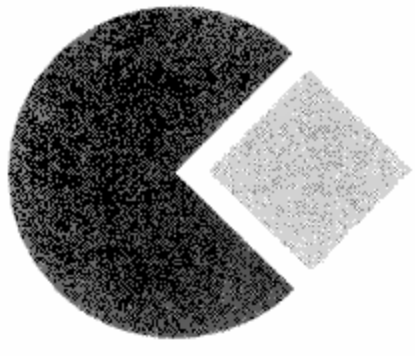


第 16 章　移植 Linux 内核

本章目标

- 了解内核源码结构，了解内核启动过程
- 掌握内核配置方法
- 移植内核同时支持 S3C2410、S3C2440
- 掌握 MTD 设备的分区方法
- 掌握 YAFFS 文件系统的移植方法

16.1　Linux 版本及特点

Linux 内核的版本号可以从源代码的顶层目录下的 Makefile 中看到，比如下面几行它们构成了 Linux 的版本号：2.6.22.6。

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 22
EXTRAVERSION = .6
```

其中的“VERSION”和“PATCHLEVEL”组成主版本号，比如 2.4、2.5、2.6 等，稳定版本的主版本号用偶数表示（比如 2.4、2.6），每隔 2～3 年出现一个稳定版本。开发中的版本号用奇数来表示（比如 2.3、2.5），它是下一个稳定版本的前身。

“SUBLEVEL”称为次版本号，它不分奇偶，顺序递增。每隔 1～2 个月发布一个稳定版本。

“EXTRAVERSION”称为扩展版本号，它不分奇偶，顺序递增。每周发布几次扩展版本号，修正最新的稳定版本的问题。值得注意的是，“EXTRAVERSION”也可以不是数字，而是类似“-rc6”的字样，表示这是一个测试版本。在新的稳定版本发布之前，会先发布几个测试版本用于测试。

Linux 内核的最初版本在 1991 年发布，这是 Linus Torvalds 为他的 386 开发的一个类 Minix 的操作系统。

Linux 1.0 的官方版发行于 1994 年 3 月，包含了 386 的官方支持，仅支持单 CPU 系统。

Linux 1.2 发行于 1995 年 3 月，它是第一个包含多平台（Alpha，Sparc，Mips 等）支持

的官方版本。

Linux 2.0发行于1996年6月，包含很多新的平台支持，但是最重要的是，它是第一个支持SMP（对称多处理器）体系的内核版本。

Linux 2.2在1999年1月发布，它带来了SMP系统性能的极大提升，同时支持更多的硬件。

Linux 2.4于2001年1月发布，它进一步地提升了SMP系统的扩展性，同时它也集成了很多用于支持桌面系统的特性：USB、PC卡（PCMCIA）的支持，内置的即插即用等。

Linux 2.6于2003年12月发布，在Linux 2.4的基础上作了极大的改进。2.6内核支持更多的平台，从小规模的嵌入式系统到服务器级的64位系统；使用了新的调度器，进程的切换更高效；内核可被抢占，使得用户的操作可以得到更快速的响应；I/O子系统也经历很大的修改，使得它在各种工作负荷下都更具响应性；模块子系统、文件系统都做了大量的改进。另外，以前使用Linux的变种μClinux来支持没有MMU的处理器，现在2.6版本的Linux中已经合入了μClinux的功能，也可以支持没有MMU的处理器。

16.2 Linux移植准备

16.2.1 获取内核源码

登录Linux内核的官方网站http://www.kernel.org/，可以看到如图16.1所示的内容。

图16.1 kernel.org网站首页

上面标明了Linux内核的最新稳定版本、正在开发的测试版本，图中间的版本号就是各种补丁的链接地址。图16.1中各种标记符的意义如表16.1所示。

表16.1 kernel.org网站首页各标记符的意义

标 记	描 述
F	全部代码，比如在图中第一行，单击“F”可以下载2.6.22.6的完整代码
B	当前的补丁基于哪个版本的内核，单击“B”可以下载这个内核
V	查看补丁文件的信息，修改了哪些文件
VI	查看与上一个扩展版本相比，修改了哪些文件
C	当前修改的记录，它更新非常频繁，可以看到一天之内有几个更改记录
Changelog	这是正式的修改记录，由开发者提供

一般而言，各种补丁文件都是基内核的某个正式版本生成的，除非使用标记符“B”指明了它所基于的版本。比如有补丁文件 patch-2.6.xx.1、patch-2.6.xx.2、patch-2.6.xx.3，它们都是基于内核 2.6.xx 生成的补丁文件。使用时可以在内核 2.6.xx 上直接打补丁 patch-2.6.xx.3，并不需要先打上补丁 patch-2.6.xx.1、patch-2.6.xx.2；相应地，如果已经打了补丁 patch-2.6.xx.2，在打补丁 patch-2.6.xx.3 前，要先去除 patch-2.6.xx.2。

本书在 Linux 2.6.22.6 上进行移植、开发，下载 linux-2.6.22.6.tar.bz2 后如下解压即可得到目录 linux-2.6.22.6，里面存放了内核源码，如下所示：

```
$ tar xjf linux-2.6.22.6.tar.bz2
```

也可以先下载内核源文件 linux-2.6.22.tar.bz2、补丁文件 patch-2.6.22.6.bz2，然后解压、打补丁（假设源文件、补丁文件放在同一个目录下），如下所示：

```
$ tar xjf linux-2.6.22.tar.bz2
$ tar xjf patch-2.6.22.6.bz2
$ cd linux-2.6.22
$ patch -p1 < ../patch-2.6.22.6
```

以下，都假设内核源码所在目录为 linux-2.6.22.6。

16.2.2 内核源码结构及 Makefile 分析

1. 内核源码结构

Linux 内核文件数目将近 2 万，除去其他架构 CPU 的相关文件，支持 S3C2410、S3C2440 这两款芯片的完整内核文件有 1 万多个。这些文件的组织结构并不复杂，它们分别位于顶层目录下的 17 个子目录，各个目录功能独立。表 16.2 描述了各目录的功能，最后 2 个目录不包含内核代码。

表 16.2　　Linux 内核子目录结构

目 录 名	描　　述
arch	体系结构相关的代码，对于每个架构的 CPU，arch 目录下有一个对应的子目录，比如 arch/arm/、arch/i386/
block	块设备的通用函数
crypto	常用加密和散列算法（如 AES、SHA 等），还有一些压缩和 CRC 校验算法
drivers	所有的设备驱动程序，里面每一个子目录对应一类驱动程序，比如 drivers/block/为块设备驱动程序，drivers/char/为字符设备驱动程序，drivers/mtd/为 NOR Flash、NAND Flash 等存储设备的驱动程序
fs	Linux 支持的文件系统的代码，每个子目录对应一种文件系统，比如 fs/jffs2/、fs/ext2/、fs/ext3/
include	内核头文件，有基本头文件（存放在 include/linux/目录下）、各种驱动或功能部件的头文件（比如 include/media/、include/mtd/、include/net / ）、各种体系相关的头文件（比如 include/asm-arm/、include/asm-i386/）。当配置内核后，include/asm/是某个 include/asm-xxx/（比如 include/asm-arm/）的链接
init	内核的初始化代码（不是系统的引导代码），其中的 main.c 文件中的 start_kernel 函数是内核引导后运行的第一个函数
ipc	进程间通信的代码

续表

目 录 名	描　述
Kernel	内核管理的核心代码，与处理器相关的代码位于 arch/*/kernel/目录下
lib	内核用到的一些库函数代码，比如 crc32.c、string.c，与处理器相关的库函数代码位于 arch/*/lib/目录下
mm	内存管理代码，与处理器相关的内存管理代码位于 arch/*/mm/目录下
net	网络支持代码，每个子目录对应于网络的一个方面
security	安全、密钥相关的代码
sound	音频设备的驱动程序
usr	用来制作一个压缩的 cpio 归档文件：initrd 的镜像，它可以作为内核启动后挂接（mount）的第一个文件系统（一般用不到）
Documentation	内核文档
scripts	用于配置、编译内核的脚本文件

对于 ARM 架构的 S3C2410、S3C2440，其体系相关的代码在 arch/arm/目录下，在后面进行 Linux 移植时，开始的工作正是修改这个目录下的文件。如图 16.2 所示为内核代码的层次结构。

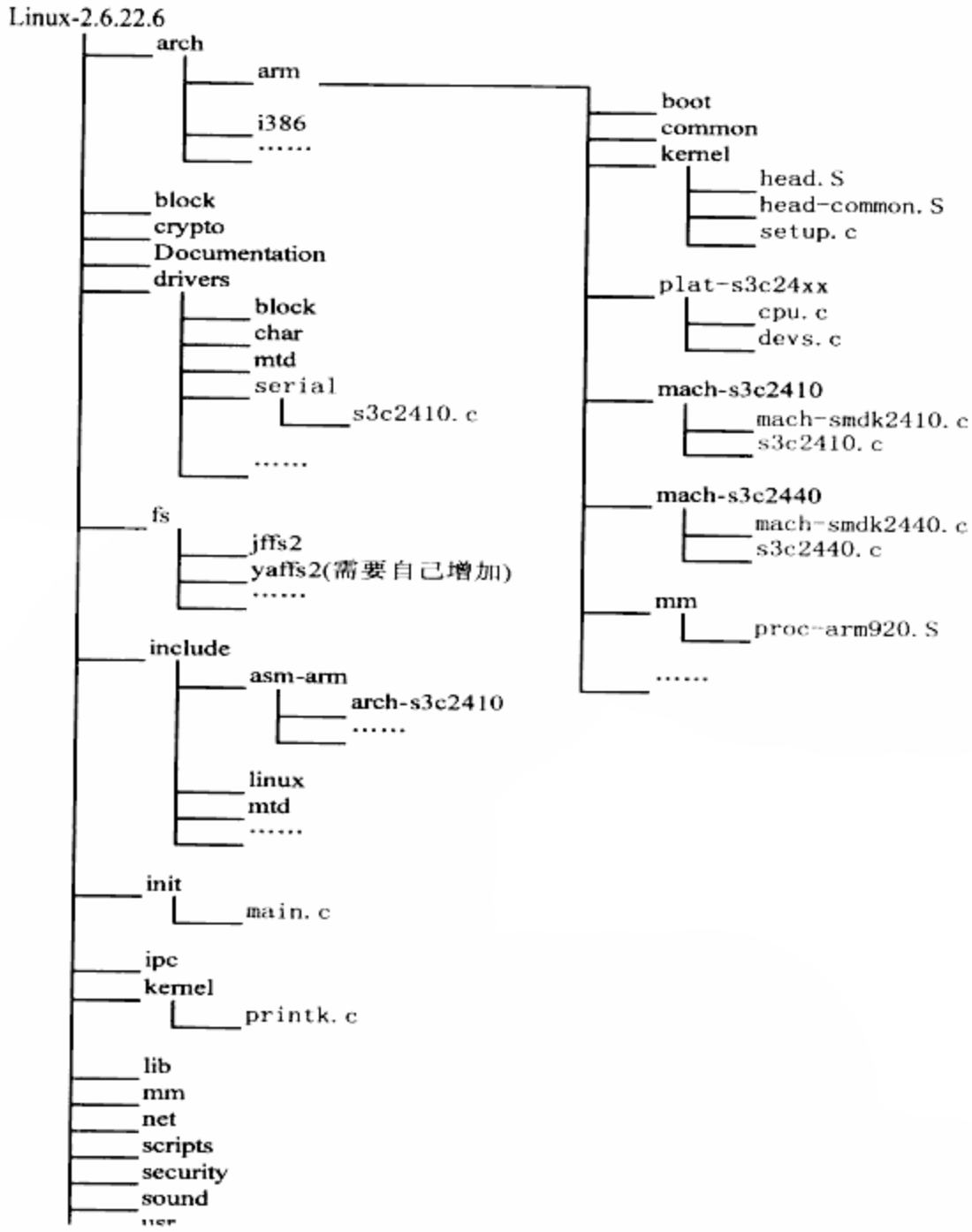


图 16.2　Linux 内核源码层次结构

2. Linux Makefile 分析

内核中的哪些文件将被编译？它们是怎样被编译的？它们连接时的顺序如何确定？哪个文件在最前面？哪些文件或函数先执行？这些都是通过 Makefile 来管理的。从最简单的角度来总结 Makefile 的作用，有以下 3 点。

（1）决定编译哪些文件。

（2）怎样编译这些文件？

（3）怎样连接这些文件，最重要的是它们的顺序如何？

Linux 内核源码中含有很多个 Makefile 文件，这些 Makefile 文件又要包含其他一些文件（比如配置信息、通用的规则等）。这些文件构成了 Linux 的 Makefile 体系，可以分为表 16.3 中的 5 类。

表 16.3　　Linux 内核 Makefile 文件分类

名　称	描　述
顶层 Makefile	它是所有 Makefile 文件的核心，从总体上控制着内核的编译、连接
.config	配置文件，在配置内核时生成。所有 Makefile 文件（包括顶层目录及各级子目录）都是根据.config 来决定使用哪些文件
arch/$(ARCH)/Makefile	对应体系结构的 Makefile，它用来决定哪些体系结构相关的文件参与内核的生成，并提供一些规则来生成特定格式的内核映象
scripts/Makefile.*	Makefile 共用的通用规则、脚本等
kbuild Makefiles	各级子目录下的 Makefile，它们相对简单，被上一层 Makefile 调用来编译当前目录下的文件

内核文档 Documentation/kbuild/makefiles.txt 对内核中 Makefile 的作用、用法讲解得非常透彻，以下根据前面总结的 Makefile 的 3 大作用分析这 5 类文件。

（1）决定编译哪些文件。

Linux 内核的编译过程从顶层 Makefile 开始，然后递归地进入各级子目录调用它们的 Makefile，分为 3 个步骤。

① 顶层 Makefile 决定内核根目录下哪些子目录将被编进内核。

② arch/$(ARCH)/Makefile 决定 arch/$(ARCH)目录下哪些文件、哪些目录将被编进内核。

③ 各级子目录下的 Makefile 决定所在目录下哪些文件将被编进内核，哪些文件将被编成模块（即驱动程序），进入哪些子目录继续调用它们的 Makefile。

先看步骤①，在顶层 Makefile 中可以看到如下内容：

