内 容 提 要

本书全面介绍了嵌入式 Linux 系统开发过程中，从底层系统支持到上层 GUI 应用的方方面面，内容涵盖 Linux 操作系统的安装及相关工具的使用、配置，嵌入式编程所需要的基础知识（交叉编译工具的选项设置、Makefile 语法、ARM 汇编指令等），硬件部件的使用及编程（囊括了常见硬件，比如 UART、I^2C、LCD 等），U-Boot、Linux 内核的分析、配置和移植，根文件系统的构造（包括移植 busybox、glibc、制作映象文件等），内核调试技术（比如添加 kgdb 补丁、栈回溯等），驱动程序编写及移植（LED、按键、扩展串口、网卡、硬盘、SD 卡、LCD 和 USB 等），GUI 系统的移植（包含两个 GUI 系统：基于 Qtopia 和基于 X），应用程序调试技术。

本书从最简单的点亮一个 LED 开始，由浅入深地讲解，使读者最终可以配置、移植、裁剪内核，编写驱动程序，移植 GUI 系统，掌握整个嵌入式 Linux 系统的开发方法。

本书由浅入深，循序渐进，适合刚接触嵌入式 Linux 的初学者学习，也可作为大、中专院校嵌入式相关专业本科生、研究生的教材。

前　　言

背景知识

嵌入式 Linux 在嵌入式领域发展迅速，需求旺盛，但是嵌入式 Linux 的入门很难。初学者多是自己琢磨，效率不高。学习过程中碰到的问题千奇百怪，解决后却往往发现是极其低级的错误，以作者为例，初学时在论坛疯狂发帖求教，现在回头一看不免感叹：怎么会提出这么弱智的问题？但是，当时就是被这类问题折磨得寝食难安。

相对于嵌入式 Linux 常识的匮乏，更大的困难是缺乏完善的知识结构：只了解硬件，或是只了解软件。对于有志于从事底层系统开发（比如改造 Bootloader、钻研内核、为新硬件编写驱动程序）的人，对于想从上层软件开发转到底层软件开发的人，应该看得懂电路原理图，看得懂芯片数据手册，清楚地知道软件是怎样和硬件发生作用的。

同样，对于想从硬件岗位转到软件岗位的人，对于想从传统单片机（比如 51 单片机）编程进一步学习“有操作系统的”嵌入式编程的人，需要找到一个学习的切入点：先掌握各个硬件部件的简单编程，再将它们组合起来构成一个相对复杂的软件系统——比如 Bootloader，进而编写基于操作系统的驱动程序，最后深入钻研操作系统内核。

对于尚未参加工作的在校生来说，缺乏实际的操作经验可能是就业的最大障碍。很多人买了开发板想进一步练习，却发现不知从何入手。

鉴于上述种种困难及需求，作者结合自己的学习经历、工作心得写成此书，期望能帮助读者加快嵌入式 Linux 的入门速度，并体会到深入学习嵌入式 Linux 的乐趣。

关于本书

本书以 S3C2410、S3C2440 开发板为例，从分析硬件上电执行的第一条指令开始，到构造出一个类似 PDA、基于 Linux 的桌面 GUI 系统，带领读者学习、掌握从最底层到最高层的软件编写方法。

本书主要涉及以下主题：

- 开发环境的搭建（包括安装 Linux 系统及日常使用的工具）；
- 开发板上各硬件部件的使用方法及实际的编程操作；

- 嵌入式 Linux 系统的构造（包括 Bootloader、内核、文件系统等）；
- 嵌入式 Linux 驱动程序的编写方法及大量实例；
- GUI 系统的移植（两个 GUI 系统：基于 Qtopia 和基于 X）；
- 调试技术（包括内核调试技术和应用程序调试技术）。

本书所有章节都以理论结合代码的方式进行讲解，并可按照书中说明进行实际操作，力求让读者“知其然，也知其所以然”。

本书内容及组织方式

本书按照嵌入式 Linux 初学者的学习过程，从简单到复杂，从底层软件到上层软件进行讲解，全书分 5 篇，共 27 章。

第 1 篇（第 1 章至第 4 章）为嵌入式 Linux 开发环境构建篇，主要讲解以下内容。

- 第 1 章介绍基于 ARM 的嵌入式 Linux 系统的基本概念。
- 第 2 章讲解嵌入式开发环境的建立，包括在 PC 上安装、配置 Linux 操作系统，安装随书光盘。
- 第 3 章介绍交叉编译工具的选项、Makefile 的语法以及本书用到的 ARM 汇编指令及相关知识，这章可以当作阅读后续章节时的参考手册。
- 第 4 章介绍了一些日常工作要用到工具，比如源码阅读、编辑工具等。

第 2 篇（第 5 章至第 14 章）为 ARM9 嵌入式系统基础实例篇，具体内容如下。

本篇首先根据 S3C2410、S3C2440 的数据手册介绍各硬件部件的使用方法，然后介绍怎样编写程序来操作它们。文中穿插介绍了连接器的很多使用技巧，读者可以由此接触到“程序的内部结构”，这是单纯的上层开发人员所缺乏的。通过读写各个硬件部件的寄存器来操作硬件，读者还可以深刻体会到“软件”和“硬件”是怎样发生作用的，是第 3 篇、第 4 篇的基础。

第 3 篇（第 15 章至第 18 章）为嵌入式 Linux 系统移植篇，主要讲解以下内容。

- 第 15 章深入分析 U-Boot（它负责引导内核）的代码结构，并详细介绍了将它移植到开发板上的方法。
- 第 16 章首先分析了内核的代码结构，然后深入分析它的启动过程，最后将它移植到开发板上。
- 第 17 章先从整体上介绍了 Linux 文件系统的目录结构——FHS 标准。然后构造文件系统：移植常用工具的集合 Busybox，移植 glibc 库，建立各个目录，建立配置文件。最后修改、编译一些工具，使用它们来制作 yaffs、jffs2 文件系统映象文件。
- 第 18 章介绍了 3 种内核调试技术：printk、kgdb 补丁、使用 Oops 信息进行栈回溯。

第 4 篇（第 19 章至第 24 章）为嵌入式 Linux 设备驱动开发篇，具体内容如下。

在第 19 章中总体介绍了驱动程序的编写、移植方法，在第 20 章介绍了内核的异常处理体系结构——就是怎样使用中断。

其他章节都是一些例子：先总体介绍相关硬件的驱动程序架构，然后根据开发板的特性进行修改。

第 5 篇（第 25 章至第 27 章）为嵌入式 Linux 系统应用开发篇，主要讲解以下内容。

- 第 25 章移植了一个基于 Qtopia 的 GUI 系统，并且以简单的“Hello, world”程序为例编写、调试 GUI 程序。

- 第 26 章移植了一个基于 X 的 GUI 系统，里面涉及众多软件，读者可以体会到上层应用的开发过程，并且获得移植大型软件的经验。这章还介绍了一个名为 Scratchbox 的交叉编译工具包，它虚拟出一个可以直接编译软件的目标机器，使得“交叉编译”变为“本地编译”，大幅减少了为非 x86 平台移植软件所需的工作量。
- 第 27 章介绍了几种简便的应用程序调试技术，包括使用 strace 工具跟踪系统调用和信号，使用 memwatch 检查程序的内存漏洞，使用库函数 backtrace 和 backtrace_symbols 来定位段错误。

本书特色

- 由浅入深，从最简单的点亮 LED 讲起直至移植 GUI 系统。
- 实例丰富，每个实例都详尽地介绍原理及分析代码。
- 结构合理，先总体介绍概念、架构，然后进行具体操作。
- 包括初学者所碰到的常见问题。

参与本书编写的人员

本书由韦东山负责编写并统编全部书稿，陈汉仪、于明俭对本书的写作提供了大力支持，在此表示感谢。

感谢我的父母和女友，在本书写作过程中给了我强大的精神支持，鼓励、支持我，使我能够坚持写完本书。

同时参与编写的还有柴作朋、单辉、丁鹏、冯发勇、付贤会、葛仕明、何国宝、何圆明、何化成、黄永华、李志宏、廖娟、林清妹、陆江萍、祁晓璐、谭爱华、魏明辉、张帮芹、周霜、朱旭琪等，在此一并表示感谢。

我们为本书开通了专用的网站，网址是 http://www.100ask.net，读者可以直接同我们交流，共同学习和提高。

由于水平有限，书中难免遗漏和不足之处，恳请广大读者提出宝贵意见。本书责任编辑的联系方式是 huangyan@ptpress.com.cn，欢迎来信交流。

编　者

2008 年 6 月

目　　录

第 1 篇　嵌入式 Linux 开发环境构建篇

第 2 篇 ARM9 嵌入式系统基础实例篇

第 3 篇 嵌入式 Linux 系统移植篇

第 4 篇　嵌入式 Linux 设备驱动开发篇

第 5 篇 嵌入式 Linux 系统应用开发篇

第 15 章　移植 U-Boot

本章目标

- 了解 Bootloader 的作用及工作流程
- 了解 U-Boot 的代码结构、编译过程
- 移植 U-Boot
- 掌握常用的 U-Boot 命令

15.1　Bootloader 简介

15.1.1　Bootloader 的概念

1．Bootloader 的引入

从前面的硬件实例可以知道，系统上电之后，需要一段程序来进行初始化：关闭 WATCHDOG、改变系统时钟、初始化存储控制器、将更多的代码复制到内存中等。如果它能将操作系统内核复制到内存中运行，无论从本地（比如 Flash）还是从远端（比如通过网络），就称这段程序为 Bootloader。

简单地说，Bootloader 就是这么一小段程序，它在系统上电时开始执行，初始化硬件设备、准备好软件环境，最后调用操作系统内核。

可以增强 Bootloader 的功能，比如增加网络功能、从 PC 上通过串口或网络下载文件、烧写文件、将 Flash 上压缩的文件解压后再运行等，这就是一个功能更为强大的 Bootloader，也称为 Monitor。实际上，在最终产品中用户并不需要这些功能，它们只是为了方便开发。

Bootloader 的实现非常依赖于具体硬件，在嵌入式系统中硬件配置千差万别，即使是相同的 CPU，它的外设（比如 Flash）也可能不同，所以不可能有一个 Bootloader 支持所有的 CPU、所有的电路板。即使是支持 CPU 架构比较多的 U-Boot，也不是一拿来就可以使用的（除非里面的配置刚好与你的板子相同），需要进行一些移植。

2．Bootloader 的启动方式

CPU 上电后，会从某个地址开始执行。比如 MIPS 结构的 CPU 会从 0xBFC00000 取第一条指令，而 ARM 结构的 CPU 则从地址 0x0000000 开始。嵌入式开发板中，需要把存储器件

ROM 或 Flash 等映射到这个地址，Bootloader 就存放在这个地址开始处，这样一上电就可以执行。

在开发时，通常需要使用各种命令操作 Bootloader，一般通过串口来连接 PC 和开发板，可以在串口上输入各种命令、观察运行结果等。这也只是对开发人员才有意义，用户使用产品时是不用接串口来控制 Bootloader 的。从这个观点来看，Bootloader 可以分为以下两种操作模式（Operation Mode）。

（1）启动加载（Boot loading）模式。

上电后，Bootloader 从板子上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。产品发布时，Bootloader 工作在这种模式下。

（2）下载（Downloading）模式。

在这种模式下，开发人员可以使用各种命令，通过串口连接或网络连接等通信手段从主机（Host）下载文件（比如内核映象、文件系统映象），将它们直接放在内存运行或是烧入 Flash 类固态存储设备中。

板子与主机间传输文件时，可以使用串口的 xmodem/ymodem/zmodem 协议，它们使用简单，只是速度比较慢；还可以使用网络通过 tftp、nfs 协议来传输，这时，主机上要开启 tftp、nfs 服务；还有其他方法，比如 USB 等。

像 Blob 或 U-Boot 等这样功能强大的 Bootloader 通常同时支持这两种工作模式，而且允许用户在这两种工作模式之间进行切换。比如，U-Boot 在启动时处于正常的启动加载模式，但是它会延时若干秒（这可以设置），等待终端用户按下任意键，而将 U-Boot 切换到下载模式。如果在指定时间内没有用户按键，则 U-Boot 继续启动 Linux 内核。

15.1.2 Bootloader 的结构和启动过程

1．概述

在移植之前先了解 Bootloader 的一些通用概念，对理解它的代码会有所帮助。

嵌入式 Linux 系统从软件的角度通常可以分为以下 4 个层次。

（1）引导加载程序，包括固化在固件（firmware）中的 boot 代码（可选）和 Bootloader 两大部分。

有些 CPU 在运行 Bootloader 之前先运行一段固化的程序（固件，firmware），比如 x86 结构的 CPU 就是先运行 BIOS 中的固件，然后才运行硬盘第一个分区（MBR）中的 Bootloader。在大多嵌入式系统中并没有固件，Bootloader 是上电后执行的第一个程序。

（2）Linux 内核。

特定于嵌入式板子的定制内核以及内核的启动参数。内核的启动参数可以是内核默认的，或是由 Bootloader 传递给它的。

（3）文件系统。

包括根文件系统和建立于 Flash 内存设备之上的文件系统。里面包含了 Linux 系统能够运行所必需的应用程序、库等，比如可以给用户提供操作 Linux 的控制界面的 shell 程序、动态连接的程序运行时需要的 glibc 或 uClibc 库等。

（4）用户应用程序。

特定于用户的应用程序，它们也存储在文件系统中。有时在用户应用程序和内核层之间

可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有：Qtopia 和 MiniGUI 等。

显然，在嵌入系统的固态存储设备上有相应的分区来存储它们，如图 15.1 所示为一个典型的分区结构。

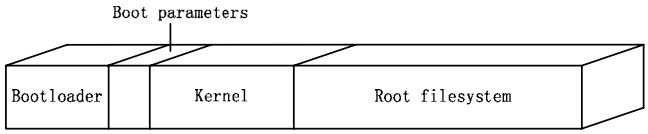

图 15.1　嵌入式 Linux 系统中的典型分区结构

“Boot parameters”分区中存放一些可设置的参数，比如 IP 地址、串口波特率、要传递给内核的命令行参数等。正常启动过程中，Bootloader 首先运行，然后它将内核复制到内存中（也有些内核可以在固态存储设备上直接运行），并且在内存某个固定的地址设置好要传递给内核的参数，最后运行内核。内核启动之后，它会挂接（mount）根文件系统（“Root filesystem”），启动文件系统中的应用程序。

2．Bootloader 的两个阶段

Bootloader 的启动过程可以分为单阶段（Single Stage）、多阶段（Multi-Stage）两种。通常多阶段的 Bootloader 能提供更为复杂的功能以及更好的可移植性。从固态存储设备上启动的 Bootloader 大多都是两阶段的启动过程。第一阶段使用汇编来实现，它完成一些依赖于 CPU 体系结构的初始化，并调用第二阶段的代码；第二阶段则通常使用 C 语言来实现，这样可以实现更复杂的功能，而且代码会有更好的可读性和可移植性。

一般而言，这两个阶段完成的功能可以如下分类。

（1）Bootloader 第一阶段的功能。

- 硬件设备初始化。
- 为加载 Bootloader 的第二阶段代码准备 RAM 空间。
- 复制 Bootloader 的第二阶段代码到 RAM 空间中。
- 设置好栈。
- 跳转到第二阶段代码的 C 入口点。

在第一阶段进行的硬件初始化一般包括：关闭 WATCHDOG、关中断、设置 CPU 的速度和时钟频率、RAM 初始化等。这些并不都是必需的，比如 S3C2410/S3C2440 的开发板所使用的 U-Boot 中，就将 CPU 的速度和时钟频率的设置放在第二阶段。

甚至，将第二阶段的代码复制到 RAM 空间中也不是必需的，对于 NOR Flash 等存储设备，完全可以在上面直接执行代码，只不过相比在 RAM 中执行效率大为降低。

（2）Bootloader 第二阶段的功能。

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射（memory map）。
- 将内核映象和根文件系统映象从 Flash 上读到 RAM 空间中。
- 为内核设置启动参数。
- 调用内核。

为了方便开发，至少要初始化一个串口以便程序员与 Bootloader 进行交互。

所谓检测内存映射，就是确定板上使用了多少内存、它们的地址空间是什么。由于嵌入式开发中 Bootloader 多是针对某类板子进行编写，所以可以根据板子的情况直接设置，不需要考虑可以适用于各类情况的复杂算法。

Flash 上的内核映象有可能是经过压缩的，在读到 RAM 之后，还需要进行解压。当然，对于有自解压功能的内核，不需要 Bootloader 来解压。

将根文件系统映象复制到 RAM 中，这不是必需的。这取决于是什么类型的根文件系统，以及内核访问它的方法。

将内核存放在适当的位置后，直接跳到它的入口点即可调用内核。调用内核之前，下列条件要满足。

（1）CPU 寄存器的设置。

- R0=0。
- R1=机器类型 ID；对于 ARM 结构的 CPU，其机器类型 ID 可以参见 linux/arch/arm/tools/mach-types。
- R2=启动参数标记列表在 RAM 中起始基地址。

（2）CPU 工作模式。

- 必须禁止中断（IRQs 和 FIQs）。
- CPU 必须为 SVC 模式。

（3）Cache 和 MMU 的设置。

- MMU 必须关闭。
- 指令 Cache 可以打开也可以关闭。
- 数据 Cache 必须关闭。

如果用 C 语言，可以像下列示例代码一样来调用内核：

```
    void (*theKernel)(int zero, int arch, u32 params_addr) = (void (*)(int, int,
u32))KERNEL_RAM_BASE;
    …
    theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

3．Bootloader 与内核的交互

Bootloader 与内核的交互是单向的，Bootloader 将各类参数传给内核。由于它们不能同时运行，传递办法只有一个：Bootloader 将参数放在某个约定的地方之后，再启动内核，内核启动后从这个地方获得参数。

除了约定好参数存放的地址外，还要规定参数的结构。Linux 2.4.x 以后的内核都期望以标记列表（tagged list）的形式来传递启动参数。标记，就是一种数据结构；标记列表，就是挨着存放的多个标记。标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。

标记的数据结构为 tag，它由一个 tag_header 结构和一个联合（union）组成。tag_header 结构表示标记的类型及长度，比如是表示内存还是表示命令行参数等。对于不同类型的标记使用不同的联合（union），比如表示内存时使用 tag_mem32，表示命令行时使用 tag_cmdline。数据结构 tag 和 tag_header 定义在 Linux 内核源码的 include/asm/setup.h 头文件中，如下所示：

```
struct tag_header {
        u32 size;
        u32 tag;
};

struct tag {
        struct tag_header hdr;
        union {
                struct tag_core         core;
                struct tag_mem32    mem;
                struct tag_videotext    videotext;
                struct tag_ramdisk ramdisk;
                struct tag_initrd  initrd;
                struct tag_serialnr     serialnr;
                struct tag_revision     revision;
                struct tag_videolfb     videolfb;
                struct tag_cmdline cmdline;

                /*
                 * Acorn specific
                 */
                struct tag_acorn    acorn;

                /*
                 * DC21285 specific
                 */
                struct tag_memclk  memclk;
        } u;
};
```

下面以设置内存标记、命令行标记为例说明参数的传递。

（1）设置标记 ATAG_CORE。

标记列表以标记 ATAG_CORE 开始，假设 Bootloader 与内核约定的参数存放地址为 0x30000100，则可以以如下代码设置标记 ATAG_CORE：

```
        params = (struct tag *) 0x30000100;

        params->hdr.tag = ATAG_CORE;
        params->hdr.size = tag_size (tag_core);
```

```
            params->u.core.flags = 0;
            params->u.core.pagesize = 0;
            params->u.core.rootdev = 0;

            params = tag_next (params);
```

其中，tag_next 定义如下，它指向当前标记的末尾：

```
    #define tag_next(t) ((struct tag *)((u32 *)(t) + (t)->hdr.size))
```

（2）设置内存标记。

假设开发板使用的内存起始地址为 0x30000000，大小为 0x4000000，则内存标记可以如下设置：

```
            params->hdr.tag = ATAG_MEM;
            params->hdr.size = tag_size (tag_mem32);

            params->u.mem.start = 0x30000000;
            params->u.mem.size  = 0x4000000;

            params = tag_next (params);
```

（3）设置命令行标记。

命令行就是一个字符串，它被用来控制内核的一些行为。比如"root=/dev/mtdblock 2 init=/linuxrc console=ttySAC0"表示根文件系统在 MTD2 分区上，系统启动后执行的第一个程序为/linuxrc，控制台为 ttySAC0（即第一个串口）。

命令行可以在 Bootloader 中通过命令设置好，然后按如下构造标记传给内核。

```
            char *p = "root=/dev/mtdblock 2 init=/linuxrc console=ttySAC0";
            params->hdr.tag = ATAG_CMDLINE;
            params->hdr.size = (sizeof (struct tag_header) + strlen (p) + 1 + 4)
>> 2;

            strcpy (params->u.cmdline.cmdline, p);

            params = tag_next (params);
```

（4）设置标记 ATAG_NONE。

标记列表以标记 ATAG_NONE 结束，如下设置：

```
            params->hdr.tag = ATAG_NONE;
            params->hdr.size = 0;
```

15.1.3 常用 Bootloader 介绍

现在 Bootloader 种类繁多，比如 x86 上有 LILO、GRUB 等。对于 ARM 架构的 CPU，有 U-Boot、Vivi 等。它们各有特点，下面列出 Linux 的开放源代码的 Bootloader 及其支持的体系架构，如表 15.1 所示。

表 15.1 开放源码的 Linux 引导程序

Bootloader	Monitor	描 述	X86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BUIS 的 Linux 引导程序	是	否	否
BLOB	是	LART 等硬件平台的引导程序	否	是	否
U-Boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是
Vivi	是	Mizi 公司针对 SAMSUNG 的 ARM CPU 设计的引导程序	否	是	否

对于本书使用的 S3C2410/S3C2440 开发板，U-Boot 和 Vivi 是两个好选择。

Vivi 是 Mizi 公司针对 SAMSUNG 的 ARM 架构 CPU 专门设计的，基本上可以直接使用，命令简单方便。不过其初始版本只支持串口下载，速度较慢。在网上出现了各种改进版本：支持网络功能、USB 功能、烧写 YAFFS 文件系统映象等。

U-Boot 则支持大多 CPU，可以烧写 EXT2、JFFS2 文件系统映象，支持串口下载、网络下载，并提供了大量的命令。相对于 Vivi，它的使用更复杂，但是可以用来更方便地调试程序。

15.2 U-Boot 分析与移植

15.2.1 U-Boot 工程简介

U-Boot，全称为 Universal Boot Loader，即通用 Bootloader，是遵循 GPL 条款的开放源代码项目。其前身是由德国 DENX 软件工程中心的 Wolfgang Denk 基于 8xxROM 的源码创建的 PPCBOOT 工程。后来整理代码结构使得非常容易增加其他类型的开发板、其他架构的 CPU（原来只支持 PowerPC）；增加更多的功能，比如启动 Linux、下载 S-Record 格式的文件、通过网络启动、通过 PCMCIA/CompactFLash/ATA disk/SCSI 等方式启动。增加 ARM 架构 CPU 及其他更多 CPU 的支持后，改名为 U-Boot。

它的名字“通用”有两层含义：可以引导多种操作系统、支持多种架构的 CPU。它支持如下操作系统：Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等，支持如下架构的 CPU：PowerPC、MIPS、x86、ARM、NIOS、XScale 等。

U-Boot 有如下特性。

- 开放源码。
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS。
- 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale。
- 较高的可靠性和稳定性。
- 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等。
- 丰富的设备驱动源码，如串口、以太网、SDRAM、Flash、LCD、NVRAM、EEPROM、RTC、键盘等。
- 较为丰富的开发调试文档与强大的网络技术支持。
- 支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统。
- 支持 NFS 挂载、从 Flash 中引导压缩或非压缩系统内核。
- 可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤对 Linux 支持最为强劲。
- 支持目标板环境变量多种存储方式，如 Flash、NVRAM、EEPROM。
- CRC32 校验，可校验 Flash 中内核、RAMDISK 镜像文件是否完好。
- 上电自检功能：SDRAM、Flash 大小自动检测，SDRAM 故障检测，CPU 型号。
- 特殊功能：XIP 内核引导。

可以从 http://sourceforge.net/projects/U-Boot 获得 U-Boot 的最新版本，如果使用过程中碰到问题或是发现 Bug，可以通过邮件列表网站 http://lists.sourcef orge.net/ lists/ listinfo/U-Boot-users/获得帮助。

15.2.2 U-Boot 源码结构

本书在 U-Boot-1.1.6 的基础上进行分析和移植，从 sourceforge 网站下载 U-Boot-1.1.6.tar.bz2 后解压即得到全部源码，U-Boot 源码目录结构比较简单。

U-Boot-1.1.6 根目录下共有 26 个子目录，可以分为 4 类。

（1）平台相关的或开发板相关的。

（2）通用的函数。

（3）通用的设备驱动程序。

（4）U-Boot 工具、示例程序、文档。

这 26 个子目录的功能与作用如表 15.2 所示。

表 15.2 U-Boot 顶层目录说明

目　录	特　性	解 释 说 明
board	开发板相关	对应不同配置的电路板（即使 CPU 相同），比如 smdk2410、sbc2410x
cpu	平台相关	对应不同的 CPU，比如 arm920t、arm925t、i386 等；在它们的子目录下仍可以进一步细分，比如 arm920t 下就有 at91rm9200、s3c24x0
lib_i386 类似		某一架构下通用的文件

续表

目　录	特　性	解 释 说 明

include	通用的函数	头文件和开发板配置文件，开发板的配置文件都放在 include/configs 目录下，U-Boot 没有 make menuconfig 类似的菜单来进行可视化配置，需要手动地修改配置文件中的宏定义
lib_generic		通用的库函数，比如 printf 等
common		通用的函数，多是对下一层驱动程序的进一步封装
disk	通用的设备驱动程序	硬盘接口程序
drivers		各类具体设备的驱动程序，基本上可以通用，它们通过宏从外面引入平台/开发板相关的函数
dtt		数字温度测量器或者传感器的驱动
fs		文件系统
nand_spl		U-Boot 一般从 ROM、NOR Flash 等设备启动，现在开始支持从 NAND Flash 启动，但是支持的 CPU 种类还不多
net		各种网络协议
post		上电自检程序
rtc		实时时钟的驱动
doc	文档	开发、使用文档
examples	示例程序	一些测试程序，可以使用 U-Boot 下载后运行
tools	工具	制作 S-Record、U-Boot 格式映象的工具，比如 mkimage

U-Boot 中各目录间也是有层次结构的，虽然这种分法不是绝对的，但是在移植过程中可以提供一些指导意义，如图 15.2 所示。

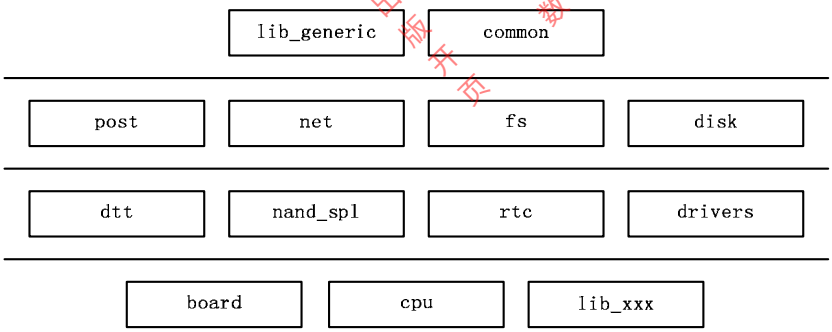

图 15.2　U-Boot 顶层目录的层次结构

比如 common/cmd_nand.c 文件提供了操作 NAND Flash 的各种命令，这些命令通过调用 drivers/nand/nand_base.c 中的擦除、读写函数来实现。这些函数针对 NAND Flash 的共性作了一些封装，将平台/开发板相关的代码用宏或外部函数来代替。而这些宏与外部函数，如果与平台相关，就要在下一层次的 cpu/xxx(xxx 表示某型号的 CPU)中实现；如果与开发板相关，就要在下一层次的 board/xxx 目录（xxx 表示某款开发板）中实现。本书移植的 U-Boot，就是在 cpu/arm920t/s3c24x0 目录下增加了一个 nand_flash.c 文件来实现这些函数。

以增加烧写 yaffs 文件系统映象的功能为例，即在 common 目录下的 cmd_nand.c 中增加

命令。比如 nand write.yaffs，这个命令要调用 drivers/nand/nand_util.c 中的相应函数，针对 yaffs 文件系统的特点依次调用擦除、烧写函数。而这些函数依赖于 drivers/nand/nand_base.c、cpu/arm920t/s3c24x0/nand_flash.c 文件中的相关函数。

目前 U-Boot-1.1.6 支持 10 种架构，根目录下有 10 个类似 lib_i386 的目录；31 个型号（类型）的 CPU，cpu 目录下有 31 个子目录；214 种开发板，board 目录下有 214 个子目录，很容易从中找到与自己的板子相似的配置，在上面稍作修改即可使用。

15.2.3 U-Boot 的配置、编译、连接过程

1．U-Boot 初体验

U-Boot-1.1.6 中有几千个文件，要想了解对于某款开发板，使用哪些文件、哪个文件首先执行、可执行文件占用内存的情况，最好的方法就是阅读它的 Makefile。

根据顶层 Readme 文件的说明，可以知道如果要使用开发板 board/<board_name>，就先执行“ make <board_name>_config ”命令进行配置，然后执行“ make all ”，就可以生成如下 3 个文件。

- U-Boot.bin：二进制可执行文件，它就是可以直接烧入 ROM、NOR Flash 的文件。
- U-Boot：ELF 格式的可执行文件。
- U-Boot.srec：Motorola S-Record 格式的可执行文件。

对于 S3C2410 的开发板，执行“ make smdk2410_config ”、“ make all ”后生成的 U-Boot.bin 可以烧入 NOR Flash 中运行。启动后可以看到串口输出一些信息后进入控制界面，等待用户的输入。

对于 S3C2440 的开发板，烧入上面生成的 U-Boot.bin，串口无输出，需要修改代码。

在修改代码之前，先看看上面两个命令“ make smdk2410_config ”、“ make all ”做了什么事情，以了解程序的流程，知道要修改哪些文件。

另外，编译 U-Boot 成功后，还会在它的 tools 子目录下生成一些工具，比如 mkimage 等。将它们复制到/usr/local/bin 目录下，以后就可以直接使用它们了，比如编译内核时，会使用 mkimage 来生成 U-Boot 格式的内核映象文件 uImage。

2．U-Boot 的配置过程

在顶层 Makefile 中可以看到如下代码：

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

假定在 U-Boot-1.1.6 的根目录下编译，则其中的 MKCONFIG 就是根目录下的 mkconfig 文件。$(@:_config=)的结果就是将“ smdk2410_config ”中的“ _config ”去掉，结果为“ smdk2410 ”。所以“ make smdk2410_config ”实际上就是执行如下命令：

