.1 Qt概述

Qt是1991年奇趣科技公司开发的一个跨平台的C++图形用户界面应用程序框架。它提供给应用程序开发者建立艺术级的图形用户界面所需的所用功能。Qt是完全面向对象的，很容易扩展，并且允许真正地组件编程。

1999 年6 月发布Qt 2.0。

2001 年10 月15 日发布Qt 3.0，在这个版本增加对Mac OS 的支持Qt/MacOS。

2005年6 月27 日发布Qt 4.0 ，这一个版本产生的重大变革是，Qt和Qte开始合为一个版本进行发布，并在Qt 4.0 开始迅速扩展到其他嵌入式平台。

2008 年5 月发布Qt 4.4，它增加了对WinCE 的支持。

2008年6 月17日TrollTech被NOKIA公司收购，更名为QT Software。并确定未来在诺基亚大量采用 Symbian 的开发平台上将使用 QT。我想这应该是诺基亚的巅峰，但此时安卓却在悄无声息地逼近。

2009年3 月发布Qt 4.5，Qt Creator包含在其中，QT 拥有统一的集成开发平台，结束了用插件在其他IDE开发的历史。

2009 年12 月发布Qt 4.6，QT 加入对Symbian 的支持。

2010 年3 月份，发布了Qt 4.7 和QtCreator 2.0。

Qt的未来，众说纷纭，但是作为初学者，我们很有必要学习其中的思想和方法。

5.2 Qt编译环境搭建

在进行Qt开发之前，先建立Qt编译环境、移植Qt是一个至关重要的步骤。

5.2.1 tslib安装

OK6410 开发平台在使用触摸屏时，因为电磁噪声的缘故，触摸屏容易存在点击不准确、有抖动等问题。tslib能够为触摸屏驱动获得的采样提供诸如滤波、去抖、校准等功能，通常作为触摸屏驱动的适配层，为上层的应用提供一个统一的接口。

在官方网站下载tslib-1.0.tar.bz2。将下载完成之后的tslib-1.0.tar.bz2存放在宿主机的任意一个目录下：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib$ ls
tslib-1.0.tar.bz2
```

将其解压出来：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib$ tar jxvf tslib-1.0.tar.bz2
```

解压完成之后进入tslib-1.0目录，如下所示：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib/tslib-1.0$ ls
acinclude.m4  autogen.sh  COPYING  m4    plugins  tests
```

AUTHORS ChangeLog etc Makefile.am tslib.pc.in
autogen-clean.sh configure.ac INSTALL NEWS src

安装autoconf、automake、libtool：

root@zhuzhaoqi-desktop:/home/zhuzhaoqi#apt-get install autoconf

root@zhuzhaoqi-desktop:/home/zhuzhaoqi#apt-get install automake

root@zhuzhaoqi-desktop:/home/zhuzhaoqi#apt-get install libtool

由于open函数的语法不符合最新的gcc，
在/tests/ts_calibrate.c中加入open的第三个参数：

```
if ((calfile = getenv("TSLIB_CALIBFILE")) != NULL) {
cal_fd = open (calfile, O_CREAT | O_RDWR, 0777);
} else {
cal_fd = open ("/etc/pointercal", O_CREAT | O_RDWR,0777);
}
```

执行编译：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib/tslib-1.0$ ./autogen.sh

……

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib/tslib-1.0$ ./configure --prefix=/usr/local/tslib-1.0/ --host=arm-linux ac_cv_func_malloc_0_nonnull=yes --enable-inputapi=no

……

--prefix=/usr/local/tslib-1.0/这是安装路径，进行编译、安装。

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib/tslib-1.0$make

……

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/tslib/tslib-1.0$sudo make install

安装完成之后在/usr/local/目录下有：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local$ ls

arm etc include lib qwt-6.0.2 sbin src

bin games info man qwt-6.0.2-arm share tslib-1.0

修改ts.conf：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local/tslib-1.0/etc$ vim ts.conf

去掉module_raw input前面的#，注意前面这个空格也得删除，如下所示：

Uncomment if you wish to use the linux input layer event interface

module_raw input

将/usr/local/tslib-1.0/目录下的所有文件拷贝到开发板，笔者是存放在usr/local/，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local/tslib-1.0$ sudo cp -r */home/zhuzhaoqi/rootfs/usr/local/

在OK6410中设置tslib环境变量，在文件系统的/etc/profile中添加如下：

//指定帧缓冲设备

export set TSLIB_FBDEVICE=/dev/fb0

//指定触摸屏设备节点

export set TSLIB_TSDEVICE=/dev/input/event0

//指定 TSLIB 配置文件的位置

export set TSLIB_CONFFILE=/usr/local/etc/ts.conf

//指定触摸屏校准文件 pintercal 的存放位置

export set TSLIB_CALIBFILE=/etc/pointercal

//指定触摸屏插件所在路径

export set TSLIB_PLUGINDIR=/usr/local/lib/ts

//设定控制台设备为 none ，否则默认为 /dev/tty ，

export TSLIB_CONSOLEDEVICE=none

在测试触摸屏之前，首先得保证在/dev目录下有触摸屏设备节点eventX：

[YJR@zhuzhaoqi]\# ls -l /dev/input/e*

crw-rw---- 1 root root 13, 64 Jan 1 08:00 /dev/input/event0

运行ts_calibrate：

[YJR@zhuzhaoqi]\# cd bin/

[YJR@zhuzhaoqi]\# ls

ts_calibrate ts_harvest ts_print ts_print_raw ts_test

[YJR@zhuzhaoqi]\# ./ts_calibrate

运行/usr/local/bin 中的 ts_calibrate 进行校准，成功的话会出现界面，并点击十字符号，完成后会生成/etc/pointercal文件，这便是触摸屏的校准配置文件。

或者可以写一个触摸屏校准脚本calibrate，存放在/bin目录下：

#!/bin/sh

export TSLIB_FBDEVICE=/dev/fb0

export TSLIB_TSDEVICE=/dev/input/event0

```
export TSLIB_CONFFILE=/usr/local/tslib/etc/ts.conf
export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_PLUGINDIR=/usr/local/tslib/lib/ts
export TSLIB_TSEVENTTYPE=H3600
export TSLIB_CONSOLEDEVICE=none
  export QWS_KEYBOARD="TTY:/dev/tty1"
  if [ -c /dev/input/event0 ]; then
    if [ -e /etc/pointercal -a ! -s /etc/pointercal ] ;
then
      rm /etc/pointercal
    fi
  fi
  export PATH=$QTDIR/bin:$PATH
  export
LD_LIBRARY_PATH=$QTDIR/plugins/qtopialmigrate/:$QTDIR/qt_pl
ugins/
imageformats/:$QTDIR/lib:/root/tslib/build/lib:$LD_LIBRARY_
PATH
  exec /usr/local/tslib/bin/ts_calibrate 1>/dev/null
2>/dev/null
  #exec /usr/local/tslib/bin/ts_test 1>/dev/null
2>/dev/null
```

执行calibrate：