```
433 init-y      := init/
434 drivers-y   := drivers/ sound/
435 net-y       := net/
436 libs-y      := lib/
437 core-y      := usr/
...
556 core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

可见，顶层 Makefile 将这 13 个子目录分为 5 类：init-y、drivers-y、net-y、libs-y 和 core-y。表 16.2 中有 17 个子目录，除去 include 目录和后面两个不包含内核代码的目录外，还有一个 arch 目录没有出现在内核中。它在 arch/$(ARCH)/Makefile 中被包含进内核，在顶层 Makefile 中直接包含了这个 Makefile，如下所示：

```
491 include $(srctree)/arch/$(ARCH)/Makefile
```

对于 ARCH 变量，可以在执行 make 命令时传入，比如“make ARCH=arm …”。另外，对于非 x86 平台，还需要指定交叉编译工具，这也可以在执行 make 命令时传入，比如“make CROSS_COMPILE=arm-linux- …”。为了方便，常在顶层 Makefile 中进行如下修改。

修改前：

```
185 ARCH            ?= $(SUBARCH)
186 CROSS_COMPILE   ?=
```

修改后：

```
185 ARCH            ?= arm
186 CROSS_COMPILE   ?= arm-linux-
```

对于步骤②的 arch/$(ARCH)/Makefile，以 ARM 体系为例，在 arch/arm/Makefile 中可以看到如下内容：

```
94 head-y       := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
...
171 core-y              += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
172 core-y              += $(MACHINE)
173 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2400/
174 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2412/
175 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2440/
...
191 libs-y             := arch/arm/lib/  $(libs-y)
...
```

从第 94 行可知，除前面的 5 类子目录外，又出现了另一类：head-y，不过它直接以文件名出现。MMUEXT 在 arch/arm/Makefile 前面定义，对于没有 MMU 的处理器，MMUEXT 的值为-nommu，使用文件 head-nommu.S；对于有 MMU 的处理器，MMUEXT 的值为空，使用文件 head.S。

arch/arm/Makefile 中类似第 171、172、173 行的代码进一步扩展了 core-y 的内容，第 191 行扩展了 libs-y 的内容，这些都是体系结构相关的目录。第 173～175 行中的 CONFIG_ARCH_S3C2410 在配置内核时定义，它的值有 3 种：y、m 或空。y 表示编进内核，m 表示编为模块，空表示不使用。

编译内核时，将依次进入 init-y、core-y、libs-y、drivers-y 和 net-y 所列出的目录中执行它们的 Makefile，每个子目录都会生成一个 built-in.o（libs-y 所列目录下，有可能生成 lib.a 文

件)。最后，head-y 所表示的文件将和这些 built-in.o、lib.a 一起被连接成内核映象文件 vmlinux。

最后，看一下步骤③是怎么进行的。

在配置内核时，生成配置文件.config（具体的配置过程在 16.2.3 小节讲述)。内核顶层 Makefile 使用如下语句间接包含.config 文件，以后就根据.config 中定义的各个变量决定编译哪些文件。之所以说是“间接”包含，是因为包含的是 include/config/auto.conf 文件，而它只是将.config 文件中的注释去掉，并根据顶层 Makefile 中定义的变量增加了一些变量而已。

```
441 # Read in config
442 -include include/config/auto.conf
```

include/config/auto.conf 文件的生成过程不再描述，它与.config 的格式相同，摘选部分内容如下（注意，下面以“#”开头的行是本书加的注释)：

```
CONFIG_ARCH_SMDK2410=y
CONFIG_ARCH_S3C2440=y
# .config 中没有下面这行，它是根据顶层 Makefile 中定义的内核版本号增加的
CONFIG_KERNELVERSION="2.6.22.6"
# .config 中没有下面这行，它是根据顶层 Makefile 中定义的 ARCH 变量增加的
CONFIG_ARCH="arm"
CONFIG_JFFS2_FS=y
CONFIG_LEDS_S3C24XX=m
```

在 include/config/auto.conf 文件中，变量的值主要有两类：“y”和“m”。各级子目录的 Makefile 使用这些变量来决定哪些文件被编进内核中，哪些文件被编成模块（即驱动程序)，要进入哪些下一级子目录继续编译，这通过以下 4 种方法来确定(obj-y、obj-m、lib-y 是 Makefile 中的变量)。

① obj-y 用来定义哪些文件被编进（built-in）内核。

obj-y 中定义的.o 文件由当前目录下的.c 或.S 文件编译生成，它们连同下级子目录的 built-in.o 文件一起被组合成(使用“$(LD) -r”命令)当前目录下的 built-in.o 文件。这个 built-in.o 文件将被它的上一层 Makefile 使用。

obj-y 中各个.o 文件的顺序是有意义的，因为内核中用 module_init()或__initcall 定义的函数将按照它们的连接顺序被调用。

例 16.1：当下面的 CONFIG_ISDN、CONFIG_ISDN_PPP_BSDCOMP 在.config 中被定义为 y 时，isdn.c 或 isdn.S、isdn_bsdcomp.c 或 isdn_bsdcomp.S 被编译成 isdn.o、isdn_bsdcomp.o。这两个.o 文件被组合进 built-in.o 文件中，最后被连接进入内核。假如 isdn.o、isdn_bsdcomp.o 中分别用 module_init(A)、module_init(B)定义了函数 A、B，则内核启动时 A 先被调用，然后是 B。

```
obj-$(CONFIG_ISDN)              += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

② obj-m 用来定义哪些文件被编译成可加载模块（Loadable module)。

obj-m 中定义的.o 文件由当前目录下的.c 或.S 文件编译生成，它们不会被编进 built-in.o 中，而是被编成可加载模块。

一个模块可以由一个或几个.o 文件组成。对于只有一个源文件的模块，在 obj-m 中直接增加它的.o 文件即可。对于有多个源文件的模块，除在 obj-m 中增加一个.o 文件外，还要定义一个<module_name>-objs 变量来告诉 Makefile 这个.o 文件由哪些文件组成。

例 16.2：当下面的 CONFIG_ISDN_PPP_BSDCOMP 在.config 文件中被定义为 m 时，isdn_bsdcomp.c 或 isdn_bsdcomp.S 将被编译成 isdn_bsdcomp.o 文件，它最后被制作成 isdn_bsdcomp.ko 模块，如下所示：

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

例 16.3：当下面的 CONFIG_ISDN 在.config 文件中被定义为 m 时，将会生成一个 isdn.o 文件，它由 isdn-objs 中定义的 isdn_net_lib.o、isdn_v110.o、isdn_common.o 等 3 个文件组合而成。isdn.o 最后被制作成 isdn.ko 模块。

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

③ lib-y 用来定义哪些文件被编成库文件。

lib-y 中定义的.o 文件由当前目录下的.c 或.S 文件编译生成，它们被打包成当前目录下的一个库文件：lib.a。

同时出现在 obj-y、lib-y 中的.o 文件，不会被包含进 lib.a 中。

要把这个 lib.a 编进内核中，需要在顶层 Makefile 中 libs-y 变量中列出当前目录。要编成库文件的内核代码一般都在这两个目录下：lib/、arch/$(ARCH)/lib/。

④ obj-y、obj-m 还可以用来指定要进入的下一层子目录。

Linux 中一个 Makefile 文件只负责生成当前目录下的目标文件，子目录下的目标文件由子目录的 Makefile 生成。Linux 的编译系统会自动进入这些子目录调用它们的 Makefile，只是在这之前指定这些子目录。

这要用到 obj-y、obj-m，只要在其中增加这些子目录名即可。

例 16.4：fs/Makefile 中有如下一行，当 CONFIG_JFFS2_FS 被定义为 y 或 m 时，在编译时将会进入 jffs2/目录进行编译。Linux 的编译系统只会根据这些信息决定是否进入下一级目录，而下一级中的文件如何编译成 built-in.o 或模块由它的 Makefile 决定。

```
101 obj-$(CONFIG_JFFS2_FS)      += jffs2/
```

（2）怎样编译这些文件。

即编译选项、连接选项是什么。这些选项分 3 类：全局的，适用于整个内核代码树；局部的，仅适用于某个 Makefile 中的所有文件；个体的，仅适用于某个文件。

全局选项在顶层 Makefile 和 arch/$(ARCH)/Makefile 中定义，这些选项的名称为：CFLAGS、AFLAGS、LDFLAGS、ARFLAGS，它们分别是编译 C 文件的选项、编译汇编文件的选项、连接文件的选项、制作库文件的选项。

需要使用局部选项时，它们在各个子目录中定义，名称为：EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS，它们的用途与前述选项相同，

只是适用范围比较小，它们针对当前 Makefile 中的所有文件。

另外，如果想针对某个文件定义它的编译选项，可以使用 CFLAGS_$@，AFLAGS_$@。前者用于编译某个 C 文件，后者用于编译某个汇编文件。$@表示某个目标文件名，比如以下代码表示编译 aha152x.c 时，选项中要额外加上“-DAHA152X_STAT -DAUTOCONF”。

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
```

需要注意的是，这 3 类选项是一起使用的，在 scripts/Makefile.lib 中可以看到。

```
_c_flags       = $(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$(basetarget).o)
```

（3）怎样连接这些文件，它们的顺序如何。

前面分析有哪些文件要编进内核时，顶层 Makefile 和 arch/$(ARCH)/Makefile 定义了 6 类目录（或文件）：head-y、init-y、drivers-y、net-y、libs-y 和 core-y。它们的初始值如下（以 ARM 体系为例）。

arch/arm/Makefile 中：

```
94 head-y       := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
...
171 core-y              += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
172 core-y              += $(MACHINE)
173 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2400/
174 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2412/
175 core-$(CONFIG_ARCH_S3C2410) += arch/arm/mach-s3c2440/
...
191 libs-y             := arch/arm/lib/  $(libs-y)
...
```

顶层 Makefile 中：

```
433 init-y      := init/
434 drivers-y   := drivers/ sound/
435 net-y       := net/
436 libs-y      := lib/
437 core-y      := usr/
...
556 core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

可见，除 head-y 外，其余的 init-y、drivers-y 等都是目录名。在顶层 Makefile 中，这些目录名的后面直接加上 built-in.o 或 lib.a，表示要连接进内核的文件，如下所示：

```
567 init-y      := $(patsubst %/, %/built-in.o, $(init-y))
568 core-y      := $(patsubst %/, %/built-in.o, $(core-y))
```

```
569 drivers-y   := $(patsubst %/, %/built-in.o, $(drivers-y))
570 net-y       := $(patsubst %/, %/built-in.o, $(net-y))
571 libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
572 libs-y2     := $(patsubst %/, %/built-in.o, $(libs-y))
573 libs-y      := $(libs-y1) $(libs-y2)
```

上面的 patsubst 是个字符串处理函数，它的用法如下：

```
$(patsubst pattern,replacement,text)
```

表示寻找“text”中符合格式“pattern”的字，用“replacement”替换它们。比如上面的 init-y 初值为“init/”，经过第 567 行的交换后，“init-y”变为“init/built-in.o”。

顶层 Makefile 中，再往下看。

```
602 vmlinux-init := $(head-y) $(init-y)
603 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
604 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
605 vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

第 604 行的 vmlinux-all 表示所有构成内核映象文件 vmlinux 的目标文件，从第 602～604 行可知这些目标文件的顺序为：head-y、init-y、core-y、libs-y、drivers-y、net-y，即 arch/arm/kernel/head.o（假设有 MMU，否则为 head-nommu.o）、arch/arm/kernel/ init_task.o、init/built-in.o、usr/built-in.o 等。

第 605 行表示连接脚本为 arch/$(ARCH)/kernel/vmlinux.lds。对于 ARM 体系，连接脚本就是 arch/arm/kernel/vmlinux.lds，它由 arch/arm/kernel/vmlinux.lds.S 文件生成，规则在 scripts/Makefile.build 中，如下所示：

```
248 $(obj)/%.lds: $(src)/%.lds.S FORCE
249     $(call if_changed_dep,cpp_lds_S)
250
```

现将生成的 arch/arm/kernel/vmlinux.lds 摘录如下：

```
286 SECTIONS
287 {
...
291  . = (0xc0000000) + 0x00008000; /* 代码段起始地址，这是个虚拟地址 */
292
293  .text.head : {
294   _stext = .;
295   _sinittext = .;
296   *(.text.head)
297  }
298
```

```
299  .init : { /* 内核初始化的代码和数据 */
...
343  }
344
...
355  .text : { /* 真正的代码段 */
356   _text = .; /* 代码段和只读数据段的开始地址 */
...
372  }
373  /* 只读数据 */
374  . = ALIGN((4096)); .rodata : AT(ADDR(.rodata) - 0) {……} . = ALIGN((4096));
375
376  _etext = .; /* 代码段和只读数据段的结束地址 */
……
386  .data : AT(__data_loc) {    /* 数据段 */
387   __data_start = .; /* 数据段起始地址 */
……
422   _edata = .;    /* 数据段结束地址 */
423  }
424  _edata_loc = __data_loc + SIZEOF(.data);    /* 数据段结束地址 */
425
426  .bss : {    /* BSS 段，没有初化或初值为 0 的全局、静态变量 */
427   __bss_start = .;   /* BSS 段起始地址 */
428   *(.bss)
429   *(COMMON)
430   _end = .;      /* BSS 段结束地址 */
431  }
432     /* 调试信息段 */
433  .stab 0 : { *(.stab) }
……
440 }
```