```
./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0
```

再来看看 mkconfig 的作用，在 mkconfig 文件开头第 6 行给出了它的用法：

```
06 # Parameters:  Target  Architecture  CPU  Board [VENDOR] [SOC]
```

这里解释一下概念，对于 S3C2410、S3C2440，它们被称为 SoC(System on Chip)，上面除 CPU 外，还集成了包括 UART、USB 控制器、NAND Flash 控制器等设备（称为片内外设）。S3C2410/S3C2440 中的 CPU 为 ARM920T。

下面分步骤分析 mkconfig 的作用。

（1）确定开发板名称 BOARD_NAME，相关代码如下：

```
11 APPEND=no    # Default: Create new config file
12 BOARD_NAME=""    # Name to print in make output
13
14 while [ $# -gt 0 ] ; do
15  case "$1" in
16  --) shift ; break ;;
17  -a) shift ; APPEND=yes ;;
18  -n) shift ; BOARD_NAME="${1%%_config}" ; shift ;;
19  *)  break ;;
20  esac
21 done
22
23 [ "${BOARD_NAME}" ] || BOARD_NAME="$1"
```

对于“ ./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0 ”命令，其中没有“ -- ”、“ -a ”、“ -n ”等符号，所以第 14～22 行没做任何事情。第 11、12 行两个变量仍维持原来的值。

执行完第 23 行后，BOARD_NAME 的值等于第 1 个参数，即“ smdk2410 ”。

（2）创建到平台/开发板相关的头文件的链接。

略过 mkconfig 文件中的一些没有起作用的行，如下所示：

```
30 #
31 # Create link to architecture specific headers
32 #
33 if [ "$SRCTREE" != "$OBJTREE" ] ; then
……
45 else
46      cd ./include
47      rm -f asm
48      ln -s asm-$2 asm
49 fi
```

```
50
```

第 33 行判断源代码目录和目标文件目录是否一样，可以选择在其他目录下编译 U-Boot，这可以令源代码目录保持干净，可以同时使用不同的配置进行编译。不过本书是直接在源代码目录下编译的，第 33 行的条件不满足，将执行 else 分支的代码。

第 46～48 行进入 include 目录，删除 asm 文件（这是上一次配置时建立的链接文件），然后再次建立 asm 文件，并令它链接向 asm-$2 目录，即 asm-arm。

继续往下看代码：

```
51 rm -f asm-$2/arch
52
53 if [ -z "$6" -o "$6" = "NULL" ] ; then
54      ln -s ${LNPREFIX}arch-$3 asm-$2/arch
55 else
56      ln -s ${LNPREFIX}arch-$6 asm-$2/arch
57 fi
58
59 if [ "$2" = "arm" ] ; then
60      rm -f asm-$2/proc
61      ln -s ${LNPREFIX}proc-armv asm-$2/proc
62 fi
63
```

第 51 行删除 asm-$2/arch 目录，即 asm-arm/arch。

对于“ ./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0 ”命令，$6 为“ s3c24x0 ”，不为空，也不是“ NULL ”，所以第 53 行的条件不满足，将执行 else 分支。

第 56 行中，LNPREFIX 为空，所以这个命令实际上就是“ ln -s arch-$6 asm-$2/arch ”，即“ ln -s arch-s3c24x0 asm-arm/arch ”。

第 60、61 行重新建立 asm-arm/proc 文件，并让它链接向 proc-armv 目录。

（3）创建顶层 Makefile 包含的文件 include/config.mk，如下所示：

```
64 #
65 # Create include file for Make
66 #
67 echo "ARCH   = $2" >  config.mk
68 echo "CPU    = $3" >> config.mk
69 echo "BOARD  = $4" >> config.mk
70
71 [ “ $5 ” ] && [ "$5" != "NULL" ] && echo "VENDOR = $5" >> config.mk
72
73 [ "$6" ] && [ "$6" != "NULL" ] && echo "SOC    = $6" >> config.mk
```

```
74
```

对于“ ./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0 ”命令，上面几行代码创建的 config.mk 文件内容如下：

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

（4）创建开发板相关的头文件 include/config.h，如下所示：

```
75 #
76 # Create board specific header file
77 #
78 if [ "$APPEND" = "yes" ]       # Append to existing config file
79 then
80      echo >> config.h
81 else
82      > config.h                 # Create new config file
83 fi
84 echo "/* Automatically generated - do not edit */" >>config.h
85 echo "#include <configs/$1.h>" >>config.h
86
```

APPEND 维持原值“ no ”，所以 config.h 被重新建立，它的内容如下：

```
/* Automatically generated - do not edit */
#include <configs/smdk2410.h>
```

现在总结一下，配置命令“ make smdk2410_config ”，实际的作用就是执行“ ./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0 ”命令。假设执行“ ./mkconfig $1 $2 $3 $4 $5 $6 ”命令，则将产生如下结果。

（1）开发板名称 BOARD_NAME 等于$1。

（2）创建到平台/开发板相关的头文件的链接，如下所示：

```
        ln -s asm-$2 asm
        ln -s arch-$6 asm-$2/arch
        ln -s proc-armv asm-$2/proc       # 如果$2 不是 arm 的话，此行没有
```

（3）创建顶层 Makefile 包含的文件 include/config.mk，如下所示：

```
ARCH   = $2
CPU    = $3
BOARD  = $4
VENDOR = $5     # $5 为空，或者是 NULL 的话，此行没有
```

```
SOC    = $6      # $6 为空，或者是 NULL 的话，此行没有
```

（4）创建开发板相关的头文件 include/config.h，如下所示：

```
/* Automatically generated - do not edit */
#include <configs/$1.h>"
```

从这 4 个结果可以知道，如果要在 board 目录下新建一个开发板<board_name>的目录，则在 include/config 目录下也要建立一个文件<board_name>.h，里面存放的就是开发板<board_name>的配置信息。

U-Boot 还没有类似 Linux 一样的可视化配置界面（比如使用 make menuconfig 来配置），要手动修改配置文件 include/config/<board_name>.h 来裁减、设置 U-Boot。

配置文件中有以下两类宏。

（1）一类是选项（Options），前缀为“ CONFIG_ ”，它们用于选择 CPU、SOC、开发板类型，设置系统时钟、选择设备驱动等。比如：

```
#define CONFIG_ARM920T        1          /* This is an ARM920T Core   */
#define  CONFIG_S3C2410       1          /* in a SAMSUNG S3C2410 SoC    */
#define CONFIG_SMDK2410       1          /* on a SAMSUNG SMDK2410 Board  */
#define CONFIG_SYS_CLK_FREQ    12000000  /* the SMDK2410 has 12MHz input clock */
#define CONFIG_DRIVER_CS8900 1           /* we have a CS8900 on-board */
```

（2）另一类是参数（Setting），前缀为“ CFG_ ”，它们用于设置 malloc 缓冲池的大小、U-Boot 的提示符、U-Boot 下载文件时的默认加载地址、Flash 的起始地址等。比如：

```
#define CFG_MALLOC_LEN      (CFG_ENV_SIZE + 128*1024)
#define CFG_PROMPT         "100ASK>"/* Monitor Command Prompt   */
#define CFG_LOAD_ADDR       0x33000000  /* default load address */
#define PHYS_FLASH_1       0x00000000   /* Flash Bank #1 */
```

从下面的编译、连接过程可知，U-Boot 中几乎每个文件都被编译和连接，但是这些文件是否包含有效的代码，则由宏开关来设置。比如对于网卡驱动 drivers/cs8900.c，它的格式为：

```
#include <common.h>          /* 将包含配置文件 include/config/<board_name>.h */
…
#ifdef CONFIG_DRIVER_CS8900
/* 实际的代码 */
…
#endif  /* CONFIG_DRIVER_CS8900 */
```

如果定义了宏 CONFIG_DRIVER_CS8900，则文件中包含有效的代码；否则，文件被注释为空。

可以这样认为，“ CONFIG_ ”除了设置一些参数外，主要用来设置 U-Boot 的功能、选择使用文件中的哪一部分；而“ CFG_ ”用来设置更细节的参数。

3．U-Boot 的编译、连接过程

配置完后，执行“ make all ”即可编译，从 Makefile 中可以了解 U-Boot 使用了哪些文件、哪个文件首先执行、可执行文件占用内存的情况。

先确定用到哪些文件，下面所示为 Makefile 中与 ARM 相关的部分。

```
117 include $(OBJTREE)/include/config.mk
118 export      ARCH CPU BOARD VENDOR SOC
119
…
127 ifeq ($(ARCH),arm)
128 CROSS_COMPILE = arm-linux-
129 endif
…
163 # load other configuration
164 include $(TOPDIR)/config.mk
165
```

第 117、164 行用于包含其他的 config.mk 文件，第 117 行所要包含文件的就是在上面的配置过程中制作出来的 include/config.mk 文件，其中定义了 ARCH、CPU、BOARD、SOC 等 4 个变量的值为 arm、arm920t、smdk2410、s3c24x0。

第 164 行包含顶层目录的 config.mk 文件，它根据上面 4 个变量的值确定了编译器、编译选项等。其中对我们理解编译过程有帮助的是 BOARDDIR、LDFLAGS 的值，如下所示：

```
88 BOARDDIR = $(BOARD)
…
91 sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk  # include board specific
rules
…
143 LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/U-Boot.lds
…
189 LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE) $(PLATFORM_LDFLAGS)
```

在 board/smdk2410/config.mk 中，定义了“ TEXT_BASE = 0x33F80000 ”。所以，最终结果如下：BOARDDIR 为 smdk2410；LDFLAGS 中有“ -T board/smdk2410/U-Boot.lds -Ttext 0x33F80000 ”字样。

继续往下看 Makefile：

```
166 #########################################################################
167 # U-Boot objects....order is important (i.e. start must be first)
168
169 OBJS  = cpu/$(CPU)/start.o
```

```
…
193 LIBS  = lib_generic/libgeneric.a
194 LIBS += board/$(BOARDDIR)/lib$(BOARD).a
195 LIBS += cpu/$(CPU)/lib$(CPU).a
…
199 LIBS += lib_$(ARCH)/lib$(ARCH).a
200 LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a \
201 fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
202 LIBS += net/libnet.a
…
212 LIBS += $(BOARDLIBS)
213
……
```

从第 169 行得知，OBJS 的第一个值为“cpu/$(CPU)/start.o”，即“cpu/arm920t/ start.o”。

第 193～213 行指定了 LIBS 变量就是平台/开发板相关的各个目录、通用目录下相应的库，比如：lib_generic/libgeneric.a、board/smdk2410/libsmdk2410.a、cpu/arm920t/ libarm920t.a、lib_arm/libarm.a、fs/cramfs/libcramfs.a fs/fat/libfat.a 等。

OBJS、LIBS 所代表的.o、.a 文件就是 U-Boot 的构成，它们通过如下命令由相应的源文件（或相应子目录下的文件）编译得到。

```
268 $(OBJS):
269             $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))
270
271 $(LIBS):
272             $(MAKE) -C $(dir $(subst $(obj),,$@))
273
274 $(SUBDIRS):
275             $(MAKE) -C $@ all
276
```

第 268、269 两行的规则表示，对于 OBJS 中的每个成员，都将进入 cpu/$(CPU)目录（即 cpu/arm920t）编译它们。现在 OBJS 为 cpu/arm920t/start.o，它将由 cpu/arm920t/start.S 编译得到。

第 271、272 两行的规则表示，对于 LIBS 中的每个成员，都将进入相应的子目录执行“make”命令。这些子目录中的 Makefile，结构相似，它们将 Makefle 中指定的文件编译、连接成一个库文件。

当所有的 OBJS、LIBS 所表示的.o 和.a 文件都生成后，就剩最后的连接了，这对应 Makefile 中如下几行：

```
246 $(obj)U-Boot.srec:     $(obj)U-Boot
247             $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
248
249 $(obj)U-Boot.bin:   $(obj)U-Boot
250             $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
251
……
262 $(obj)U-Boot:       depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
263              UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed  -n -e 's/.*\(__u
_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
264              cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJS) \
265                --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
266                -Map U-Boot.map -o U-Boot
267
```

先使用第 262～266 的规则连接得到 ELF 格式的 U-Boot，最后转换为二进制格式 U-Boot.bin、S-Record 格式 U-Boot.srec。LDFLAGS 确定了连接方式，其中的“ -T board/smdk2410/U-Boot.lds -Ttext 0x33F80000 ”字样指定了程序的布局、地址。board/smdk2410/U-Boot.lds 文件如下：

```
28 SECTIONS
29 {
30      . = 0x00000000;
31
32      . = ALIGN(4);
33      .text      :
34      {
35        cpu/arm920t/start.o   (.text)
36        *(.text)
37      }
38
39      . = ALIGN(4);
40      .rodata : { *(.rodata) }
41
42      . = ALIGN(4);
43      .data : { *(.data) }
44
45      . = ALIGN(4);
46      .got : { *(.got) }
47
```

```
48          . = .;
49          __u_boot_cmd_start = .;
50          .u_boot_cmd : { *(.u_boot_cmd) }
51          __u_boot_cmd_end = .;
52
53          . = ALIGN(4);
54          __bss_start = .;
55          .bss : { *(.bss) }
56          _end = .;
57 }
```

从第 35 行可知，cpu/arm920t/start.o 被放在程序的最前面，所以 U-Boot 的入口点在 cpu/arm920t/start.S 中。

现在来总结一下 U-Boot 的编译流程。

（1）首先编译 cpu/$(CPU)/start.S，对于不同的 CPU，还可能编译 cpu/$(CPU)下的其他文件。

（2）然后，对于平台/开发板相关的每个目录、每个通用目录都使用它们各自的 Makefile 生成相应的库。

（3）将 1、2 步骤生成的.o、.a 文件按照 board/$(BOARDDIR)/config.mk 文件中指定的代码段起始地址、board/$(BOARDDIR)/U-Boot.lds 连接脚本进行连接。

（4）第 3 步得到的是 ELF 格式的 U-Boot，后面 Makefile 还会将它转换为二进制格式、S-Record 格式。

15.2.4　U-Boot 的启动过程源码分析

本书使用的 U-Boot 从 NOR Flash 启动，下面以开发板 smdk2410 的 U-Boot 为例。

U-Boot 属于两阶段的 Bootloader，第一阶段的文件为 cpu/arm920t/start.S 和 board/smdk2410/lowlevel_init.S，前者是平台相关的，后者是开发板相关的。

1．U-Boot 第一阶段代码分析

（1）硬件设备初始化。

依次完成如下设置：将 CPU 的工作模式设为管理模式（svc），关闭 WATCHDOG，设置 FCLK、HCLK、PCLK 的比例（即设置 CLKDIVN 寄存器），关闭 MMU、CACHE。

代码都在 cpu/arm920t/start.S 中，注释也比较完善，读者有不明白的地方可以参考前面硬件实验的相关章节。

（2）为加载 Bootloader 的第二阶段代码准备 RAM 空间。

所谓准备 RAM 空间，就是初始化内存芯片，使它可用。对于 S3C2410/S3C2440，通过在 start.S 中调用 lowlevel_init 函数来设置存储控制器，使得外接的 SDRAM 可用。代码在 board/smdk2410/lowlevel_init.S 中。

> 注意　lowlevel_init.S 文件是开发板相关的，这表示如果外接的设备不一样，可以修改 lowlevel_init.S 文件中的相关宏。

lowlevel_init 函数并不复杂，只是要注意这时的代码、数据都只保存在 NOR Flash 上，内存中还没有，所以读取数据时要变换地址。代码如下：

```
129 _TEXT_BASE:
130 .word   TEXT_BASE
131
132 .globl lowlevel_init
133 lowlevel_init:
134     /* memory control configuration */
135     /* make r0 relative the current location so that it */
136     /* reads SMRDATA out of FLASH rather than memory ! */
137     ldr     r0, =SMRDATA
138     ldr r1, _TEXT_BASE
139     sub r0, r0, r1
140     ldr r1, =BWSCON /* Bus Width Status Controller */
141     add     r2, r0, #13*4
142 0:
143     ldr     r3, [r0], #4
144     str     r3, [r1], #4
145     cmp     r2, r0
146     bne     0b
147
148     /* everything is fine now */
149     mov pc, lr
150
151     .ltorg
152 /* the literal pools origin */
153
154 SMRDATA:        /* 13个寄存器的值 */
155     .word ……
156     .word ……
```


第 137～139 行进行地址变换，因为这时候内存中还没有数据，不能使用连接程序时确定的地址来读取数据。

第 137 行中 SMRDATA 表示这 13 个寄存器的值存放的开始地址（连接地址），值为 0x33F8xxxx，处于内存中。

第 138 行获得代码段的起始地址，它就是第 130 行中的“ TEXT_BASE ”，其值在 board/smdk2410/config.mk 中定义为“ TEXT_BASE = 0x33F80000 ”。

第 139 行将 0x33F8xxxx 与 0x33F80000 相减，这就是 13 个寄存器值在 NOR Flash 上存放的开始地址。

（3）复制 Bootloader 的第二阶段代码到 RAM 空间中。

这里将整个 U-Boot 的代码（包括第一、第二阶段）都复制到 SDRAM 中，这在 cpu/arm920t/start.S 中实现，如下所示：

```
164 relocate:                           /* 将 U-Boot 复制到 RAM 中 */
165     adr     r0, _start              /* r0:当前代码的开始地址 */
166     ldr     r1, _TEXT_BASE          /* r1:代码段的连接地址 */
167     cmp     r0, r1                  /* 测试现在是在 Flash 中还是在 RAM 中 */
168     beq     stack_setup             /* 如果已经在 RAM 中(这通常是调试时直接下载到 RAM 中),
                                        * 则不需要复制
                                        */
169
170     ldr     r2, _armboot_start/* _armboot_start 在前面定义，是第一条指令的运行地址 */
171     ldr     r3, _bss_start          /* 在连接脚本 U-Boot.lds 中定义,是代码段的结束地址 */
172     sub     r2, r3, r2              /* r2 = 代码段长度 */
173     add     r2, r0, r2              /* r2 = NOR Flash 上代码段的结束地址 */
174
175 copy_loop:
176     ldmia   r0!, {r3-r10}           /* 从地址[r0]处获得数据 */
177     stmia   r1!, {r3-r10}           /* 复制到地址[r1]处 */
178     cmp     r0, r2                  /* 判断是否复制完毕 */
179     ble     copy_loop               /* 没复制完，则继续 */
```

（4）设置好栈。

栈的设置灵活性很大，只要让 sp 寄存器指向一段没有使用的内存即可。

```
182     /* Set up the stack                        */
183 stack_setup:
184     ldr r0, _TEXT_BASE              /* _TEXT_BASE 为代码段的开始地址，值为 0x33F80000 */
185     sub r0, r0, #CFG_MALLOC_LEN     /* 代码段下面，留出一段内存以实现 malloc */
186     sub r0, r0, #CFG_GBL_DATA_SIZE  /* 再留出一段内存，存一些全局参数 */
187 #ifdef CONFIG_USE_IRQ
188     sub r0, r0, # (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)       /*
IRQ、FIQ 模式的栈 */
189 #endif
190     sub sp, r0, #12                 /* 最后，留出 12 字节的内存给 abort 异常，
                                        * 往下的内存就都是栈了
                                        */
191
```

到了这一步，读者可以知道内存的使用情况了，如图 15.3 所示（图中与上面的划分稍有

不同，这是因为在 cpu/arm920t/cpu.c 中的 cpu_init 函数中才真正为 IRQ、FIQ 模式划分了栈）。

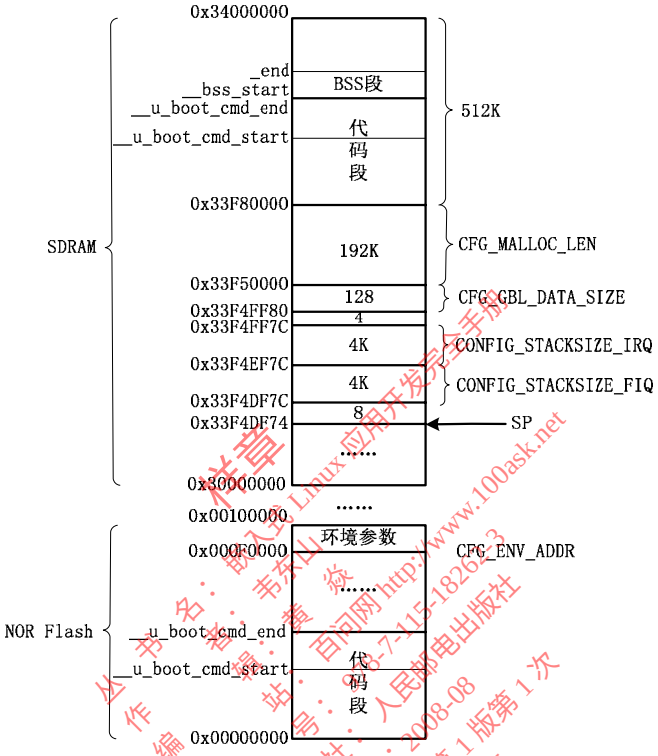

图 15.3 U-Boot 内存使用情况

（5）跳转到第二阶段代码的 C 入口点。

在跳转之前，还要清除 BSS 段（初始值为 0、无初始值的全局变量、静态变量放在 BSS 段），代码如下：

```
    192 clear_bss:
    193     ldr     r0, _bss_start      /* BSS 段的开始地址，它的值在连接脚本
U-Boot.lds 中确定 */
    194     ldr     r1, _bss_end        /* BSS 段的结束地址，它的值在连接脚本
U-Boot.lds 中确定 */
    195     mov r2, #0x00000000
    196
    197 clbss_l:str r2, [r0]          /* 往 BSS 段中写入 0 值 */
    198     add     r0, r0, #4
    199     cmp     r0, r1
    200     ble     clbss_l
    201
```

现在，C 函数的运行环境已经完全准备好，通过如下命令直接跳转（这之后，程序才在内存中执行），它将调用 lib_arm/board.c 中的 start_armboot 函数，这是第二阶段的入口点。