```
[YJR@zhuzhaoqi]\# ./calibrate
```

如果执行的是ts_calibrate测试，效果如图5.1所示。

如果执行的是ts_test测试，选择draw画图选项，效果如图5.2所示。

图5.1 ts_calibrate测试效果

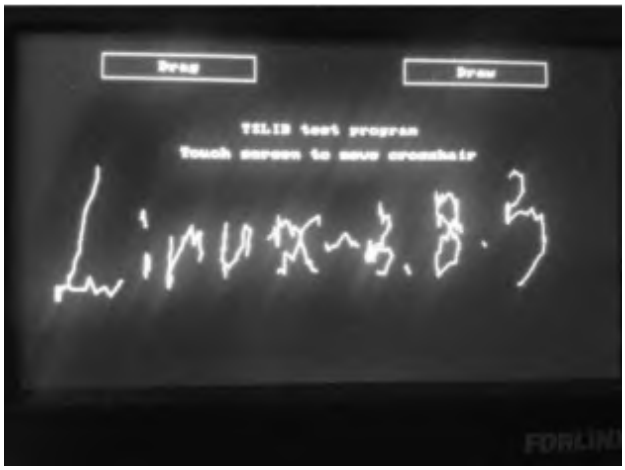


图5.2 ts_test测试效果

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第五章01课（tslib安装）。

5.2.2 安装Linux/x11版Qt-4.8.4

在官方网站下载Qt libraries 4.8.4 for Linux/x11 （225 MB）（实际是：qt-everywhere-opensource-src-4.8.4.tar.gz）。

完成之后在Ubuntu宿主机解压：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4$ tar zxvf qt-everywhere-opensource-src-4.8.4.tar.gz

在装有gold linker的系统里，编译脚本会加入-fuse-ld=gold 选项，但这个选项gcc是不支持的。解决办法是移除该选项，找到文件src/3rdparty/webkit/Source/common.pri，屏蔽QMAKE_LFLAGS+=-fuse-ld=gold。

```
linux-g++ {
isEmpty($$(SBOX_DPKG_INST_ARCH)):exists(/usr/bin/ld.gold)
{
    message(Using gold linker)
#  QMAKE_LFLAGS+=-fuse-ld=gold
}
}
```

配置安装路径：

```
zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/qt-
everywhere-opensource-src-4.8.4$ ./configure --
prefix=/usr/local/qt-4.8.4-x11
```

……

Type 'c' if you want to use the Commercial Edition.

Type 'o' if you want to use the Open Source Edition.

c是商业，o是开源，选择o

……

Type 'yes' to accept this license offer.

Type 'no' to decline this license offer.

选择yes。

……

Qt is now configured for building. Just run 'make'.

Once everything is built, you must run 'make install'.

Qt will be installed into /usr/local/qt-4.8.4-x11

To reconfigure, run 'make confclean' and 'configure'.

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/qt-everywhere-opensource-src-4.8.4$

Qt输出信息提示我们进行make编译：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/qt-everywhere-opensource-src-4.8.4$ make

这个编译过程比较久，依每个人的电脑配置而定，大概需要1～3个小时。

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/qt-everywhere-opensource-src-4.8.4$ make install

安装好之后，在/usr/local目录下面有：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local$ ls

arm etc include lib qt-4.8.4-arm qwt-6.0.2 sbin src

bin games info man qt-4.8.4-x11 qwt-6.0.2-arm tslib-1.0

5.2.3 安装Embedded版Qt-4.8.4

Embedded版Qt4.8.4源码和Linux/x11版Qt-4.8.4是一样的，将下载的源码解压在另一个文件夹，配置Embedded版配置：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/arm/qt-everywhere-opensource-src-4. 8.4$ ./configure -prefix /usr/local/qt-4.8.4-arm/ -shared -no-fast -no-largefile -no-exceptions -qt-sql-sqlite -qt3support -no-xmlpatterns -multimedia -no-svg -no-mmx -no-3dnow -no-sse -no-sse2 -qt-zlib -no-webkit -qt-libtiff -qt-libpng -qt-libjpeg -make libs

-nomake examples -nomake docs -nomake demo -no-optimized-qmake -no-nis -no-cups -no-iconv -no-dbus -no-separate-debug-info -no-openssl -xplatform qws/linux-arm-g++ -embedded arm -little-endian -no-freetype -depths 4,8,16,32 -qt-gfx-linuxfb -no-gfx-multiscreen -no-gfx-vnc -no-gfx-qvfb -qt-kbd-linuxinput -no-kbd-tty -no-glib -armfpa -no-mouse-qvfb -qt-mouse-pc -qt-mouse-tslib -I/usr/local/tslib-1.0/include -L/usr/local/tslib-1.0/lib

执行make进行编译：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/arm/qt-everywhere-opensource-src-4. 8.4$ make

执行make install进行安装：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4/arm/qt-everywhere-opensource-src-4. 8.4$ make install

安装好之后，在/usr/local目录下面有：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local$ ls

arm etc include lib qt-4.8.4-arm qwt-6.0.2 sbin src

bin games info man qt-4.8.4-x11 qwt-6.0.2-arm tslib-1.0

在qt-4.8.4-arm目录下有：

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local/qt-4.8.4-arm$ ls

bin imports include lib mkspecs plugins

在文件系统/opt/目录下新建Qt-4.8.4目录，如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/rootfs/opt$ sudo mkdir Qt-4.8.4/

将imports、lib、mkspecs、plugins拷贝至/opt/Qt-4.8.4/。

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local/qt-4.8.3-arm$ sudo cp -rimportslibmkspecsplugins /home/zhuzhaoqi/rootfs/opt/Qt-4.8.4/

在开发板的文件系统/usr/目录下新建/qt/目录，将/qt-4.8.4-arm/lib/目录下的所有文件拷贝到/usr/qt/目录中。

zhuzhaoqi@zhuzhaoqi-desktop:~/rootfs/usr$ sudo mkdir qt

zhuzhaoqi@zhuzhaoqi-desktop:/usr/local/qt-4.8.4-arm/lib$ sudo cp -r */home/zhuzhaoqi/rootfs/usr/qt/

为OK6410开发平台添加Qt启动环境参数，在/etc/profile中添加：