下面对本节分析 Makefile 的结果作一下总结。

（1）配置文件.config 中定义了一系列的变量，Makefile 将结合它们来决定哪些文件被编进内核、哪些文件被编成模块、涉及哪些子目录。

（2）顶层 Makefile 和 arch/$(ARCH)/Makefile 决定根目录下哪些子目录、arch/$(ARCH)目录下哪些文件和目录将被编进内核。

（3）最后，各级子目录下的 Makefile 决定所在目录下哪些文件将被编进内核，哪些文件将被编成模块（即驱动程序），进入哪些子目录继续调用它们的 Makefile。

（4）顶层 Makefile 和 arch/$(ARCH)/Makefile 设置了可以影响所有文件的编译、连接选

项：CFLAGS、AFLAGS、LDFLAGS、ARFLAGS。

（5）各级子目录下的 Makefile 中可以设置能够影响当前目录下所有文件的编译、连接选项：EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS；还可以设置可以影响某个文件的编译选项：CFLAGS_$@，AFLAGS_$@。

（6）顶层 Makefile 按照一定的顺序组织文件，根据连接脚本 arch/$(ARCH)/ kernel/vmlinux.lds 生成内核映象文件 vmlinux。

16.2.3　内核的 Kconfig 分析

在内核目录下执行“make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-”时，就会看到一个如图 16.3 所示的菜单，这就是内核的配置界面。通过配置界面，可以选择芯片类型、选择需要支持的文件系统，去除不需要的选项等，这就称为“配置内核”。注意，也有其他形式的配置界面，比如“make config”命令启动字符配置界面，对于每个选项都会依次出现一行提示信息，逐个回答；“make xconfig”命令启动 X-windows 图形配置界面。

所有配置工具都是通过读取 arch/$(ARCH)/Kconfig 文件来生成配置界面，这个文件是所有配置文件的总入口，它会包含其他目录的 Kconfig 文件。配置界面如图 16.3 所示。

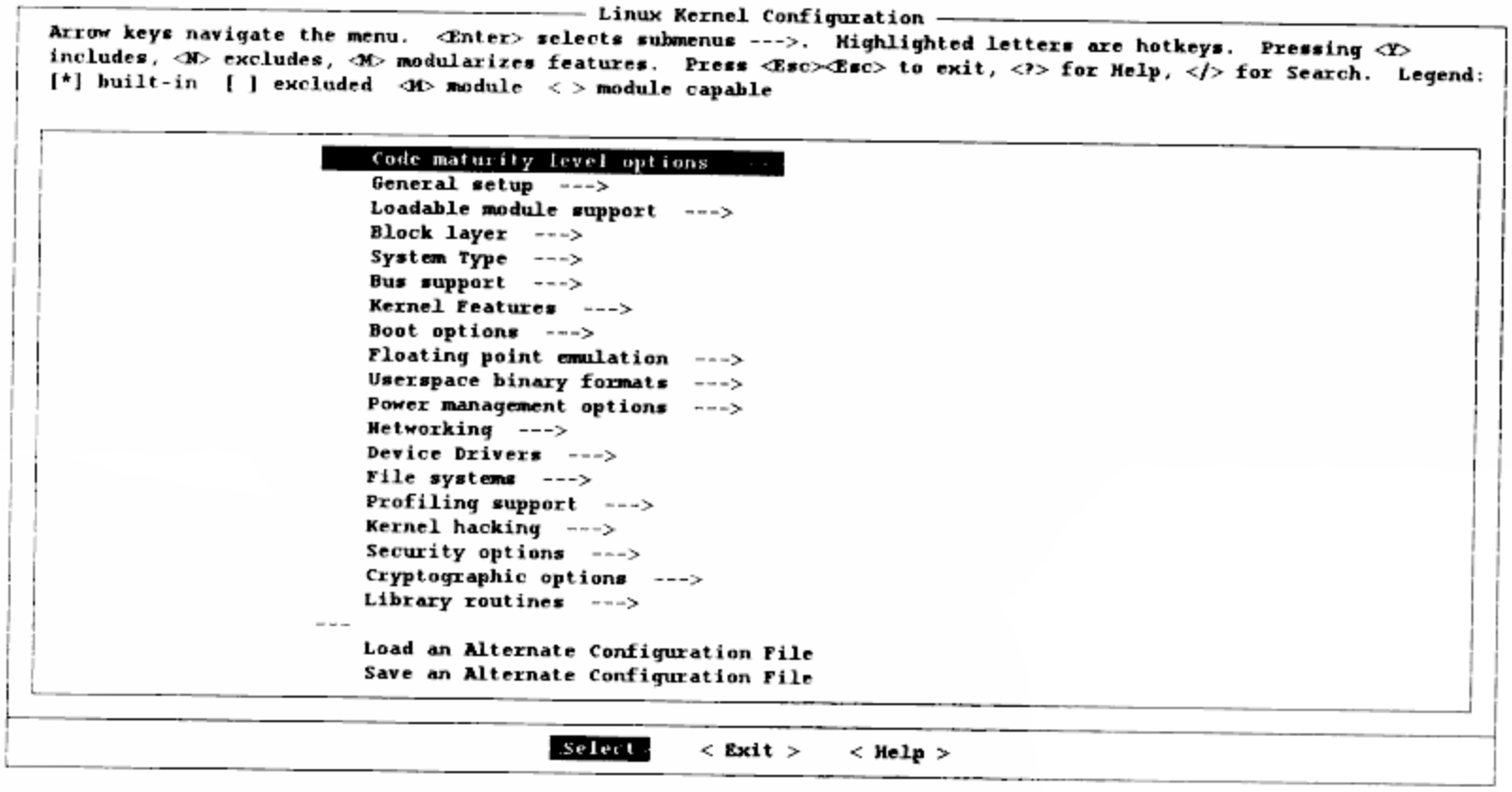


图 16.3　内核配置界面（菜单形式）

内核源码每个子目录中，都有一个 Makefile 文件和 Kconfig 文件。Makefile 的作用前面已经讲述，Kconfig 用于配置内核，它就是各种配置界面的源文件。内核的配置工具读取各个 Kconfig 文件，生成配置界面供开发人员配置内核，最后生成配置文件.config。

内核的配置界面以树状的菜单形式组织，主菜单下有若干个子菜单，子菜单下又有子菜单或配置选项。每个子菜单或选项可以有依赖关系，这些依赖关系用来确定它们是否显示。只有被依赖项的父项已经被选中，子项才会显示。

Kconfig 文件的语法可以参考 Documentation/kbuild/kconfig-language.txt 文件，下面讲述几个常用的语法，并在最后介绍菜单形式的配置界面操作方法。

1. Kconfig 文件的基本要素：config 条目（entry）

config 条目常被其他条目包含，用来生成菜单、进行多项选择等。

config 条目用来配置一个选项，或者这么说，它用于生成一个变量，这个变量会连同它的值一起被写入配置文件.config 中。比如有一个 config 条目用来配置 CONFIG_LEDS_S3C24XX，根据用户的选择，.config 文件中可能出现下面 3 种配置结果中的一个。

```
CONFIG_LEDS_S3C24XX=y        # 对应的文件被编进内核
CONFIG_LEDS_S3C24XX=m        # 对应的文件被编成模块
#CONFIG_LEDS_S3C24XX          # 对应的文件没有被使用
```

以一个例子说明 config 条目格式，下面代码选自 fs/Kconfig 文件，它用于配置 CONFIG_JFFS2_FS_POSIX_ACL 选项。

```
1255 config JFFS2_FS_POSIX_ACL
1256     bool "JFFS2 POSIX Access Control Lists"
1257     depends on JFFS2_FS_XATTR
1258     default y
1259     select FS_POSIX_ACL
1260     help
1261       Posix Access Control Lists (ACLs) support permissions for users and
1262       groups beyond the owner/group/world scheme.
1263
1264       To learn more about Access Control Lists, visit the Posix ACLs for
1265       Linux website <http://acl.bestbits.at/>.
1266
1267       If you don't know what Access Control Lists are, say N
1268
```

代码中包含了几乎所有的元素，下面一一说明。

第 1255 行中，config 是关键字，表示一个配置选项的开始；紧跟着的 JFFS2_FS_POSIX_ACL 是配置选项的名称，省略了前缀“CONFIG_”。

第 1256 行中，boot 表示变量类型，即 CONFIG_JFFS2_FS_POSIX_ACL 的类型。有 5 种类型：bool、tristate、string、hex 和 int，其中的 tristate 和 string 是基本的类型，其他类型是它们的变种。boot 变量取值有两种：y 和 n；tristate 变量取值有 3 种：y、n 和 m；string 变量取值为字符串；hex 变量取值为十六进制的数据；int 变量取值为十进制的数据。

“boot”之后的字符串是提示信息，在配置界面中上下移动光标选中它时，就可以通过按空格或回车键来设置 CONFIG_JFFS2_FS_POSIX_ACL 的值。提示信息的完整格式如下，如果使用“if <expr>”，则当 expr 为真时才显示提示信息。在实际使用时，prompt 关键字可以省略。

```
"prompt" <prompt> ["if" <expr>]
```

第 1257 行表示依赖关系，格式如下。只有 JFFS2_FS_XATTR 配置选项被选中时，当前配置选项的提示信息才会出现，才能设置当前配置选项。注意，如果依赖条件不满足，则它取默认值。

```
"depends on"/"requires" <expr>
```

第 1258 行的表示默认值为 y，格式如下：

```
"default" <expr> ["if" <expr>]
```

第 1259 行表示当前配置选项 FFS2_FS_POSIX_ACL 被选中时，配置选项 FS_POSIX_ACL 也会被自动选中，格式如下：

```
"select" <symbol> ["if" <expr>]
```

第 1260 行表示下面几行是帮助信息，帮助信息的关键字有如下两种，它们完全一样。当遇到一行的缩进距离比第一行帮助信息的缩进距离小时，表示帮助信息已经结束。比如第 1268 行的缩进距离比第 126 的缩进距离小，帮助信息到第 1267 行结束。

```
"help" or "---help---"
```

注意　除第 1255 行的关键字及配置选项的名称、第 256 行的变量类型外，其他信息都是可以省略的。

2. menu 条目

menu 条目用于生成菜单，格式如下：

```
"menu" <prompt>
<menu options>
<menu block>
"endmenu"
```

它的实际使用并不如它的标准格式那样复杂，下面是一个例子。

```
menu "Floating point emulation"

config FPE_NWFPE
        ……
config FPE_NWFPE_XP
        ……
……
endmenu
```

menu 之后的字符串是菜单名，“menu”和“endmenu”之间有很多 config 条目。在配置界面上会出现如下字样的菜单，移动光标选中它后按回车键进入，就会看到这些 config 条目定义的配置选项。

```
Floating point emulation  --->
```

3. choice 条目

choice 条目将多个类似的配置选项组合在一起，供用户单选或多选，格式如下：

```
"choice"
<choice options>
<choice block>
"endchoice"
```

实际使用中，也是在“choice”和“endchoice”之间定义多个 config 条目，比如 arch/arm/Kconfig 中有如下代码：

```
choice
        prompt "ARM system type"
        default ARCH_VERSATILE

config ARCH_AAEC2000
        ...
config ARCH_INTEGRATOR
        ...
endchoice
```

prompt "ARM system type"给出提示信息“ARM system type”，光标选中它后按回车键进入，就可以看到多个 config 条目定义的配置选项。

choice 条目中定义的变量类型只能有两种：bool 和 tristate，不能同时有这两种类型的变量。对于 bool 类型的 choice 条目，只能在多个选项中选择一个；对于 tristate 类型的 choice 条目，要么就把多个（可以是一个）选项都设为 m；要么就像 bool 类型的 choice 条目一样，只能选择一个。这是可以理解的，比如对于同一个硬件，它有多个驱动程序，可以选择将其中之一编译进内核中（配置选项设为 y），或者把它们都编译为模块（配置选项设为 m）。

4. comment 条目

comment 条目用于定义一些帮助信息，它在配置过程中出现在界面的第一行；并且这些帮助信息会出现在配置文件中（作为注释），格式如下：

```
"comment" <prompt>
<comment options>
```

实际使用中也很简单，比如 arch/arm/Kconfig 中有如下代码：

```
menu "Floating point emulation"

comment "At least one emulation must be selected"
...
```

进入菜单“Floating point emulation　--->”之后，在第一行会看到如下内容：

```
--- At least one emulation must be selected
```

而在.config 文件中也会看到如下内容：

```
#
# At least one emulation must be selected
#
```

5. source 条目

source 条目用于读入另一个 Kconfig 文件，格式如下：

```
"source" <prompt>
```

下面是一个例子，取自 arch/arm/Kconfig 文件，它读入 net/Kconfig 文件。

```
source "net/Kconfig"
```

6. 菜单形式的配置界面操作方法

配置界面的开始几行就是它的操作方法，如图 16.4 所示。

```
                    Linux Kernel Configuration
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
built-in  [ ] excluded  <M> module  < > module capable
```

图 16.4　菜单形式的配置界面的操作方法

内核 scripts/kconfig/mconf.c 文件的注释给出了更详细的操作方法，讲解如下。

一些特定功能的文件可以直接编译进内核中，或者编译成一个可加载模块，或者根本不使用它们。还有一些内核参数，必须给它们赋一个值：十进制数、十六进制数，或者一个字符串。

配置界面中，以[*]、<M>或[ ]开头的选项表示相应功能的文件被编译进内核中、被编译成一个模块，或者没有使用。尖括号<>表示相应功能的文件可以被编译成模块。

要修改配置选项，先使用方向键高亮选中它，按<Y>键选择将它编译进内核，按<M>键选择将它编译成模块，按<N>键将不使用它。也可以按空格键进行循环选择，例如：Y→N→M→Y。

上/下方向键用来高亮选中某个配置选项，如果要进入某个子菜单，先选中它，然后按回车键进入。配置选项的名字后有“--->”表示它是一个子菜单。配置选项的名称中有一个高亮的字母，它被称为“热键”（hotkey），直接输入热键就可以选中该配置选项，或者循环选中具有相同热键的配置选项。