```
223     ldr     pc, _start_armboot
224
225 _start_armboot: .word start_armboot
226
```

2．U-Boot 第二阶段代码分析

它与 15.1.2 节中描述的 Bootloader 第二阶段所完成的功能基本上一致，只是顺序有点小差别。另外，U-Boot 在启动内核之前可以让用户决定是否进入下载模式，即进入 U-Boot 的控制界面。

第二阶段从 lib_arm/board.c 中的 start_armboot 函数开始，程序流程如图 15.4 所示。

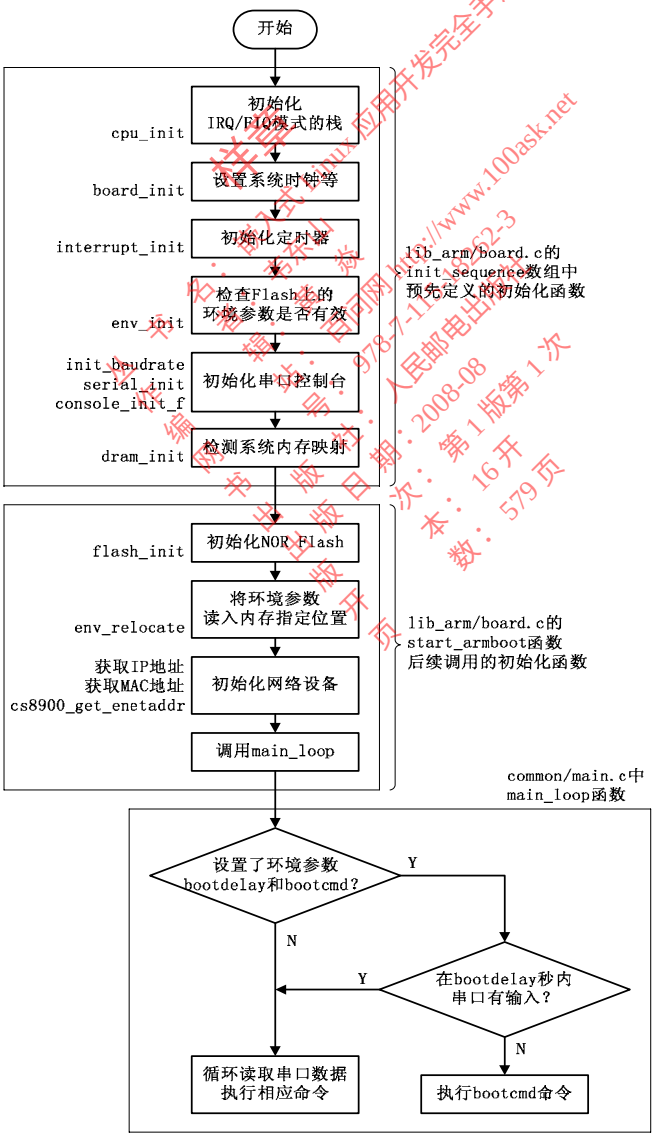

图 15.4 U-Boot 第二阶段流程图

移植 U-Boot 的主要工作在于对硬件的初始化、驱动，所以下面讲解时将重点放在硬件的操作上。

（1）初始化本阶段要使用到的硬件设备。

最主要的是设置系统时钟、初始化串口，只要这两个设置好了，就可以从串口看到打印信息。

board_init 函数设置 MPLL、改变系统时钟，它是开发板相关的函数，在 board/smdk2410/smdk2410.c 中实现。值得注意的是，board_init 函数中还保存了机器类型 ID，这将在调用内核时传给内核，代码如下：

```
        /* arch number of SMDK2410-Board */
        gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;   /* 值为 193 */
```

串口的初始化函数主要是 serial_init，它设置 UART 控制器，是 CPU 相关的函数，在 cpu/arm920t/s3c24x0/serial.c 中实现。

（2）检测系统内存映射（memory map）。

对于特定的开发板，其内存的分布是明确的，所以可以直接设置。board/smdk2410/smdk2410.c 中的 dram_init 函数指定了本开发板的内存起始地址为 0x30000000，大小为 0x4000000。代码如下：

```
int dram_init  (void)
{
        gd->bd->bi_dram[0].start = PHYS_SDRAM_1;      /* 即 0x300000000 */
        gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;  /* 即 0x4000000 */

        return 0;
}
```

这些设置的参数，将在后面向内核传递参数时用到。

（3）U-Boot 命令的格式。

从图 15.3 可以知道，即使是内核的启动，也是通过 U-Boot 命令来实现的。U-Boot 中每个命令都通过 U_BOOT_CMD 宏来定义，格式如下：

```
U_BOOT_CMD(name,maxargs,repeatable,command,"usage","help")
```

各项参数的意义如下。

① name：命令的名字，注意，它不是一个字符串（不要用双引号括起来）。

② maxargs：最大的参数个数。

③ repeatable：命令是否可重复，可重复是指运行一个命令后，下次敲回车即可再次运行。

④ command：对应的函数指针，类型为(*cmd)(struct cmd_tbl_s *, int, int, char *[])。

⑤ usage：简短的使用说明，这是个字符串。

⑥ help：较详细的使用说明，这是个字符串。

宏 U_BOOT_CMD 在 include/command.h 中定义，如下所示：

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage,
help}
```

Struct_Section 也是在 include/command.h 中定义，如下所示：

```
#define Struct_Section  __attribute__ ((unused,section (".u_boot_cmd")))
```

比如对于 bootm 命令，它如下定义：

```
U_BOOT_CMD(
        bootm,  CFG_MAXARGS,    1,  do_bootm,
         "string1",
          "string2"
);
```

宏 U_BOOT_CMD 扩展开后如下所示：

```
cmd_tbl_t __u_boot_cmd_bootm __attribute__ ((unused,section (".u_boot_cmd")))
= {"bootm", CFG_MAXARGS, 1, do_bootm, "string1", "string2"};
```

对于每个使用 U_BOOT_CMD 宏来定义的命令，其实都是在".u_boot_cmd"段中定义一个 cmd_tbl_t 结构。连接脚本 U-Boot.lds 中有如下代码：

```
        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;
```

程序中就是根据命令的名字在内存段_ _u_boot_cmd_start～_ _u_boot_cmd_end 找到它的 cmd_tbl_t 结构，然后调用它的函数（请参考 common/command.c 中的 find_cmd 函数）。

内核的复制和启动，可以通过如下命令来完成：bootm 从内存、ROM、NOR Flash 中启动内核，bootp 则通过网络来启动，而 nboot 从 NAND Flash 启动内核。它们都是先将内核映象从各种媒介中读出，存放在指定的位置；然后设置标记列表以给内核传递参数；最后跳到内核的入口点去执行。具体实现的细节不再描述，有兴趣的读者可以阅读 common/cmd_boot.c、common/cmd_net.c、common/cmd_nand.c 来了解它们的实现。

（4）为内核设置启动参数。

U-Boot 也是通过标记列表向内核传递参数。并且，在 15.1.2 小节中内存标记、命令行标记的示例代码就是取自 U-Boot 中的 setup_memory_tags、setup_commandline_tag 函数，它们都是在 lib_arm/armlinux.c 中定义。一般而言，设置这两个标记就可以了，在配置文件 include/configs/smdk2410.h 中增加如下两个配置项即可：

```
#define CONFIG_SETUP_MEMORY_TAGS    1
#define CONFIG_CMDLINE_TAG          1
```

对于 ARM 架构的 CPU，都是通过 lib_arm/armlinux.c 中的 do_bootm_linux 函数来启动内

核。这个函数中，设置标记列表，最后通过“ theKernel (0, bd→bi_arch_number, bd→bi_boot_params) ”调用内核。其中，theKernel 指向内核存放的地址（对于 ARM 架构的 CPU，通常是 0x30008000），bd→bi_arch_number 就是前面 board_init 函数设置的机器类型 ID，而 bd→bi_boot_params 就是标记列表的开始地址。

15.2.5　U-Boot 的移植

开发板 smdk2410 的配置适用于大多数 S3C2410 开发板，或是只需要极少的修改即可使用。但是目前 U-Boot 中没有对 S3C2440 的支持，需要我们自己移植。

本书基于的 S3C2410、S3C2440 两款开发板，它们的外接硬件相同。

- BANK0 外接容量为 1MB，位宽为 8 的 NOR Flash 芯片 AM29LV800。
- BANK3 外接 10M 网卡芯片 CS8900，位宽为 16。
- BANK6 外接两片容量为 32MB、位宽为 16 的 SDRAM 芯片 K4S561632，组成容量为 64MB、位宽为 32 的内存。
- 通过 NAND Flash 控制器外接容量为 64MB，位宽为 8 的 NAND Flash 芯片 K9S1208。

对于 NOR Flash 和 NAND Flash，如图 15.5 所示划分它们的使用区域。由于 NAND Flash 的“ 位反转 ”现象比较常见，为保证数据的正确，在读写数据时需要使用 ECC 校验。另外，NAND Flash 在使用过程中、运输过程中还有可能出现坏块。所以本书选择在 NOR Flash 中保存 U-Boot，在 NAND Flash 中保存内核和文件系统，并在使用 U-Boot 烧写内核、文件系统时进行坏块检查、ECC 校验。这样，即使 NAND Flash 出现坏块导致内核或文件系统不能使用，也可以通过 NOR Flash 中的 U-Boot 来重新烧写。

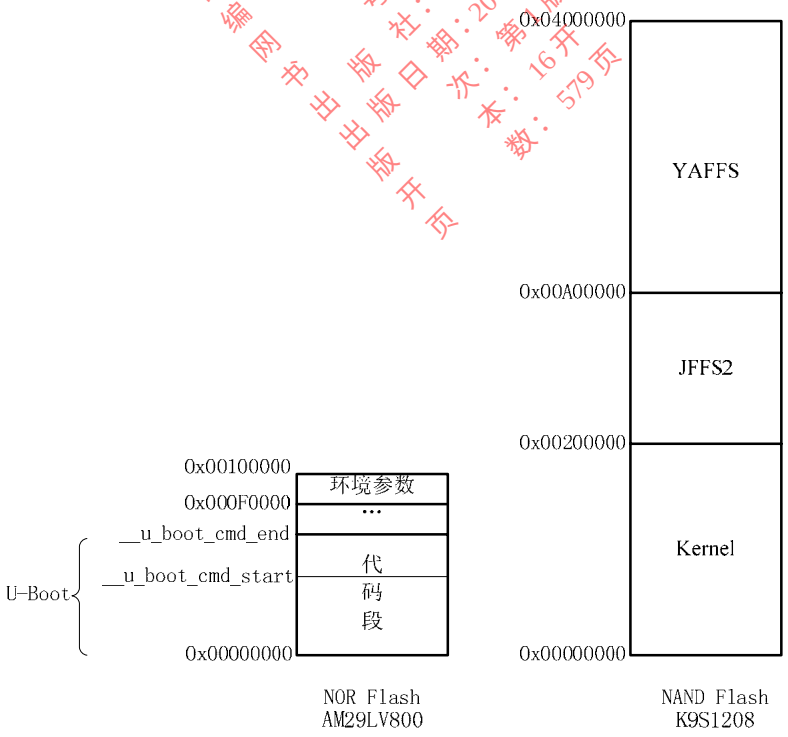

图 15.5　开发板固态存储器分区划分

smdk2410 开发板已经支持 NOR Flash 芯片 AM29LV800，U-Boot 本身也已经支持 JFFS2 文件系统映象的烧写。下面一步一步移植 U-Boot（所有的修改都在补丁文件 U-boot-1.1.6_100ask_24 X 0.Patch 里，读者可以直接打补丁），增加如下新功能。

- 同时支持本书使用的 S3C2410 和 S3C2440 开发板。
- 支持串口 xmodem 协议。
- 支持网卡芯片 CS8900。
- 支持 NAND Flash 读写。
- 支持烧写 yaffs 文件系统映象。

1．同时支持 S3C2410 和 S3C2440

我们将在开发板 smdk2410 的基础上进行移植。

（1）新建一个开发板的相应目录和文件。

为了不破坏原来的代码，在 board 目录下将 smdk2410 复制为 100ask24x0 目录，并将 board/100ask24x0/smdk2410.c 改名为 100ask24x0.c。

根据前面描述的配置过程可知，还要在 include/configs 目录下建立一个配置文件 100ask24x0.h，可以将 include/configs/smdk2410.h 直接复制为 100ask24x0.h。

还要修改两个 Makefile，首先在顶层 Makefile 中增加如下两行：

```
100ask24x0_config   :       unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t 100ask24x0 NULL s3c24x0
```

然后在 board/100ask24x0/Makefile 中进行如下修改（因为前面将 smdk2410.c 文件改名为 100ask24x0.c 了）：

```
COBJS   := smdk2410.o flash.o
改为：
COBJS   := 100ask24x0.o flash.o
```

（2）修改 SDRAM 的配置。

SDRAM 的初始化在 U-Boot 的第一阶段完成，就是在 board/100ask24x0/lowlevel_init.S 文件中设置存储控制器。

检查一下 BANK6 的设置：位宽为 32，宏 B6_BWSCON 刚好为 DW32（表示 32 位），无需改变；另外还要根据 HCLK 设置 SDRAM 的刷新参数，主要是 REFCNT 寄存器。

本书所用开发板的 HCLK 都设为 100MHz，需要根据 SDRAM 芯片的具体参数重新计算 REFCNT 寄存器的值（请参考第 6 章）。代码修改如下：

```
126 #define REFCNT  1113    /* period=15.6μs, HCLK=60Mhz, (2048+1-15.6*60) */
改为
126 #define REFCNT  0x4f4   /*        period=7.8125μs,        HCLK=100Mhz,
(2048+1-7.8125*100) */
```

对于其他 BANK，比如网卡芯片 CS8900 所在的 BANK2，原来的设置刚好匹配，无需更改；而对于 BANK1、BANK2、BANK4、BANK5、BANK7，在 U-Boot 中并没有使用到它们

外接的设备，也不需要理会。

（3）增加对 S3C2440 的支持。

S3C2440 是 S3C2410 的改进版，它们的操作基本相似。不过在系统时钟的设置、NAND Flash 控制器的操作等方面有一些小差别。它们的 MPLL、UPLL 计算公式不一样，FCLK、HCLK 和 PCLK 的分频化设置也不一样，这在下面的代码中可以看到。NAND Flash 控制器的差别在增加对 NAND Flash 的支持时讲述。

本章的目标是令同一个 U-Boot 二进制代码既可以在 S3C2410 上运行，也可以在 S3C2440 上运行。首先需要在代码中自动识别是 S3C2410 还是 S3C2440，这可以通过读取 GSTATUS1 寄存器的值来分辨：0x32410000 表示 S3C2410，0x32410002 表示 S3C2410A，0x32440000 表示 S3C2440，0x32440001 表示 S3C2440A。S3C2410 和 S3C2410A、S3C2440 和 S3C2440A，对本书来说没有区别。

对于 S3C2410 开发板，将 FCLK 设为 200MHz，分频比为 FCLK:HCLK:PCLK=1:2:4；对于 S3C2440 开发板，将 FCLK 设为 400MHz，分频比为 FCLK:HCLK:PCLK=1:4:8。还将 UPLL 设为 48MHz，即 UCLK 为 48MHz，以在内核中支持 USB 控制器。

首先修改 board/100ask24x0/100ask24x0.c 中的 board_init 函数，下面是修改后的代码：

```
33 /* S3C2440: MPLL = (2*m * Fin) / (p * 2^s), UPLL = (m * Fin) / (p * 2^s)
34  * m = M (the value for divider M)+ 8, p = P (the value for divider P) + 2
35  */
36 #define S3C2440_MPLL_400MHZ     ((0x5c<<12)|(0x01<<4)|(0x01))
37 #define S3C2440_UPLL_48MHZ      ((0x38<<12)|(0x02<<4)|(0x02))
38 #define S3C2440_CLKDIV          0x05  /* FCLK:HCLK:PCLK = 1:4:8, UCLK = UPLL */
39
40 /* S3C2410: Mpll,Upll = (m * Fin) / (p * 2^s)
41  * m = M (the value for divider M)+ 8, p = P (the value for divider P) + 2
42  */
43 #define S3C2410_MPLL_200MHZ     ((0x5c<<12)|(0x04<<4)|(0x00))
44 #define S3C2410_UPLL_48MHZ      ((0x28<<12)|(0x01<<4)|(0x02))
45 #define S3C2410_CLKDIV          0x03    /* FCLK:HCLK:PCLK = 1:2:4 */
46
```

上面代码针对 S3C2410、S3C2440 分别定义了 MPLL、UPLL 寄存器的值。开发板输入时钟为 12MHz(这在 include/configs/100ask24x0.h 中的宏 CONFIG_SYS_CLK_FREQ 中定义)，读者可以根据代码中的计算公式针对自己的开发板修改系统时钟。

下面是针对 S3C2410、S3C2440 分别使用不同的宏设置系统时钟。

```
58 int board_init (void)
59 {
60     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
61     S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
62
```

```
63      /* 设置 GPIO */
64      gpio->GPACON = 0x007FFFFF;
65      gpio->GPBCON = 0x00044555;
66      gpio->GPBUP = 0x000007FF;
67      gpio->GPCCON = 0xAAAAAAAA;
68      gpio->GPCUP = 0x0000FFFF;
69      gpio->GPDCON = 0xAAAAAAAA;
70      gpio->GPDUP = 0x0000FFFF;
71      gpio->GPECON = 0xAAAAAAAA;
72      gpio->GPEUP = 0x0000FFFF;
73      gpio->GPFCON = 0x000055AA;
74      gpio->GPFUP = 0x000000FF;
75      gpio->GPGCON = 0xFF95FFBA;
76      gpio->GPGUP = 0x0000FFFF;
77      gpio->GPHCON = 0x002AFAAA;
78      gpio->GPHUP = 0x000007FF;
79
80      /* 同时支持 S3C2410 和 S3C2440 */
81      if ((gpio->GSTATUS1 == 0x32410000) || (gpio->GSTATUS1 == 0x32410002))
82      {
83          /* FCLK:HCLK:PCLK = 1:2:4 */
84          clk_power->CLKDIVN = S3C2410_CLKDIV;
85
86          /* 修改为异步总线模式 */
87          __asm__(    "mrc    p15, 0, r1, c1, c0, 0\n"    /* read ctrl register   */
88                      "orr    r1, r1, #0xc0000000\n"      /* Asynchronous         */
89                      "mcr    p15, 0, r1, c1, c0, 0\n"    /* write ctrl register  */
90                      :::"r1"
91                      );
92
93          /* 设置 PLL 锁定时间 */
94          clk_power->LOCKTIME = 0xFFFFFF;
95
96          /* 配置 MPLL */
97          clk_power->MPLLCON = S3C2410_MPLL_200MHZ;
98
99          /* 配置 MPLL 后，要延时一段时间再配置 UPLL */
100         delay (4000);
101
```

```
102         /* 配置 UPLL */
103         clk_power->UPLLCON = S3C2410_UPLL_48MHZ;
104
105         /* 再延时一会 */
106         delay (8000);
107
108         /* 机器类型 ID，这在调用 Linux 内核时用到 */
109         gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
110     }
111     else
112     {
113         /* FCLK:HCLK:PCLK = 1:4:8 */
114         clk_power->CLKDIVN = S3C2440_CLKDIV;
115
116         /* 修改为异步总线模式 */
117         __asm__(    "mrc    p15, 0, r1, c1, c0, 0\n"    /* read ctrl register   */
118                     "orr    r1, r1, #0xc0000000\n"      /* Asynchronous         */
119                     "mcr    p15, 0, r1, c1, c0, 0\n"    /* write ctrl register  */
120                     :::"r1"
121                     );
122
123         /* 设置 PLL 锁定时间 */
124         clk_power->LOCKTIME = 0xFFFFFF;
125
126         /* 配置 MPLL */
127         clk_power->MPLLCON = S3C2440_MPLL_400MHZ;
128
129         /* 配置 MPLL 后，要延时一段时间再配置 UPLL */
130         delay (4000);
131
132         /* 配置 UPLL */
133         clk_power->UPLLCON = S3C2440_UPLL_48MHZ;
134
135         /* 再延时一会 */
136         delay (8000);
137
138         /* 机器类型 ID，这在调用 Linux 内核时用到，这个值要与内核相对应 */
139         gd->bd->bi_arch_number = MACH_TYPE_S3C2440;
140     }
```

```
141
142     /* 启动内核时，参数存放位置。这个值在构造标记列表时用到 */
143     gd->bd->bi_boot_params = 0x30000100;
144
145     icache_enable();
146     dcache_enable();
147
148     return 0;
149 }
150
```

最后一步：获取系统时钟的函数需要针对 S3C2410、S3C2440 的不同进行修改。

在后面设置串口波特率时需要获得系统时钟，就是在 U-Boot 的第二阶段，lib_arm/board.c 中 start_armboot 函数调用 serial_init 函数初始化串口时，会调用 get_PCLK 函数。它在 cpu/arm920t/s3c24x0/speed.c 中定义，与它相关的还有 get_HCLK、get_PLLCLK 等函数。

前面的 board_init 函数在识别出 S3C2410 或 S3C2440 后，设置了机器类型 ID：gd→bd→bi_arch_number，后面的函数可以通过它来分辨是 S3C2410 还是 S3C2440。首先要在程序的开头增加如下一行，这样才可以使用 gd 变量。

```
DECLARE_GLOBAL_DATA_PTR;
```

S3C2410 和 S3C2440 的 MPLL、UPLL 计算公式不一样，所以 get_PLLCLK 函数也需要修改，如下所示：

```
56 static ulong get_PLLCLK(int pllreg)
57 {
58     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
59     ulong r, m, p, s;
60
61     if (pllreg == MPLL)
62         r = clk_power->MPLLCON;
63     else if (pllreg == UPLL)
64         r = clk_power->UPLLCON;
65     else
66         hang();
67
68     m = ((r & 0xFF000) >> 12) + 8;
69     p = ((r & 0x003F0) >> 4) + 2;
70     s = r & 0x3;
71
72     /* 同时支持 S3C2410 和 S3C2440 */
73     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
```

```
74          return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
75      else
76          return((CONFIG_SYS_CLK_FREQ * m * 2) / (p << s));   /* S3C2440 */
77  }
78
```

由于分频系数的设置方法也不一样，get_HCLK、get_PCLK也需要修改。对于S3C2410，沿用原来的计算方法，else分支中是S3C2440的代码，如下所示：

```
85  /* for s3c2440 */
86  #define S3C2440_CLKDIVN_PDIVN       (1<<0)
87  #define S3C2440_CLKDIVN_HDIVN_MASK  (3<<1)
88  #define S3C2440_CLKDIVN_HDIVN_1     (0<<1)
89  #define S3C2440_CLKDIVN_HDIVN_2     (1<<1)
90  #define S3C2440_CLKDIVN_HDIVN_4_8   (2<<1)
91  #define S3C2440_CLKDIVN_HDIVN_3_6   (3<<1)
92  #define S3C2440_CLKDIVN_UCLK        (1<<3)
93
94  #define S3C2440_CAMDIVN_CAMCLK_MASK (0xf<<0)
95  #define S3C2440_CAMDIVN_CAMCLK_SEL  (1<<4)
96  #define S3C2440_CAMDIVN_HCLK3_HALF  (1<<8)
97  #define S3C2440_CAMDIVN_HCLK4_HALF  (1<<9)
98  #define S3C2440_CAMDIVN_DVSEN       (1<<12)
99
100 /* return HCLK frequency */
101 ulong get_HCLK(void)
102 {
103     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
104     unsigned long clkdiv;
105     unsigned long camdiv;
106     int hdiv = 1;
107
108     /* 同时支持S3C2410和S3C2440 */
109     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
110         return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
111     else
112     {
113         clkdiv = clk_power->CLKDIVN;
114         camdiv = clk_power->CAMDIVN;
115
```

```
116         /* 计算分频比 */
117
118         switch (clkdiv & S3C2440_CLKDIVN_HDIVN_MASK) {
119         case S3C2440_CLKDIVN_HDIVN_1:
120             hdiv = 1;
121             break;
122
123         case S3C2440_CLKDIVN_HDIVN_2:
124             hdiv = 2;
125             break;
126
127         case S3C2440_CLKDIVN_HDIVN_4_8:
128             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK4_HALF) ? 8 : 4;
129             break;
130
131         case S3C2440_CLKDIVN_HDIVN_3_6:
132             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK3_HALF) ? 6 : 3;
133             break;
134        }
135
136         return get_FCLK() / hdiv;
137     }
138 }
139
140 /* return PCLK frequency */
141 ulong get_PCLK(void)
142 {
143     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
144     unsigned long clkdiv;
145     unsigned long camdiv;
146     int hdiv = 1;
147
148     /* 同时支持 S3C2410 和 S3C2440 */
149     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
150         return((clk_power->CLKDIVN & 0x1) ? get_HCLK()/2 : get_HCLK());
151     else
152     {
153         clkdiv = clk_power->CLKDIVN;
154         camdiv = clk_power->CAMDIVN;
```

```
155
156         /* 计算分频比 */
157
158         switch (clkdiv & S3C2440_CLKDIVN_HDIVN_MASK) {
159         case S3C2440_CLKDIVN_HDIVN_1:
160             hdiv = 1;
161             break;
162
163         case S3C2440_CLKDIVN_HDIVN_2:
164             hdiv = 2;
165             break;
166
167         case S3C2440_CLKDIVN_HDIVN_4_8:
168             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK4_HALF) ? 8 : 4;
169             break;
170
171         case S3C2440_CLKDIVN_HDIVN_3_6:
172             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK3_HALF) ? 6 : 3;
173             break;
174        }
175
176         return get_FCLK() / hdiv / ((clkdiv & S3C2440_CLKDIVN_PDIVN)? 2:1);
177     }
178 }
179
```

现在重新执行“ make 100ask24x0_config ”和“ make all ”生成的 U-Boot.bin 文件既可以运行于 S3C2410 开发板，也可以运行于 S3C2440 开发板。将它烧入 NOR Flash 后启动，就可以在串口工具（设置为 115200,8N1）中看到提示信息，可以输入各种命令操作 U-Boot 了。

（4）选择 NOR Flash 的型号。

但是，现在还无法通过 U-Boot 命令烧写 NOR Flash。本书所用开发板中的 NOR Flash 型号为 AM29LV800，而配置文件 include/configs/100ask24x0.h 中的默认型号为 AM29LV400。修改如下：

```
#define CONFIG_AMD_LV400 1   /* uncomment this if you have a LV400 flash */
#if 0
#define CONFIG_AMD_LV800 1   /* uncomment this if you have a LV800 flash */
#endif
改为：
#if 0
```

```
#define CONFIG_AMD_LV400 1   /* uncomment this if you have a LV400 flash */
#endif
#define CONFIG_AMD_LV800 1   /* uncomment this if you have a LV800 flash */
```

本例中 NOR Flash 的操作函数在 board/100ask24x0/flash.c 中实现，它支持 AM29LV400y 和 AM29LV800。对于其他型号的 NOR Flash，如果符合 CFI 接口标准，则可以在使用 drivers/cfi_flash.c 中的接口函数；否则，只好自己编写了。如果要使用 cfi_flash.c，如下修改两个文件。

在 include/configs/100ask24x0.h 中增加以下一行：

```
#define CFG_FLASH_CFI_DRIVER    1
```

在 board/100ask24x0/Makefile 中去掉 flash.o：

```
COBJS   := 100ask24x0.o flash.o
改为：
COBJS   := 100ask24x0.o
```

修改好对 NOR Flash 的支持后，重新编译 U-Boot：make clean、make all。运行后可以在串口中看到如下字样：

```
Flash:  1 MB
```

现在可以使用 loadb、loady 等命令通过串口下载文件，然后使用 erase、cp 命令分别擦除、烧写 NOR Flash 了，它们的效率比 JTAG 高好几倍。

2．支持串口 xmodem 协议

上面的 loadb 命令需要配合 Linux 下的 kermit 工具来使用，loady 命令通过串口 ymodem 协议来传输文件。Windows 下的超级终端虽然支持 ymodem，但是它的使用界面实在不友好。而本书推荐使用的 Windows 工具 SecureCRT 只支持 xmodem 和 zmodem。为了方便在 Windows 下开发，现在修改代码增加对 xmodem 的支持，即增加一个命令 loadx。

依照 loady 的实现来编写代码，首先使用 U_BOOT_CMD 宏来增加 loadx 命令：

```
/* 支持 xmodem, www.100ask.net */
U_BOOT_CMD(
        loadx, 3, 0,   do_load_serial_bin,
        "loadx   - load binary file over serial line (xmodem mode)\n",
        "[ off ] [ baud ]\n"
        "    - load binary file over serial line"
        " with offset 'off' and baudrate 'baud'\n"
);
```