```
export QTDIR=/usr/qt
export QPEDIR=$QTDIR
export QT_PLUGIN_PATH=/usr/qt
export T_ROOT=/usr/local/tslib
export PATH=$QTDIR/:$PATH
export QWS_MOUSE_PROTO=Tslib:/dev/event0
export LD_LIBRARY_PATH=$T_ROOT/lib:$QTDIR
export QT_QWS_FONTDIR=/usr/qt
```

至此，Qt移植就完成了。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第五章02课（安装Linux和embedded版本Qt-4.8.4）。

5.2.4 安装Qt Creator

在官方网站下载 qt-creator-linux-x86-opensource-2.7.0.bin，然后执行 qt-creator-linux-x86-opensource-2.7.0.bin：

zhuzhaoqi@zhuzhaoqi-desktop:~/Qt-4.8.4/Qt-4.8.4$ ./qt-creator-linux-x86-opensource-2.7.0.bin

根据提示进行安装，这个操作很简单。安装完成之后进行设置。

打开Qt Creator，界面如图5.3 所示。

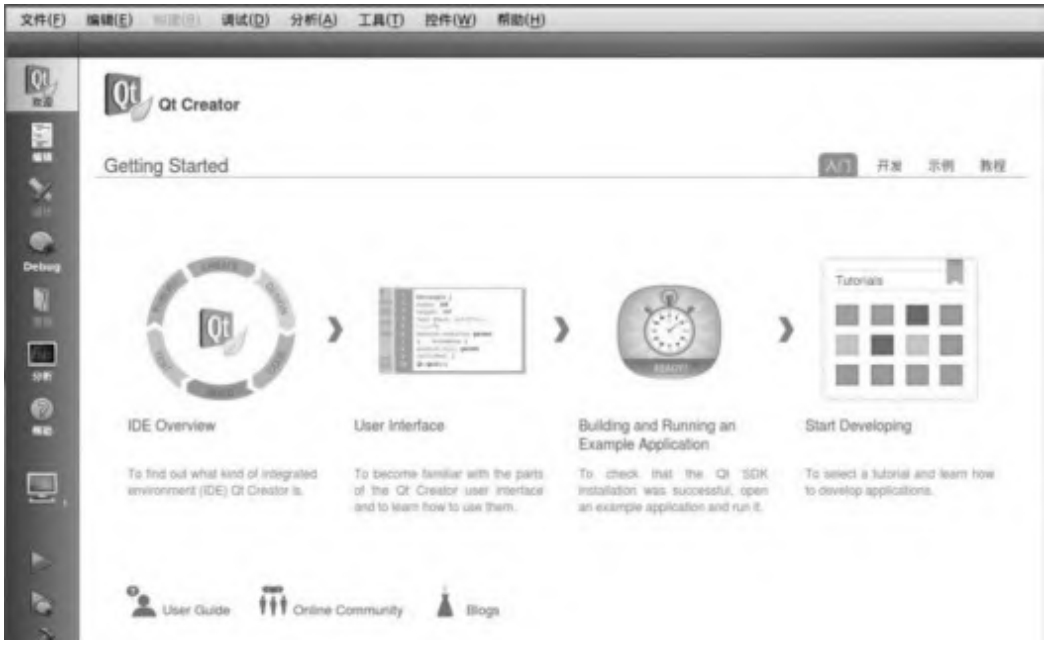


图5.3 Qt Creator 界面

单击“工具”菜单下的“选项”子菜单，进入如图5.4所示的设置界面。

图5.4 设置界面

接着单击左侧的“构建和运行”选项，再单击右侧的“Qt版本”选项卡。单击“添加”按钮，将x11和ARM的qmake路径添加进去，如图5.5所示。

图5.5 qmake配置

添加交叉编译工具链，如图5.6所示。

图5.6 添加编译器

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第五章03课（安装QtCreator编译环境）。

5.3 初体验Hello Word

搭建好Qt编译环境和将Qt-4.8.4移植至OK6410开发平台之后，这章用最经常用到的“HelloWord”程序打开OK6410的Qt程序设计。

单击“文件”菜单中的“新建文件或项目”，进入如图5.7所示的界面。

图5.7 新建工程

选择“文件和类”，再选择“Qt Gui应用”，然后单击“选择”按钮，进入如图5.8所示的界面。

图5.8 项目介绍和位置

在“名称”文本框中输入新建工程的名称，在“创建路径”中输入新建工程的保存路径。单击“下一步”按钮，进入如图5.9所示的界面。

图5.9 选择构建套件

在图5.9中勾选“桌面”复选框，以下两个目录存放着Qt应用程序和编译之后能在Ubuntu系统下运行的执行文件。勾选“arm”复选框，

以下两个目录存放着Qt应用程序调试和漏译之后能在arm单板中运行的执行文件。完成之后单击“下一步”按钮，如图5.10所示的界面。

图5.10 选择类信息

基类有3种：QWidget、QMainWindow和QDialog。

QWidget 类是所有用户界面对象的基类。窗口部件是用户界面的一个基本单元，它从窗口系统接收鼠标、键盘和其他事件，并且在屏幕上绘制自己。每一个窗口部件都是矩形的，并且它们按z轴顺序排列。一个窗口部件可以被它的父窗口部件或者它前面的窗口部件盖住一部分。

QMainWindow 类提供一个有菜单条、锚接窗口（例如工具条）和一个状态条的主应用程序窗口。主窗口通常用在提供一个大的中央窗口部件（例如文本编辑或者绘制画布）以及周围菜单、工具条和一个状态条。QMainWindow常常被继承，因为这使得封装中央部件、菜单和工具条以及窗口状态条变得更容易，当用户单击菜单项或者工具条按钮时，槽会被调用。

QDialog 类是对话框窗口的基类。对话框窗口主要用于短期任务以及和用户进行简要通信的顶级窗口。QDialog可以是模态对话框，也

可以是非模态对话框。QDialog支持扩展性并且可以提供返回值。它们可以有默认按钮。QDialog 也可以有一个 QSizeGrip 在它的右下角，使用setSizeGripEnabled()。

QDialog 是最普通的顶级窗口。一个不会被嵌入到父窗口部件的窗口部件叫做顶级窗口部件。通常情况下，顶级窗口是有框架和标题栏的窗口。在Qt中，QMainWindow和不同的QDialog的子类是最普通的顶级窗口。

如果是顶级对话框，那就是基于QDialog创建的；如果是主窗体，那就是基于QmainWindow创建的；如果不确定，或者有可能作为顶级窗体，或有可能嵌入到其他窗体中，则是基于QWidget创建的。当然了，实际中，还可以基于任何其他部件类来派生。

这里选择Qdialog类，完成之后单击“下一步”按钮，进入如图5.11所示的界面。

图5.11 项目管理

这里无需配置，单击“完成”按钮，则名为“HelloWord”的Qt工程建立完成。进入“HelloWord”程序设计，如图5.12所示。

图5.12 HelloWord工程

单击左侧“界面文件”下的“dialog.ui”，进入界面设计，如图5.13所示。

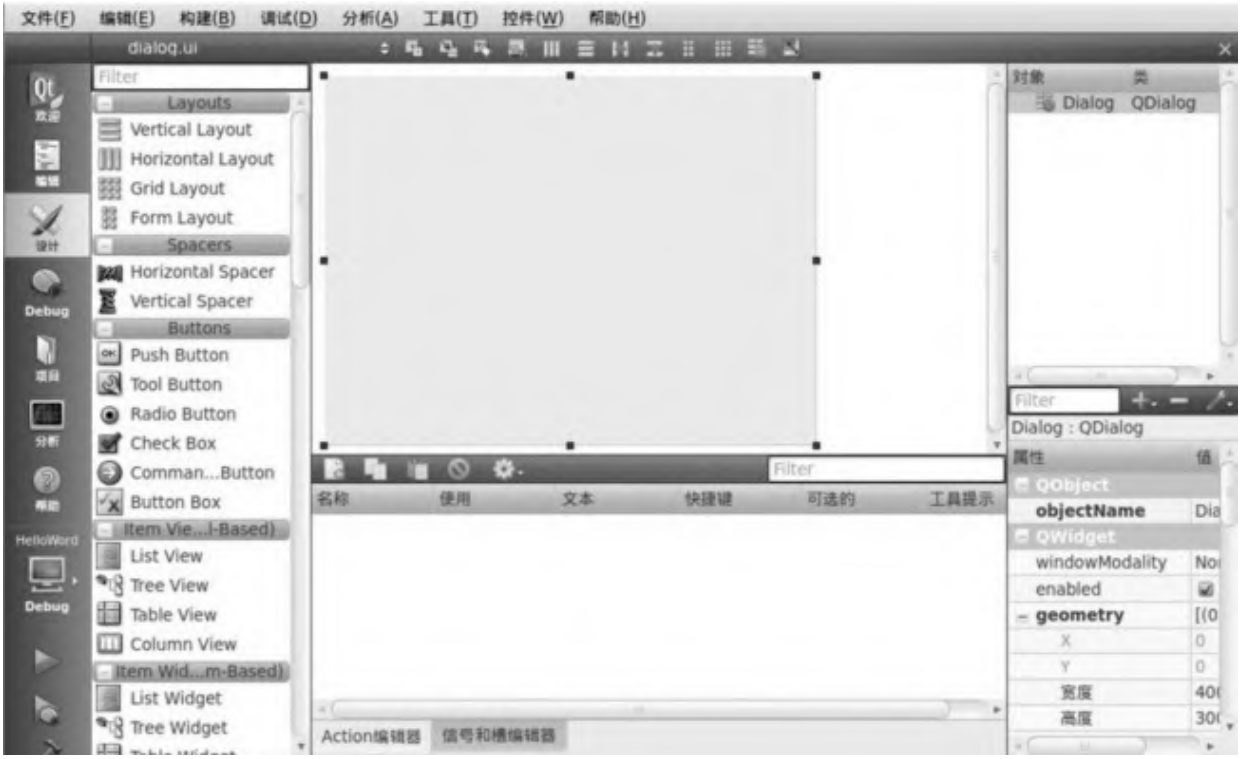


图5.13 界面设计

由于OK6410的LCD分辨率为480*272，所以高度和宽度应该修改适合OK6410开发平台，如图5.14所示。

在Qt界面设计的左侧有很多与Qt相关的属性设计类，本章就简单设计一个Qt界面，显示“HelloWord”字样。这里选择“Lable”，接着输入“Hello Word”，如图5.15 所示。

属性	值
enabled	☑
geometry	[(0, 0), 480 ...
X	0
Y	0
宽度	480
高度	272
sizePolicy	[Preferred, ...
minimumSize	0 x 0
maximumSize	16777215 x...
sizeIncrement	0 x 0

图5.14 Qt界面高度和宽度设置

图5.15 Hello Word设计

完成Qt程序设计之后，单击左下方的“Release”进行调试和生成相应的应用文件。如图5.16所示。

选择“arm”中的“Release”之后，单击左下方的“绿色三角形”生成应用文件。完成之后在/qtcreator-2.7.0/app目录下将生成

相应的Release目录。

zhuzhaoqi@zhuzhaoqi-desktop:~/qtcreator-2.7.0/app$ ls

build-HelloWord-arm-Release build-HelloWord-桌面-Release HelloWord

图5.16 Debug和Release

进入build-HelloWord-arm-Release目录，当中的HelloWord即为运行在OK6410开发板的Qt文件。如下所示：

zhuzhaoqi@zhuzhaoqi-desktop:~/qtcreator-2.7.0/app/build-HelloWord-arm-Release$ ls

dialog.o main.o moc_dialog.cpp ui_dialog.h

HelloWord Makefile moc_dialog.o

将HelloWord文件拷贝到文件系统的/usr/bin/目录下：

zhuzhaoqi@zhuzhaoqi-desktop:~/qtcreator-2.7.0/app/build-HelloWord-arm-Release$ sudo cp HelloWord /home/zhuzhaoqi/rootfs/usr/bin/

在/usr/bin/目录下建立env脚本，为Qt环境参数进行设置：

#!/bin/sh

exportPATH=/opt/Qt-4.8.4/lib:/opt/Qt-4.8.4/bin:/sbin:/usr/sbin:/bin:/usr/bin

export QPEDIR=/opt/Qt-4.8.4

export QTDIR=/opt/Qt-4.8.4

export QT_QWS_FONTDIR=/opt/Qt-4.8.4/lib/fonts

export QWS_DISPLAY=LinuxFb:mmWidth152:mmHeight88:1

export QWS_MOUSE_PROTO=tslib:/dev/input/event0

export TSLIB_CALIBFILE=/etc/pointercal

export TSLIB_CONFFILE=/usr/local/tslib/etc/ts.conf

export TSLIB_CONSOLEDEVICE=none

export TSLIB_FBDEVICE=/dev/fb0

export TSLIB_PLUGINDIR=/usr/local/tslib/lib/ts

export TSLIB_ROOT=/usr/local/tslib

export TSLIB_TSDEVICE=/dev/input/event0

export TSLIB_TSEVENTTYPE=H3600

export QT_PLUGIN_PATH=/opt/Qt-4.8.4/plugins

export LD_LIBRARY_PATH=/opt/Qt-4.8.4/lib

执行env脚本之后，在开发板中执行HelloWord文件：

[YJR@zhuzhaoqi]\# ./HelloWord -qws

此时可以看到OK6410开发平台显示HelloWord窗口，效果如所示。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第五章 04 课（Qt初体验之Hello）。

5.4 字符设备驱动Qt应用程序

其实字符驱动设备驱动程序的应用程序在第4章已经初步尝试过，只是没有漂亮的界面操作，本节使用Qt给应用程序添加漂亮的Qt图形界面。

5.4.1 基于Qt-4.8.4的LED应用程序

前一章中的LED设备驱动程序我们已经使用简单的应用程序测试过了，本节我们要完成基于Qt的LED应用程序。基于Qt的应用程序与之前的应用程序原理相通。

先建立一个LED的Qt工程，这里选择QWidget类，如图5.17所示。

图5.17 LED工程建立

建立好工程之后，进入LED应用程序的界面设计，如图5.18所示。

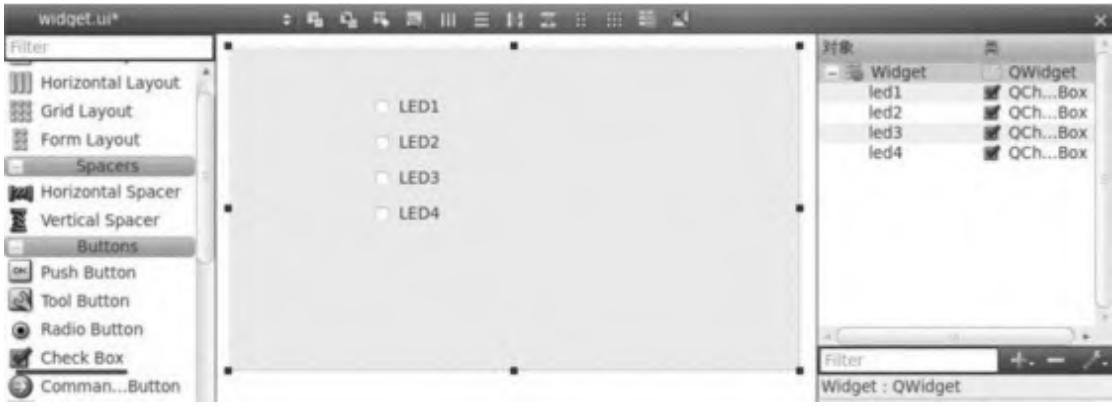


图5.18 LED应用程序界面设计

这里使用到了“Check Box”属性，将其拖放至界面，并更名为“LED1～LED4”，同时将右边“Widget”属性下面更名为“led1～led4”。完成之后进入编程。

在widget.h 中添加“Check Box”的相应函数：

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
namespace Ui {
class Widget;
}
  class Widget : public QWidget
  {
    Q_OBJECT
  public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();
  private:
    Ui::Widget *ui;
    int led_fd;
```

```
  private slots:
    void CheckBoxClicked();
  };
  #endif // WIDGET_H
```