可以使用翻页键<PAGE UP>和<PAGE DOWN>来移动配置界面中的内容。

要退出配置界面，使用左/右方向键选中<Exit>按钮，然后按回车键。如果没有配置选项使用后面这些按键作为热键的话，也可以按两次<ESC>键或<E> <X>键退出。

按<TAB>键可以在<Select>、<Exit>和<Help>这 3 个按钮中循环选中它们。

要想阅读某个配置选项的帮助信息，选中它之后，再选中<Help>按钮，按回车键；也可以选中配置选项后，直接按<H>或<?>键。

对于 choise 条目中的多个配置选项，使用方向键高亮选中某个配置选项，按<S>或空格

键选中它；也可以通过输入配置选项的首字母，然后按<S>或空格键选中它。

对于 int、hex 或 string 类型的配置选项，要输入它们的值时，先高亮选中它，按回车键，输入数据，再按回车键。对于十六进制数据，前缀 0x 可以省略。

配置界面的最下面，有如下两行：

```
Load an Alternate Configuration File
Save an Alternate Configuration File
```

前者用于加载某个配置文件，后者用于将当前的配置保存到某个配置文件中去。需要注意的是，如果不使用这两个选项，配置的加载文件、输出文件都默认为.config 文件；如果加载了其他的文件（假设文件名为 A），然后在它的基础上进行修改，最后退出保存时，这些变动会保存到 A 中去，而不是.config。

当然，可以先加载（Load an Alternate Configuration File）文件 A，然后修改，最后保存（Save an Alternate Configuration File）到.config 中去。

16.2.4 Linux 内核配置选项

Linux 内核配置选项多达上千个，一个个地进行选择既耗费时间，对开发人员的要求也比较高（需要了解每个配置选项的作用）。一般的做法是在某个默认配置文件的基础上进行修改，比如我们可以先加载配置文件 arch/arm/configs/s3c2410_defconfig，再增加、去除某些配置选项。

下面分 3 部分介绍内核配置选项，先从整体介绍主菜单的类别，然后分别介绍和移植系统关系比较密切的“System Type”、“Device Drivers”菜单。

1. 配置界面主菜单的类别

表 16.4 讲解了主菜单的类别，以后读者配置内核时，可以根据自己所要设置的功能进入某个菜单，然后根据其中各个配置选项的帮助信息进行配置。

表 16.4　　配置界面主菜单的类别/功能

配置界面主菜单	描　　述
Code maturity level options	代码成熟度选项：用于包含一些正在开发的或者不成熟的代码、驱动程序。一般不设置
General setup	常规设置：比如增加附加的内核版本号、支持内存页交换（swap）功能、System V 进程间通信等。除非很熟悉其中的内容，否则一般使用默认配置
Loadable module support	可加载模块支持：一般都会打开可加载模块支持（Enable loadable module support）、允许卸载已经加载的模块（Module unloading）、让内核通过运行 modprobe 来自动加载所需要的模块（Automatic kernel module loading）
Block layer	块设备层：用于设置块设备的一些总体参数，比如是否支持大于 2TB 的块设备、是否支持大于 2TB 的文件、设置 I/O 调度器等。一般使用默认值即可
System Type	系统类型：选择 CPU 的架构、开发板类型等与开发板相关的配置选项
Bus support	PCMCIA/CardBus 总线的支持，对于本书的开发板，不用设置
Kernel Features	用于设置内核的一些参数，比如是否支持内核抢占（这对实时性有帮助）、是否支持动态修改系统时钟（timer tick）等
Boot options	启动参数：比如设置默认的命令行参数等，一般不用理会

续表

配置界面主菜单	描　述
Floating point emulation	浮点运算仿真功能：目前 Linux 还不支持硬件浮点运算，所以要选择一个浮点仿真器，一般选择“NWFPE math emulation”
Userspace binary formats	可执行文件格式：一般都选择支持 ELF、a.out 格式
Power management options	电源管理选项
Networking	网络协议选项：一般都选择“Networking support”以支持网络功能，选择“Packet socket”以支持 socket 接口功能，选择“TCP/IP networking”以支持 TCP/IP 网络协议。通常可以在选择“Networking support”后，使用默认配置
Device Drivers	设备驱动程序：几乎包含了 Linux 的所有驱动程序
File systems	文件系统：可以在里面选择要支持的文件系统，比如 EXT2、JFFS2 等
Profiling support	对系统的活动进行分析，仅供内核开发者使用
Kernel hacking	调试内核时的各种选项
Security options	安全选项：一般使用默认配置
Cryptographic options	加密选项
Library routines	库子程序：比如 CRC32 检验函数、zlib 压缩函数等。不包含在内核源码中的第三方内核模块可能需要这些库，可以全不选，内核中若有其他部分依赖它，会自动选上

2. “System Type”菜单：系统类型

对于 arm 平台（在顶层 Makefile 中修改“ARCH ?= arm”），执行“make menuconfig”后在配置界面可以看到“System Type”字样，进入它得到另一个界面，如图 16.5 所示。

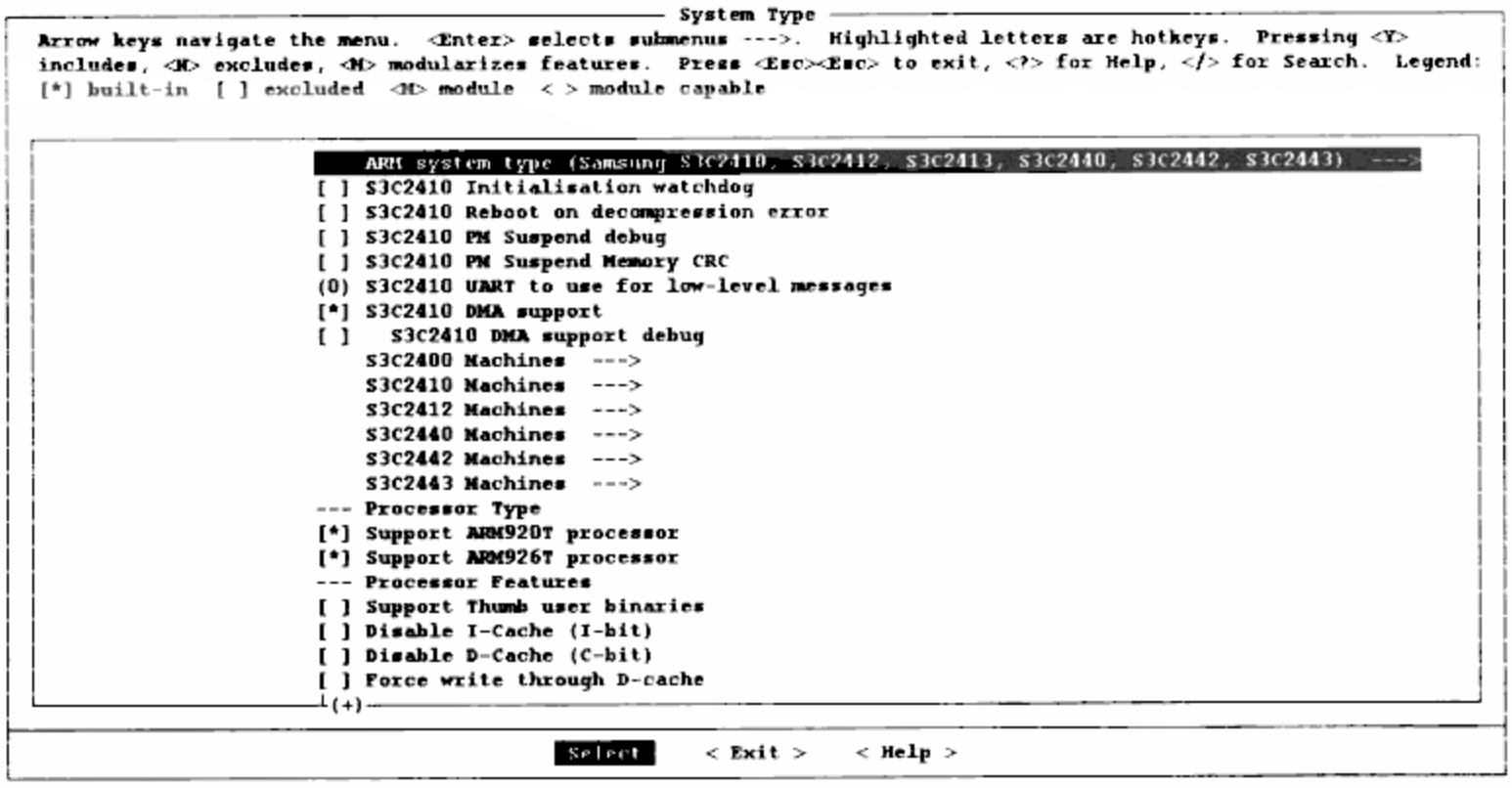


图 16.5　“System Type”菜单的配置界面

第一行“ARM system type”用来选择体系结构，进入它之后选中“Samsung S3C2410, S3C2412, S3C2413, S3C2440, S3C2442, S3C2443”，查看帮助信息可以知道它对应 CONFIG_ARCH_S3C2410 配置项。

下面几行用来设置 S3C2410（包括 S3C2412 等）系统的特性，比如选中“S3C2410 UART to use for low-level messages”后按回车键，可以输入数字，表示使用哪个串口来输入内核打印信息；选中“S3C2410 DMA support”表示支持 DAM 功能。

再往下的“S3C2410 Machines ---->”、“S3C2440 Machines ---->”表示这又是一个菜单，它们用来选择开发板类型。比如进入“S3C2410 Machines”菜单后，可以看到如下内容：

```
[*] SMDK2410/A9M2410
[ ] IPAQ H1940
[ ] Acer N30
[ ] Simtec Electronics BAST (EB2410ITX)
[ ] NexVision OTOM Board
[ ] AML M5900 Series
[ ] Thorcom VR1000
[*] QT2410
```

它们表示目前内核中支持 S3C2410 的 8 种开发板。选中某个开发板后，它相应的文件就会被编译进内核中。比如对于开发板“SMDK2410/A9M2410”，它的配置项为 CONFIG_ARCH_SMDK2410（可以查看帮助信息知道这点），在 arch/arm/mach-s3c2410/Makefile 中可以看到如下一行，表示如果选择支持该开发板，则 arch/arm/mach-s3c2410/mach-smdk2410.c 文件被编进内核中。

```
obj-$(CONFIG_ARCH_SMDK2410)     += mach-smdk2410.o
```

在移植内核时，可以选中某个配置相似的开发板，然后在上面进行修改。

后面的内容一看名字就可以了解它们的功能，不再介绍。

3. “Device Drivers”菜单：设备驱动程序

执行“make menuconfig”后在配置界面可以看到“Device Drivers”字样，进入它则进入如图 16.6 所示界面。

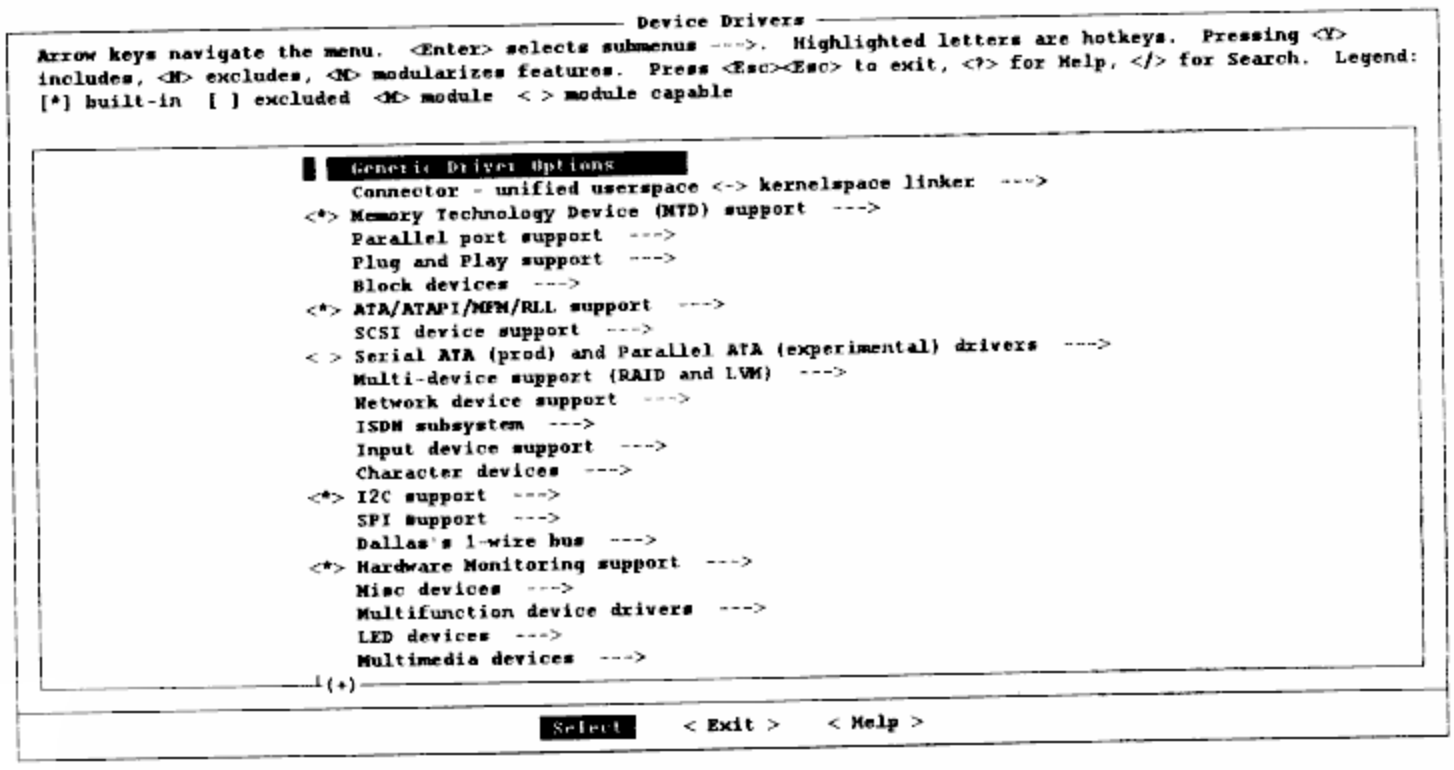


图 16.6 “Device Drivers”菜单的配置界面

图 16.6 中的各个子菜单与内核源码 drivers/目录下各个子目录一一对应，如表 16.5 所示。在配置过程中可以参考这个表格找到对应的配置选项；在添加新驱动时，也可以参考它来决定代码放在哪个目录下。