其次，在 do_load_serial_bin 函数中增加对 loadx 命令的处理分支。也是依照 loady 来实现：

```
481     /* 支持 xmodem */
```

```
482    if (strcmp(argv[0],"loadx")==0) {
483        printf ("## Ready for binary (xmodem) download "
484            "to 0x%08lX at %d bps...\n",
485            offset,
486            load_baudrate);
487
488        addr = load_serial_xmodem (offset);
489
490    } else if (strcmp(argv[0],"loady")==0) {
491        printf ("## Ready for binary (ymodem) download "
492            "to 0x%08lX at %d bps...\n",
......
```

第 481～490 行就是为 loadx 命令增加的代码。

在第 288 行调用 load_serial_xmodem 函数，它是依照 load_serial_ymodem 实现的一个新函数：

```
36 #if (CONFIG_COMMANDS & CFG_CMD_LOADB)
37 /* 支持 xmodem */
38 static ulong load_serial_xmodem (ulong offset);
39 static ulong load_serial_ymodem (ulong offset);
40 #endif
......
995 /* 支持 xmodem */
996 static ulong load_serial_xmodem (ulong offset)
997 {
......
1003      char xmodemBuf[1024];              /* 原来是ymodemBuf,这只是为了与函数名称一致 */
......
1008      info.mode = xyzModem_xmodem;     /* 原来是 xyzModem_ymodem，对应 ymodem */
......
```

首先在文件开头增加 load_serial_xmodem 函数的声明，然后复制 load_serial_ymodem 函数为 load_serial_xmodem，稍作修改。

① 将局部数组 ymodemBuf 改名为 xmodemBuf，并在后面使用到的地方统一修改。这只是为了与函数名称一致。

② info.mode 的值从 xyzModem_ymodem 改为 xyzModem_xmodem。

重新编译、烧写 U-Boot.bin 后，就可以使用 loadx 命令下载文件了。

3．支持网卡芯片 CS8900

使用串口来传输文件的速率太低，现在增加对网卡芯片 CS8900 的支持。

本书使用开发板的网卡芯片 CS8900 的连接方式与 smdk2410 完全一样，所以现在的 U-Boot 中已经支持 CS8900 了，它的驱动程序为 drivers/cs8900.c。只要在 U-Boot 控制界面中稍加配置就可以使用网络功能。使用网络之前，先设置开发板 IP 地址、MAC 地址，服务器 IP 地址，比如可以在 U-Boot 中执行以下命令：

```
setenv ipaddr 192.168.1.17
setenv ethaddr 08:00:3e:26:0a:5b
setenv serverip 192.168.1.11
saveenv
```

然后就可以使用 tftp 或 nfs 命令下载文件了，注意：服务器上要开启 tftp 或 nfs 服务。比如可以使用如下命令将 U-Boot.bin 文件下载到内存 0x30000000 中：

```
tftp 0x30000000 U-Boot.bin
或
nfs 0x30000000 192.168.1.57:/work/nfs_root/U-Boot.bin
```

可以修改配置文件，让网卡的各个默认值就是上面设置的值。在此之前，先了解网卡的相关文件，这有助于移植代码以支持其他连接方式的 CS8900。

首先，CS8900 接在 S3C2410、S3C2440 的 BANK3，位宽为 16，使用 WAIT、nBE 信号。在设置存储控制器时要设置好 BANK3。代码在 board/100ask24x0/lowlevel_init.S 中，如下所示：

```
#define B3_BWSCON       (DW16 + WAIT + UBLB)
......
/* 时序参数 */
#define B3_Tacs         0x0 /*  0clk */
#define B3_Tcos         0x3 /*  4clk */
#define B3_Tacc         0x7 /* 14clk */
#define B3_Tcoh         0x1 /*  1clk */
#define B3_Tah          0x0 /*  0clk */
#define B3_Tacp         0x3  /*  6clk */
#define B3_PMC          0x0 /* normal */
```

接下来，还要确定 CS8900 的基地址。这在配置文件 include/configs/100ask24x0.h 中定义，如下所示：

```
#define CONFIG_DRIVER_CS8900    1           /* 使用 CS8900 */
#define CS8900_BASE             0x19000300  /* 基地址 */
#define CS8900_BUS16            1           /* 位宽为 16 */
```

网卡 CS8900 的访问基址为 0x19000000，之所以再偏移 0x300 是由它的特性决定的。

最后，还是在配置文件 include/configs/100ask24x0.h 中定义 CS8900 的各个默认地址，如下所示：

```
#define CONFIG_ETHADDR      08:00:3e:26:0a:5b
```

```
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.1.17
#define CONFIG_SERVERIP     192.168.1.11
```

如果要增加 ping 命令，还可以在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_PING，如下所示：

```
#define CONFIG_COMMANDS \
                      (CONFIG_CMD_DFL    | \
                      CFG_CMD_CACHE  | \
                      CFG_CMD_PING       | \
……
```

4．支持 NAND Flash

U-Boot 1.1.6 中对 NAND Flash 的支持有新旧两套代码，新代码在 drivers/nand 目录下，旧代码在 drivers/nand_legacy 目录下。文档 doc/README.nand 对这两套代码有所说明：使用旧代码需要定义更多的宏，而新代码移植自 Linux 内核 2.6.12，它更加智能，可以自动识别更多型号的 NAND Flash。目前之所以还保留旧的代码，是因为两个目标板 NETTA、NETTA_ISDN 使用 JFFS 文件系统，它们还依赖于旧代码。当相关功能移植到新代码之后，旧的代码将从 U-Boot 中去除。

要让 U-Boot 支持 NAND Flash，首先在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_NAND，如下所示：

```
#define CONFIG_COMMANDS \
                      (CONFIG_CMD_DFL    | \
                      CFG_CMD_CACHE  | \
                      CFG_CMD_PING       | \
                      CFG_CMD_NAND   | \
…
```

然后选择使用哪套代码：在配置文件中定义宏 CFG_NAND_LEGACY 则使用旧代码，否则使用新代码。

使用旧代码时，需要实现 drivers/nand_legacy/nand_legacy.c 中使用到的各种宏，比如：

```
#define NAND_WAIT_READY(nand)   /* 等待 Nand Flash 的状态为“就绪”，代码依赖于具体的开发板 */
#define WRITE_NAND_COMMAND(d, adr)   /* 写 NAND Flash 命令，代码依赖于具体的开发板 */
```

本书使用新代码，下面讲述移植过程。

代码的移植没有现成的文档，可以在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_NAND 后就编译代码，然后一个一个地解决出现的错误。编译结果中出现的错误和警告如下：

```
nand.h:412: error: 'NAND_MAX_CHIPS' undeclared here (not in a function)
```

```
nand.c:35: error: 'CFG_MAX_NAND_DEVICE' undeclared here (not in a function)
nand.c:38: error: 'CFG_NAND_BASE' undeclared here (not in a function)
nand.c:35: error: storage size of 'nand_info' isn't known
nand.c:37: error: storage size of 'nand_chip' isn't known
nand.c:38: error: storage size of 'base_address' isn't known
nand.c:37: warning: 'nand_chip' defined but not used
nand.c:38: warning: 'base_address' defined but not used
```

在配置文件 include/configs/100ask24x0.h 中增加如下 3 个宏就可以解决上述错误。在 Flash 的驱动程序中，设备是逻辑上的概念，表示一组相同结构、访问函数相同的 Flash 芯片。在本书所用开发板中，只有一个 NAND Flash 芯片，所以设备数为 1，芯片数也为 1。

```
#define CFG_NAND_BASE          0    /* 无实际意义：基地址，这在 board_nand_init
中重新指定 */
#define CFG_MAX_NAND_DEVICE    1    /* NAND Flash“设备”的数目为 1 */
#define NAND_MAX_CHIPS         1    /* 每个 NAND Flash“设备”由 1 个 NAND Flash
“芯片”组成 */
```

修改配置文件后再次编译，现在只有一个错误了，“board_nand_init 函数未定义”，如下所示：

```
nand.c:50: undefined reference to `board_nand_init'
```

调用 board_nand_init 函数的过程为：NAND Flash 的初始化入口函数是 nand_init，它在 lib_arm/board.c 的 start_armboot 函数中被调用；nand_init 函数在 drivers/ nand/nand.c 中实现，它调用相同文件中的 nand_init_chip 函数；nand_init_chip 函数首先调用 board_nand_init 函数来初始化 NAND Flash 设备，最后才是统一的识别过程。

从 board_nand_init 函数的名称就可以知道它是平台/开发板相关的函数，需要自己编写。本书在 cpu/arm920t/s3c24x0 目录下新建一个文件 nand_flash.c，在里面针对 S3C2410、S3C2440 实现了统一的 board_nand_init 函数。

在编写 board_nand_init 函数的之前，需要针对 S3C2410、S3C2440 NAND Flash 控制器的不同来定义一些数据结构和函数：

（1）在 include/s3c24x0.h 文件中增加 S3C2440_NAND 数据结构。

```
typedef struct {
    S3C24X0_REG32   NFCONF;
    S3C24X0_REG32   NFCONT;
    S3C24X0_REG32   NFCMD;
    S3C24X0_REG32   NFADDR;
    S3C24X0_REG32   NFDATA;
    S3C24X0_REG32   NFMECCD0;
    S3C24X0_REG32   NFMECCD1;
    S3C24X0_REG32   NFSECCD;
```

```
        S3C24X0_REG32   NFSTAT;
        S3C24X0_REG32   NFESTAT0;
        S3C24X0_REG32   NFESTAT1;
        S3C24X0_REG32   NFMECC0;
        S3C24X0_REG32   NFMECC1;
        S3C24X0_REG32   NFSECC;
        S3C24X0_REG32   NFSBLK;
        S3C24X0_REG32   NFEBLK;
    } /*__attribute__((__packed__))*/ S3C2440_NAND;
```

（2）在 include/s3c2410.h 文件中仿照 S3C2410_GetBase_NAND 函数定义 S3C2440_GetBase_NAND 函数。

```
/* for s3c2440 */
static inline S3C2440_NAND * const S3C2440_GetBase_NAND(void)
{
    return (S3C2440_NAND * const)S3C2410_NAND_BASE;
}
```

既然新的 NAND Flash 代码是从 Linux 内核 2.6.12 中移植来的，那么 cpu/arm920t/s3c24x0/nand_flash.c 文件也可以仿照内核中对 S3C2410、S3C2440 的 NAND Flash 进行初始化的 drivers/mtd/nand/s3c2410.c 文件来编写。为了方便阅读，先把 cpu/arm920t/s3c24x0/nand_flash.c 文件的代码全部列出来，如下所示：

```
/*
 * s3c2410/s3c2440 的 NAND Flash 控制器接口
 * 修改自 Linux 内核 2.6.13 文件 drivers/mtd/nand/s3c2410.c
 */

#include <common.h>

#if (CONFIG_COMMANDS & CFG_CMD_NAND) && !defined(CFG_NAND_LEGACY)
#include <s3c2410.h>
#include <nand.h>

DECLARE_GLOBAL_DATA_PTR;

#define S3C2410_NFSTAT_READY    (1<<0)
#define S3C2410_NFCONF_nFCE     (1<<11)

#define S3C2440_NFSTAT_READY    (1<<0)
```

```
18 #define S3C2440_NFCONT_nFCE    (1<<1)
19
20
21 /* S3C2410：NAND Flash 的片选函数 */
22 static void s3c2410_nand_select_chip(struct mtd_info *mtd, int chip)
23 {
24     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
25
26     if (chip == -1) {
27         s3c2410nand->NFCONF |= S3C2410_NFCONF_nFCE; /* 禁止片选信号 */
28     } else {
29         s3c2410nand->NFCONF &= ~S3C2410_NFCONF_nFCE;  /* 使能片选信号 */
30     }
31 }
32
33 /* S3C2410：命令和控制函数
34  *
35  * 注意，这个函数仅仅根据各种命令来修改“写地址”IO_ADDR_W 的值(这称为 tglx 方法)，
36  * 这种方法使得平台/开发板相关的代码很简单。
37  * 真正发出命令是在上一层 NAND Flash 的统一的驱动中实现，
38  * 它首先调用这个函数修改“写地址”，然后才分别发出控制、地址、数据序列。
39 */
40 static void s3c2410_nand_hwcontrol(struct mtd_info *mtd, int cmd)
41 {
42     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
43     struct nand_chip *chip = mtd->priv;
44
45     switch (cmd) {
46     case NAND_CTL_SETNCE:
47     case NAND_CTL_CLRNCE:
48         printf("%s: called for NCE\n", __FUNCTION__);
49         break;
50
51     case NAND_CTL_SETCLE:
52         chip->IO_ADDR_W = (void *)&s3c2410nand->NFCMD;
53         break;
54
55     case NAND_CTL_SETALE:
56         chip->IO_ADDR_W = (void *)&s3c2410nand->NFADDR;
```

```
57          break;
58
59          /* NAND_CTL_CLRCLE: */
60          /* NAND_CTL_CLRALE: */
61      default:
62          chip->IO_ADDR_W = (void *)&s3c2410nand->NFDATA;
63          break;
64      }
65 }
66
67 /* S3C2410：查询 NAND Flash 状态
68  *
69  * 返回值：0 表示忙,1 表示就绪
70  */
71 static int s3c2410_nand_devready(struct mtd_info *mtd)
72 {
73      S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
74
75      return (s3c2410nand->NFSTAT & S3C2410_NFSTAT_READY);
76 }
77
78
79 /* S3C2440：NAND Flash 的片选函数 */
80 static void s3c2440_nand_select_chip(struct mtd_info *mtd, int chip)
81 {
82      S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
83
84      if (chip == -1) {
85          s3c2440nand->NFCONT |= S3C2440_NFCONT_nFCE; /* 禁止片选信号 */
86      } else {
87          s3c2440nand->NFCONT &= ~S3C2440_NFCONT_nFCE;  /* 使能片选信号 */
88      }
89 }
90
91 /* S3C2440：命令和控制函数，与 s3c2410_nand_hwcontrol 函数类似 */
92 static void s3c2440_nand_hwcontrol(struct mtd_info *mtd, int cmd)
93 {
94      S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
95      struct nand_chip *chip = mtd->priv;
```

```
96
97      switch (cmd) {
98      case NAND_CTL_SETNCE:
99      case NAND_CTL_CLRNCE:
100         printf("%s: called for NCE\n", __FUNCTION__);
101         break;
102
103      case NAND_CTL_SETCLE:
104         chip->IO_ADDR_W = (void *)&s3c2440nand->NFCMD;
105         break;
106
107      case NAND_CTL_SETALE:
108          chip->IO_ADDR_W = (void *)&s3c2440nand->NFADDR;
109          break;
110
111          /* NAND_CTL_CLRCLE: */
112          /* NAND_CTL_CLRALE: */
113      default:
114         chip->IO_ADDR_W = (void *)&s3c2440nand->NFDATA;
115         break;
116     }
117 }
118
119 /* S3C2440：查询 NAND Flash 状态
120  *
121  * 返回值：0 表示忙,1 表示就绪
122  */
123 static int s3c2440_nand_devready(struct mtd_info *mtd)
124 {
125     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
126
127     return (s3c2440nand->NFSTAT & S3C2440_NFSTAT_READY);
128 }
129
130 /*
131  * Nand flash 硬件初始化:
132  * 设置 NAND Flash 的时序, 使能 NAND Flash 控制器
133  */
134 static void s3c24x0_nand_inithw(void)
```

```
135 {
136     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
137     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
138
139 #define TACLS   0
140 #define TWRPH0  4
141 #define TWRPH1  2
142
143     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
144     {
145         /* 使能 NAND Flash 控制器，初始化 ECC，使能片选信号，设置时序 */
146         s3c2410nand->NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)| (TWRPH0<<4)|
(TWRPH1<<0);
147     }
148     else
149     {
150         /* 设置时序 */
151         s3c2440nand->NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
152         /* 初始化 ECC，使能 NAND Flash 控制器，使能片选信号 */
153         s3c2440nand->NFCONT = (1<<4)|(0<<1)|(1<<0);
154     }
155 }
156
157 /*
158  * 被 drivers/nand/nand.c 调用，初始化 NAND Flash 硬件，初始化访问接口函数
159  */
160 void board_nand_init(struct nand_chip *chip)
161 {
162     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
163     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
164
165     s3c24x0_nand_inithw();  /* Nand flash 硬件初始化 */
166
167     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410) {
168         chip->IO_ADDR_R    = (void *)&s3c2410nand->NFDATA;
169         chip->IO_ADDR_W    = (void *)&s3c2410nand->NFDATA;
170         chip->hwcontrol    = s3c2410_nand_hwcontrol;
171         chip->dev_ready    = s3c2410_nand_devready;
172         chip->select_chip  = s3c2410_nand_select_chip;
```

```
173         chip->options     = 0; /* 设置位宽等，位宽为 8 */
174     } else {
175         chip->IO_ADDR_R    = (void *)&s3c2440nand->NFDATA;
176         chip->IO_ADDR_W    = (void *)&s3c2440nand->NFDATA;
177         chip->hwcontrol    = s3c2440_nand_hwcontrol;
178         chip->dev_ready    = s3c2440_nand_devready;
179         chip->select_chip  = s3c2440_nand_select_chip;
180         chip->options     = 0; /* 设置位宽等，位宽为 8 */
181     }
182
183     chip->eccmode      = NAND_ECC_SOFT; /* ECC 校验方式：软件 ECC */
184 }
185
186 #endif
```

文件中分别针对 S3C2410、S3C2440 实现了 NAND Flash 最底层访问函数，并进行了一些硬件的设置（比如时序、使能 NAND Flash 控制器等）。新的代码对 NAND Flash 的封装做得很好，只要向上提供底层初始化函数 board_nand_init 来设置好平台/开发板相关的初始化、提供底层接口即可。

最后，只要将新建的 nand_flash.c 文件编入 U-Boot 中就可以擦除、读写 NAND Flash 了。如下修改 cpu/arm920t/s3c24x0/Makefile 文件即可。

修改前：

```
COBJS   = i2c.o interrupts.o serial.o speed.o \
          usb_ohci.o
```

修改后：

```
COBJS   = i2c.o interrupts.o serial.o speed.o \
          usb_ohci.o nand_flash.o
```

现在，可以使用新编译的 U-Boot.bin 烧写内核映象到 NAND Flash 去了。

5．支持烧写 yaffs 文件系统映象

在实际生产中，可以通过烧片器等手段将内核、文件系统映象烧入固态存储设备中，Bootloader 不需要具备烧写功能。但为了方便开发，通常在 Bootloader 中增加烧写内核、文件系统映象文件的功能。

增加了 NAND Flash 功能的 U-Boot 1.1.6 已经可以通过“ nand write… ”、“ nand write.jffs2… ”等命令来烧写内核，烧写 cramfs、jffs2 文件系统映象文件。但是在 NAND Flash 上，yaffs 文件系统的性能更佳，下面增加“ nand write.yaffs… ”命令以烧写 yaffs 文件系统映象文件。

“ nand write.yaffs… ”字样的命令中，“ nand ”是具体命令，“ write.yaffs… ”是参数。nand

命令在 common/cmd_nand.c 中实现如下：

```
    U_BOOT_CMD(nand, 5, 1, do_nand,
        "nand    - NAND sub-system\n",
        "info                  - show available NAND devices\n"
        "nand device [dev]     - show or set current device\n"
        "nand read[.jffs2]     - addr off|partition size\n"
        "nand write[.jffs2]    - addr off|partiton size - read/write 'size' bytes starting\n"
        "    at offset 'off' to/from memory address 'addr'\n"
    …
```

先在其中增加“ nand write.yaffs… ”的使用说明，如下所示：

```
    U_BOOT_CMD(nand, 5, 1, do_nand,
        "nand    - NAND sub-system\n",
        "info                  - show available NAND devices\n"
        "nand device [dev]     - show or set current device\n"
        "nand read[.jffs2]     - addr off|partition size\n"
        "nand write[.jffs2]    - addr off|partiton size - read/write 'size' bytes starting\n"
        "    at offset 'off' to/from memory address 'addr'\n"
        "nand read.yaffs addr off size - read the 'size' byte yaffs image starting\n"
        "    at offset 'off' to memory address 'addr'\n"
        "nand write.yaffs addr off size - write the 'size' byte yaffs image starting\n"
        "    at offset 'off' from memory address 'addr'\n"
    …
```

然后，在 nand 命令的处理函数 do_nand 中增加对“ write.yaffs… ”的支持。do_nand 函数仍在 common/cmd_nand.c 中实现，代码修改如下：

```
            (!strcmp(s, ".jffs2") || !strcmp(s, ".e") || !strcmp(s, ".i"))) {
…
        }else if (  s != NULL && !strcmp(s, ".yaffs")){
            if (read) {
                /* read */
                nand_read_options_t opts;
                memset(&opts, 0, sizeof(opts));
                opts.buffer = (u_char*) addr;
                opts.length = size;
```

```
    361                 opts.offset = off;
    362                 opts.readoob = 1;
    363                 opts.quiet     = quiet;
    364                 ret = nand_read_opts(nand, &opts);
    365             } else {
    366                 /* write */
    367                 nand_write_options_t opts;
    368                 memset(&opts, 0, sizeof(opts));
    369               opts.buffer = (u_char*) addr;    /* yaffs 文件系统映象存放的地址 */
    370                 opts.length = size;          /* 长度 */
    371               opts.offset = off;             /* 要烧写到的 NAND Flash 的偏移地址 */
    372               /* opts.forceyaffs = 1; */     /* 计算 ECC 码的方法，没有使用 */
    373                 opts.noecc = 1;                  /* 不需要计算 ECC，yaffs 映
象中有 OOB 数据 */
    374                 opts.writeoob = 1;           /* 写 OOB 区 */
    375                 opts.blockalign = 1;         /* 每个“ 逻辑上的块 ”大小为 1 个“ 物
理块 ” */
    376                 opts.quiet     = quiet;      /* 是否打印提示信息 */
    377                 opts.skipfirstblk = 1;         /* 跳过第一个可用块 */
    378                 ret = nand_write_opts(nand, &opts);
    379             }
    380         } else {
    …
    385         }
    386
```

第 354～379 行就是针对命令“ nand read.yaffs… ”、“ nand write.yaffs… ”增加的代码。有兴趣的读者可以自己分析“ if (read) ”分支的代码，下面只讲解“ else ”分支，即“ nand write.yaffs… ”命令的实现。

NAND Flash 每一页大小为（512+16）字节（还有其他格式的 NAND Flash，比如每页大小为(256+8)、(2048+64)等），其中的 512 字节就是一般存储数据的区域，16 字节称为 OOB（Out Of Band）区。通常在 OOB 区存放坏块标记、前面 512 字节的 ECC 校验码等。

cramfs、jffs2 文件系统映象文件中并没有 OOB 区的内容，如果将它们烧入 NOR Flash 中，则是简单的“ 平铺 ”关系；如果将它们烧入 NAND Flash 中，则 NAND Flash 的驱动程序首先根据 OOB 的标记略过坏块，然后将一页数据（512 字节）写入后，还会计算这 512 字节的 ECC 校验码，最后将它写入 OOB 区，如此循环。cramfs、jffs2 文件系统映象文件的大小通常是 512 的整数倍。

而 yaffs 文件系统映象文件的格式则跟它们不同，文件本身就包含了 OOB 区的数据（里面有坏块标记、ECC 校验码、其他 yaffs 相关的信息）。所以烧写时，不需要再计算 ECC 值，首先检查是否坏块（是则跳过），然后写入 512 字节的数据，最后写入 16 字节的 OOB 数据，

如此循环。yaffs 文件系统映象文件的大小是（512+16）的整数倍。

注意 烧写 yaffs 文件系统映象时，分区上第一个可用的（不是坏块）块也要跳过。

下面分析上面的代码。

第 369～371 行设置源地址、目的地址、长度。烧写 yaffs 文件系统映象前，一般通过网络将它下载到内存某个地址处（比如 0x30000000），然后通过类似“ nand write.yaffs 0x30000000 0x00A00000 $(filesize) ”的命令烧到 NAND Flash 的偏移地址 0x00A00000 处。对于这个命令，第 369 行中 opts.buffer 等于 0x30000000，第 370 行中 opts.length 等于$(filesize)的值，就是前面下载的文件的大小，第 371 行中的 opts.offset 等于 0x00A00000。

这里列出不使用的第 372 行，是因为 opts.forceyaffs 这个名字很有欺骗性，它其实是指计算 ECC 校验码的一种方法。烧写 yaffs 文件系统映象时，不需要计算 ECC 校验码。

第 373、374 行指定烧写数据时不计算 ECC 校验码、而是烧入文件中的 OOB 数据。

第 375 行指定“ 逻辑块 ”的大小，“ 逻辑块 ”可以由多个“ 物理块 ”组成，在 yaffs 文件系统映象中，它们是 1:1 的关系。

第 377 行的 opts.skipfirstblk 是新加的项，nand_write_options_t 结构中没有 skipfirstblk 成员。它表示烧写时跳过第一个可用的逻辑块，这是由 yaffs 文件系统的特性决定的。

既然 skipfirstblk 是在 nand_write_options_t 结构中新加的项，那么就要重新定义 nand_write_options_t 结构，并在下面调用的 nand_write_opts 函数中对它进行处理。

首先在 include/nand.h 中进行如下修改，增加 skipfirstblk 成员。

```
struct nand_write_options {
        u_char *buffer;     /* memory block containing image to write */
        ulong length;       /* number of bytes to write */
        ulong offset;       /* start address in NAND */
        int quiet;          /* don't display progress messages */
        int autoplace;      /* if true use auto oob layout */
        int forcejffs2;     /* force jffs2 oob layout */
        int forceyaffs;     /* force yaffs oob layout */
        int noecc;          /* write without ecc */
        int writeoob;       /* image contains oob data */
        int pad;            /* pad to page size */
       int blockalign;      /* 1|2|4 set multiple of eraseblocks to align to */
        int skipfirstblk;   /* 新加，烧写时跳过第一个可用的逻辑块 */
};
typedef struct nand_write_options nand_write_options_t;
```

然后，修改 nand_write_opts 函数，增加对 skipfirstblk 成员的支持。它在 drivers/nand/nand_util.c 文件中，下面的第 301、第 421～424 行的新加的。

```
285 int nand_write_opts(nand_info_t *meminfo, const nand_write_options_t
*opts)
286 {
```

```
…
300     int result;
301     int skipfirstblk = opts->skipfirstblk;
…
397         while (blockstart != (mtdoffset & (~erasesize_blockalign+1))) {
…
419                 }
420
421                 if (baderaseblock) {
422                     mtdoffset = blockstart
423                         + erasesize_blockalign;
424                 }
…
```

进行了上面的移植后，U-Boot 已经可以烧写 yaffs 文件系统映象了。由于前面设置“ opts.noecc = 1 ”不使用 ECC 校验码，在烧写过程中会出现很多的提示信息，如下所示：

```
Writing data without ECC to NAND-FLASH is not recommended
```

可以修改 drivers/nand/nand_base.c 文件的 nand_write_page 函数，将它去掉。

修改前：

```
917 case NAND_ECC_NONE:
918     printk (KERN_WARNING "Writing data without ECC to NAND-FLASH is not
recommended\n");
```

修改后：

```
917 case NAND_ECC_NONE:
918     //printk (KERN_WARNING "Writing data without ECC to NAND-FLASH is not
recommended\n");
```