在widget.cpp中添加LED相应的处理代码。

```
#include "widget.h"
#include "ui_widget.h"
//checkbox所需的头文件
#include <qcheckbox.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
  {
    ui->setupUi(this);
  //打开设备文件
    led_fd = ::open("/dev/led", O_RDONLY);
    if (led_fd < 0)
    {
      led_fd = ::open("/dev/led", O_RDONLY);
    }
```

```
    connect(ui->led1, SIGNAL(clicked()), this,
SLOT(CheckBoxClicked()) );
    connect(ui->led2, SIGNAL(clicked()), this,
SLOT(CheckBoxClicked()) );
    connect(ui->led3, SIGNAL(clicked()), this,
SLOT(CheckBoxClicked()) );
    connect(ui->led4, SIGNAL(clicked()), this,
SLOT(CheckBoxClicked()) );
    CheckBoxClicked();
  }
  Widget::~Widget()
  {
    delete ui;
  }
  void Widget::CheckBoxClicked()
  {
    ioctl(led_fd, int(ui->led1->isChecked()), 0);
    ioctl(led_fd, int(ui->led2->isChecked()), 1);
    ioctl(led_fd, int(ui->led3->isChecked()), 2);
    ioctl(led_fd, int(ui->led4->isChecked()), 3);
  }
```

这里应用到了Qt的信号与槽。

```
connect(ui->led1, SIGNAL(clicked()),
this,SLOT(CheckBoxClicked()) );
```