表 16.5　　设备驱动程序配置子菜单分类/功能

“Device Drivers”子菜单	描　述
Generic Driver Options	对应 drivers/base 目录，这是设备驱动程序中一些基本和通用的配置选项
Connector – unified userspace <-> kernelspace linker	对应 drivers/connector 目录，一般不用理会
Memory Technology Device (MTD) support	对应 drivers/mtd 目录，它用于支持各种新型的存储设备，比如 NOR Flash、NAND Flash 等
Parallel port support	对应 drivers/parport 目录，它用于支持各种并口设备，在一般嵌入式开发板中用不到
Plug and Play support	对应 drivers/pnp 目录，支持各种“即插即用”的设备
Block devices	对应 drivers/block 目录，包括回环设备、RAMDISK 等的驱动
ATA/ATAPI/MFM/RLL support	对应 drivers/ide 目录，它用来支持 ATA/ATAPI/MFM/RLL 接口的硬盘、软盘、光盘等
SCSI device support	对应 drivers/scsi 目录，支持各种 SCSI 接口的设备
Serial ATA (prod) and Parallel ATA (experimental) drivers	对应 drivers/ata 目录，支持 SATA 与 PATA 设备
Multi-device support (RAID and LVM)	对应 drivers/md 目录，表示多设备支持(RAID 和 LVM)。RAID 和 LVM 的功能是使多个物理设备组建成一个单独的逻辑磁盘
Network device support	对应 drivers/net 目录，用来支持各种网络设备，比如 CS8900、DM9000 等
ISDN subsystem	对应 drivers/isdn 目录，用来提供综合业务数字网（Integrated Service Digital Network）的驱动程序
Input device support	对应 drivers/input 目录，支持各类输入设备，比如键盘、鼠标等
Character devices	对应 drivers/char 目录，它包含各种字符设备的驱动程序。串口的配置选项也是从这个菜单调用的，但是串口的代码在 drivers/serial 目录下
I2C support	对应 drivers/i2c 目录，支持各类 I^2C 设备
SPI support	对应 drivers/spi 目录，支持各类 SPI 设备
Dallas's 1-wire bus	对应 drivers/w1 目录，支持一线总线。
Hardware Monitoring support	对应 drivers/hwmon 目录。当前主板大多都有一个监控硬件健康的设备用于监视温度/电压/风扇转速等，这些功能需要 I^2C 的支持。在嵌入式开发板中一般用不到
Misc devices	对应 drivers/misc 目录，用来支持一些不好分类的设备，称为杂项设备

续表

“Device Drivers”子菜单	描　述
Multifunction device drivers	对应 drivers/mfd 目录，用来支持多功能的设备，比如 SM501，它既可用于显示图像，也可以用作串口等
LED devices	对应 drivers/leds 目录，包含各种 LED 驱动程序
Multimedia devices	对应 drivers/media 目录，包含多媒体驱动，比如 V4L（Video for Linux），它用于向上提供统一的图像、声音接口
Graphics support	对应 drivers/video 目录，提供图形设备/显卡的支持
Sound	对应 sound/目录（它不在 drivers/目录下），用来支持各种声卡
HID Devices	对应 drivers/hid 目录，用来支持各种 USB-HID 设备，或者符合 USB-HID 规范的设备（比如蓝牙设备）。HID 表示 human interface device，比如各种 USB 接口的鼠标/键盘/游戏杆/手写板等输入设备
USB support	对应 drivers/usb 目录，包括各种 USB Host 和 USB Device 设备
MMC/SD card support	对应 drivers/mmc 目录，用来支持各种 MMC/SD 卡
Real Time Clock	对应 drivers/rtc 目录，用来支持各种实时时钟设备。比如 S3C24x0 上就集成了 RTC 芯片

16.3 Linux 内核移植

本节将修改 linux-2.6.22.6 内核，使得它可以同时在本书所用的 S3C2410、S3C2440 开发板上运行，并修改相关驱动使它支持网络功能、支持 JFFS2、YAFFS 文件系统，同时修改 MTD 设备分区，使得内核可以挂接 NAND Flash 上的文件系统。首先讲解内核启动过程，然后详细讲解移植步骤。

16.3.1 Linux 内核启动过程概述

与移植 U-Boot 的过程相似，在移植 Linux 之前，先了解它的启动过程。Linux 的启动过程可以分为两部分：架构/开发板相关的引导过程、后续的通用启动过程。如图 16.7 所示是 ARM 架构处理器上 Linux 内核 vmlinux 的启动过程。之所以强调是 vmlinux，是因为其他格式的内核在进行与 vmlinux 相同的流程之前会有一些独特的操作。比如对于压缩格式的内核 zImage，它首先进行自解压得到 vmlinux，然后执行 vmlinux 开始“正常的”启动流程。

引导阶段通常使用汇编语言编写，它首先检查内核是否支持当前架构的处理器，然后检查是否支持当前开发板。通过检查后，就为调用下一阶段的 start_kernel 函数作准备了。这主要分如下两个步骤。

① 连接内核时使用的虚拟地址，所以要设置页表、使能 MMU。

② 调用 C 函数 start_kernel 之前的常规工作，包括复制数据段、清除 BSS 段、调用 start_kernel 函数。

第二阶段的关键代码主要使用 C 语言编写。它进行内核初始化的全部工作，最后调用 rest_init 函数启动 init 过程，创建系统第一个进程：init 进程。在第二阶段，仍有部分架构/开发板相关的代码，比如图 16.7 中的 setup_arch 函数用于进行架构/开发板相关的设置（比如重新设置页表、设置系统时钟、初始化串口等）。

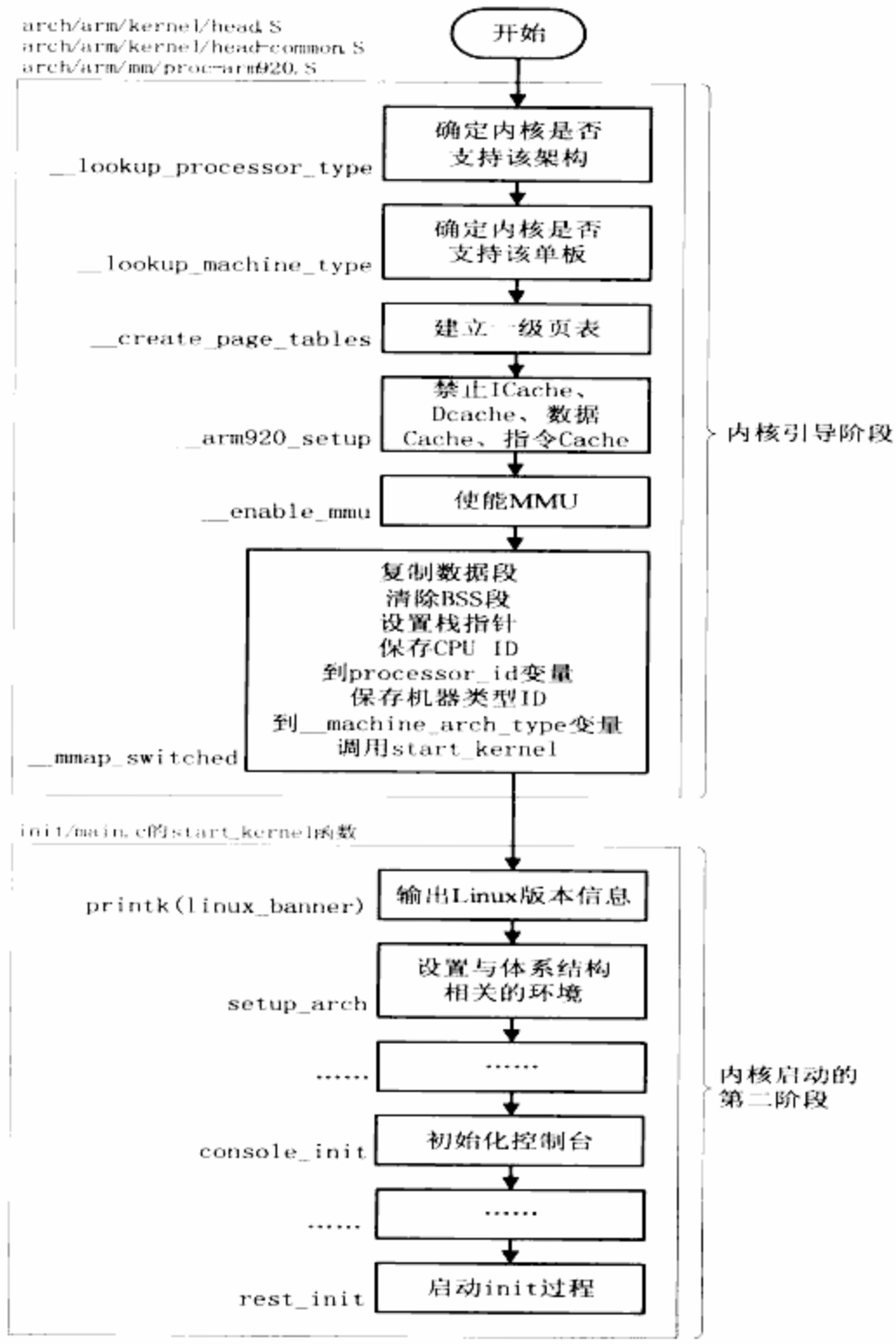


图 16.7 ARM 处理器的 Linux 内核启动过程

16.3.2 修改内核以支持 S3C2410/S3C2440 开发板

首先配置、编译内核，确保内核可以正确编译。得到内核源码后，先修改顶层 Makefile，如下所示：

```
185 ARCH          ?= $(SUBARCH)
186 CROSS_COMPILE   ?=
```

改为：

```
185 ARCH            ?= arm
186 CROSS_COMPILE   ?= arm-linux-
```

然后执行如下命令，使用 arch/arm/configs/smdk2410_defconfig 文件来配置内核，它生成.config 配置文件，以后就可以直接使用“make menuconfig”修改配置了。

```
make smdk2410_defconfig
```

最后是编译生成内核，执行“make”命令将在顶层目录下生成内核映象文件 vmlinux；执行“make uImage”除生成 vmlinux 外，还在 arch/arm/boot/目录下生成 U-Boot 格式的内核映象文件 uImage。我们使用“make uImage”命令。

对于 S3C2410 开发板，上面生成的 uImage 是可以使用的。在 U-Boot 控制界面中使用如下命令下载 uImage 并启动它：

```
tftp  0x32000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
bootm 0x32000000
```

在串口可以看到内核的启动信息，只是在最后看到如下的 panic 信息，这是因为没有修改 MTD 分区，没有增加对 yaffs 文件系统的支持。

```
VFS: Unable to mount root fs via NFS, trying floppy.
VFS: Cannot open root device "mtdblock/2" or unknown-block(2,0)
Please append a correct "root=" boot option; here are the available partitions:
1f00          16 mtdblock0 (driver?)
1f01        2048 mtdblock1 (driver?)
1f02        4096 mtdblock2 (driver?)
1f03        2048 mtdblock3 (driver?)
1f04        4096 mtdblock4 (driver?)
1f05       10240 mtdblock5 (driver?)
1f06       24576 mtdblock6 (driver?)
1f07       16384 mtdblock7 (driver?)
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
```

对于 S3C2440 开发板，使用同样的命令启动 uImage，在打印如下信息之后（注意，这些信息是 U-Boot 打印的），就出现了一大堆乱码，如下所示：

```
Starting kernel ...

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

所以，Linux 2.6.22.6 还不支持本书所用的 S3C2440 开发板，这个开发板的配置与内核所支持的开发板不完全一致。

要让内核支持本书所用的 S3C2440 开发板，需要进行一些修改，至于要修改哪些文件，

这需要详细了解内核的启动代码。

1. 引导阶段代码分析

由前面对内核 Makefile 的分析，可知 arch/arm/kernel/head.S 是内核执行的第一个文件。另外，U-Boot 调用内核时，r1 寄存器中存储“机器类型 ID”，内核会用到它。

移植 Linux 内核时，对于 arch/arm/kernel/head.S，只需要关注开头几条指令，如下所示：