6．修改默认配置参数以方便使用

前面移植网卡芯片 CS8900 时，已经设置过默认 IP 地址等。为了使用 U-Boot 时减少一些设置，现在修改配置文件 include/configs/100ask24x0.h，增加默认配置参数，其中一些在移植过程中已经增加的选项这里也再次说明。

（1）Linux 启动参数。

增加如下 3 个宏：

```
#define CONFIG_SETUP_MEMORY_TAGS    1       /* 向内核传递内存分布信息 */
#define CONFIG_CMDLINE_TAG          1       /* 向内核传递命令行参数 */
/* 默认命令行参数 */
#define CONFIG_BOOTARGS         "noinitrd root=/dev/mtdblock 2 init=/linuxrc
```

```
console=ttySAC0"
```

（2）自动启动命令。

增加如下 2 个宏：

```
/* 自动启动前延时 3s */
#define CONFIG_BOOTDELAY        3
/* 自动启动的命令 */
#define CONFIG_BOOTCOMMAND      "nboot 0x32000000 0 0; bootm 0x32000000"
```

自动启动时（开机 3s 内无输入），首先执行“ nboot 0x32000000 0 0 ”命令将第 0 个 NAND Flash 偏移地址 0 上的映象文件复制到内存 0x32000000 中；然后执行“ bootm 0x32000000 ”命令启动内存中的映象。

（3）默认网络设置。

根据具体网络环境增加、修改下面 4 个宏：

```
#define CONFIG_ETHADDR      08:00:3e:26:0a:5b
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.1.17
#define CONFIG_SERVERIP     192.168.1.11
```

15.2.6　U-Boot 的常用命令

1．U-Boot 的常用命令的用法

进入 U-Boot 控制界面后，可以运行各种命令，比如下载文件到内存，擦除、读写 Flash，运行内存、NOR Flash、NAND Flash 中的程序，查看、修改、比较内存中的数据等。

使用各种命令时，可以使用其开头的若干个字母代替它。比如 tftpboot 命令，可以使用 t、tf、tft、tftp 等字母代替，只要其他命令不以这些字母开头即可。

当运行一个命令之后，如果它是可重复执行的（代码中使用 U_BOOT_CMD 定义这个命令时，第 3 个参数是 1），若想再次运行可以直接输入回车。

U-Boot 接收的数据都是十六进制，输入时可以省略前缀 0x、0X。

下面介绍常用的命令。

（1）帮助命令 help。

运行 help 命令可以看到 U-Boot 中所有命令的作用，如果要查看某个命令的使用方法，运行“ help 命令名 ”，比如“ help bootm ”。

可以使用“ ? ”来代替“ help ”，比如直接输入“ ? ”、“ ? bootm ”。

（2）下载命令。

U-Boot 支持串口下载、网络下载，相关命令有：loadb、loads、loadx、loady 和 tftpboot、nfs。

前几个串口下载命令使用方法相似，以 loadx 命令为例，它的用法为“ loadx [off] [baud] ”。“ [] ”表示里面的参数可以省略，off 表示文件下载后存放的内存地址，baud 表示使用的波特率。如果 baud 参数省略，则使用当前的波特率；如果 off 参数省略，存放的地址为配置文件中定义的宏 CFG_LOAD_ADDR。

tftpboot 命令使用 TFTP 协议从服务器下载文件，服务器的 IP 地址为环境变量 serverip。

用法为“ tftpboot [loadAddress] [bootfilename] ”，loadAddress 表示文件下载后存放的内存地址，bootfilename 表示要下载的文件的名称。如果 loadAddress 省略，存放的地址为配置文件中定义的宏 CFG_LOAD_ADDR；如果 bootfilename 省略，则使用开发板的 IP 地址构造一个文件名，比如开发板 IP 为 192.168.1.17，则默认的文件名为 C0A80711.img。

nfs 命令使用 NFS 协议下载文件，用法为“ nfs [loadAddress] [host ip addr:bootfilename] ”。“ loadAddress、bootfilename ”的意义与 tftpboot 命令一样，“ host ip addr ”表示服务器的 IP 地址，默认为环境变量 serverip。

下载文件成功后，U-Boot 会自动创建或更新环境变量 filesize，它表示下载的文件的长度，可以在后续命令中使用“ $(filesize) ”来引用它。

（3）内存操作命令。

常用的命令有：查看内存命令 md、修改内存命令 md、填充内存命令 mw、复制命令 cp。这些命令都可以带上后缀“ .b ”、“ .w ”或“ .l ”，表示以字节、字（2 个字节）、双字（4 个字节）为单位进行操作。比如“ cp.l 30000000 31000000 2 ”将从开始地址 0x30000000 处，复制 2 个双字到开始地址为 0x31000000 的地方。

md 命令用法为“ md[.b, .w, .l] address [count] ”，表示以字节、字或双字（默认为双字）为单位，显示从地址 address 开始的内存数据，显示的数据个数为 count。

mm 命令用法为“ mm[.b, .w, .l] address ”，表示以字节、字或双字（默认为双字）为单位，从地址 address 开始修改内存数据。执行 mm 命令后，输入新数据后回车，地址会自动增加，按“ Ctrl+C ”键退出。

mw 命令用法为“ mw[.b, .w, .l] address value [count] ”，表示以字节、字或双字（默认为双字）为单位，往开始地址为 address 的内存中填充 count 个数据，数据值为 value。

cp 命令用法为“ cp[.b, .w, .l] source target count ”，表示以字节、字或双字（默认为双字）为单位，从源地址 source 的内存复制 count 个数据到目的地址的内存。

（4）NOR Flash 操作命令。

常用的命令有查看 Flash 信息的 flinfo 命令、加/解写保护命令 protect、擦除命令 erase。由于 NOR Flash 的接口与一般内存相似，所以一些内存命令可以在 NOR Flash 上使用，比如读 NOR Flash 时可以使用 md、cp 命令，写 NOR Flash 时可以使用 cp 命令（cp 根据地址分辨出是 NOR Flash，从而调用 NOR Flash 驱动完成写操作）。

直接运行“ flinfo ”即可看到 NOR Flash 的信息，有 NOR Flash 的型号、容量、各扇区的开始地址、是否只读等信息。比如对于本书基于的开发板，flinfo 命令的结果如下：

```
Bank # 1: AMD: 1x Amd29LV800BB (8Mbit)
  Size: 1 MB in 19 Sectors
  Sector Start Addresses:
    00000000 (RO) 00004000 (RO) 00006000 (RO) 00008000 (RO) 00010000 (RO)
    00020000 (RO) 00030000      00040000      00050000       00060000
    00070000      00080000      00090000      000A0000       000B0000
    000C0000      000D0000      000E0000      000F0000 (RO)
```

其中的 RO 表示该扇区处于写保护状态，只读。

对于只读的扇区，在擦除、烧写它之前，要先解除写保护。最简单的命令为“ protect off all ”，解除所有 NOR Flash 的写保护。

erase 命令常用的格式为“ erase start end ”，擦除的地址范围为 start～end；“ erase start +len ”，擦除的地址范围为 start～（star+tlen−1），“ erase all ”，表示擦除所有 NOR Flash。

> 注意 其中的地址范围，刚好是一个扇区的开始地址到另一个（或同一个）扇区的结束地址。比如要擦除 Amd29LV800BB 的前 5 个扇区，执行的命令为“ erase 0 0x2ffff ”，而非“ erase 0 0x30000 ”。

（5）NAND Flash 操作命令。

NAND Flash 操作命令只有一个：nand，它根据不同的参数进行不同操作，比如擦除、读取、烧写等。

“ nand info ”查看 NAND Flash 信息。

“ nand erase [clean] [off size] ”擦除 NAND Flash。加上“ clean ”时，表示在每个块的第一个扇区的 OOB 区加写入清除标记；off、size 表示要擦除的开始偏移地址的长度，如果省略 off 和 size，表示要擦除整个 NAND Flash。

“ nand read[.jffs2] addr off size ”从 NAND Flash 偏移地址 off 处读出 size 个字节的数据存放到开始地址为 addr 的内存中。是否加后缀“ .jffs ”的差别只是读操作时的 ECC 校验方法不同。

“ nand write[.jffs2] addr off size ”把开始地址为 addr 的内存中的 size 个字节数据写到 NAND Flash 的偏移地址 off 处。是否加后缀“ .jffs ”的差别只是写操作时的 ECC 校验方法不同。

“ nand read.yaffs addr off size ”从 NAND Flash 偏移地址 off 处读出 size 个字节的数据（包括 OOB 区域），存放到开始地址为 addr 的内存中。

“ nand write.yaffs addr off size ”把开始地址为 addr 的内存中的 size 个字节数据（其中有要写入 OOB 区域的数据）写到 NAND Flash 的偏移地址 off 处。

“ nand dump off ”将 NAND Flash 偏移地址 off 的一个扇区的数据打印出来，包括 OOB 数据。

（6）环境变量命令。

“ printenv ”命令打印全部环境变量，“ printenv name1 name2… ”打印名字为 name1、name2、…的环境变量。

“ setenv name value ”设置名字为 name 的环境变量的值为 value。

“ setenv name ”删除名字为 name 的环境变量。

上面的设置、删除操作只是在内存中进行，“ saveenv ”将更改后的所有环境变量写入 NOR Flash 中。

（7）启动命令。

不带参数的“ boot ”、“ bootm ”命令都是执行环境变量 bootcmd 所指定的命令。

“ bootm [addr [arg…]] ”命令启动存放在地址 addr 处的 U-Boot 格式的映象文件（使用 U-Boot 目录 tools 下的 mkimage 工具制作得到），[arg…]表示参数。如果 addr 参数省略，映象文件所在地址为配置文件中定义的宏 CFG_LOAD_ADDR。

“ go addr [arg…] ”与 bootm 命令类似，启动存放在地址 addr 处的二进制文件，[arg...]表示参数。

“ nboot [[[loadAddr] dev] offset] ”命令将 NAND Flash 设备 dev 上偏移地址 off 处的映象文

件复制到内存 loadAddr 处，然后，如果环境变量 autostart 的值为“ yes ”，就启动这个映象。如果 loadAddr 参数省略，存放地址为配置文件中定义的宏 CFG_LOAD_ADDR；如果 dev 参数省略，则它的取值为环境变量 bootdevice 的值；如果 offset 参数省略，则默认为 0。

2．U-Boot 命令使用实例

下面通过一个例子来演示如何使用各种命令烧写内核映象文件、yaffs 映象文件，并启动系统。

（1）制作内核映象文件。

对于本书使用的 Linux 2.6.22.6 版本，编译内核时可以直接生成 U-Boot 格式的映象文件 uImage。

对于不能直接生成 uImage 的内核，制作方法在 U-Boot 根目录下的 README 文件中有说明，假设已经编译好的内核文件为 vmlinux，它是 ELF 格式的。mkimage 是 U-Boot 目录 tools 下的工具，它在编译 U-Boot 时自动生成。执行以下 3 个命令将内核文件 vmlinux 制作为 U-Boot 格式的映象文件 uImage，它们首先将 vmlinux 转换为二进制格式，然后压缩，最后构造头部信息（里面包含有文件名称、大小、类型、CRC 校验码等），如下所示。

```
① arm-linux-objcopy -O binary -R .note -R .comment -S vmlinux linux.bin
② gzip -9 linux.bin
③ mkimage -A arm -O linux -T kernel -C gzip -a 0x30008000 -e 0x30008000 -n
"Linux Kernel Image" -d linux.bin.gz uImage
```

（2）烧写内核映象文件 uImage。

首先将 uImage 放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务。

然后运行如下命令下载文件，擦除、烧写 NAND Flash，如下所示。

```
① tftp 0x30000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
② nand erase 0x0 0x00200000
③ nand write.jffs2 0x30000000 0x0 $(filesize)
```

第 3 条命令之所以使用“ nand write.jffs2 ”而不是“ nand write ”，是因为前者不要求文件的长度是页对齐的（512 字节对齐）。也可以使用“ nand write ”，但是需要将命令中的长度参数改为$(filesize)向上进行 512 取整（比如，513 向上进行 512 取整，结果为 512 × 2=1024）后的值。比如 uImage 的大小为 1540883，向上进行 512 取整后为 1541120（即 0x178400），可以使用命令“ nand write 0x30000000 0x0 0x178400 ”进行烧写。

（3）烧写 yaffs 文件系统映象。

假设 yaffs 文件系统映象的文件名为 yaffs.img，首先将它放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务；然后执行如下命令下载、擦除、烧写，如下所示。

```
① tftp 0x30000000 yaffs.img 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
yaffs.img
② nand erase 0xA00000 0x3600000
③ nand write.yaffs 0x30000000 0xA00000 $(filesize)
```

这时，重启系统，在 U-Boot 倒数 3s 之后，就会自动启动 Linux 系统。

（4）烧写 jffs2 文件系统映象。

假设 jffs2 文件系统映象的文件名为 jffs2.img，首先将它放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务；然后执行如下命令下载、擦除、烧写，如下所示。

```
    ① tftp 0x30000000 jffs2.img 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
jffsz.img
    ② nand erase 0x200000 0x800000
    ③ nand write.jffs2 0x30000000 0x200000 $(filesize)
```

系统启动后，就可以使用“ mount -t jffs2 /dev/mtdblock1 /mnt ”挂接 jffs2 文件系统。

15.2.7　使用 U-Boot 来执行程序

在前面的实例中使用 JTAG 烧写程序到 NAND Flash，烧写过程十分缓慢。如果使用 U-Boot 来烧写 NAND Flash，效率会高很多。烧写二进制文件到 NAND Flash 中所使用的命令与上面烧写内核映象文件 uImage 的过程类似，只是不需要将二进制文件制作成 U-Boot 格式。

另外，可以将程序下载到内存中，然后使用 go 命令执行它。假设有一个程序的二进制可执行文件 test.bin，连接地址为 0x30000000。首先将它放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务；然后将它下载到内存 0x30000000 处，最后使用 go 命令执行它，如下所示。

```
    ① tftp 0x30000000 test.bin 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
test.bin
    ② go 0x30000000
```

第 18 章　Linux 内核调试技术

本章目标

- 掌握几种调试内核的方法：printk、kgdb、分析 Oops、栈回溯
- 使用调试工具：gdb、ddd

18.1　内核打印函数 printk

18.1.1　printk 的使用

1．printk 函数的记录级别

调试内核、驱动的最简单方法，是使用 printk 函数打印信息。printk 函数与用户空间的 printf 函数格式完全相同，它所打印的字符串头部可以加入“ <n> ”样式的字符，其中 n 为 0～7，表示这条信息的记录级别。

在内核代码 include/linux/kernel.h 中，下面几个宏控制了 printk 函数所能输出的信息的记录级别。

```
#define console_loglevel (console_printk[0])
#define default_message_loglevel (console_printk[1])
#define minimum_console_loglevel (console_printk[2])
#define default_console_loglevel (console_printk[3])
```

举例说明这几个宏的含义。

① 对于 printk(“ <n>… ”)，只有 n 小于 console_loglevel 时，这个信息才会被打印。

② 假设 default_message_loglevel 的值等于 4，如果 printk 的参数开头没有“ <n> ”样式的字符，则在 printk 函数中进一步处理前会自动加上“ <4> ”。

③ minimum_console_loglevel 是一个预设值，平时不起作用。通过其他工具来设置 console_loglevel 的值时，这个值不能小于 minimum_console_loglevel。

④ default_console_loglevel 也是一个预设值，平时不起作用。它表示设置 console_loglevel 时的默认值，通过其他工具来设置 console_loglevel 的值时，会用到这个值。

minimum_console_logleve 和 default_console_loglevel 这两个值的作用，可以参考内核源文件 kernel/printk.c 的 do_syslog 函数。

上面代码中，console_printk 是一个数组，它在 kernel/printk.c 中定义：

```
/* printk's without a loglevel use this.. */
#define DEFAULT_MESSAGE_LOGLEVEL 4 /* KERN_WARNING */

/* We show everything that is MORE important than this.. */
#define MINIMUM_CONSOLE_LOGLEVEL 1 /* Minimum loglevel we let people use */
#define DEFAULT_CONSOLE_LOGLEVEL 7 /* anything MORE serious than KERN_DEBUG */
......
int console_printk[4] = {
    DEFAULT_CONSOLE_LOGLEVEL,   /* console_loglevel */
    DEFAULT_MESSAGE_LOGLEVEL,   /* default_message_loglevel */
    MINIMUM_CONSOLE_LOGLEVEL,   /* minimum_console_loglevel */
    DEFAULT_CONSOLE_LOGLEVEL,   /* default_console_loglevel */
};
```

2．在用户空间修改 printk 函数的记录级别

挂接 proc 文件系统后，读取/proc/sys/kernel/printk 文件可以得知 console_loglevel、default_message_loglevel、minimum_console_loglevel 和 default_console_loglevel 这 4 个值。

比如执行以下命令，它的结果“ 7 4 1 7 ”表示这 4 个值。

```
# cat /proc/sys/kernel/printk
7       4       1       7
```

也可以直接修改/proc/sys/kernel/printk 文件来改变这 4 个值，比如：

```
# echo "1 4 1 7" > /proc/sys/kernel/printk
```

这使得 console_loglevel 被改为 1，于是所有的 printk 信息都不会被打印。

3．printk 函数记录级别的名称及使用

在内核代码 include/linux/kernel.h 中有如下代码，它们表示 0～7 这 8 个记录级别的名称。

```
#define KERN_EMERG      "<0>"   /* system is unusable           */
#define KERN_ALERT      "<1>"   /* action must be taken immediately    */
#define KERN_CRIT       "<2>"   /* critical conditions          */
#define KERN_ERR        "<3>"   /* error conditions         */
#define KERN_WARNING    "<4>"   /* warning conditions           */
#define KERN_NOTICE     "<5>"   /* normal but significant condition    */
#define KERN_INFO       "<6>"   /* informational            */
```

```
#define KERN_DEBUG      "<7>"   /* debug-level messages        */
```

在使用 printk 函数时，可以这样使用记录级别；

```
printk(KERN_WARNING"there is a warning here!\n")
```

18.1.2　串口控制台

1．串口与 printk 函数的关系

在嵌入式 Linux 开发中，printk 信息常常从串口输出，这时串口被称为串口控制台。从内核 kernel/printk.c 的 printk 函数开始，往下查看它的调用关系，可以知道 printk 函数是如何与具体设备的输出函数挂钩的。

printk 函数调用的子函数的主要脉落如下：

```
printk ->
    vprintk ->
        emit_log_char  // 把要打印的数据写入一个全局缓冲区(名为 log_buf)中
        release_console_sem ->
            call_console_drivers -> 
                _call_console_drivers ->
                    __call_console_drivers ->
                        con->write  // con 是 console_drivers 链表的表项，调用具体的输出函数
```

对于可以作为控制台的设备，在初始化时会通过 register_console 函数向 console_drivers 链表注册一个 console 结构，里面有 write 函数指针。

以 drivers/serial/s3c2410.c 文件中的串口初始化函数 s3c24xx_serial_initconsole 为例，它的部分代码如下：

```
1892 static int s3c24xx_serial_initconsole(void)
1893 {
…
1927     register_console(&s3c24xx_serial_console);
1928     return 0;
1928 }
```

第 1927 行的 s3c24xx_serial_console 就是 console 结构，它在相同的文件中定义，部分内容如下：

```
1882 static struct console s3c24xx_serial_console =
1883 {
1884     .name       = S3C24XX_SERIAL_NAME,       // 这个宏被定义为“SAC”
1885     .device     = uart_console_device,     // init 进行、用户程序打开/dev/console 时用到
1886     .flags      = CON_PRINTBUFFER,          // 打印先前在 log_buf 中保存的信息
```

```
1887     .index      = -1,                          // 表示使用哪个串口由命令行参数决定
1888     .write      = s3c24xx_serial_console_write,  // 串口控制台的输出函数
1889     .setup      = s3c24xx_serial_console_setup   // 串口控制台的设置函数
1890 };
```

第 1886 行的 CON_PRINTBUFFER 表示注册这个结构后，要把 log_buf 缓冲区中的所有信息打印出来。这表明，在实际的硬件被初始化之前，就可以使用 printk 函数，只不过这时的打印信息是保存在 log_buf 缓冲区中，还没有真正输出。

第 1888 行的 s3c24xx_serial_console_write 是串口输出函数，它会调用 s3c24xx_serial_console_putchar 函数将要打印的字符一个个地从串口输出。

s3c24xx_serial_console_putchar 是最底层的函数，代码如下：

```
static void
s3c24xx_serial_console_putchar(struct uart_port *port, int ch)
{
    unsigned int ufcon = rd_regl(cons_uart, S3C2410_UFCON);
    while (!s3c24xx_serial_console_txrdy(port, ufcon))
        barrier();
    wr_regb(cons_uart, S3C2410_UTXH, ch);
}
```

从上面的代码可以知道，从串口中输出 printk 打印信息时，是一个字符一个字符地发送、等待发送完成、发送、接着等待，……，效率很低。调试完毕后，通常要将 printk 信息去掉。

2．设置内核命令行参数使用串口控制台

第 15 章中使用 U-Boot 时，设置了命令行参数“ console=ttySAC0 ”，它使得 printk 的信息从串口 0 中输出。

内核是怎样根据这些命令行参数确定 printk 的输出设备呢？在 kernel/printk.c 中有如下代码：

```
__setup("console=", console_setup);
```

内核开始执行时，发现形如“ console=… ”的命令行参数时，就会调用 console_setup 函数进行解析。对于命令行参数“ console=ttySAC0 ”，它会解析出：设备名（name）为 ttySAC，索引（index）为 0，这些信息被保存在类型为 console_cmdline、名称为 console_cmdline 的全局数组中（数据光类型、数组名相同，请勿混淆）。

在后面使用“ register_console (&s3c24xx_serial_console) ”注册控制台（参考前面的代码 drivers/serial/s3c2410.c 中第 1927 行）时，会将 s3c24xx_serial_console 结构与 console_cmdline 数组中的设备进行比较，发现名字、索引相同。

① s3c24xx_serial_console 结构中名字（name）为 S3C24XX_SERIAL_NAME，即“ ttySAC ”，而根据“ console=ttySAC0 ”解析出来的名字也是“ ttySAC ”。

② s3c24xx_serial_console 结构中索引（index）为−1，表示使用命令行中解析出来的索

引0，表示串口0。

综上所述，命令行参数“ console=ttySAC0 ”决定printk信息将通过s3c24xx_serial_console结构中的相关函数，从串口0输出。

最后，既然printk输出的信息是先保存在缓冲区log_buf中的，那么也可以读取log_buf以获得这些信息：系统启动后，想查看printk信息时，直接运行dmesg命令即可。通过其他非串口的手段（比如ssh、telnet）登录系统时，也可以使用dmesg命令查看printk信息。

18.2 内核源码级别的调试方法

18.2.1 内核调试工具KGDB的作用与原理

1．KGDB介绍

KGDB是一个源码级别的Linux内核调试器。使用KGDB调试内核时，需要结合GDB一起使用。它们使得调试内核就像调试应用程序一样，可以在内核代码中设置断点、一步一步地执行指令、观察变量的值。

使用KGDB时，需要两台机器，即主机和目标机，两者通过串口线相连。要调试的内核需要增加KGDB功能，它在目标机上运行，GDB在主机上运行。串口线被GDB用来与内核通信。

KGDB是一个内核补丁，目前支持i386、x86_64、ppc、s390、ARM等架构。将内核打上KGDB补丁后才能够使用GDB来调试。

2．KGDB的原理

安装KGDB调试环境需要为Linux内核加上kgdb补丁，补丁实现GDB远程调试所需要的功能，包括命令处理、陷阱处理及串口通信3个主要的部分。KGDB补丁的主要作用是在Linux内核中添加了一个调试Stub。调试Stub是Linux内核中的一小段代码，是运行GDB的开发机和所调试内核之间的一个媒介。GDB和调试stub之间通过GDB串行协议进行通信。GDB串行协议是一种基于消息的ASCII码协议，包含了各种调试命令。当设置断点时，KGDB将断点的指令替换为一条trap指令，当执行到断点时控制权就转移到调试stub中去。此时，调试stub的任务就是使用远程串行通信协议将当前环境传送给GDB，然后从GDB处接收命令。GDB命令告诉stub下一步该做什么，当stub收到继续执行的命令时，将恢复程序的运行环境，把对CPU的控制权重新交还给内核。

KGDB补丁给内核添加以下3个部件。

（1）GDB stub。

GDB stub被称为调试插桩（简称为stub），是KGDB调试器的核心。它是Linux内核中的一小段代码，用来处理主机上GDB发来的各种请求；并且在内核处于被调试状态时，控制目标机板上的处理器。

（2）修改异常处理函数。

当这个异常发生时，内核将控制权交给KGDB调试器，程序进入KGDB提供的异常处

理函数中。在里面，可以分析程序的各种情况。

（3）串口通信。

GDB 和 stub 之间通过 GDB 串行协议进行通信。它是一种基于消息的 ASCII 码协议，包含了各种调试命令。

除串口外，也可以使用网卡进行通信。

以设置内核断点为例说明 KGDB 与 GDB 之间的工作过程。设置断点时，KGDB 修改内核代码，将断点位置的指令替换成一条异常指令（在 ARM 中这是一条未定义的指令）。当执行到断点时发生异常，控制权转移到 stub 的异常处理函数中。此时，stub 的任务就是使用 GDB 串行通信协议将当前环境传送给 GDB，然后从 GDB 处接收命令，GDB 命令告诉 stub 下一步该做什么。当 stub 收到继续执行的命令时，将恢复原来替换的指令、恢复程序的运行环境，把对 CPU 的控制权重新交还给内核。

18.2.2 给内核添加 KGDB 功能支持 S3C2410/S3C2440

如果读者使用了前面第 16 章提到的内核补丁文件 linux2.6.22.6_100ask24x0.patch，则本节中对代码的修改可以忽略，只需要关注对内核的配置（补丁文件生成的 config_ok 文件对 KGDB 也已经配置好了）。

如果/work/system/linux-2.6.22.6 曾经应用了补丁文件 linux2.6.22.6_100ask24x0.patch，那么它（基于随书光盘容量的考虑，Xorg_git_20071119.tar.bz2 中删除了 doc 目录，这无关紧要）。

对于本书使用的 Linux 2.6.22 内核，有对应的 KGDB 补丁。但是对于 S3C2410、S3C2440，还需要自己编写串口初始化函数、发送、接收字符函数，以供 stub 调用。

1．给内核添加 KGDB 补丁

要使用的 KGDB 补丁的分支版本为 linux2_6_22_uprev，有 3 种获取方法。

① 从 web 网页下载，地址如下。

```
http://kgdb.cvs.sourceforge.net/kgdb/kgdb-2/?pathrev=linux2_6_22_uprev
```

② 使用 cvs 工具下载，执行以下命令即可。

```
$ cd /work/debug
$ cvs -z3 -d:pserver:anonymous@kgdb.cvs.sourceforge.net:/cvsroot/kgdb co -P
-r linux2_6_22_uprev kgdb-2
```

③ 也可以使用已经下载好了的，即/work/debug/kgdb-2_linux2_6_22_uprev.tar.bz2。

在下载或解压后得到 kgdb-2 目录里，除了各种补丁文件外，还有一个名为 series 的文件，它表示这些补丁文件使用的顺序。可以参考 series 文件一个个地打补丁，也可以使用“ quilt push -a ”命令一次全部打上：先把 kgdb-2 目录复制到内核目录下，并改名为 patches；然后在内核目录下执行“ quilt push -a ”命令，命令如下：

```
$ cd /work/system/linux-2.6.22.6
$ cp -rf /work/debug/kgdb-2 patches
$ quilt push -a
```