这里每个“CheckBox”发出一个信号，相应就会有一个槽接收：

```
ioctl(led_fd, int(ui->led1->isChecked()), 0);
```

信号和槽机制是Qt的核心机制，要熟悉Qt编程就必须对信号与槽进行深入了解。信号和槽是一种高级接口，应用于对象之间的通信，它是Qt的核心特性，也是Qt区别于其他工具包的重要地方。信号和槽是Qt自行定义的一种通信机制，它独立于标准的C/C++语言，因此要正确地处理信号与槽，必须借助一个称谓moc的Qt工具，该工具是C++的预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

通过调用QObject对象的connect函数来将对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。函数原型：

```
bool QObject::connect ( const QObject * sender, const char * signal,
        const QObject * receiver, const char * member )
[static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。当指定信号 signal 时必须使用Qt的宏SIGNAL()，当指定槽函数member 时必须使用SLOT()，SIGNAL()和SLOT()是把参数转换成字符串。如果发射者和接收者属于同一对象的话，那么在connect调用中接收者参数可以省略。

下面定义两个对象：标签对象 label 和滚动条对象 scroll，并将 valueChanged() 信号与标签对象的setNum()相关联，另外信号还携带了一个整型参数，这样标签总是显示滚动条所处位置的值。

```
QLabel*label = new QLabel;
QScrollBar *scroll = new QScrollBar;
QObject::connect( scroll, SIGNAL(valueChanged(int)),
    label, SLOT(setNum(int)) );
```

一个信号可以和多个槽相互关联：

connect(slider, SIGNAL(valueChanged(int)),spinBox, SLOT(setValue(int)));

connect(slider, SIGNAL(valueChanged(int)),this, SLOT(Indicator(int)));

这样槽虽然会一个接一个的被调用，但是调用的顺序是不定的。

多个信号亦可以和一个槽相互关联：

connect(lcd, SIGNAL(overflow()),this, SLOT(handleMathError()));

connect(calculator, SIGNAL(divisionByZero()),this, SLOT(handleMathError()));

只要任意一个信号发出，槽就会被调用。

一个信号可以连接到另外一个信号：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
    this, SIGNAL(updateRecord(const QString &)));
```

当lineEdit信号发出的时候，this这个信号亦会被发出。

槽可以被取消连接：

disconnect(lcd, SIGNAL(overflow()),this, SLOT(handleMathError()));

这种情况并不经常出现，因为当一个对象delete之后，Qt自动取消所有连接到这个对象上面的槽。

为了正确地连接信号槽，信号和槽的参数个数、类型以及出现的顺序都必须相同：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
    this, SLOT(processReply(int, const QString &)));
```

如果信号的参数多于槽的参数，那么这个参数之后的那些参数都会被忽略掉：

connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),

this, SLOT(checkErrorCode(int)));

const QString &这个参数就会被槽忽略掉。

设计好LED应用程序，进行Release之后，将其生成的LED应用文件拷贝至OK6410开发平台的文件系统中。

启动开发板，加载led驱动：

[YJR@zhuzhaoqi]\# insmod led.ko

建立led驱动的设备节点：

[YJR@zhuzhaoqi]\# mknod /dev/led c 240 0

执行LED的Qt应用程序：

[YJR@zhuzhaoqi]\# ./LED -qws

效果如图5.19所示。

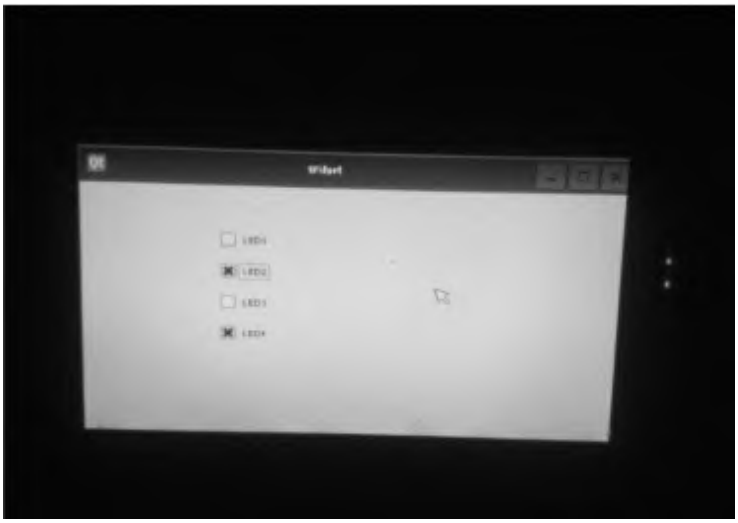


图5.19 LED应用程序执行效果

单击Qt界面上的LEDX，即可熄灭相应的LED。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第五章 05 课（Qt之LED）。

5.4.2 基于Qt-4.8.4的ADC应用程序

ADC 的驱动程序在上一章中已经详细讲解过了，这一小节就 ADC 采样回来的值显示在 Qt界面进行学习。

建立一个ADC工程，使用widget类型，在adc.ui界面中添加QLCDNumber类用来显示ADC采样值和转换之后的电压值。布局如图5.20所示。

图5.20 ADC应用界面

完成之后进行程序设计。ADC采样需要一个采样周期，因此需要在widget.h中添加时间事件。

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
namespace Ui {
  class Widget;
  }
  class Widget : public QWidget
```

```
{
  Q_OBJECT
public:
  explicit Widget(QWidget *parent = 0);
  ~Widget();
protected:
  void timerEvent(QTimerEvent *);
private:
  Ui::Widget *ui;
};
#endif // WIDGET_H
```

timerEvent（QTimerEvent *）函数的入口参数单位是ms，经过设定的时间将会响应一次这个函数。

在widget.cpp函数中打开ADC设备文件，提取ADC采样值，间隔为1000ms。

```
#include "widget.h"
#include "ui_widget.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/fs.h>
#include <errno.h>
#include <string.h>
```

```
Widget::Widget(QWidget *parent) :
  QWidget(parent),
  ui(new Ui::Widget)
{
  ui->setupUi(this);
  //1000ms响应一次
  startTimer(1000);
}
Widget::~Widget()
{
  delete ui;
}
void Widget::timerEvent(QTimerEvent *)
{
  int fd = ::open("/dev/zzqadc", O_RDWR);
  if (fd < 0)
  {
    return;
  }
  int adc_data = 0;
  adc_data = read(fd,NULL,0);
  ui->adValue->display(adc_data);
  ui->vValue->display( ((float)(adc_data)) * 3.3 / 1024
);
  ::close(fd);
}
```

这里显示了两个数据，一个是采样回来的 adc_data，即为采样值；第二个是经过转换之后得到的电压值，由于这里采用的是10位AD、3.3V的参考电压，所以采样值和电压值之间的关系是：3.3 / 1024。

接着进行Release，将生成的adc应用文件拷贝至开发板的文件系统下。

zhuzhaoqi@zhuzhaoqi-desktop:~/qtcreator-2.7.0/app/build-adc-arm-Release$ sudo cp adc/home/zhuzhaoqi/rootfs/usr/bin/

加载ADC驱动程序：

[YJR@zhuzhaoqi]\# insmod zzqadc.ko

dev_init return ret: 0

[YJR@zhuzhaoqi]\# lsmod

zzqadc 1459 0 - Live 0xbf000000

加载成功之后运行ADC的Qt应用程序：

[YJR@zhuzhaoqi]\# ./adc -qws

可以看到开发板上成功显示采样值和电压值。

本节配套视频位于光盘中“嵌入式 Linux 开发实用教程视频”目录下第五章 06 课（Qt之ADC）。

第6章 嵌入式Linux学习拓展

前5章，从Linux基础到U-Boot移植，从Linux内核移植到Linux设备驱动，再从Qt的移植到Qt应用程序的设计，终于对嵌入式Linux有了一定的认识。本章将抽取笔者曾经完成的“基于S3C6410的生理参数采集系统”部分模块进行实战演练，让读者能对前5章知识进行巩固和升华。