```
78 ENTRY(stext)
79     msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE          @ 确保进入管理(svc)模式
80                                                            @ 并且禁止中断
81     mrc p15, 0, r9, c0, c0                                 @ 读取 CPU ID，存入 r9 寄存器
82     bl  __lookup_processor_type                            @ 调用函数,输入参数 r9=cpuid,返回值 r5=procinfo
83     movs    r10, r5                                        @ 如果不支持当前 CPU，则返回值 r5=0
84     beq __error_p                                          @ 如果 r5=0，则打印错误
85     bl  __lookup_machine_type                              @ 调用函数，返回值 r5=machinfo
86     movs    r8, r5                                         @ 如果不支持当前机器(即开发板)，则返回值 r5=0
87     beq __error_a                                          @ 如果 r5=0，则打印错误
……
```

第 79 行通过设置 CPSR 寄存器来确保处理器进入管理（svc）模式，并且禁止中断。

第 82 行读取协处理器 CP15 的寄存器 C0 获得 CPU ID，CPU ID 格式如图 16.8 和表 16.6 所示。

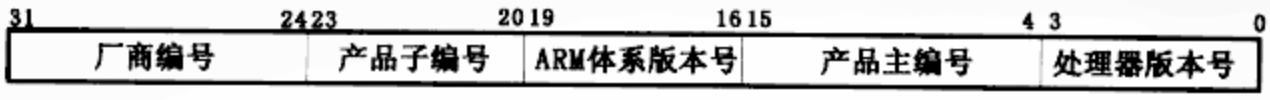


图 16.8　ARM7 之后架构的 CPU ID 格式

表 16.6　　ARM7 之后架构的 CPU ID 中各字段的定义

位	含　义
[31:24]	厂商编号目前有以下值。 0x41 = A，表示 ARM 公司 0x44 = D，表示 Digital Equipment 公司 0x69 = I，表示 Intel 公司
[23:20]	由厂商定义，当产品主编号相同时，使用子编号来区分不同的产品子类，如产品中不同的高速缓存大小等
[19:16]	ARM 体系版本号，目前取值如下。 0x01，表示 ARM 体系版本 4 0x02，表示 ARM 体系版本 4T 0x03，表示 ARM 体系版本 5 0x04，表示 ARM 体系版本 5T 0x05，表示 ARM 体系版本 5TE
[15:4]	产品主编号
[3:0]	处理器版本号

比如，S3C2410 的 CPU ID 为 0x41129200，S3C2440 的 CPU ID 也是 0x41129200。注意，S3C2410 和 S3C2440 称为片上系统（SoC），除 CPU 外，还集成了包括 UART、USB 控制器、NAND Flash 控制器等设备。从它们的 CPU ID 可知，它们的 CPU 是相同的，只是片上外设不一样。

第 82 行调用__lookup_processor_type 函数（这个函数将在下面讲述），确定内核是否支持当前 CPU。如果支持，r5 寄存器返回一个用来描述处理器的结构体的地址，否则 r5 的值为 0。

第 85 行调用__lookup_machine_type 函数（这个函数将在下面讲述），确定内核是否支持当前机器（即开发板）。如果支持，r5 寄存器返回一个用来描述这个开发板的结构体的地址，否则 r5 的值为 0。

如果__lookup_processor_type、__lookup_machine_type 这两个函数中有一个返回值为 0，则内核不能启动，如果配置内核时选择了 CONFIG_DEBUG_LL，还会打印一些提示信息。

__lookup_processor_type、__lookup_machine_typ 函数都是在 arch/arm/kernel/head -common.S 中定义的，先讲解前者。

内核映象中，定义了若干个 proc_info_list 结构（它的结构体原型在 include/ asm-arm/procinfo.h 中定义），表示它支持的 CPU。对于 ARM 架构的 CPU，这些结构体的源码在 arch/arm/mm/目录下，比如 proc-arm920.S 中的如下代码，它表示 arm920 CPU 的 proc_info_list 结构。

```
448 .section ".proc.info.init", #alloc, #execinstr
449
450 .type   __arm920_proc_info,#object
451 __arm920_proc_info:
452 .long   0x41009200
453 .long   0xff00fff0
...
```

不同的 proc_info_list 结构被用来支持不同的 CPU，它们都是定义在“.proc.info.init”段中。在连接内核时，这些结构体被组织在一起，开始地址为__proc_info_begin，结束地址为__proc_info_end。这可以从连接脚本文件 arch/arm/kernel/vmlinux.lds 中看出来。

```
302    __proc_info_begin = .;  /* proc_info_list 结构的开始地址 */
303    *(.proc.info.init)
304    __proc_info_end = .;    /* proc_info_list 结构的结束地址 */
```

__lookup_processor_type 函数就是根据前面读出的 CPU ID（存在 r9 寄存器中），从这些 proc_info_list 结构中找出匹配的，它的代码如下（在 arch/arm/kernel/head-common.S 中）：

```
145     .type   __lookup_processor_type, %function
146   __lookup_processor_type:
147     adr r3, 3f              @ r3 = 第 178 行代码的物理地址，下面会讲解这条指令
148     ldmda   r3, {r5 - r7}   @ r5 = __proc_info_begin，r6 =__proc_info_end，
它们是虚拟地址
```

```
    @ r7 = 第178行代码的虚拟地址
    149     sub r3, r3, r7          @ r3 = r3 - r7，即物理地址和虚拟地址的差值
    150     add r5, r5, r3          @ r5 = _ _proc_info_begin 对应的物理地址
    151     add r6, r6, r3          @ r6 = _ _proc_info_end 对应的物理地址
    152 1: ldmia   r5, {r3, r4}     @ r3、r4 = proc_info_list 结构中的 cpu_val、cpu_mask
    153     and r4, r4, r9          @ r4 = r4 & r9 = cpu_mask & 传入的 CPU ID
    154     teq r3, r4              @ 比较
    155     beq 2f                  @ 如果相等，表示找到匹配的 proc_info_list 结构，跳
到第160行
    156     add r5, r5, #PROC_INFO_SZ       @ r5 指向下一个 proc_info_list 结构
    @ PROC_INFO_SZ = sizeof(proc_info_list)
    157     cmp r5, r6              @ 是否已经比较完所有的 proc_info_list 结构?
    158     blo 1b                  @ 没有则继续比较
    159    mov r5, #0               @ 比较完毕，但是没有匹配的 proc_info_list 结构，r5 = 0
    160 2:  mov pc, lr              @ 返回
    161
    ...
    172 /*
    173  * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
    174  * more information about the _ _proc_info and _ _arch_info structures.
    175  */
    176     .long   _ _proc_info_begin  @ proc_info_list 结构的开始地址，这是连接地址，
也是虚拟地址
    177     .long   _ _proc_info_end    @ proc_info_list 结构的结束地址，这是连接地址，
也是虚拟地址
    178 3:  .long   .                  @ "." 号表示当前这行代码编译连接后的虚拟地址
```

请参考图16.7，在调用_ _enable_mmu函数之前使用的都是物理地址，而内核却是以虚拟地址连接的。所以在访问proc_info_list结构前，先将它的虚拟地址转换为物理地址，上面第147～151行就是用来转换地址的。

第147行用来获得第178行代码的物理地址。adr指令基于pc寄存器计算地址值，由于这时候还没使能MMU，pc寄存器中使用的还是物理地址，所以执行"adr r3, 3f"后，r3寄存器中存放的就是第178行代码的物理地址。

第148行用来获得第176～178行定义的数据：_ _proc_info_begin、_ _proc_info_end和"."。这3个数据都是在连接内核时确定，它们是虚拟地址，前两个表示proc_info_list结构的开始地址和结束地址，"."表示当前行的代码在编译连接后的虚拟地址。

第149行计算物理地址和虚拟地址的差值，第150～151根据这个差值计算_ _proc_info_begin、_ _proc_info_end的物理地址。

下面的代码依次读取每个proc_info_list结构前面的两个成员（cpu_val和cpu_mask），判断cpu_val是否等于（r9 & cpu_mask），r9是arch/arm/kernel/head.S中调用_ _lookup_processor_type

时传入的 CPU ID。如果比较相等，则表示当前 proc_info_list 结构适用于这个 CPU，直接返回这个结构的地址（存在 r5 中）。如果_ _proc_info_begin、_ _proc_info_end 之间的所有 proc_info_list 结构都不支持这个 CPU，则返回 0（r5 等于 0）。

对于 S3C2410、S3C2440 开发板，它们的 CPU ID 都是 0x41129200，而在 arch/arm/mm/proc-arm920.S 中定义的_ _arm920_proc_info 结构中，cpu_val、cpu_mask 等于 0x41009200、0xff00fff0，刚好匹配。内核中要包含这个文件，在 arch/arm/mm/Makefile 中可以看到下面这行，它表示需要配置 CONFIG_CPU_ARM920T（配置菜单中，System Type -> Support ARM920T processor）。

```
57 obj-$(CONFIG_CPU_ARM920T)    += proc-arm920.o
```

下面讲解_ _lookup_machine_typ 函数，它和_ _lookup_processor_type 函数代码相似。

内核中对于每种支持的开发板都会使用宏 MACHINE_START、MACHINE_END 来定义一个 machine_desc 结构，它定义了开发板相关的一些属性及函数，比如机器类型 ID、起始 I/O 物理地址、Bootloader 传入的参数的地址、中断初始化函数、I/O 映射函数等。比如对于 SDMK2440 开发板，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中定义如下：

```
192 MACHINE_START(S3C2440, "SMDK2440")
193     /* Maintainer: Ben Dooks <ben@fluff.org> */
194     .phys_io    = S3C2410_PA_UART,
195     .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
196     .boot_params    = S3C2410_SDRAM_PA + 0x100,
197
198     .init_irq   = s3c24xx_init_irq,
199     .map_io     = smdk2440_map_io,
200     .init_machine   = smdk2440_machine_init,
201     .timer      = &s3c24xx_timer,
202 MACHINE_END
```

第 192、202 行的宏 MACHINE_START、MACHINE_END 在 include/asm-arm/mach/arch.h 文件中定义，如下所示：

```
50 #define MACHINE_START(_type,_name)                  \
51 static const struct machine_desc _ _mach_desc_##_type  \
52  _ _used                                \
53  _ _attribute_ _((_ _section_ _(".arch.info.init"))) = {  \
54   .nr       = MACH_TYPE_##_type,          \
55   .name         = _name,
56
57 #define MACHINE_END                        \
58 };
```

所以上一段代码扩展开来就是：

```
static const struct machine_desc __mach_desc_S3C2440
__used
__attribute__((__section__(".arch.info.init"))) = {
        .nr = MACH_TYPE_S3C2440,
        .name = "SMDK2440",
...
};
```

其中的 MACH_TYPE_S3C2440 在 arch/arm/tools/mach-types 中定义，它最后会被转换成一个头文件 include/asm-arm/ mach-types.h 供其他文件包含。 machine_desc 结构在 include/asm-arm/mach/arch.h 文件中定义。所有的 machine_desc 结构都处于“.arch.info.init”段中，在连接内核时，它们被组织在一起，开始地址为__arch_info_begin，结束地址为__arch_info_end。这可以从连接脚本文件 arch/arm/kernel/vmlinux.lds 中看出来：

```
305   __arch_info_begin = .;      /* machine_desc 结构的开始地址 */
306    *(.arch.info.init)
307   __arch_info_end = .;        /* machine_desc 结构的结束地址 */
```

不同的 machine_desc 结构用于不同的开发板，U-Boot 调用内核时，会在 r1 寄存器中给出开发板的标记（机器类型 ID）。__lookup_machine_typ 函数将这个值与 machine_desc 结构中的 nr 成员比较，如果两者相等则表示找到匹配的 machine_desc 结构，于是返回它的地址（存在 r5 中）。如果__arch_info_begin、__arch_info_end 之间所有 machine_desc 结构的 nr 成员都不等于 r1 寄存器的值，则返回 0（r5 等于 0）。

对于本书所用的 S3C2410、S3C2440 开发板，U-Boot 传入的机器类型 ID 为 MACH_TYPE_SMDK2410、MACH_TYPE_S3C2440。它们对应的 machine_desc 结构分别在 arch/arm/mach-s3c2410/mach-smdk2410.c 和 arch/arm/mach-s3c2440/mach-smdk2440.c 中定义，所以这两个文件要编进内核中。在配置菜单中，选中下面两个开发板即可。

```
System Type -> S3C2410 Machines -> SMDK2410/A9M2410
System Type -> S3C2440 Machines -> SMDK2440
```

__lookup_machine_type 函数的代码如下（在 arch/arm/kernel/head-common.S 中）：

```
178 3:  .long   .
179     .long   __arch_info_begin
180     .long   __arch_info_end
...
193     .type   __lookup_machine_type, %function
194 __lookup_machine_type:
195     adr r3, 3b                  @ r3 = 第 178 行代码的物理地址
196     ldmia   r3, {r4, r5, r6}    @ r4 = 第 178 行代码的虚拟地址
@ r5 = __arch_info_begin,r6 = __arch_info_end,它们是虚拟地址
```

```
197     sub r3, r3, r4              @ r3 = r3 - r4，即物理地址和虚拟地址的差值
198     add r5, r5, r3              @ r5 = _ _arch_info_begin 对应的物理地址
199     add r6, r6, r3              @ r6 = _ _arch_info_end 对应的物理地址
200 1:  ldr r3, [r5, #MACHINFO_TYPE]    @ r5 是 machine_desc 结构的地址
@ r3 = machine_desc 结构中定义的 nr 成员，即机器类型 ID
201     teq r3, r1                  @ r1 是 Bootloader 调用内核时传入的机器类型 ID，它
们是否相等?
202     beq 2f                      @ 若相等，跳到第 207 行
203     add r5, r5, #SIZEOF_MACHINE_DESC    @ 否则，r5 指向下一个 machine_desc 结构
                                    @ SIZEOF_MACHINE_DESC = sizeof(machine_desc)
204     cmp r5, r6                  @ 是否已经比较完所有的 machine_desc 结构?
205     blo 1b                      @ 没有则继续比较
206     mov r5, #0                  @ 比较完毕，但是没有匹配的 machine_desc 结构，r5 = 0
207 2:  mov pc, lr                  @ 返回
```

如果_ _lookup_processor_type、_ _lookup_machine_type 函数都返回成功，则图 16.7 的后续引导程度将继续执行下去。其中的_ _create_page_tables 函数用来创建一级页表以建立虚拟地址到物理地址的映射关系，它用到_ _lookup_processor_type 函数返回的 proc_info_list 结构。在引导阶段的最后，调用 start_kernel 函数进入内核启动的第二阶段。_ _lookup_machine_type 函数确定的 machine_desc 结构将在第二阶段中多次使用。

2．start_kernel 函数部分代码分析

进入 start_kernel 函数（在 init/main.c 中）之后，如果在串口上没有看到内核的启动信息，一般而言有两个原因：Bootloader 传入的命令行参数不对，或者 setup_arch 函数（在 arch/arm/kernel/setup.c 中）针对开发板的设置不正确。

从图 16.7 可知，在调用 setup_arch 函数之前已经调用“printk(linux_banner)”了，但是这个时候 printk 函数只是将打印信息放在缓冲区中，并没有打印到控制台上（比如串口、LCD 屏等），因为这个时候控制台还没有初始化。当读者阅读 start_kernel 函数的代码时，请注意，printk 函数打印的内容在 console_init 函数注册、初始化控制台之后才真正输出。

移植 U-Boot 时，U-Boot 传给内核的参数有两类：预先存在某个地址的 tag 列表和调用内核时在 r1 寄存器中指定的机器类型 ID。后者在引导阶段的_ _lookup_machine_typ 函数已经用到，而 tag 列表将在 setup_arch 函数中进行初步处理。本节将重点介绍 setup_arch 函数、console_init 函数，以 tag 列表的处理（内存 tag、命令行 tag）、串口控制台的初始化为主线。

（1）setup_arch 函数分析。

先看 setup_arch 函数，它在 arch/arm/kernel/setup.c 中定义，其部分代码如下，图 16.9 是它的流程图。