2．修改补丁本身带入的错误

修改include/asm-arm/system.h第380行，这是一个笔误（“－”号表示原来的代码，“＋”号表示新代码）：

```
-               pref = *p;
+               prev = *p;
```

3．编写S3C2410/S3C2440的KGDB串口函数

目前的KGDB补丁不支持S3C2410/S3C2440的串口，需要自己编写相关函数。可以参考arch/arm/mach-pxa/kgdb-serial.c，在arch/arm/mach-s3c2410/目录下也建立一个kgdb-serial.c文件。

KGDB只需要3个串口函数：初始化函数、发送单字符函数、接收单字符函数。然后将它们填入同一文件中，一个名为kgdb_io_ops的struct kgdb_io结构中。

下面分段介绍这3个函数及文件中其他内容，完整的代码请参考linux-2.6.22.6_ok.tar.bz2。

① 串口初始化函数。

```
53 static int kgdb_serial_init(void)
54 {
55     struct clk *clock_p;
56     u32 pclk;
57     u32 ubrdiv;
58     u32 val;
59     u32 index = CONFIG_KGDB_PORT_NUM;
60
61     clock_p = clk_get(NULL, "pclk");
62     pclk = clk_get_rate(clock_p);
63
64     ubrdiv = (pclk / (UART_BAUDRATE * 16)) - 1;
65
66     /* 设置GPIO用作串口，并且禁止内部上拉
67      * GPH2、GPH3用作TXD0、RXD0
68      * GPH4、GPH5用作TXD1、RXD1
69      * GPH6、GPH7用作TXD2、RXD2
70      */
71     if (index < MAX_PORT)
72     {
73         index = 2 + index * 2;
74
```

```
75          val = inl(S3C2410_GPHUP) | (0x3 << index);
76          outl(val, S3C2410_GPHUP);
77
78          index *= 2;
79          val = (inl(S3C2410_GPHCON) & ~(~(0xF << index))) | \
80                (0xA << index);
81          outl(val, S3C2410_GPHCON);
82      }
83      else
84      {
85          return -1;
86      }
87
88      // 8N1(8 个数据位，无校验位，1 个停止位)
89      wr_regl(CONFIG_KGDB_PORT_NUM, S3C2410_ULCON, 0x03);
90
91      // 中断/查询方式，UART 时钟源为 PCLK
92      wr_regl(CONFIG_KGDB_PORT_NUM, S3C2410_UCON, 0x3c5);
93
94      // 使用 FIFO
95      wr_regl(CONFIG_KGDB_PORT_NUM, S3C2410_UFCON, 0x51);
96
97      // 不使用流控
98      wr_regl(CONFIG_KGDB_PORT_NUM, S3C2410_UMCON, 0x00);
99
100      // 设置波特率
101      wr_regl(CONFIG_KGDB_PORT_NUM, S3C2410_UBRDIV, ubrdiv);
102
103      return 0;
104 }
105
```

要使用串口，需要选择相关的 GPIO 引脚用作串口，并且设置串口的数据格式、时钟源、波特率等。

② 发送单字符函数。

```
106 static void kgdb_serial_putchar(u8 c)
107 {
108     /* 等待，直到发送缓冲区中的数据已经全部发送出去 */
109     while (!(rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_UTRSTAT) & S3C2410_UTRSTAT_TXE));
110
```

```
    111     /* 向UTXH寄存器中写入数据，UART即自动将它发送出去 */
    112     wr_regb(CONFIG_KGDB_PORT_NUM, S3C2410_UTXH, c);
    113 }
    114
```

③ 接收单字符函数。

```
    115 static int kgdb_serial_getchar(void)
    116 {
    117     /* 等待，直到接收缓冲区中有数据 */
    118     while (!(rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_UTRSTAT) & S3C2410_
UTRSTAT_RXDR));
    119
    120     /* 直接读取URXH寄存器，即可获得接收到的数据 */
    121     return rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_URXH);
    122 }
    123
```

④ 使用这些函数构建 kgdb_io_ops 结构。

```
    124 struct kgdb_io kgdb_io_ops = {
    125     .init = kgdb_serial_init,
    126     .read_char = kgdb_serial_getchar,
    127     .write_char = kgdb_serial_putchar,
    128 };
```

kgdb_io_ops 结构将在 kernel/kgdb.c 中被用到，这个结构封装了开发板相关的串口操作函数。其他的 KGDB 代码都是具体开发板无关的。

4．修改内核配置文件、Makefile

① 修改 arch/arm/mach-s3c2410/Makefile，将新增的 kgdb-serial.c 文件编译进内核。

+ obj-$(CONFIG_KGDB_S3C24XX_SERIAL) += kgdb-serial.o

② 上面的 CONFIG_KGDB_S3C24XX_SERIAL 是新加的配置项，要修改配置文件 lib/Kconfig.kgdb 来支持它。

修改了 4 个地方，下面的修改内容仿照补丁文件的格式，首字母为“ - ”的行表示是老文件中的代码，首字母为“ + ”的行表示是新文件中的代码。

- 在“ Method for KGDB communication ”下增加一个选择项。

```
    choice
            prompt "Method for KGDB communication"
            depends on KGDB
    +         default KGDB_S3C24XX_SERIAL if ARCH_S3C2410
```

- 用来配置 KGDB_S3C24XX_SERIAL 选项。

```
+ config KGDB_S3C24XX_SERIAL
+         bool "KGDB: On the S3C24xx serial port"
+         depends on ARCH_S3C2410
+         help
+                 Enables the KGDB serial driver for S3C24xx
```

- 配置 KGDB_S3C24XX_SERIAL 后，也可以设置 KGDB 所用串口的波特率。

```
config KGDB_BAUDRATE
        int "Debug serial port baud rate"
        depends on (KGDB_8250 && KGDB_SIMPLE_SERIAL) || \
                KGDB_MPSC || KGDB_CPM_UART || \
-               KGDB_TXX9 || KGDB_PXA_SERIAL || KGDB_AMBA_PL011
+                   KGDB_TXX9 || KGDB_PXA_SERIAL || KGDB_AMBA_PL011 ||
KGDB_S3C24XX_SERIAL
```

- 配置 KGDB_S3C24XX_SERIAL 后，也可以设置 KGDB 使用哪个串口，默认使用第 1 个。

```
config KGDB_PORT_NUM
     int "Serial port number for KGDB"
     range 0 1 if KGDB_MPSC
     range 0 3
-    depends on (KGDB_8250 && KGDB_SIMPLE_SERIAL) || KGDB_MPSC || KGDB_TXX9
-    default "1"
+    depends on (KGDB_8250 && KGDB_SIMPLE_SERIAL) || KGDB_MPSC || KGDB_TXX9 ||
KGDB_S3C24XX_SERIAL
+    default "0"
```

5．配置内核，使能 KGDB 功能

执行“ make menuconfig ”来配置内核，如下配置以使能 KGDB 功能。

```
Kernel hacking  --->
    [*] KGDB: kernel debugging with remote gdb     // 表示使能 KGDB 功能
    [*] KGDB: Console messages through gdb         // 表示控制台信息(printk)
会发送到 GDB
        Method for KGDB communication (KGDB: On the S3C24xx serial port)  --->
//S3C24xx 串口
       < > KGDB: On ethernet (NEW)
       (115200) Debug serial port baud rate (NEW)     // 波特率为 115200
```

(0) Serial port number for KGDB (NEW)　　// 使用第 1 个 S3C24xx 串口

然后执行“ make uImage ”即可生成内核 vmlinux、arch/arm/boot/uImage。

18.2.3　结合可视化图形前端 DDD 和 GDB 来调试内核

1．DDD 介绍及安装

DDD 是“ Data Display Debugger ”的简称，是命令行调试程序，是 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 等的可视化图形前端，这意味着可以不用记忆、输入各种调试命令，可以使用各种按钮进行调试。它特有的图形数据显示功能（Graphical Data Display）可以把数据结构按照图形的方式显示出来。

DDD 的功能非常强大，可以调试用 C/C++、Ada、Fortran、Pascal、Modula-2 和 Modula-3 编写的程序；可以以超文本方式浏览源代码；能够进行断点设置、回溯调试和历史纪录编辑；具有程序在终端运行的仿真窗口，并在远程主机上进行调试的能力。图形数据显示功能（Graphical Data Display）是创建该调试器的初衷之一，能够显示各种数据结构之间的关系，并将数据结构以图形化形式显示；具有 GDB/DBX/XDB 的命令行界面，包括完全的文本编辑、历史纪录、搜寻引擎。

通过 DDD 调用 GDB 来调试内核，可以在图形界面上完成调试工作。

可以在 Ubuntu 7.10 中通过网络安装 DDD，命令如下：

```
$ sudo apt-get install ddd
```

2．GDB 介绍及安装

通过 GDB 这类调试器，程序员可以知道一个程序执行时内部动作过程，可以知道一个程序崩溃时发生了什么事。

GDB 可以完成以下 4 个主要功能，这可以帮助程序员捕捉到程序的错误。

① 启动程序，并指定各类能够影响程序运行的参数。

② 使程序在指定条件下停止运行。

③ 当程序停止时，观察各种状态，检查发生了什么事情。

④ 修改程序的执行参数，比如修改某个变量，这使得在查错时可以试验各种参数。

GDB 支持多种编程语言，可以调试用 C/C++、Modula-2 和 Fortran 等语言编写的程序。GDB 是基于命令行的，GDB 启动后，在它的控制界面使用各种命令进行操作。

Ubuntu 7.10 自带的 GDB 工具是基于 x86 系列的，需要自己下载源码为 ARM 平台编译一个 GDB 工具，为便于区分，将它命名为 arm-linux-gdb。

从网站 http://www.gnu.org/software/gdb/下载 gdb-6.7.tar.bz2，或者使用/work/debug/ gdb-6.7.tar.bz2。执行以下命令编译、安装 arm-linux-gdb。

```
$ tar xjf gdb-6.7.tar.bz2
$ cd gdb-6.7/
$ ./configure --target=arm-linux
$ make
```

```
$ sudo make install
```

3．使用 arm-linux-gdb 调试内核（命令行方式）

先启动支持 KGDB 的内核，然后在主机上启动 arm-linux-gdb。

（1）启动内核。

要使用 KGDB 功能，需要增加两个命令参数：console=kgdb 和 kgdbwait。前者表示内核打印信息会被发送给 GDB，即通过上面增加的 kgdb-serial.c 中的相关函数进行发送；后者表示内核启动时先停住，等待 GDB 的连接。

假设将上面编译好的内核 uImage 放在/work/nfs_root 目录下，则可以在 U-Boot 上使用以下命令设置命令行参数、启动内核。

```
100ask> set bootargs noinitrd root=/dev/mtdblock 2 console=kgdb kgdbwait
100ask> nfs 0x31000000 192.168.1.57:/work/nfs_root/uImage
```

```
100ask> bootm 0x31000000
```

这时可以看到以下启动信息：

```
Starting kernel ...

Uncompressing
Linux.....................................................................
..................................... done, booting the kernel.
```

内核在等待主机 arm-linux-gdb 的连接。

（2）启动 arm-linux-gdb。

> 注意 由于 arm-linux-gdb 要用到串口，所以如果是在 vmware 上运行 Linux，还要设置 vmware 的特性，增加串口（物理串口）。

启动 arm-linux-gdb 之前，先退出刚才操作 U-Boot 所用的串口工具，因为 arm-linux-gdb 也要使用这个串口。

然后在主机上进入内核目录，启动 arm-linux-gdb，可以执行以下命令：

```
$ cd /work/system/linux-2.6.22.6
```

```
$ sudo arm-linux-gdb ./vmlinux
```

这时会看到 arm-linux-gdb 的启动信息，进入控制界面：

```
GNU gdb 6.7
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-linux"...
(gdb)
```

最后，执行两个命令设置口、连接目标板。

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
```

这时可以看到如下信息，表明已经连接上了目标板，目标板在 kernel/kgdb.c 的 1775 行暂停运行。

```
Remote debugging using /dev/ttyS0
0xc0067a28 in breakpoint () at kernel/kgdb.c:1775
1775            atomic_set(&kgdb_setting_breakpoint, 1);
(gdb)
```

现在就可以使用如种 GDB 的命令控制内核的执行、进行调试了，读者可以自行参考 GDB 的手册。比如输入 n 命令执行下一条指令，输入 c 命令全速运行，输出 q 命令退出。GDB 命令的使用方法请参考 GDB 手册。

为了避免每次启动 arm-linux-gdb 时手工设置串口、连接目标板，可以在内核目录下建立一个名为“.gdbinit”文件，内容如下：

```
set remotebaud 115200
target remote /dev/ttyS0
```

4．通过 DDD 调用 arm-linux-gdb 来调试内核（图形界面）

arm-linux-gdb 是通过 DDD 来启动的，DDD 封装了对 arm-linux-gdb 的操作，提供一个图形化的操作界面，操作步骤如下。

（1）启动内核。

（2）启动 DDD。

DDD 要在桌面系统中启动，在远程登录工具 ssh 等的命令行中无法启动 DDD。

首先，退出操作 U-Boot 的串口工具。

然后，确保内核目录下有.gdbinit 文件。

最后，在桌面系统的控制台里，进入内核目录，启动 DDD。执行以下命令即可。

```
$ cd /work/system/linux-2.6.22.6
$ sudo ddd --debugger arm-linux-gdb ./vmlinux
```

这时，可以看到如图 18.1 所示的启动界面，在里面可以很方便地使用各类按钮进行设置断点、单步执行等操作。

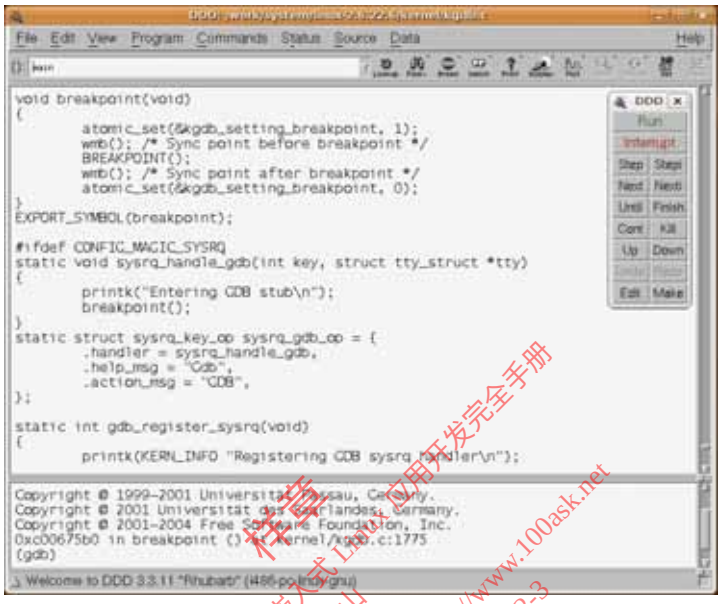

图 18.1 DDD 调用 arm-linux-gdb 来调试内核的启动界面

18.3 Oops 信息及栈回溯

18.3.1 Oops 信息来源及格式

Oops 这个单词含义为“惊讶”，当内核出错时（比如访问非法地址）打印出来的信息被称为 Oops 信息。

Oops 信息包含以下几部分内容。

① 一段文本描述信息。

比如类似“Unable to handle kernel NULL pointer dereference at virtual address 00000000”的信息，它说明了发生的是哪类错误。

② Oops 信息的序号。

比如是第 1 次、第 2 次等。这些信息与下面类似，中括号内的数据表示序号。

```
Internal error: Oops: 805 [#1]
```

③ 内核中加载的模块名称，也可能没有，以下面字样开头。

```
Modules linked in:
```

④ 发生错误的 CPU 的序号，对于单处理器的系统，序号为 0，比如：

```
CPU: 0    Not tainted  (2.6.22.6 #36)
```

⑤ 发生错误时 CPU 的各个寄存器值。

⑥ 当前进程的名字及进程 ID，比如：

```
Process swapper (pid: 1, stack limit = 0xc0480258)
```

这并不是说发生错误的是这个进程，而是表示发生错误时，当前进程是它。错误可能发生在内核代码、驱动程序，也可能就是这个进程的错误。

⑦ 栈信息。

⑧ 栈回溯信息，可以从中看出函数调用关系，形式如下：

```
Backtrace:
[<c001a6f4>] (s3c2410fb_probe+0x0/0x560) from [<c01bf4e8>] (platform_drv_probe+0x20/0x24)
...
```

⑨ 出错指令附近的指令的机器码，比如（出错指令在小括号里）：

```
Code: e24cb004 e24dd010 e59f34e0 e3a07000 (e5873000)
```

18.3.2　配置内核使 Oops 信息的栈回溯信息更直观

Linux 2.6.22 自身具备的调试功能，可以使得打印出的 Oops 信息更直观。通过 Oops 信息中 PC 寄存器的值可以知道出错指令的地址，通过栈回溯信息可以知道出错时的函数调用关系，根据这两点可以很快定位错误。

要让内核出错时能够打印栈回溯信息，编译内核时要增加“ -fno-omit-frame-pointer ”选项，这可以通过配置 CONFIG_FRAME_POINTER 来实现。查看内核目录下的配置文件.config，确保 CONFIG_FRAME_POINTER 已经被定义，如果没有，执行“ make menuconfig ”命令重新配置内核。CONFIG_FRAME_POINTER 有可能被其他配置项自动选上。

18.3.3　使用 Oops 信息调试内核的实例

1．获得 Oops 信息

本小节故意修改 LCD 驱动程序 drivers/video/s3c2410fb.c，加入错误代码：在 s3c2410fb_probe 函数的开头增加下面两条代码：

```
    int *ptest = NULL;
     *ptest = 0x1234;
```

重新编译内核，启动后会出错并打印出如下 Oops 信息：

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = c0004000
[00000000] *pgd=00000000
Internal error: Oops: 805 [#1]
Modules linked in:
CPU: 0    Not tainted  (2.6.22.6 #36)
PC is at s3c2410fb_probe+0x18/0x560
LR is at platform_drv_probe+0x20/0x24
pc : [<c001a70c>]    lr : [<c01bf4e8>]    psr: a0000013
sp : c0481e64  ip : c0481ea0  fp : c0481e9c
```

```
r10: 00000000  r9 : c0024864  r8 : c03c420c
r7 : 00000000  r6 : c0389a3c  r5 : 00000000  r4 : c036256c
r3 : 00001234  r2 : 00000001  r1 : c04c0fc4  r0 : c0362564
Flags: NzCv  IRQs on  FIQs on  Mode SVC_32  Segment kernel
Control: c000717f  Table: 30004000  DAC: 00000017
Process swapper (pid: 1, stack limit = 0xc0480258)
Stack: (0xc0481e64 to 0xc0482000)
1e60:c02b1f70 00000020 c03625d4 c036256c c036256c 00000000 c0389a3c
1e80: c0389a3c c03c420c c0024864 00000000 c0481eac c0481ea0 c01bf4e8 c001a704
1ea0: c0481ed0 c0481eb0 c01bd5a8 c01bf4d8 c0362644 c036256c c01bd708 c0389a3c
1ec0: 00000000 c0481ee8 c0481ed4 c01bd788 c01bd4d0 00000000 c0481eec c0481f14
1ee0: c0481eec c01bc5a8 c01bd718 c038dac8 c038dac8 c03625b4 00000000 c0389a3c
1f00: c0389a44 c038d9dc c0481f24 c0481f18 c01bd808 c01bc568 c0481f4c c0481f28
1f20: c01bcd78 c01bd7f8 c0389a3c 00000000 00000000 c0480000 c0023ac8 00000000
1f40: c0481f60 c0481f50 c01bdc84 c01bcd0c 00000000 c0481f70 c0481f64 c01bf5fc
1f60: c01bdc14 c0481f80 c0481f74 c019479c c01bf5a0 c0481ff4 c0481f84 c0008c14
1f80: c0194798 e3c338ff e0222423 00000000 00000001 e2844004 00000000 00000000
1fa0: 00000000 c0481fb0 c002bf24 c0041328 00000000 00000000 c0008b40 c00476ec
1fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1fe0: 00000000 00000000 00000000 c0481ff8 c00476ec c0008b50 c03cdf50 c0344178
Backtrace:
[<c001a6f4>] (s3c2410fb_probe+0x0/0x560) from [<c01bf4e8>] (platform_drv_
probe+0x20/0x24)
[<c01bf4c8>] (platform_drv_probe+0x0/0x24) from [<c01bd5a8>] (driver_probe_
device+0xe8/0x18c)
[<c01bd4c0>] (driver_probe_device+0x0/0x18c) from [<c01bd788>] (__driver_
attach+0x80/0xe0)
 r8:00000000 r7:c0389a3c r6:c01bd708 r5:c036256c r4:c0362644
[<c01bd708>] (_ _driver_attach+0x0/0xe0) from [<c01bc5a8>] (bus_for_each_
dev+0x50/0x84)
 r5:c0481eec r4:00000000
[<c01bc558>] (bus_for_each_dev+0x0/0x84) from [<c01bd808>] (driver_attach+
0x20/0x28)
 r7:c038d9dc r6:c0389a44 r5:c0389a3c r4:00000000
[<c01bd7e8>] (driver_attach+0x0/0x28) from [<c01bcd78>] (bus_add_driver+
0x7c/0x1b4)
[<c01bccfc>] (bus_add_driver+0x0/0x1b4) from [<c01bdc84>] (driver_register+
0x80/0x88)
[<c01bdc04>] (driver_register+0x0/0x88) from [<c01bf5fc>] (platform_driver_
```

```
register+0x6c/0x88)
     r4:00000000
    [<c01bf590>] (platform_driver_register+0x0/0x88) from [<c019479c>] (s3c2410fb_
init+0x14/0x1c)
    [<c0194788>] (s3c2410fb_init+0x0/0x1c) from [<c0008c14>] (kernel_init+0xd4/
0x28c)
    [<c0008b40>] (kernel_init+0x0/0x28c) from [<c00476ec>] (do_exit+0x0/0x760)
    Code: e24cb004 e24dd010 e59f34e0 e3a07000 (e5873000)
    Kernel panic - not syncing: Attempted to kill init!
```

2．分析Oops信息

（1）明确出错原因。

由出错信息“Unable to handle kernel NULL pointer dereference at virtual address 00000000”可知内核是因为非法地址访问出错，使用了空指针。

（2）根据栈回溯信息找出函数调用关系。

内核崩溃时，可以从pc寄存器得知崩溃发生时的函数、出错指令。但是很多情况下，错误有可能是它的调用者引入的，所以找出函数的调用关系也很重要。

部分栈回溯信息如下：

```
    [<c001a6f4>] (s3c2410fb_probe+0x0/0x560) from [<c01bf4e8>] (platform_drv_
probe+0x20/0x24)
```

这行信息分为两部分，表示后面的platform_drv_probe函数调用了前面的s3c2410fb_probe函数。

前半部含义为：“c001a6f4”是s3c2410fb_probe函数首地址偏移0的地址，这个函数大小为0x560。

后半部含义为：“c01bf4e8”是platform_drv_probe函数首地址偏移0x20的地址，这个函数大小为0x24。

另外，后半部的“[<c01bf4e8>]”表示s3c2410fb_probe执行后的返回地址。

对于类似下面的栈回溯信息，其中是r8～r4表示driver_probe_device函数刚被调用时这些寄存器的值。

```
    [<c01bd4c0>] (driver_probe_device+0x0/0x18c) from [<c01bd788>] (__driver_
attach+0x80/0xe0)
     r8:00000000 r7:c0389a3c r6:c01bd708 r5:c036256c r4:c0362644
```

从上面的栈回溯信息可以知道内核出错时的函数调用关系如下，最后在s3c2410fb_probe函数内部崩溃。

```
    do_exit ->
        kernel_init ->
            s3c2410fb_init ->
```

```
                platform_driver_register ->
                    driver_register ->
                        bus_add_driver ->
                            driver_attach ->
                                bus_for_each_dev ->
                                    __driver_attach ->
                                      driver_probe_device ->
                                          platform_drv_probe ->
                                                                s3c2410fb_probe
```

（3）根据 pc 寄存器的值确定出错位置。

上述 Oops 信息中出错时的寄存器值如下：

```
PC is at s3c2410fb_probe+0x18/0x560
LR is at platform_drv_probe+0x20/0x24
pc : [<c001a70c>]    lr : [<c01bf4e8>]    psr: a0000013
...
```

“ PC is at s3c2410fb_probe+0x18/0x560 ”表示出错指令为 s3c2410fb_probe 函数中偏移为 0x18 的指令。

“ pc : [<c001a70c>] ”表示出错指令的地址为 c001a70c(十六进制)。

（4）结合内核源代码和反汇编代码定位问题。

先生成内核的反汇编代码 vmlinux.dis，执行以下命令：

```
$ cd /work/system/linux-2.6.22.6
```

$ arm-linux-objdump -D vmlinux > vmlinux.dis

出错地址 c001a70c 附近的部分汇编代码如下：

```
c001a6f4 <s3c2410fb_probe>:
c001a6f4:   e1a0c00d    mov ip, sp
c001a6f8:   e92ddff0    stmdb   sp!, {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
c001a6fc:   e24cb004    sub fp, ip, #4  ; 0x4
c001a700:   e24dd010    sub sp, sp, #16 ; 0x10
c001a704:   e59f34e0    ldr r3, [pc, #1248] ; c001abec <.init+0x1284c>
c001a708:   e3a07000    mov r7, #0  ; 0x0
c001a70c:   e5873000    str r3, [r7]     <===========出错指令
c001a710:   e59030fc    ldr   r3, [r0, #252]
```

出错指令为“ str r3, [r7] ”，它把 r3 寄存器的值放到内存中，内存地址为 r7 寄存器的值。根据 Oops 信息中的寄存器值可知：r3 为 0x00001234，r7 为 0。0 地址不可访问，所以出错。

s3c2410fb_probe 函数的部分 C 代码如下：

```
static int __init s3c2410fb_probe(struct platform_device *pdev)
{
    struct s3c2410fb_info *info;
    struct fb_info    *fbinfo;
    struct s3c2410fb_hw *mregs;
    int ret;
    int irq;
    int i;
    u32 lcdcon1;

    int *ptest = NULL;
    *ptest = 0x1234;

    mach_info = pdev->dev.platform_data;
```

结合反汇编代码，很容易知道是“*ptest = 0x1234;”导致错误，其中的ptest为空。

对于大多数情况，从反汇编代码定位到C代码并不会如此容易，这需要较强的阅读汇编程序的能力。通过栈回溯信息知道函数的调用关系，这已经可以帮助定位很多问题了。

18.3.4 使用Oops的栈信息手工进行栈回溯

前面说过，从Oops信息的pc寄存器值可知得知崩溃发生时的函数、出错指令。但是错误有可能是它的调用者引入的，所以还要找出函数的调用关系。

由于内核配置了CONFIG_FRAME_POINTER，当出现Oops信息时，会打印栈回溯信息。如果内核没有配置CONFIG_FRAME_POINTER，这时可以自己分析栈信息，找到函数的调用关系。