6.1 学习拓展简介

“基于 S3C6410 的生理参数采集系统”可以实现对人体的生理参数如心电、心音、脉搏、血压参数、体表温度、呼吸与睡眠鼾声等进行检测。本生理参数采集系统主要用于儿童、老人、病人等的监测，通过GPRS模块将采集系统采集的数据发送至监护人手机，让监护人实时了解监测者的生理参数。

生理参数采集的各个模块都是通过传感器将模拟信号转换成数字信号，所以笔者在本章就抛砖引玉，选取体表温度采集进行深入分析，将采集的温度信息通过 GPRS 发送至监护人手机中。笔者希望等读者完成本章学习之后，能独自将“基于S3C6410的生理参数采集系统”剩余部分完成。

6.2 Linux驱动程序设计

作为连通硬件和应用程序的驱动程序，在项目设计中有着举足轻重的地位，U-Boot、Linux内核和文件系统在之前章节已经完成建立，本节就从驱动着手，学习温度采集和GPRS的原理，完成驱动程序的设计。

6.2.1 温度传感器模块

1．温度传感器简介

能对温度进行采集的传感器是多样性的，有铂电阻、DS18B20等、由于采集系统是 24 小时对生理参数进行监测的，对温度瞬时性变化要求不是很高，所以从采集的方便性、实用性考虑，本采集系统采用DS18B20温度传感器对体表温度进行采集。

市场上的DS18B20传感器有多种形状，常见的有TO-92封装、SOSI封装等，但是不管何种封装，原理图都相似，如图6.1所示。从本采集系统的实用性考虑，采用TO-92封装，如图6.2所示。

DS18B20温度传感器适应电压范围为：3V～5.5V，测量温度范围为：－55℃～125℃，测量分辨率为9～12位，对应的可分辨温度分别为0.5℃、0.25℃、0.125℃和0.0625℃。DS18B20传感器采用独特的单线接口方式，与微处理器连接使用仅需要DQ单根数字信号线即可实现微处理器与DS18B20的双向通信，可实现组网多点测温。

DS18B20传感器和S3C6410的连接原理图如图6.3所示。

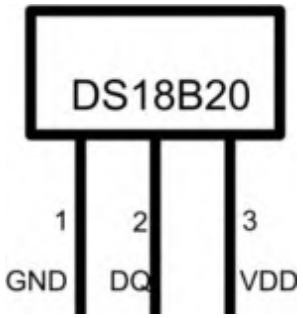


图6.1 DS18B20传感器原理图

图6.2 DS18B20的TO-92封装

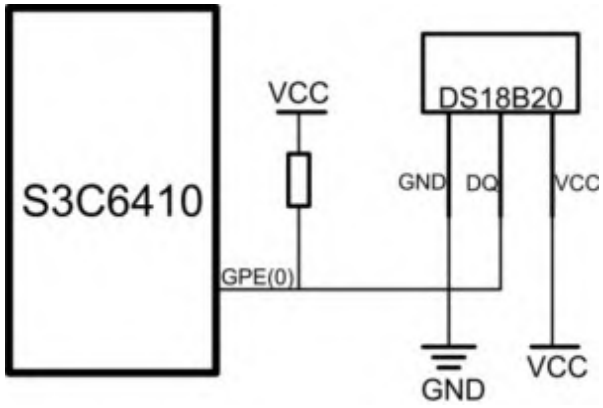


图6.3 DS18B20传感器与S3C6410连接原理图

这里我们采用GPE（0）引脚作为与DS18B20的数据线连接引脚，当然读者可以取任何一个具有输入输出功能的引脚使用。

2．驱动程序设计

通过上一小节，已经对DS18B20有了初步的认识，接下来就对DS18B20传感器设计编写驱动程序。对于驱动程序的编写，阅读芯片手册是每个驱动工程师的必会技能。从《DS18B20手册》中，我们可以初

步将驱动程序分为四部分：初始化、写操作、读操作、读取温度值。由于DS18B20的驱动程序并不复杂，因此这里我们一边阅读《DS18B20手册》，一边写驱动程序。

（1）宏定义

```
//初始化失败标记
#define DS18B20_ERROR  0x01
//驱动模块名称
#define DEVICE_NAME  "zzqds18b20"
//设备号
#define DS18B20_MAJOR  243
```

如果DS18B20初始化失败，则会返回0x01。这里宏定义DS18B20的驱动模块名称和设备驱动号。

```
//设置GPE(0)引脚IO为推挽输出模式
#define SetGPE0out()  s3c_gpio_cfgpin( S3C64XX_GPE(0), S3C_GPIO_SFN(1) )
//设在GPE(0)引脚IO输入模式
#define SetGPE0in()    s3c_gpio_cfgpin( S3C64XX_GPE(0), S3C_GPIO_SFN(0) )
//向GPE(0)写入数据
#define WriteGPE0(data)  gpio_set_value( S3C64XX_GPE(0), data )
//读取GPE(0)的当前数据值
#define ReadGPE0()    gpio_get_value( S3C64XX_GPE(0) )
```

（2）初始化

《DS18B20手册》所提供的复位时序图如图6.4所示。

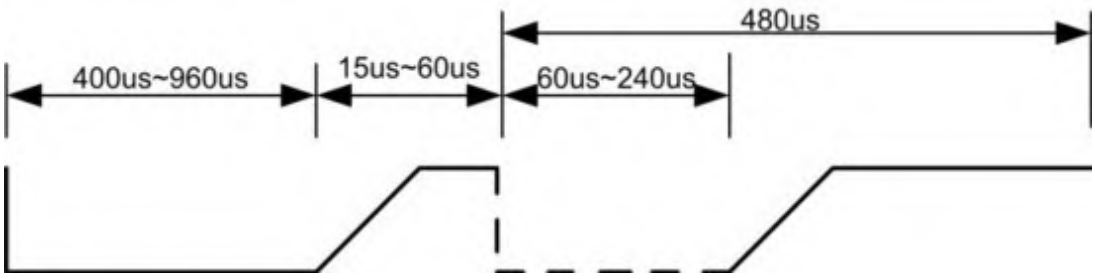


图6.4 DS18B20初始化时序图

从图6.4可知上电之后第一时刻给DS18B20传感器发出一个最少为480us低电平的复位脉冲，接着进入接收状态，DS182传感器在检测到总线的上升沿之后等待15 us～60us，然后DS1820发出持续60us～240us低电平的存在脉冲。根据这个原理，我们可以写出如下初始化函数：

```
/*
* 复位函数
*/
static int DS18B20Reset(void)
{
  int iStatus = 0;
  /* 设置GPE(0)为输出模式 */
  SetGPE0Out();
  /* GPE(0)输出一个测试高电平 */
  WriteGPE0(1);
  udelay(100);
  /* GPE(0)输出复位低电平脉冲 */
  WriteGPE0(0);
  /* 480us～960us，这里我们取600us */
  udelay(600);
  /* GPE(0)输出一个测试高电平 */
```

```
    WriteGPE0(1);
    udelay(100);
    /* GPE(0)进入接收状态 */
    SetGPE0In();
    /* 读取回复信号 */
    iStatus = ReadGPE0();
    udelay(100);
    SetGPE0Out();
    /* 返回复位的状态，成功抑或失败 */
    return (iStatus);
  }
```