```
770 void _ _init setup_arch(char **cmdline_p)
771 {
...
```

```
776     setup_processor();                          // 进行处理器相关的一些设置
777    mdesc = setup_machine(machine_arch_type);  // 获得开发板的 machine_desc 结构
...
783    if (mdesc->boot_params)                     // 定义了 Bootloader 传入参数的地址?
784      tags = phys_to_virt(mdesc->boot_params); // 这个地址就是 tag 列表的首地址
...
798     if (tags->hdr.tag == ATAG_CORE) {
799       if (meminfo.nr_banks != 0)                // 如果已经在内核中定义了 meminfo 结构
800           squash_mem_tags(tags);                // 则忽略内存 tag
801       parse_tags(tags);                         // 解释每个 tag
802     }
...
809     memcpy(boot_command_line, from, COMMAND_LINE_SIZE);
810     boot_command_line[COMMAND_LINE_SIZE-1] = '\0';
811    parse_cmdline(cmdline_p, from);              // 对命令行进行一些先期的处理
812     paging_init(&meminfo, mdesc);               // 重新初始化页表
...
```

首先，第 776 行的 setup_processor 函数被用来进行处理器相关的一些设置，它会调用引导阶段的 lookup_processor_type 函数（它的主体是前面分析过的__lookup_processor_type 函数）以获得该处理器的 proc_info_list 结构。

接下来，第 777 行的 setup_machine 函数被用来获得开发板的 machine_desc 结构，这通过调用引导阶段 lookup_machine_type 函数（它的主体是前面分析过的 lookup_ machine_type 函数）来实现。以后，就会根据开发板的 machine_desc 结构来进行一些开发板相关的操作。

第 783～784 行用来确定 Bootloader 传入的启动参数的地址，它在开发板的 machine_desc 结构中指定，第 784 行将它转换为虚拟地址。比如对于 S3C2440 开发板，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中有如下定义。启动参数的地址就是（S3C2410_SDRAM_PA + 0x100），即 0x30000100。

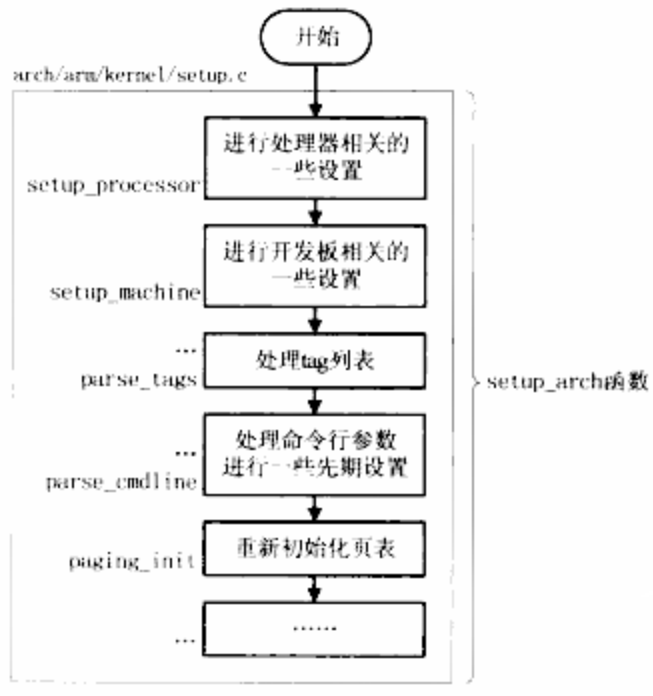


图 16.9　setup_arch 函数流程

```
192 MACHINE_START(S3C2440, "SMDK2440")
...
196     .boot_params    = S3C2410_SDRAM_PA + 0x100,
```

第 801 行处理每个 tag。文件 arch/arm/kernel/setup.c 对每种 tag 都定义了相应的处理函数，比如对于内存 tag、命令行 tag，使用如下两行代码指定了它们的处理函数为 parse_tag_mem32、parse_tag_cmdline。

```
__tagtable(ATAG_MEM, parse_tag_mem32);
__tagtable(ATAG_CMDLINE, parse_tag_cmdline);
```

parse_tag_mem32 函数根据内存 tag 定义的内存起始地址、长度，在全局结构变量 meminfo 中增加内存的描述信息。以后内核就可以通过 meminfo 结构了解开发板的内存信息。

parse_tag_cmdline 只是简单地将命令行 tag 的内容复制到字符串 default_command_line 中保存下来，后面才进一步处理。

第 811 行扫描命令行参数，对其中的一些参数进行先期的处理。这些参数使用“__early_param”来定义，比如 arch/arm/kernel/setup.c 中下面的一行代码，它表示如果命令行中有“mem=…”的字样，就调用 early_mem 对它进行处理（宏__early_param 在 include/asm-arm/setup.h 中定义）：

```
__early_param("mem=", early_mem);
```

“mem=…”用来强制限制 Linux 系统所能使用的内存总量，比如“mem=60M”使得系统只能使用 60MB 的内存，即使内存 tag 中指明了共有 64MB 内存。类似的参数还有“initrd=”等，有兴趣的读者可以阅读相关代码，本书的 U-Boot 中没有设置这类命令行参数。

注意，命令行的处理还没有结束，在 setup_arch 函数之外还会进行一系列的后续处理，比如 start_kernel 函数中调用的如下代码：

```
526    setup_command_line(command_line);
...
545    parse_early_param();
546    parse_args("Booting kernel", static_command_line, __start__param,
547        __stop__param - __start__param,
548        &unknown_bootoption);
```

比如对于命令行中的“console=ttySAC0”，它的处理过程就是第 546 行的 parse_args 函数调用第 548 行传入的 unknown_bootoption 函数，最后调用下面代码指定的处理函数 console_setup（在 kernel/printk.c 中定义）。

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

命令行参数“console=…”用来指定要使用的控制台的名称、序号、参数。比如对于“console=ttySAC0,115200”，表示要使用的控制台名称为 ttySAC，序号为 0（即第一个串口），波特率为 115200。经过 console_setup 处理后，会在全局结构变量 console_cmdline 中保存这些信息，在后面 console_init 函数初始化控制台时会根据这些信息选择要使用的控制台。

setup_arch 函数后面会调用 paging_init 函数，这也是一个开发板相关的函数，在下面说明。

（2）paging_init 函数分析。

这个函数在 setup_arch 函数中的调用形式如下：

```
paging_init(&meminfo, mdesc);
```

meminfo 中存放内存的信息，前面解释内存 tag 时确定构建了这个全局结构。

mdesc 就是前面 lookup_machine_type 函数返回的 machine_desc 结构。对于 S3C2440 开发板，这个结构在 arch/arm/mach-s3c2440/mach-smdk2440.c 中有如下定义：

```
MACHINE_START(S3C2440, "SMDK2440")
    /* Maintainer: Ben Dooks <ben@fluff.org> */
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C2410_SDRAM_PA + 0x100,

    .init_irq   = s3c24xx_init_irq,
    .map_io     = smdk2440_map_io,
    .init_machine   = smdk2440_machine_init,
    .timer      = &s3c24xx_timer,
MACHINE_END
```

上面这几行代码是移植 Linux 必须关注的数据结构。对于 S3C2410 开发板，它在 arch/arm/mach-s3c2410/mach-smdk2410.c 中定义。

paging_init 函数在 arch/arm/mm/mmu.c 中定义，根据我们的移植目的（让内核可以在 S3C2440 上运行），只需要关注如下流程：

```
paging_init -> devicemaps_init -> mdesc->map_io()
```

对于 S3C2440 开发板，就是调用 smdk2440_map_io 函数，它也是在 arch/arm/mach-s3c 24 40/mach-smdk2440.c 中定义，如下所示：

```
177 static void _ _init smdk2440_map_io(void)
178 {
179     s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
180     s3c24xx_init_clocks(16934400);
181    s3c24xx_init_uarts(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
182 }
```

第 179～181 行的 3 个函数所实现的功能，从它们的名字即可看出。注意第 180 行的参数值，它表示开发板晶振的频率。在本书所用的开发板中，这个频率是 12MHz，不是 16934400，这就是在 S3C2440 开发板上启动 uImage 时串口输出乱码的原因，将它改为 12000000 就好了。

（3）console_init 函数分析。

虽然上面已经找到内核无法正常输出信息的原因，但我们不该止步于此。在 2.4 版本的内核中，命令行参数常用“console=ttyS0”来指定控制台为串口 0，在 2.6 版本的内核中改为“console=ttySAC0”。分析 console_init 函数的功能就可以了解这点。

console_init 函数被 start_kernel 函数调用，它在 drivers/char/tty_io.c 文件中定义如下：

```
3967 void _ _init console_init(void)
3968 {
```

```
3969    initcall_t *call;
...
3978    call = _ _con_initcall_start;
3979    while (call < _ _con_initcall_end) {
3980        (*call)();
3981        call++;
3982    }
3983 }
```

它调用地址范围_ _con_initcall_start 至_ _con_initcall_end 之间的定义的每个函数，这些函数使用 console_initcall 宏来指定，比如 drivers/serial/s3c2410.c 中：

```
console_initcall(s3c24xx_serial_initconsole);
```

s3c24xx_serial_initconsole 函数也是在 drivers/serial/s3c2410.c 中定义，它初始化 S3C24xx 类 SoC 的串口控制台，部分代码如下：

```
1892 static int s3c24xx_serial_initconsole(void)
1893 {
...
1927    register_console(&s3c24xx_serial_console);
1928    return 0;
1929 }
```

s3c24xx_serial_console 结构在 drivers/serial/s3c2410.c 中定义如下：

```
static struct console s3c24xx_serial_console =
{
    .name       = S3C24XX_SERIAL_NAME,    // 即 "ttySAC"
    .device     = uart_console_device,    // 以后使用/dev/console 时，用来构造设备节点
    .flags      = CON_PRINTBUFFER,        // 控制台可用之前，printk 已经在缓冲区中打印了
                                          // 很多信息，CON_PRINTBUFFER 表示注册控制台之后，
                                          // 打印这些 "过去的" 的信息
    .index      = -1,                     // -1 可以匹配任意序号，比如 ttySAC0/1/2
    .write      = s3c24xx_serial_console_write,   // 打印函数
    .setup      = s3c24xx_serial_console_setup    // 设置函数
};
```

第 1927 行在内核中注册控制台，就是把 s3c24xx_serial_console 结构链入一个全局链表 console_drivers 中（它在 kernel/printk.c 中定义）。并且使用其中的名字（name）和序号（index）与前面 "console=…" 指定的控制台相比较，如果相符，则以后的 printk 信息从这个控制台输出。

对于本书的情况，"console=ttySAC0"，而 s3c24xx_serial_console 结构中名字为"ttySAC"，序号为-1（表示可以取任意值），所以两者匹配，printk 信息将从串口 0 输出。

现在总结一下上面分析的内核启动第二阶段的函数调用过程（以S3C2440开发板为例），相同缩进的函数表示它们是在同一个函数中被调用：

```
start_kernel  ->
    setup_arch  ->
        setup_processor
        setup_machine
        ...
        parse_tags
        ...
        parse_cmdline
        paging_init  ->
            devicemaps_init ->
                mdesc->map_io() ->
                    s3c24xx_init_io
                    s3c24xx_init_clocks
                    s3c24xx_init_uarts
        ...
        console_init  ->
            s3c24xx_serial_initconsole
                register_console(&s3c24xx_serial_console)
        ...
```

3. 修改内核

在arch/arm/mach-s3c2440/mach-smdk2440.c中做如下修改：

修改前：

```
180     s3c24xx_init_clocks(16934400);
```

修改后：

```
180     s3c24xx_init_clocks(12000000);
```

然后执行“make uImage”生成uImage。

对于S3C2410、S3C2440开发板，上面生成的uImage都可以使用了。

把uImage放到tftp服务器的目录下，或者放到Linux中/work/nfs_root目录下，然后在U-Boot控制界面中使用如下命令下载uImage并启动它。