1．栈的作用

一个程序包含代码段、数据段、BSS段、堆、栈；其中数据段用来中存储初始值不为0的全局数据，BSS段用来存储初始值为0的全局数据，堆用于动态内存分配，栈用于实现函数调用、存储局部变量。

被调用函数在执行之前，它会将一些寄存器的值保存在栈中，其中包括返回地址寄存器lr。如果知道了所保存的lr寄存的值，那么就可以知道它的调用者是谁。在栈信息中，一个函数一个函数地往上找出所有保存的lr值，就可以知道各个调用函数，这就是栈回溯的原理。

2．栈回溯实例分析

仍以前面的LCD驱动程序为例，使用上面的Oops信息的栈信息进行分析，栈信息如下：

```
Stack: (0xc0481e64 to 0xc0482000)
1e60:c02b1f70 00000020 c03625d4 c036256c c036256c 00000000 c0389a3c
1e80: c0389a3c c03c420c c0024864 00000000 c0481eac c0481ea0 c01bf4e8 c001a704
1ea0: c0481ed0 c0481eb0 c01bd5a8 c01bf4d8 c0362644 c036256c c01bd708 c0389a3c
1ec0: 00000000 c0481ee8 c0481ed4 c01bd788 c01bd4d0 00000000 c0481eec c0481f14
```

```
1ee0: c0481eec c01bc5a8 c01bd718 c038dac8 c038dac8 c03625b4 00000000 c0389a3c
…
```

① 根据 pc 寄存器值找到第一个函数，确定它的栈大小，确定调用函数。

从 Oops 信息可知 pc 值为 c001a70c，使用它在内核反汇编程序 vmlinux.dis 中可以知道它位于 s3c2410fb_probe 函数内。

根据这个函数开始部分的汇编代码可以知道栈的大小、lr 返回值在栈中保存的位置，代码如下：

```
c001a6f4 <s3c2410fb_probe>:
c001a6f4:   e1a0c00d    mov ip, sp
c001a6f8:   e92ddff0    stmdb   sp!, {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
c001a6fc:   e24cb004    sub fp, ip, #4  ; 0x4
c001a700:   e24dd010    sub sp, sp, #16 ; 0x10
…
c001a70c:   e5873000    str r3, [r7]     // pc 值 c001a70c 对应的指令
…
```

{r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}这 11 个寄存器都保存在栈中，指令“ sub sp, sp, #16 ”又使得栈向下扩展了 16 字节，所以本函数的栈大小为（11 × 4+16）字节，即 15 个双字。

栈信息开始部分的 15 个数据就是本函数的栈内容，下面列出了它们所保存的寄存器。

```
1e60:         c02b1f70 00000020 c03625d4 c036256c c036256c 00000000 c0389a3c
                                                    r4         r5         r6
1e80: c0389a3c c03c420c c0024864 00000000 c0481eac c0481ea0 c01bf4e8 c001a704
         r7         r8         r9         sl         fp         ip         lr         pc
```

其中 lr 值为 c01bf4e8，表示函数 s3c2410fb_probe 执行完后的返回地址，它是调用函数中的地址。下面使用 lr 值再次重复本步骤的回溯过程。

② 根据 lr 寄存器值找到调用函数，确定它的栈大小，确定上一级调用函数。

根据上步得到的 lr 值（c01bf4e8）在内核反汇编程序 vmlinux.dis 中可以知道它位于 platform_drv_probe 函数内。

根据这个函数开始部分的反汇编代码可以知道栈的大小、lr 返回值在栈中保存的位置。代码如下：

```
c01bf4c8 <platform_drv_probe>:
c01bf4c8:   e1a0c00d    mov ip, sp
c01bf4cc:   e92dd800    stmdb   sp!, {fp, ip, lr, pc}
…
c01bf4e8:   e89da800    ldmia   sp, {fp, sp, pc}    // lr 值(c01bf4e8)对应的指令
```

{fp, ip, lr, pc}这 4 寄存器都保存在栈中，本函数的栈大小为 4 个双字。Oops 栈信息中，前一个函数 s3c2410fb_probe 的栈下面的 4 个数据就是函数 platform_drv_probe 的栈内容，如下所示：

```
1ea0: c0481ed0 c0481eb0 c01bd5a8 c01bf4d8
        fp       ip       lr       pc
```

其中 lr 值为 c01bd5a8，表示函数 platform_drv_probe 执行完后的返回地址，它是上一级调用函数中的地址。使用 lr 值，重复本步骤的查找过程，直到栈信息分析完毕或者再也无法分析，这样就可以找出所有的函数调用关系。

有些函数很简单，没有使用栈（sp 值在这个函数中没有变化），或者没有在栈中保存 lr 值。这些情况需要读者灵活处理，较强的汇编程序阅读能力是关键。

第 20 章　Linux 异常处理体系结构

本章目标

- 了解 Linux 异常处理体系结构
- 掌握 Linux 中断处理体系结构，了解几种重要的数据结构
- 学习中断处理函数的注册、处理、卸载流程
- 掌握在驱动程序中使用中断的方法

20.1　Linux 异常处理体系结构概述

20.1.1　Linux 异常处理的层次结构

内核的中断处理结构有很好的扩充性，并适当屏蔽了一些实现细节。但是开发人员应该深入“ 黑盒子 ”了解其中的实现原理。

1．异常的作用

异常，就是可以打断 CPU 正常运行流程的一些事情，比如外部中断、未定义的指令、试图修改只读的数据、执行 swi 指令（Software Interrupt Instruction，软件中断指令）等。当这些事情发生时，CPU 暂停当前的程序，先处理异常事件，然后再继续执行被中断的程序。操作系统中经常通过异常来完成一些特定的功能，除第 9 章介绍的“ 中断 ”外，还有下面举的例子（但不限于这些例子）。

- 当 CPU 执行未定义的机器指令时将触发“ 未定义指令异常 ”，操作系统可以利用这个特点使用一些自定义的机器指令，它们在异常处理函数中实现。
- 可以将一块数据设为只读的，然后提供给多个进程共用，这样可以节省内存。当某个进程试图修改其中的数据时，将触发“ 数据访问中止异常 ”，在异常处理函数中将这块数据复制出一份可写的副本，提供给这个进程使用。
- 当用户程序试图读写的数据或执行的指令不在内存中时，也会触发一个“ 数据访问中止异常 ”或“ 指令预取中止异常 ”，在异常处理函数中将这些数据或指令读入内存（内存不足时还可以将不使用的数据、指令换出内存），然后重新执行被中断的程序。这样可以节省内存，还使得操作系统可以运行这类程序：它们使用的内存远大于实际的

物理内存。

- 当程序使用不对齐的地址访问内存时，也会触发“ 数据访问中止异常 ”，在异常处理程序中先使用多个对齐的地址读出数据；对于读操作，从中选取数据组合好后返回给被中断的程序；对于写操作，修改其中的部分数据后再写入内存。这使得程序（特别是应用程序）不用考虑地址对齐的问题。
- 用户程序可以通过“ swi ”指令触发“ swi 异常 ”，操作系统在 swi 异常处理函数中实现各种系统调用。

2．Linux 内核对异常的设置

内核在 start_kernel 函数（源码在 init/main.c 中）中调用 trap_init、init_IRQ 两个函数来设置异常的处理函数。

（1）trap_init 函数分析。

trap_init 函数（代码在 arch/arm/kernel/traps.c 中）被用来设置各种异常的处理向量，包括中断向量。所谓“ 向量 ”，就是一些被安放在固定位置的代码，当发生异常时，CPU 会自动执行这些固定位置上的指令。ARM 架构 CPU 的异常向量基址可以是 0x00000000，也可以是 0xffff0000，Linux 内核使用后者。trap_init 函数将异常向量复制到 0xffff0000 处，部分代码如下：

```
708 void __init trap_init(void)
709 {
…
721     memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
722     memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);
…
734 }
```

第 721 行中，vectors 等于 0xffff0000。地址__vectors_start～__vectors_end 之间的代码就是异常向量，在 arch/arm/kernel/entry-armv.S 中定义，它们被复制到地址 0xffff0000 处。

异常向量的代码很简单，它们只是一些跳转指令。发生异常时，CPU 自动执行这些指令，跳转去执行更复杂的代码，比如保存被中断程序的执行环境，调用异常处理函数，恢复被中断程序的执行环境并重新运行。这些“ 更复杂的代码 ”在地址__stubs_start～__stubs_end 之间，它们在 arch/arm/kernel/entry-armv.S 中定义。第 722 行将它们复制到地址 0xffff0000+0x200 处。

异常向量、异常向量跳去执行的代码都是使用汇编写的，为给读者一个形象概念，下面讲解部分代码，它们在 arch/arm/kernel/entry-armv.S 中。

异常向量的代码如下，其中的“ stubs_offset ”用来重新定位跳转的位置（向量被复制到地址 0xffff0000 处，跳转的目的代码被复制到地址 0xffff0000+0x200 处）。

```
1059    .equ    stubs_offset, __vectors_start + 0x200 - __stubs_start
1060
1061    .globl  __vectors_start
```

```
1062 __vectors_start:
1063    swi SYS_ERROR0                        /* 复位时，CPU 将执行这条指令 */
1064    b  vector_und + stubs_offset          /* 未定义异常时，CPU 将执行这条指令 */
1065    ldr pc, .LCvswi + stubs_offset         /* swi 异常 */
1066    b  vector_pabt + stubs_offset         /* 指令预取中止 */
1067    b  vector_dabt + stubs_offset         /* 数据访问中止 */
1068    b  vector_addrexcptn + stubs_offset   /* 没有用到 */
1069    b  vector_irq + stubs_offset          /* irq 异常 */
1070    b  vector_fiq + stubs_offset          /* fiq 异常 */
1071
1072    .globl  __vectors_end
1073 __vectors_end:
```

其中的 vector_und、vector_pabt 等表示要跳转去执行的代码。以 vector_und 为例，它仍在 arch/arm/kernel/entry-armv.S 中，通过 vector_stub 宏来定义，代码如下：

```
1002    vector_stub und, UND_MODE
1003
1004    .long   __und_usr           @  0 (USR_26 / USR_32)，在用户模式执行了未定义的指令
1005    .long   __und_invalid       @  1 (FIQ_26 / FIQ_32)，在 FIQ 模式执行了未定义的指令
1006    .long   __und_invalid       @  2 (IRQ_26 / IRQ_32)，在 IRQ 模式执行了未定义的指令
1007    .long   __und_svc           @  3 (SVC_26 / SVC_32)，在管理模式执行了未定义的指令
1008    .long   __und_invalid       @  4
1009    .long   __und_invalid       @  5
1010    .long   __und_invalid       @  6
1011    .long   __und_invalid       @  7
1012    .long   __und_invalid       @  8
1013    .long   __und_invalid       @  9
1014    .long   __und_invalid       @  a
1015    .long   __und_invalid       @  b
1016    .long   __und_invalid       @  c
1017    .long   __und_invalid       @  d
1018    .long   __und_invalid       @  e
1019    .long   __und_invalid       @  f
```

第 1002 行的 vector_stub 是一个宏，它根据后面的参数“ und, UND_MODE ”定义了以“ vector_und ”为标号的一段代码。vector_stub 宏的功能为：计算处理完异常后的返回地址、

保存一些寄存器（比如 r0、lr、spsr），然后进入管理模式，最后根据被中断的工作模式调用第 1004～1019 行中的某个跳转分支。当发生异常时，CPU 会根据异常的类型进入某个工作模式，但是很快 vector_stub 宏又会强制 CPU 进入管理模式，在管理模式下进行后续处理，这种方法简化了程序设计，使得异常发生前的工作模式要么是用户模式，要么是管理模式。

第 1004～1019 行中的代码表示在各个工作模式下执行未定义指令时，发生的异常的处理分支。比如 1004 行的__und_usr 表示在用户模式下执行未定义指令时，所发生的未定义异常将由它来处理；第 1007 行的__und_svc 表示在管理模式下执行未定义指令时，所发生的未定义异常将由它来处理。在其他工作模式下不可能发生未定义指令异常，否则使用“ __und_invalid ”来处理错误。ARM 架构 CPU 中使用 4 位数据来表示工作模式（目前只有 7 种工作模式），所以共有 16 个跳转分支。

不同的跳转分支（比如__und_usr、__und_svc）只是在它们的入口处（比如保存被中断程序的寄存器）稍有差别，后续的处理大体相同，都是调用相应的 C 函数。比如未定义指令异常发生时，最终会调用 C 函数 do_undefinstr 来进行处理。各种的异常的 C 处理函数可以分为 5 类，它们分布在不同的文件中。

① 在 arch/arm/kernel/traps.c 中。

未定义指令异常的 C 处理函数在这个文件中定义，总入口函数为 do_undefinstr。

② 在 arch/arm/mm/fault.c 中。

与内存访问相关的异常的 C 处理函数在这个文件中定义，比如数据访问中止异常、指令预取中止异常。总入口函数为 do_DataAbort、do_PrefetchAbort。

③ 在 arch/arm/mm/irq.c 中。

中断处理函数的在这个文件中定义，总入口函数为 asm_do_IRQ，它调用其他文件注册的中断处理函数。

④ 在 arch/arm/kernel/calls.S 中。

在这个文件中，swi 异常的处理函数指针被组织成一个表格；swi 指令机器码的位[23:0]被用来作为索引。这样，通过不同的“ swi index ”指令就可以调用不同的 swi 异常处理函数，它们被称为系统调用，比如 sys_open、sys_read、sys_write 等。

⑤ 没有使用的异常。

在 Linux 2.6.22.6 中没有使用 FIQ 异常。

trap_init 函数搭建了各类异常的处理框架。当发生异常时，各种 C 处理函数会被调用。这些 C 函数还要进一步细分异常发生的情况，分别调用更具体的处理函数。比如未定义指令异常的 C 处理函数总入口为 do_undefinstr，这个函数里还要根据具体的未定义指令调用它的模拟函数。

除了中断外，内核已经为各类异常准备了细致而完备的处理函数，比如 swi 异常处理函数为每一种系统调用都准备了一个“ sys_ ”开头的函数，数据访问中止异常的处理函数为对齐错误、页权限错误、段翻译错误等具体异常都准备了相应的处理函数。这些异常的处理函数与开发板的配置无关，基本不用修改。

（2）init_IRQ 函数分析。

中断也是一种异常，之所以把它单独提出来，是因为中断的处理与具体开发板密切相

关，除一些必须、共用的中断（比如系统时钟中断、片内外设 UART 中断）外，必须由驱动开发者提供处理函数。内核提炼出中断处理的共性，搭建了一个非常容易扩充的中断处理体系。

init_IRQ 函数（代码在 arch/arm/kernel/irq.c 中）被用来初始化中断的处理框架，设置各种中断的默认处理函数。当发生中断时，中断总入口函数 asm_do_IRQ 就可以调用这些函数作进一步处理。

这一节从总体上介绍了异常处理体系结构，并没有太多地深入具体的代码，读者可以根据本节提供的线索自行深入了解。如图 20.1 所示为异常处理体系结构。

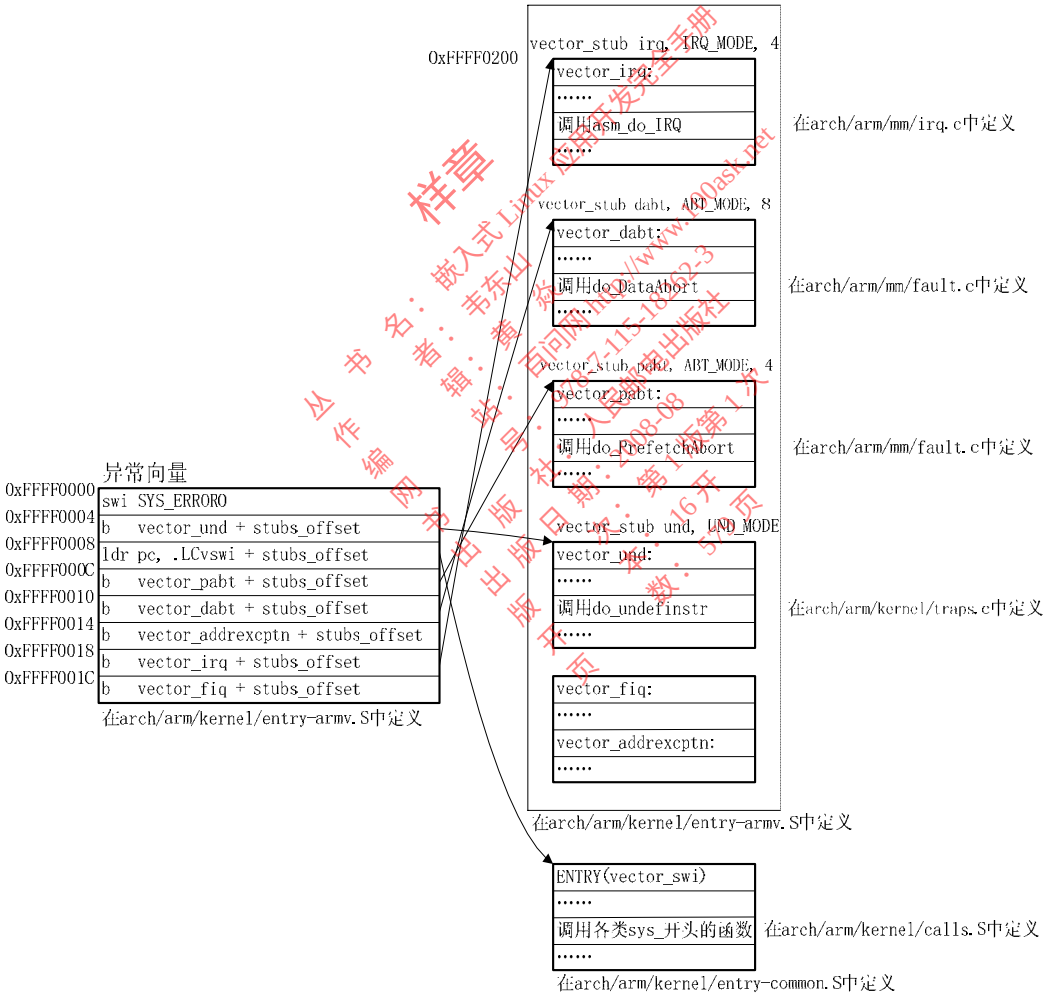

图 20.1　ARM 架构 Linux 内核的异常处理体系结构

20.1.2　常见的异常

ARM 架构 Linux 内核中，只用到了 5 种异常，在它们的处理函数中进一步细分发生这些异常的原因。表 20.1 列出了常见的异常。

表 20.1　　ARM 架构 Linux 中常见的异常

异 常 总 类	异 常 细 分
未定义指令异常	ARM 指令 break
	Thumb 指令 break
	ARM 指令 mrc
指令预取中止异常	取指时地址翻译错误（translation fault），系统中还没有为这个指令的地址建立映射关系
数据访问中止异常	访问数据时段地址翻译错误（section translation fault）
	访问数据时页地址翻译错误（page translation fault）
	地址对齐错误
	段权限错误（section permission fault）
	页权限错误（page permission fault）
	…
中断异常	GPIO 引脚中断、WDT 中断、定时器中断、USB 中断、UART 中断等
swi 异常	各类系统调用，sys_open、sys_read、sys_write 等

20.2 Linux 中断处理体系结构

20.2.1 中断处理体系结构的初始化

1．中断处理体系结构

Linux 内核将所有的中断统一编号，使用一个 irq_desc 结构数组来描述这些中断：每个数组项对应一个中断（也有可能是一组中断，它们共用相同的中断号），里面记录了中断的名称、中断状态、中断标记（比如中断类型、是否共享中断等），并提供了中断的低层硬件访问函数（清除、屏蔽、使能中断），提供了这个中断的处理函数入口，通过它可以调用用户注册的中断处理函数。

通过 irq_desc 结构数组就可以了解中断处理体系结构，irq_desc 结构的数据类型在 include/linux/irq.h 中定义，如下所示：

```
151 struct irq_desc {
152     irq_flow_handler_t  handle_irq; /* 当前中断的处理函数入口 */
153     struct irq_chip     *chip;      /* 低层的硬件访问 */
……
157     struct irqaction    *action;    /* 用户提供的中断处理函数链表 */
158     unsigned int        status;     /* IRQ 状态 */
……
175     const char      *name;          /* 中断名称 */
176 } ____cacheline_internodealigned_in_smp;
```

第 152 行的 handle_irq 是这个或这组中断的处理函数入口。发生中断时，总入口函数 asm_do_IRQ 将根据中断号调用相应 irq_desc 数组项中的 handle_irq。handle_irq 使用 chip 结构中的函数来清除、屏蔽或者重新使能中断，还一一调用用户在 action 链表中注册的中断处理函数。

第 153 行的 irq_chip 结构类型也是在 include/linux/irq.h 中定义，其中的成员大多用于操作底层硬件，比如设置寄存器以屏蔽中断、使能中断、清除中断等。这个结构的部分成员如下：

```
struct irq_chip {
 const char  *name;
 unsigned int  (*startup)(unsigned int irq); /* 启动中断,如果不设置,缺省为"enable" */
 void    (*shutdown)(unsigned int irq);    /* 关闭中断，如果不设置，缺省为"disable" */
 void    (*enable)(unsigned int irq);      /* 使能中断，如果不设置，缺省为"unmask" */
 void   (*disable)(unsigned int irq);    /* 禁止中断，如果不设置，缺省为"mask" */

 void    (*ack)(unsigned int irq);  /* 响应中断,通常是清除当前中断使得可以接收下一个中断*/
 void    (*mask)(unsigned int irq);       /* 屏蔽中断源 */
 void    (*mask_ack)(unsigned int irq);   /* 屏蔽和响应中断 */
 void    (*unmask)(unsigned int irq);     /* 开启中断源 */
…
}
```

irq_desc 结构中第 157 行的 irqaction 结构类型在 include/linux/interrupt.h 中定义。用户注册的每个中断处理函数用一个 irqaction 结构来表示，一个中断（比如共享中断）可以有多个处理函数，它们的 irqactio 结构链接成一个链表，以 action 为表头。irq_desc 结构定义如下：

```
struct irqaction {
    irq_handler_t handler;     /* 用户注册的中断处理函数 */
   unsigned long flags;        /* 中断标志,比如是否共享中断、电平触发还是边沿触发等 */
    cpumask_t mask;            /* 用于 SMP(对称多处理器系统) */
    const char *name;          /* 用户注册的中断名字，"cat /proc/interrupts"时可以看到 */
    void *dev_id;              /* 用户传给上面的 handler 的参数，还可以用来区分共享中断 */
    struct irqaction *next;
    int irq;                /* 中断号 */
    struct proc_dir_entry *dir;
};
```

irq_desc 结构数组、它的成员“ struct irq_chip *chip ”、“ struct irqaction *action ”，这 3 种数据结构构成了中断处理体系的框架。这 3 者的关系如图 20.2 所示。

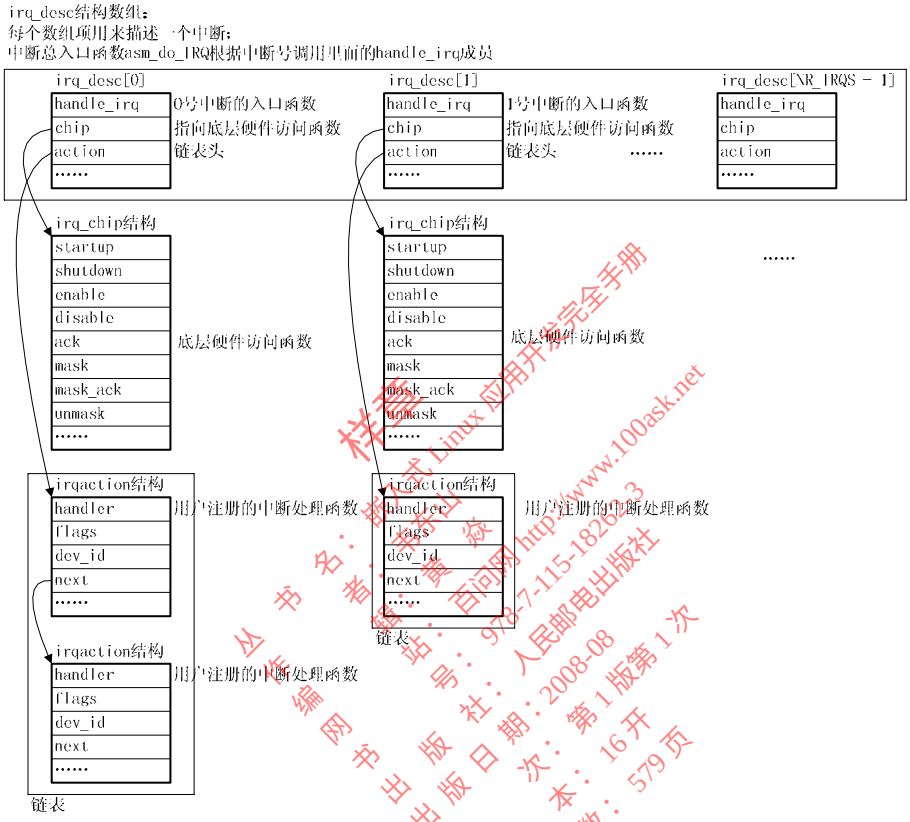

图 20.2 Linux 中断处理体系结构

中断的处理流程如下。

（1）发生中断时，CPU 执行异常向量 vector_irq 的代码。

（2）在 vector_irq 里面，最终会调用中断处理的总入口函数 asm_do_IRQ。

（3）asm_do_IRQ 根据中断号调用 irq_desc 数组项中的 handle_irq。

（4）handle_irq 会使用 chip 成员中的函数来设置硬件，比如清除中断、禁止中断、重新使能中断等。

（5）handle_irq 逐个调用用户在 action 链表中注册的处理函数。

可见，中断体系结构的初始化就是构造这些数据结构，比如 irq_desc 数组项中的 handle_irq、chip 等成员；用户注册中断时就是构造 action 链表；用户卸载中断时就是从 action 链表中去除不需要的项。

2．中断处理体系结构的初始化

init_IRQ 函数被用来初始化中断处理体系结构，代码在 arch/arm/kernel/irq.c 中。

```
156 void __init init_IRQ(void)
157 {
```

```
158    int irq;
159
160    for (irq = 0; irq < NR_IRQS; irq++)
161        irq_desc[irq].status |= IRQ_NOREQUEST | IRQ_NOPROBE;
…
167    init_arch_irq();
168 }
```

第 160～161 行初始化 irq_desc 结构数组中每一项的中断状态。

第 167 行调用架构相关的中断初始化函数。对于本书所用的 S3C2410、S3C2440 开发板，这个函数就是 s3c24xx_init_irq，移植 Linux 内核时讲述的 machine_desc 结构中的 init_irq 成员就指向这个函数。

s3c24xx_init_irq 函数在 arch/arm/plat-s3c24xx/irq.c 中定义，它为所有的中断设置了芯片相关的数据结构（irq_desc[irq].chip），设置了处理函数入口（irq_desc[irq]. handle_irq）。以外部中断 EINT4～EINT23 为例，用来设置它们的代码如下：

```
760    for (irqno = IRQ_EINT4; irqno <= IRQ_EINT23; irqno++) {
761        irqdbf("registering irq %d (extended s3c irq)\n", irqno);
762        set_irq_chip(irqno, &s3c_irqext_chip);
763        set_irq_handler(irqno, handle_edge_irq);
764        set_irq_flags(irqno, IRQF_VALID);
765    }
```