（3）写操作

写操作分为：控制器写0时序（如图6.5所示）和控制器写1时序（如图6.6所示）这两种。

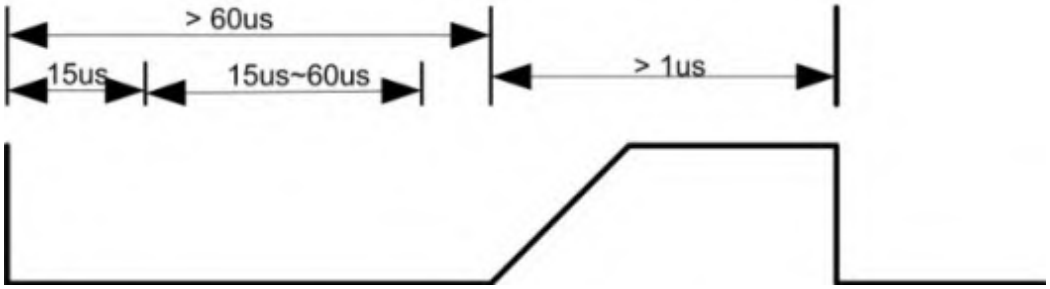


图6.5 控制器写0时序时序图

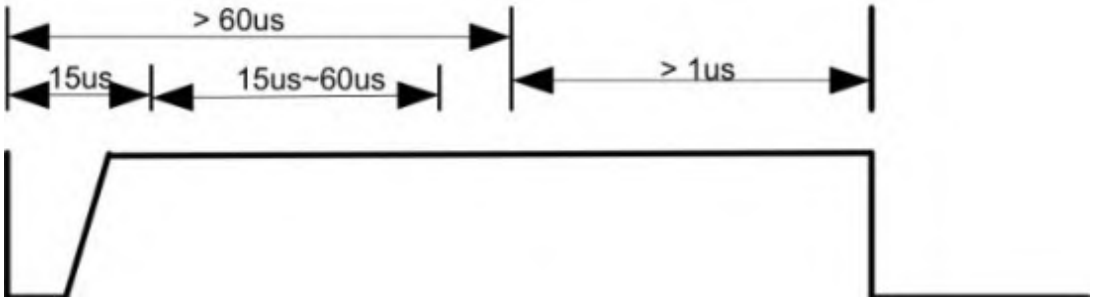


图6.6 控制器写1时序

当DQ从高电平拉为低电平，此时将在15us之内将所需写的位送至DQ总线上，在之后的15us～60us之内对DQ总线进行采样。如果是写0时序，则是写入0；如果是写1时序，则是写入1。

这里我们采用写0时序进行写操作，驱动程序如下：

```
/*
  * 写操作函数
  */
static void DS18B20WriteData(int iData)
{
  int i;
  /* 设置GPE(0)为输出模式 */
  SetGPE0Out();
  /* 写入数据 */
  for(i = 0; i < 8; i++)
  {
    WriteGPE0(0);
    udelay(12);
    WriteGPE0(iData & 0x01);
    udelay(30);
    WriteGPE0(1);
    iData >>= 1;
    udelay(2);
  }
}
```

（4）读操作

读操作与写操作相对应，也就是：控制器读0时序和控制器读1时序。读者可参考写操作进行自行分析，读操作代码如下：

```
/*
  * 读操作
  */
static int DS18B20ReadData(void)
{
  int i, iData = 0;
  for(i = 0; i < 8; i++)
  {
    SetGPE0Out();
    WriteGPE0(0);
    iData >>= 1;
    udelay(12);
    WriteGPE0(1);
    SetGPE0In();
    udelay(1);
    if(ReadGPE0())
      iData |= 0x80;
    udelay(60);
  }
  return iData;
}
```

（5）读取温度值

当S3C6410检测到DS18B20存在，便可对其发送出ROM的控制命令，如下所示：

[33H]读ROM

[55H]匹配ROM

[CCH]跳过ROM

[F0H]搜索ROM

[ECH]告警搜索

DS18B20自身也有6条控制命令，如下所示：

[44H] 温度转换启动DS18B20 进行温度转换

[BEH] 读暂存器9 位二进制数字

[4EH] 写暂存器将数据写入暂存器的TH、TL 字节

[48H] 复制暂存器把暂存器的TH、TL 字节写到E2RAM 中

[B8H] 重新调E2RAM，把E2RAM 中的TH、TL 字节写到暂存器TH、TL 字节中

[B4H] 读电源供电方式启动DS18B20 发送电源供电方式的信号给主CPU

读取温度值的第一步是对DS18B20进行复位操作，并且判断是否复位成功。如下：

```
//如果复位失败
if( 1 == DS18B20_Reset() )
{
    return DS18B20_ERROR;
}
//否则是成功，则进行读取温度值操作
else
{
//读取温度值
}
```

一般跳过ROM检测即可，开始读温度值，如下：

```
udelay(400);
    DS18B20WriteData(0xcc);
    DS18B20WriteData(0xbe);
```

DS18B20的温度操作是16位，如图6.7所示。

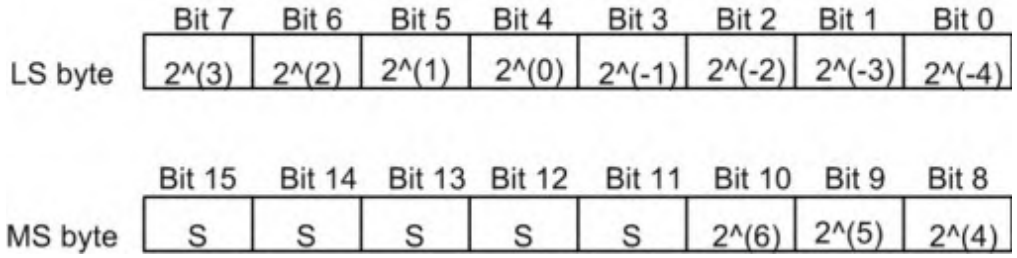


图6.7 温度寄存器格式

Bit11～Bit15这5位是温度正负标志位，当Bit11～Bit15全为1时，则温度为负；若Bit11～Bit15全为0时，则温度为正。

进行温度读取，这里我们使用一个数组存储温度的高8位和低8位，如下：

```
//温度低8位
    TempValue[0] = DS18B20ReadData();
    //温度高8位
    TempValue[1] = DS18B20ReadData();
```

读取完成之后需要对其进行转换处理，可以是整型、保留一位小数抑或保留两位小数，情况如下：

```
//保留两位小数
Temp = ( (TempValue[1]<< 8) | TempValue[0]) * (0.0625 *100)
//保留一位小数
Temp = ( (TempValue[1]<< 8) | TempValue[0]) * (0.0625 *10)
//保留0位小数
Temp = ( (TempValue[1]<< 8) | TempValue[0]) * (0.0625 *1)
```

由于这里是测量体温，因此不存在负数温度的问题。如果是负数，则要进行转换：

```
//高5位全部为1(0xf8)则为负数,为负数时取反加1,并且设置负数标志
if( 0xf8 == ( TempValue[1] & 0xf8) )
{
  TempValue[1] = ～TempValue[1] ;
  TempValue[0] = ～TempValue[0] + 1 ;
  if( 0x00 == TempValue[0])
  {
    TempValue[1]++;
  }
  //负数标志
iFlag = 1;
  }
```

（6）驱动程序核心控制

前面已经完成了 DS18B20 传感器的基本操作，接下来将使用 Linux 字符驱动操作函数调用。

```
static ssize_t DS18B20_read(struct file *file, char __user *buff,
  size_t size, loff_t *loff)
{
  ……
  temp = DS18B20_ReadTemper();//读取温度
  ……
}
```

整个驱动程序的核心所在如下：

```
struct file_operations ds18b20_fops = {
    .owner= THIS_MODULE,
```

```
    .write= DS18B20_write,
    .read= DS18B20_read,
    .unlocked_ioctl = DS18B20_ioctl,
    .release= DS18B20_release,
};
```