第 762 行 set_irq_chip 函数的作用就是“ irq_desc[irqno].chip = &s3c_irqext_chip ”。以后就可以通过 irq_desc[irqno].chip 结构中的函数指针设置这些外部中断的触发方式（电平触发、边沿触发等）、使能中断、禁止中断等。

第 763 行设置这些中断的处理函数入口为 handle_edge_irq，即“ irq_desc[irqno]. handle_irq = handle_edge_irq ”。发生中断时，handle_edge_irq 函数会调用用户注册的具体处理函数。

第 764 行设置中断标志为“ IRQF_VALID ”，表示可以使用它们。

对于本书使用的 S3C2410、S3C2440 开发板，init_IRQ 函数执行完后，图 20.2 中各个 irq_desc 数组项的 chip、handle_irq 成员都被设置好了。

20.2.2　用户注册中断处理函数的过程

用户（即驱动程序）通过 request_irq 函数向内核注册中断处理函数，request_irq 函数根据中断号找到 irq_desc 数组项，然后在它的 action 链表中添加一个表项。

request_irq 函数在 kernel/irq/manage.c 中定义，函数原型如下：

```
int request_irq(unsigned int irq, irq_handler_t handler,
       unsigned long irqflags, const char *devname, void *dev_id)
```

request_irq 函数首先使用这 4 个参数构造一个 irqaction 结构，然后调用 setup_irq 函数将它链入链表中，代码如下：

```
527     action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
…
531     action->handler = handler;
532     action->flags = irqflags;
533     cpus_clear(action->mask);
534     action->name = devname;
535     action->next = NULL;
536     action->dev_id = dev_id;
…
559     retval = setup_irq(irq, action);
```

setup_irq 函数也是在 kernel/irq/manage.c 中定义，它完成如下 3 个功能（本书忽略了其他不感兴趣的功能）。

（1）将新建的 irqaction 结构链入 irq_desc[irq]结构的 action 链表中，这有两种可能。

① 如果 action 链表为空，则直接链入。

② 否则先判断新建的 irqaction 结构和链表中的 irqaction 结构所表示的中断类型是否一致；即是否都声明为“可共享的”（IRQF_SHARED）、是否都使用相同的触发方式（电平、边沿、极性），如果一致，则将新建的 irqaction 结构链入。

（2）设置 irq_desc[irq]结构中 chip 成员的还没设置的指针，让它们指向一些默认函数。

> **注意** chip 成员在 init_IRQ 函数初始化中断体系结构的时候已经被设置，这里只是设置其中还没设置的指针。

这通过 irq_chip_set_defaults 函数来完成，它在 kernel/irq/chip.c 中定义。

```
251 void irq_chip_set_defaults(struct irq_chip *chip)
252 {
253     if (!chip->enable)
254         chip->enable = default_enable;      /* 它调用 chip->unmask */
255     if (!chip->disable)
256         chip->disable = default_disable;    /* 此函数为空 */
257     if (!chip->startup)
258         chip->startup = default_startup;    /* 它调用 chip->enable */
259     if (!chip->shutdown)
260         chip->shutdown = chip->disable;
261     if (!chip->name)
262         chip->name = chip->typename;
263     if (!chip->end)
264         chip->end = dummy_irq_chip.end;
265 }
```

（3）设置中断的触发方式。

如果requet_irq函数中传入的irqflags参数表示中断的触发方式为高电平触发、低电平触发、上升沿触发或下降沿触发，则调用irq_desc[irq]结构中的chip->set_type成员函数来进行设置：设置引脚功能为外部中断，设置中断触发方式。

注意　如果原来的action链表非空，表示以前已经设置过这个中断的触发方式，就不用再次设置了。

（4）启动中断。

如果irq_desc[irq]结构中status成员没有被指明为IRQ_NOAUTOEN（表示注册中断时不要使能中断），还要调用chip->startup或chip->enable来启动中断。所谓启动中断通常就是使能中断。

一般来说，只有那些“可以自动使能的”中断对应的irq_desc[irq].status才会被指明为IRQ_NOAUTOEN。所以，无论哪种情况，执行reqeust_irq注册中断之后，这个中断就已经被使能了，在编写驱动程序时要注意这点。

总结一下使用reqeust_irq函数注册中断后的“成果”。

（1）irq_desc[irq]结构中的action链表中已经链入了用户注册的中断处理函数。

（2）中断的触发方式已经被设好。

（3）中断已经被使能。

总之，执行reqeust_irq函数之后，中断就可以发生并能够被处理了。

20.2.3　中断的处理过程

asm_do_IRQ是中断的C语言总入口函数，它在arch/arm/kernel/irq.c中定义，部分代码如下：

```
111 asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
112 {
113     struct pt_regs *old_regs = set_irq_regs(regs);
114     struct irq_desc *desc = irq_desc + irq;
…
125     desc_handle_irq(irq, desc);
…
132 }
```

第125行的desc_handle_irq函数直接调用desc结构中的handle_irq成员函数，它就是irq_desc[irq].handle_irq。

需要说明的是，asm_do_IRQ函数中参数irq的取值范围为IRQ_EINT0～(IRQ_EINT0 + 31)，只有32个取值。它可能是一个实际中断的中断号，也可能是一组中断的中断号。这是由S3C2410、S3C2440的芯片特性决定的：发生中断时INTPND寄存器的某一位被置1，INTOFFSET寄存器中记录了是哪一位（0～31），中断向量调用asm_do_IRQ之前根据INTOFFSET寄存器的值确定irq参数。每一个实际的中断在irq_desc数组中都有一项与它对应，它们的数目不止32。当asm_do_IRQ函数中参数irq表示的是“一组”中断时，irq_desc[irq].handle_irq成员函数还需要先分辨出是哪一个中断（假设中断号为irqno），然后调用irq_desc[irqno].handle_irq来进一步处理。

以外部中断EINT8～EINT23为例，它们通常是边沿触发。

（1）它们被触发时，INTOFFSET寄存器中的值都是5，asm_do_IRQ函数中参数irq的值

为（IRQ_EINT0+5），即 IRQ_EINT8t23。上面代码中第 125 行将调用 irq_desc[IRQ_EINT8t23].handle_irq 来进行处理。

（2）irq_desc[IRQ_EINT8t23].handle_irq 在前面 init_IRQ 函数初始化中断体系结构的时候被设为 s3c_irq_demux_extint8。

（3）s3c_irq_demux_extint8 函数的代码在 arch/arm/plat-s3c24xx/irq.c 中，它首先读取 EINTPEND、EINTMASK 寄存器，确定发生了哪些中断，重新计算它们的中断号，然后调用 irq_desc 数组项中的 handle_irq 成员函数。

代码如下：

```
558 static void
559 s3c_irq_demux_extint8(unsigned int irq,
560                struct irq_desc *desc)
561 {
562     unsigned long eintpnd = __raw_readl(S3C24XX_EINTPEND); /* EINT8～EINT23 发生时,相应位被置 1 */
563     unsigned long eintmsk = __raw_readl(S3C24XX_EINTMASK);  /* 屏蔽寄存器 */
564 
565     eintpnd &= ~eintmsk; /* 清除被屏蔽的位 */
566     eintpnd &= ~0xff;     /* 清除低 8 位(EINT8 对应位 8, …) */
567 
568     /* 循环处理所有的子中断 */
569 
570     while (eintpnd) {
571         irq = __ffs(eintpnd);   /* 确定 eintpnd 中为 1 的最高位 */
572         eintpnd &= ~(1<<irq);   /* 将此位清 0 */
573 
574         irq += (IRQ_EINT4 - 4); /* 重新计算中断号:前面计算出 irq 等于 8 时,中断号为 IRQ_EINT8 */
575         desc_handle_irq(irq, irq_desc + irq);   /* 调用这个中断的真正的处理函数入口 */
576     }
577 
578 }
```


（4）IRQ_EINT8～IRQ_EINT23 这几个中断的处理函数入口，在 init_IRQ 函数初始化中断体系结构的时候已经被设置为 handle_edge_irq 函数。上面第 575 行的代码就是调用这个函数，它在 kernel/irq/chip.c 中定义。从它的名字可以知道，它用来处理边沿触发的中断（处理电平触发的中断为 handle_level_irq）。以下的讲解中，只关心一般的情形，忽略有关中断嵌套的代码，部分代码如下：

```
void fastcall
```

```
446 handle_edge_irq(unsigned int irq, struct irq_desc *desc)
447 {
…
466     kstat_cpu(cpu).irqs[irq]++;
467
468     /* Start handling the irq */
469     desc->chip->ack(irq);
…
497     action_ret = handle_IRQ_event(irq, action);
…
507 }
```

第 466 行用来统计中断发生的次数。

第 469 行响应中断，通常是清除当前中断使得可以接收下一个中断。对于 IRQ_EINT8～IRQ_EINT23 这几个中断，desc->chip 在前面 init_IRQ 函数初始化中断体系结构的时候被设为 s3c_irqext_chip。desc->chip->ack 就是 s3c_irqext_ack 函数（arch/arm/plat-s3c24xx/ irq.c），它用来清除中断。

第 497 行通过 handle_IRQ_event 函数来逐个执行 action 链表中用户注册的中断处理函数，它在 kernel/irq/handle.c 中定义，关键代码如下：

```
139     do {
140         ret = action->handler(irq, action->dev_id); /* 执行用户注册的中断处
理函数 */
141         if (ret == IRQ_HANDLED)
142             status |= action->flags;
143         retval |= ret;
144         action = action->next;                  /* 下一个 */
145     } while (action);
```

从第 140 行可以知道，用户注册的中断处理函数的参数为中断号 irq、action->dev_id。后一个参数是通过 request_irq 函数注册中断时传入的 dev_id 参数。它由用户自己指定、自己使用，可以为空，当这个中断是“ 共享中断 ”时除外（这在下面卸载中断时说明）。

对于电平触发的中断，它们的 irq_desc[irq].handle_irq 通常是 handle_level_irq 函数。它也是在 kernel/irq/chip.c 中定义，其功能与上述 handle_edge_irq 函数相似，关键代码如下：

```
335 void fastcall
336 handle_level_irq(unsigned int irq, struct irq_desc *desc)
337 {
…
343     mask_ack_irq(desc, irq);
…
```

```
348     kstat_cpu(cpu).irqs[irq]++;
…
364     action_ret = handle_IRQ_event(irq, action);
…
371         desc->chip->unmask(irq);
…
374 }
```

第 343 行用来屏蔽和响应中断，响应中断通常就是清除中断，使得可以接收下一个中断。

> 注意　这时即使触发了下一个中断，也只是记录寄存器中而已，只有在中断被再次使能后才能被处理。

第 348 行用来统计中断发生的次数。

第 364 行通过 handle_IRQ_event 函数来逐个执行 action 链表中用户注册的中断处理函数。

第 371 行开启中断，与前面第 343 行屏蔽中断对应。

在 handle_edge_irq、handle_level_irq 函数的开头都清除了中断。所以一般来说，在用户注册的中断处理函数中就不用再次清除中断了。但是对于电平触发的中断也有例外：虽然 handle_level_irq 函数已经清除了中断，但是它只限于清除 SoC 内部的信号；如果外设输入到 SoC 的中断信号仍然有效，这就会导致当前中断处理完毕后，会误认为再次发生了中断。对于这种情况，需要在用户注册的中断处理函数中清除中断：先清除外设的中断，然后再清除 SoC 内部的中断信号。

忽略上述的中断号重新计算过程（这不影响对整体流程的理解），中断的处理流程可以总结如下。

（1）中断向量调用总入口函数 asm_do_IRQ，传入根据中断号 irq。

（2）asm_do_IRQ 函数根据中断号 irq 调用 irq_desc[irq].handle_irq，它是这个中断的处理函数入口。对于电平触发的中断，这个入口通常为 handle_level_irq；对于边沿触发的中断，这个入口通常为 handle_edge_irq。

（3）入口函数首先清除中断，入口函数是 handle_level_irq 时还要屏蔽中断。

（4）逐个调用用户在 irq_desc[irq].action 链表中注册的中断处理函数。

（5）入口函数是 handle_level_irq 时还要重新开启中断。

20.2.4　卸载中断处理函数

中断是一种很稀缺的资源，当不再使用一个设备时，应该释放它占据的中断。这通过 free_irq 函数来实现，它与 request_irq 一样，也是在 kernel/irq/manage.c 中定义。它的函数原型如下。

```
void free_irq(unsigned int irq, void *dev_id)
```

它需要用到两个参数：irq 和 dev_id，它们与通过 reqeust_irq 注册中断函数时使用的参数一样。使用中断号 irq 定位 action 链表，再使用 dev_id 在 action 链表中找到要卸载的表项。所以，同一个中断的不同中断处理函数必须使用不同的 dev_id 来区分，这就要求在注册共享中断时参数 dev_id 必须惟一。

free_irq 函数的处理过程与 reqeust_irq 函数相反。

（1）根据中断号 irq、dev_id 从 action 链表中找到表项，将它移除。

（2）如果它是惟一的表项，还要调用 IRQ_DESC[IRQ].CHIP->SHUTDOWN 或 IRQ_DESC[IRQ]. CHIP->DISABLE 来关闭中断。

20.3　使用中断的驱动程序示例

20.3.1　按键驱动程序源码分析

开发板上有 4 个按键，它们的连线如图 20.3 所示。

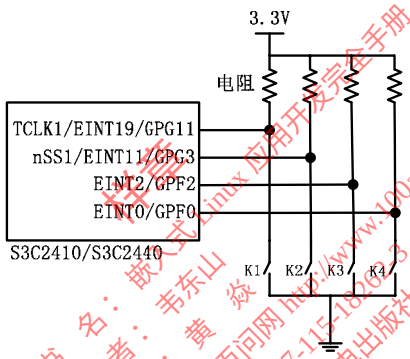

图 20.3　按键连线图

按键驱动程序就是光盘上的 drivers_and_test/buttons/s3c24xx_buttons.c 文件，下面按照函数调用的顺序进行讲解。

1．模块的初始化函数和卸载函数

代码如下：

```
130 /*
131  * 执行"insmod s3c24xx_buttons.ko"命令时就会调用这个函数
132  */
133 static int __init s3c24xx_buttons_init(void)
134 {
135     int ret;
136
137     /* 注册字符设备驱动程序
138      * 参数为主设备号、设备名字、file_operations 结构；
139      * 这样，主设备号就和具体的 file_operations 结构联系起来了，
140      * 操作主设备为 BUTTON_MAJOR 的设备文件时,就会调用 s3c24xx_buttons_fops 中的相关成员函数
141      * BUTTON_MAJOR 可以设为 0，表示由内核自动分配主设备号
142      */
143     ret = register_chrdev(BUTTON_MAJOR, DEVICE_NAME, &s3c24xx_buttons_fops);
```

```
144     if (ret < 0) {
145       printk(DEVICE_NAME " can't register major number\n");
146       return ret;
147     }
148
149     printk(DEVICE_NAME " initialized\n");
150     return 0;
151 }
152
153 /*
154  * 执行"rmmod s3c24xx_buttons.ko"命令时就会调用这个函数
155  */
156 static void __exit s3c24xx_buttons_exit(void)
157 {
158     /* 卸载驱动程序 */
159     unregister_chrdev(BUTTON_MAJOR, DEVICE_NAME);
160 }
161
162 /* 这两行指定驱动程序的初始化函数和卸载函数 */
163 module_init(s3c24xx_buttons_init);
164 module_exit(s3c24xx_buttons_exit);
165
```


与 LED 驱动相似，执行“ insmod s3c24xx_buttons.ko ”命令加载驱动时就会调用这个驱动初始化函数 s3c24xx_buttons_init；执行“ rmmod s3c24xx_buttons.ko ”命令卸载驱动时就会调用卸载函数 s3c24xx_buttons_exit。前者调用 register_chrdev 函数向内核注册驱动程序，后者调用 s3c24xx_buttons_exit 函数卸载这个驱动程序。

驱动程序的核心是 s3c24xx_buttons_fops 结构，定义如下：

```
119 /* 这个结构是字符设备驱动程序的核心
120  * 当应用程序操作设备文件时所调用的 open、read、write 等函数，
121  * 最终会调用这个结构中的对应函数
122  */
123 static struct file_operations s3c24xx_buttons_fops = {
124     .owner   =   THIS_MODULE,    /* 这是一个宏，指向编译模块时自动创建的__
this_module 变量 */
125     .open    =   s3c24xx_buttons_open,
126     .release =   s3c24xx_buttons_close,
127     .read    =   s3c24xx_buttons_read,
128 };
```

```
129
```

第 123～125 行的 3 个函数在下面依次讲述。

2．s3c24xx_buttons_open 函数

在应用程序执行 open("/dev/buttons",…)系统调用时，s3c24xx_buttons_open 函数将被调用。它用来注册 4 个按键的中断处理程序，代码如下：

```
54 /* 应用程序对设备文件/dev/buttons 执行 open(…)时，
55  * 就会调用 s3c24xx_buttons_open 函数
56  */
57 static int s3c24xx_buttons_open(struct inode *inode, struct file *file)
58 {
59     int i;
60     int err;
61
62     for (i = 0; i < sizeof(button_irqs)/sizeof(button_irqs[0]); i++) {
63         // 注册中断处理函数
64             err = request_irq(button_irqs[i].irq, buttons_interrupt, button_
irqs[i].flags,
65                               button_irqs[i].name, (void *)&press_cnt[i]);
66         if (err)
67             break;
68     }
69
70     if (err) {
71         // 如果出错，释放已经注册的中断
72         i--;
73         for (; i >= 0; i--)
74             free_irq(button_irqs[i].irq, (void *)&press_cnt[i]);
75         return -EBUSY;
76     }
77
78     return 0;
79 }
80
```

第 64 行向内核注册中断处理函数，request_irq 函数的作用在前面已经讲解过。注册成功后，这 4 个按键所用 GPIO 引脚的功能被设为外部中断，触发方式为下降沿触发，中断处理函数为 buttons_interrupt。最后一个参数“ (void *)&press_cnt[i] ”将在 buttons_interrupt 函数中用到，它用来存储按键被按下的次数。

第 70～76 行是出错处理的代码，如果前面有某个中断没有注册成功，这几行代码用来卸载已经注册的中断。

第 64 行中用到的参数 button_irqs 定义如下，表示了这 4 个按键的中断号、中断触发方式、中断名称（名称只是供执行“ cat /proc/interrupts ”时显示用）。

```
15 struct button_irq_desc {
16     int irq;             /* 中断号 */
17     unsigned long flags;   /* 中断标志，用来定义中断的触发方式 */
18     char *name;           /* 中断名称 */
19 };
20
21 /* 用来指定按键所用的外部中断引脚及中断触发方式、名字 */
22 static struct button_irq_desc button_irqs [] = {
23     {IRQ_EINT19, IRQF_TRIGGER_FALLING, "KEY1"}, /* K1 */
24     {IRQ_EINT11, IRQF_TRIGGER_FALLING, "KEY2"}, /* K2 */
25     {IRQ_EINT2,  IRQF_TRIGGER_FALLING, "KEY3"}, /* K3 */
26     {IRQ_EINT0,  IRQF_TRIGGER_FALLING, "KEY4"}, /* K4 */
27 };
28
```

3．s3c24xx_buttons_close 函数

s3c24xx_buttons_close 函数的作用与 s3c24xx_buttons_open 函数相反，它用来卸载 4 个按键的中断处理函数，代码如下：

```
82 /* 应用程序对设备文件/dev/buttons 执行 close(…)时，
83  * 就会调用 s3c24xx_buttons_close 函数
84  */
85 static int s3c24xx_buttons_close(struct inode *inode, struct file *file)
86 {
87     int i;
88
89     for (i = 0; i < sizeof(button_irqs)/sizeof(button_irqs[0]); i++) {
90         // 释放已经注册的中断
91         free_irq(button_irqs[i].irq, (void *)&press_cnt[i]);
92     }
93
94     return 0;
95 }
96
```

4．s3c24xx_buttons_read 函数

中断处理函数会在 press_cnt 数组中记录按键被按下的次数。s3c24xx_buttons_read 函数首先判断是否有按键被再次按下，如果没有则休眠等待；否则读取 press_cnt 数组的数据，代码如下：

```
32 /* 等待队列:
33  * 当没有按键被按下时，如果有进程调用 s3c24xx_buttons_read 函数，
34  * 它将休眠
35  */
36 static DECLARE_WAIT_QUEUE_HEAD(button_waitq);
37
38 /* 中断事件标志，中断服务程序将它置 1，s3c24xx_buttons_read 将它清 0 */
39 static volatile int ev_press = 0;
…
98 /* 应用程序对设备文件/dev/buttons 执行 read(…)时，
99  * 就会调用 s3c24xx_buttons_read 函数
100  */
101 static int s3c24xx_buttons_read(struct file *filp, char __user *buff,
102                                 size_t count, loff_t *offp)
103 {
104     unsigned long err;
105
106     /* 如果 ev_press 等于 0，休眠 */
107     wait_event_interruptible(button_waitq, ev_press);
108
109     /* 执行到这里时 ev_press 肯定等于 1，将它清 0 */
110     ev_press = 0;
111
112     /* 将按键状态复制给用户，并清 0 */
113     err = copy_to_user(buff, (const void *)press_cnt, min(sizeof(press_cnt),
count));
114     memset((void *)press_cnt, 0, sizeof(press_cnt));
115
116     return err ? -EFAULT : 0;
117 }
118
```

第 107 行的 wait_event_interruptible 首先会判断 ev_press 是否为 0，如果为 0 才会令当前进程进入休眠；否则向下继续执行。它的第一个参数 button_waitq 是一个等待队列，在前面第 36 行中定义；第二个参数 ev_press 用来表示中断是否已经发生，中断服务程序将它置 1，

s3c24xx_buttons_read 将它清 0。如果 ev_press 为 0，则当前进程会进入休眠，中断发生时中断处理函数 buttons_interrupt 会把它唤醒。

第 110 行将 ev_press 清 0。

第 113 行将 press_cnt 数组的内容复制到用户空间。buff 参数表示的缓冲区位于用户空间，使用 copy_to_user 向它赋值。

第 114 行将 press_cnt 数组清 0。

5．中断处理函数 buttons_interrupt

这 4 个按键的中断处理函数都是 buttons_interrupt，代码如下：

```
42 static irqreturn_t buttons_interrupt(int irq, void *dev_id)
43 {
44     volatile int *press_cnt = (volatile int *)dev_id;
45
46     *press_cnt = *press_cnt + 1;            /* 按键计数加 1 */
47     ev_press = 1;                           /* 表示中断发生了 */
48     wake_up_interruptible(&button_waitq);   /* 唤醒休眠的进程 */
49
50     return IRQ_RETVAL(IRQ_HANDLED);
51 }
52
```

buttons_interrupt 函数被调用时，第一个参数 irq 表示发生的中断号，第二个参数 dev_id 就是前面使用 request_irq 注册中断时传入的“ &press_cnt[i] ”（请参考前面第 65 行代码）。

第 46 行将按键计数加 1。

第 47～48 行将 ev_press 设为 1，唤醒休眠的进程。

将 s3c24xx_buttons.c 放到内核源码目录 drivers/char 下，在 drivers/char/Makefile 中增加如下一行：

```
obj-m   += s3c24xx_buttons.o
```

在内核根目录下执行“ make modules ”命令即可在 drivers/char 目录下生成可加载模块 s3c24xx_buttons.ko，把它放到开发板根文件系统的/lib/modules/2.6.22.6/目录下，就可以使用“ insmod s3c24xx_buttons ”、“ rmmod s3c24xx_buttons ”命令进行加载、卸载了。

20.3.2 测试程序情景分析

按键测试程序源码为 drivers_and_test/buttons/button_test.c，在这个目录下执行“ make ”命令即可生成可执行程序 button_test，然后把它放到开发板根文件系统/usr/bin/目录下。

在开发板根文件系统中建立设备文件：

```
# mknod /dev/buttons c 232 0
```

然后使用“ insmod s3c24xx_buttons ”命令加载模块。

现在直接运行 button_test 即可进行测试：按下 K1～K4，就可以在控制台上观察到类似如下的打印输出：

K1 has been pressed 2 times!

要终止 button_test，如果它在前台运行，可以输入“ Ctrl + C ”，如果它在后台运行，可以输入“ killall button_test ”。

测试程序 button_test.c 代码很简单，下面按照使用过程进行分析。

1．加载模块

执行“ insmod s3c24xx_buttons ”即可加载模块，这时在控制台执行“ cat /proc/devices ”命令可以看到内核中已经有了 buttons 设备，可以看到如下字样：

```
Character devices:
…
232 buttons
```

这表明按键设备属于字符设备，主设备号为 232。

2．测试程序打开设备

运行测试程序 button_test 后，/dev/buttons 设备就被打开了，可以使用“ cat /proc/interrupts ”命令看到注册了 4 个中断。为了便于输入其他命令，在后台运行测试程序 button_test，在命令的最后加上“ & ”就可以了，如下所示：

```
# button_test &
```

这时候执行“ cat /proc/interrupts ”命令可以看到中断的注册、使用情况。第一列数据表示中断号；第二列数据表示这个中断发生的次数；第三列的文字表示这个中断的硬件访问结构“ struct irq_chip *chip ”的名字，它在初始化中断体系结构时指定；第四列文字表示中断的名称，它在 reqeust_irq 中指定。对于这 4 个按键，可以看到如下字样：

```
# cat /proc/interrupts
           CPU0
 16:          1    s3c-ext0  KEY4
 18:          0    s3c-ext0  KEY3
…
 55:          0     s3c-ext  KEY2
 63:         22     s3c-ext  KEY1
…
```

添试程序中打开设备的代码如下：

```
01 #include <stdio.h>
```

```
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <sys/ioctl.h>
05
06 int main(int argc, char **argv)
07 {
08     int i;
09     int ret;
10     int fd;
11     int press_cnt[4];
12
13     fd = open("/dev/buttons", 0);  // 打开设备
14     if (fd < 0) {
15         printf("Can't open /dev/buttons\n");
16         return -1;
17     }
18
```

3．测试程序读取数据

读取数据的代码如下：

```
19     // 这是个无限循环，进程有可能在 read 函数中休眠，当有按键被按下时，它才返回
20     while (1) {
21         // 读出按键被按下的次数
22         ret = read(fd, press_cnt, sizeof(press_cnt));
23         if (ret < 0) {
24             printf("read err!\n");
25             continue;
26         }
27
28         for (i = 0; i < sizeof(press_cnt)/sizeof(press_cnt[0]); i++) {
29             // 如果被按下的次数不为 0，打印出来
30             if (press_cnt[i])
31                 printf("K%d has been pressed %d times!\n", i+1, press_cnt[i]);
32         }
33     }
34
```

第 22 行用来读取数据，如果以前（或自从上次读取之后）没有按键被按下，则进程进入休眠，可以使用“ ps ”命令观察到这点，如下所示：

```
# ps
  PID  Uid       VSZ Stat Command
 ......
  752 0         120 S   button_test
```

上面的“ S ”表示 button_test 进程处于休眠状态。