笔者将DS18B20的核心框架已经搭建好，希望读者能联系之前的设备驱动将驱动程序补充完整和自行编写测试应用程序进行测试。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第六章01课（项目拓展学习（1））。

6.2.2 GPRS模块

在“基于S3C6410的生理参数采集系统”项目中使用的GPRS是华为GTM900，如图6.8所示。华为GTM900无线模块是一款三频段GSM/GPRS的无线模块，它支持标准的AT命令及增强AT命令，提供丰富的语音和数据业务等功能，是高速数据传输等各种应用的理想解决方案。

图6.8 GTM900型号

用户可以通过 AT 指令集进行呼叫、短信、电话本、数据业务、补充业务、传真等方面的控制。下面通过GPRS模块与PC机连接测试，对常用的AT指令集进行详细讲解。

使用串口线连接GPRS与PC机，将SIM卡放入GPRS的卡槽，接上电源线之后上电。

打开超级终端，输入连接名称，选择串口（COM1 或COM2，根据你准备工作中Modem 连接的串口选择），对端口进行设置：

波特率：115200，

数据位：8，

奇偶校验：无，

停止位：1，

数据流控制：无。

在超级终端中输入“AT”之后按下回车键，返回“OK”，说明Modem 处于正常工作状态，如下所示：

at

OK

AT指令集的命令是不区分大小写的，“at”等效为“AT”。

1．发送短信

发送短信有两种格式：PDU方式和TEXT方式。PDU方式指的是包括所有头信息的短消息，是以二进制的方式传送（写成十六进制的格式）的；TEXT 方式指的是命令和响应均为ASCII字符。

at+cmgf=0 //PDU 方式

at+cmgf=1 //TEXT 方式

使用TEXT方式发送短信：

at

OK

at+cmgf=1

OK

at+cmgs=188xxxxxxx

>Linux-3.8.3

+CMGS: 64

OK

使用TEXT 方式发短信时，“at+cmgs=188 xxxx xxxx”，“188 xxxx xxxx”是被叫短信号码。在输入“Linux-3.8.3”信息完成之后按下“Ctrl+Z”，即为发送信息。如果回应的是“OK”，说明将短信发送出去了。

2. 读取短信

读取短消息和发送是相对的，也分为PDU方式和TEXT方式。

at+cmgr=0 //PDU 方式

at+cmgr=1 //TEXT 方式

使用TEXT方式读取短信：

at

OK

at+cmgr=1

+CMGR: "REC UNREAD","86188xxxxxxxx",,"13/05/01,20:22:20+32",145,4,0,0,"861380075

5500",145,5

Linux

最后一行“Linux”即为“188xxxxxxxx”号码所发的信息。

3. 列举信息

使用“at+cmgl”命令来读取已存储的短信。PDU 方式和 TEXT 方式所对应的列举信息如表6.1所示。

表6.1 列举信息方式

PDU 方式	TEXT 方式	读取说明
0	“REC UNREAD”	接收未读
1	“REC READ”	接受已读
2	“STO UNSENT”	存储未发送
3	“STO SENT”	存储已发送
4	“ALL”	所有消息

列举所有信息：

at+cmgl="ALL"

+CMGL: 1,"REC READ","86188xxxxxxxx",,"13/05/01,20:22:20+32",145,5

Linux

GPRS 不仅仅具有短信服务功能，还有电话服务、网络服务等功能。笔者希望读者了解短信服务功能之后，能自行参考《AT指令集》学习其他功能。

6.3 Qt应用程序设计

通过上一节，读者对DS18B20温度传感器和GTM900GPRS的原理有了认识，本小节笔者要带领读者完成的是基于Qt的应用程序设计，给“基于S3C6410的生理参数采集系统”一个漂亮的操作界面。

6.3.1 DS18B20温度传感器

在“基于S3C6410的生理参数采集系统”中，DS18B20是针对人体温度进行实时监测的，因此从效果上，只需要显示实时温度即可。

新建一个名为“DS18B20”的Qt工程，进入工程的界面设计“widget.ui”进行Qt界面设计。从视觉和效果考虑，温度显示采用

“LCD Number”，Qt界面设计如图6.9 所示。

图6.9 DS18B20界面设计

在widget.h文件中添加时间响应函数，如下所示：

```
class Widget : public QWidget
{
  Q_OBJECT
……
protected:
  void timerEvent(QTimerEvent *);
……
};
```

在widget.cpp文件中添加读取DS18B20温度传感器的时间响应函数实现，如下所示：

```
Widget::Widget(QWidget *parent) :
  QWidget(parent),
  ui(new Ui::Widget)
{
  ui->setupUi(this);
  //1s
  startTimer(1000);
}
```

```
/*
  * dispaly the temper
  */
void Widget::timerEvent(QTimerEvent *)
{
  int fd = ::open("/dev/zzqds18b20",O_RDWR);//open
DS18B20
  if (fd < 0)
  {
    return;
  }
  int temp_data = 0;
  temp_data = read(fd,NULL,0);
  ui->TempValue->display(temp_data / 10);
  ::close(fd);
}
```

实现读取温度程序较为简单，将读取出来的值除以10（是因为在驱动程序处理时为了保留一位小数，乘以了10得以转换成整数，因此这里除以10），得到温度原来的值。这里读取温度的刷新时间为1s。

笔者希望读者能把这个生理参数采集系统完成，并且增加自己的设计，将其功能更加完善，这对于读者编程能力和硬件设计能力都会有很大的提高。

本节配套视频位于光盘中“嵌入式Linux开发实用教程视频”目录下第六章02课（项目拓展学习（2））。

读累了记得休息一会哦~

网站：https://elib.cc

百万电子书免费下载

图书在版编目（CIP）数据

嵌入式Linux开发实用教程/朱兆祺，李强，袁晋蓉编著.--北京：人民邮电出版社，2014.4

ISBN 978-7-115-33483-1

Ⅰ.①嵌… Ⅱ.①朱…②李…③袁… Ⅲ.①Linux操作系统—程序设计—教材 Ⅳ.①TP316.89

中国版本图书馆CIP数据核字（2013）第254297号

内容提要

嵌入式Linux是将日益流行的Linux操作系统进行裁剪修改，使之能在嵌入式计算机系统上运行的一种操作系统。既继承了Internet上无限的开放源代码资源，又具有嵌入式操作系统的特性，其优势及应用已获得众多企业的青睐。

本书以一个嵌入式 Linux 学习者的角度，由浅入深地总结了从入门到进行项目工程实践的所有学习历程，旨在帮助读者快速入门，以实例为导向扎实掌握嵌入式开放技术。全书共分 6 章，主要内容包括嵌入式Linux基础、U-Boot移植、Linux移植、Linux驱动程序、Qt移植和程序设计以及举一反三的综合拓展学习。由于嵌入式Linux是一门非常复杂的软件技术，入门较难，因此借以此书为自学者提供一条成功入门的捷径。本书光盘包含了作者在本书基础上录制的40集学习视频，涵盖嵌入式Linux基础、U-Boot移植、Linux移植、Linux驱动程序设计、Qt移植等。本书的所有程序以及源码都在光盘中，读者可自行参考。

本书内容详实，结构明确，适合作为初学者的课程教材，也可作为嵌入式系统爱好者的自学参考资料。

◆编著 朱兆祺 李强 袁晋蓉

责任编辑 俞彬

责任印制 程彦红 焦志炜

◆人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆开本：787×1092　1/16
印张：16.75
字数：415千字　　2014年4月第1版
印数：1－3500册　　2014年4月北京第1次印刷
定价：45.00元（附光盘）
读者服务热线：(010)81055410　印装质量热线：
(010)81055316
反盗版热线：(010)81055315