```
tftp  0x32000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
bootm 0x32000000
```

可以看到内核的启动信息，最后出现panic信息（这需要修改mtd分区、增加对yaffs文件系统的支持）。

16.3.3 修改 MTD 分区

MTD（Memory Technology Device），即内存技术设备，是 Linux 中对 ROM、NOR Flash、NAND Flash 等存储设备抽象出来的一个设备层，它向上提供统一的访问接口：读、写、擦除等；屏蔽了底层硬件的操作、各类存储设备的差别。得益于 MTD 设备的作用，重新划分 NAND Flash 的分区很简单。

本节分为两部分，先介绍一下内核对 NAND Flash 的识别过程（这个过程也适用于其他设备），再给出具体的代码修改方法（读者可以直接参考第二部分进行修改、实验）。

1. 驱动对设备的识别过程

驱动程序识别设备时，有以下两种方法。

（1）驱动程序本身带有设备的信息，比如开始地址、中断号等；加载驱动程序时，就可以根据这些信息来识别设备。

（2）驱动程序本身没有设备的信息，但是内核中已经（或以后）根据其他方式确定了很多设备的信息；加载驱动程序时，将驱动程序与这些设备逐个比较，确定两者是否匹配（match）。如果驱动程序与某个设备匹配，就可以通过该驱动程序操作这个设备了。

内核常使用第二种方法来识别设备，这可以将各种设备集中在一个文件中管理，当开发板的配置改变时，便于修改代码。在内核文件 include/linux/platform_device.h 中，定义了两个数据结构来表示这些设备和驱动程序：platform_device 结构用来描述设备的名称、ID、所占用的资源（比如内存地址/大小、中断号）等；platform_driver 结构用来描述各种操作函数，比如枚举函数、移除设备函数、驱动的名称等。

内核启动后，首先构造链表将描述设备的 platform_device 结构组织起来，得到一个设备的列表；当加载某个驱动程序的 platform_driver 结构时，使用一些匹配函数来检查驱动程序能否支持这些设备，常用的检查方法很简单：比较驱动程序和设备的名称。

以 S3C2440 开发板为例，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中定义了如下设备：

```
static struct platform_device *smdk2440_devices[ ] _ _initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
};
```

在 arch/arm/plat-s3c24xx/common-smdk.c 中定义了如下设备：

```
static struct platform_device _ _initdata *smdk_devs[ ] = {
    &s3c_device_nand,
    &smdk_led4,
    &smdk_led5,
    &smdk_led6,
```

```
    &smdk_led7,
};
```

这些设备在 smdk2410_init 函数（对应 S3C2410）或 smdk2440_machine_init 函数（对应 S3C2440）中，通过 platform_add_devices 函数注册进内核中。

NAND Flash 设备 s3c_device_nand 在 arch/arm/plat-s3c24xx/devs.c 中的定义如下：

```
struct platform_device s3c_device_nand = {
    .name           = "s3c2410-nand",
    .id             = -1,
    .num_resources  = ARRAY_SIZE(s3c_nand_resource),
    .resource       = s3c_nand_resource,
};
```

对于 S3C2440 开发板，s3c_device_nand 结构的名字会在 s3c244x_map_io 函数中修改为“s3c2440-nand”，这个函数在 arch/arm/plat-s3c24xx/s3c244x.c 中的定义如下：

```
void __init s3c244x_map_io(struct map_desc *mach_desc, int size)
{
...
    s3c_device_i2c.name  = "s3c2440-i2c";
    s3c_device_nand.name = "s3c2440-nand";
    s3c_device_usbgadget.name = "s3c2440-usbgadget";
}
```

有了 NAND Flash 设备，还要有 NAND Flash 驱动程序，内核针对 S3C2410、S3C2412、S3C2440 定义了 3 个驱动。它们在 drivers/mtd/nand/s3c2410.c 中的 s3c2410_nand_init 函数中注册进内核，如下所示：

```
static int __init s3c2410_nand_init(void)
{
    printk("S3C24XX NAND Driver, (c) 2004 Simtec Electronics\n");

    platform_driver_register(&s3c2412_nand_driver);
    platform_driver_register(&s3c2440_nand_driver);
    return platform_driver_register(&s3c2410_nand_driver);
}
```

其中的 s3c2440_nand_driver 结构也是在相同的文件中定义，如下所示：

```
static struct platform_driver s3c2440_nand_driver = {
        .probe      = s3c2440_nand_probe,
        .remove     = s3c2410_nand_remove,
        .suspend    = s3c24xx_nand_suspend,
```

```
        .resume     = s3c24xx_nand_resume,
        .driver     = {
        .name       = "s3c2440-nand",
        .owner      = THIS_MODULE,
        },
};
```

可见，s3c_device_nand 结构和 s3c2440_nand_driver 结构中的 name 成员相同，都是“s3c2440-nand”。platform_driver_register 函数就是根据这点确定它们是匹配的，所以调用 s3c2440_nand_probe 函数来枚举 NAND Flash 设备 s3c_device_nand。

从 s3c2440_nand_probe 函数开始，可以一直找到对 NAND Flash 分区的识别，如下所示：

```
s3c2440_nand_probe(&s3c_device_nand)  -> // 这个参数是笔者为了便于理解换上去的
        s3c24xx_nand_probe(&s3c_device_nand, TYPE_S3C2440)  ->
            struct s3c2410_platform_nand *plat = to_nand_plat(pdev);    //
plat= &smdk_nand_info
            ...
            s3c2410_nand_add_partition(info, nmtd, sets)  ->        // sets 就
是 smdk_nand_sets
            add_mtd_partitions                                  // 实际的参数为 smdk_
default_nand_part
```

这些函数都在 drivers/mtd/nand/s3c2410.c 中定义，最后的 add_mtd_partitions 函数根据 smdk_default_nand_part 结构来确定分区。这个结构在 arch/arm/plat-s3c24xx/common-smdk.c 中定义，要改变分区时修改它即可。

2. 修改 MTD 分区

如上所述，要改变分区时，修改 arch/arm/plat-s3c24xx/common-smdk.c 文件中的 smdk_default_nand_part 结构即可。本书将 NAND Flash 划为 3 个分区，前 2MB 用于存放内核，接下来的 8MB 用于存放 JFFS2 文件系统，剩下的用来存放 YAFFS 文件系统。

smdk_default_nand_part 结构如下修改：

```
static struct mtd_partition smdk_default_nand_part[] = {
    [0] = {
        .name   = "kernel",
        .size   = SZ_2M,
        .offset = 0,
    },
    [1] = {
        .name   = "jffs2",
        .offset = MTDPART_OFS_APPEND,
```

```
        .size   = SZ_8M,
    },
    [2] = {
        .name   = "yaffs",
        .offset = MTDPART_OFS_APPEND,
        .size   = MTDPART_SIZ_FULL,
    }
};
```

其中的 MTDPART_OFS_APPEND 表示当前分区紧接着上一个分区，MTDPART_SIZ_FULL 表示当前分区的大小为剩余的 Flash 空间。

执行“make uImage”重新生成内核，在 U-Boot 控制界面中使用如下命令下载 uImage 并启动它。

```
tftp 0x32000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/U-Boot.bin
bootm 0x32000000
```

可以看到内核打印出如下分区信息。

```
Creating 3 MTD partitions on "NAND 64MiB 3,3V 8-bit":
0x00000000-0x00200000 : "kernel"
0x00200000-0x00a00000 : "jffs2"
0x00a00000-0x04000000 : "yaffs"
```

由于目标开发板上还没有写入文件系统映象，也没有设置命令行使用网络文件文件系统（nfs），内核启动到最后还是会出现 panci 信息。

16.3.4 移植 YAFFS 文件系统

1. YAFFS 文件系统介绍

YAFFS（yet another flash file system）是一种类似于 JFFS/JFFS2、专门为 NAND Flash 设计的嵌入式文件系统，适用于大容量的存储设备。它是日志结构的文件系统，提供了损耗平衡和掉电保护，可以有效地避免意外掉电对文件系统一致性和完整性的影响。与 JFFS 相比，它减少了一些功能，因此速度更快，占用内存更少。

YAFFS 充分考虑了 NAND Flash 的特点，根据 NAND Flash 以页面为单位存取的特点，将文件组织成固定大小的数据段。利用 NAND Flash 提供的每个页面 16 字节的 OOB 空间来存放 ECC（Error Correction Code）和文件系统的组织信息，不仅能够实现错误检测和坏块处理，也能够提高文件系统的加载速度。YAFFS 采用一种多策略混合的垃圾回收算法，结合了贪心策略的高效性和随机选择的平均性，达到了兼顾损耗平均和系统开销的目的。

YAFFS 文件系统具有很好的可移植性，可以在 Linux、Windows CE、pSOS、ThreadX、DSP-BIOS 等多种操作系统上工作。为 NAND Flash 提供了一种可靠的操作系统，并且特别适用于对能耗要求比较高的嵌入式系统。

YAFFS 文件系统目前已经发展到第二版本：YAFFS2，它向前兼容 YAFFS1（有时候也用 YAFFS 来表示 YAFFS1），主要特点是支持每页容量大于 512 字节的 NAND Flash。YAFFS2 的性能与 YAFFS1 相比有很大提高，比较结果如表 16.7 所示。

表 16.7　　YAFFS2 与 YAFFS1 的性能比较

比　较	YAFFS2	YAFFS1
写操作	快 1～3 倍	1.5MB/s～4.5MB/s	1.5MB/s
读操作	快 1～2 倍	7.6MB/s～16.7MB/s	7.6MB/s
删除操作	快 4～34 倍	7.8MB/s～62.5MB/s	1.8MB/s
垃圾回收	快 2～7 倍	2.1MB/s～7.7MB/s	1.1MB/s
内存消耗	减少 25%～50%	—	—

注：① 表中 YAFFS2 的最差性能表现在每页 512Byte 的 NAND Flash 上，与 YAFFS1 相似。

② 表中 YAFFS2 的较佳性能表现在每页 2kB（并且总线位宽为 16）的 NAND Flash 上。

一般而言，在 NOR Flash 上使用 JFFS2 文件系统，在 NAND Flash 上使用 YAFFS 文件系统。JFFS2 与 YAFFS 的性能比较如表 16.8 所示。

表 16.8　　JFFS2 与 YAFFS 的性能比较

性　能	JFFS2	YAFFS
内存消耗	每个节点（node）占用 16 字节 128MB 的 Flash 将占用 4MB 内存	每页占用 2 字节 128MB 的 Flash 将占用 512KB 内存
第一次启动时的扫描时间	128MB Flash 上时间为 25s	只需要读取 OOB，时间为 3s
是否压缩	压缩	不压缩
代码复杂度	复杂，特别是垃圾回收部分	简单
适用的操作系统	Linux、eCos	很多，容易移植
启动时间	Flash 容量为 4MB（或 8MB）时为 4s	Flash 容量为 30MB 时为 7s

YAFFS 有 GPL 版和商业版，它们的代码完全一致。使用 GPL 版本的 YAFFS 可以避免付费，但是需要公开二进制代码和源代码；如果不想公开，需要购买授权。

2．YAFFS 文件系统移植

从 http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/获取源代码文件 root.tar.gz，解压后得到 Development 目录，里面有两个子目录：yaffs 和 yaffs2。yaffs 目录已经不再维护，本书使用 yaffs2 目录下的代码，它向前兼容 YAFFS1。

也可以使用源代码文件/work/system/yaffs_source.tar.gz，它只是将 root.tar.gz 改了个名字。

移植 yaffs 分两个步骤。

（1）将 yaffs2 代码加入内核。

这可以通过 yaffs2 目录下的脚本文件 patch-ker.sh 来给内核打补丁，用法如下：

```
usage: ./patch-ker.sh c/l kernelpath
if c/l is c, then copy. If l then link
```

这表明，如果“c/l”为“c”，则 yaffs2 的代码会被复制到内核目录下；如果是“l”，则仅仅在内核目录下创建一些连接文件。

假设下载后解压所得的 yaffs2 源码目录为/work/system/Development/yaffs2，内核源码目录为/work/system/linux-2.6.22.6，执行以下命令打补丁：

```
$ cd /work/system/Development/yaffs2
$ ./patch-ker.sh c /work/system/linux-2.6.22.6
```

上述命令完成以下 3 件事情。

① 修改内核 fs/Kconfig 文件，增加下面两行：

```
# Patched by YAFFS
source "fs/yaffs2/Kconfig"
```

② 修改内核 fs/Makefile 文件，增加下面两行：

```
# Patched by YAFFS
obj-$(CONFIG_YAFFS_FS)                  += yaffs2/
```

③ 在内核 fs/目录下创建 yaffs2 子目录，然后如下复制文件。

将 yaffs2 源码目录下的 Makefile.kernel 文件复制为内核 fs/yaffs2/Makefile 文件。

将 yaffs2 源码目录下的 Kconfig 文件复制到内核 fs/yaffs2/目录下。

将 yaffs2 源码目录下的的*.c、*.h 文件（不包括子目录下的文件）复制到内核 fs/yaffs2/目录下。

（2）配置、编译内核。

阅读内核 fs/yaffs2/Kconfig 文件可以了解各个配置选项的作用。

下面讲解用到的几个选项。

① CONFIG_YAFFS_FS：支持 YAFFS 文件系统。

② CONFIG_YAFFS_YAFFS1：支持 YAFFS1 文件系统。

对于每页大小为 512 字节的 NAND Flash，要选上这个配置项。

③ CONFIG_YAFFS_YAFFS2：支持 yaffs1 文件系统。

对于每页大小为 2048 字节的 NAND Flash，要选上这个配置项。本书所用 NAND Flash 每页为 512 字节，这个配置项可以不选。

④ CONFIG_YAFFS_AUTO_YAFFS2：自动选择 YAFFS2 格式。

如果不设置这个配置项，必须使用“yaffs2”字样来表示 YAFFS2 文件系统格式；如果设置了这个配置项，则可以使用“yaffs”字样来统一表示 YAFFS、YAFFS2 文件系统格式，驱动程序会根据 NAND Flash 页的大小自动分辨是 YAFFS 还是 YAFFS2。

⑤ CONFIG_YAFFS_9BYTE_TAGS。

老的 YAFFS1 文件系统中，使用 oob 区中 9 个字节作为文件系统的标记（tag），比新的 YAFFS1 多了 1 字节——“pageStatus”，它用来表示页的状态。

如果要使用老的 YAFFS1，这个配置项要选上，另外还要修改 MTD 设备层以使用老的

oob layout 结构，oob layout 就是内核文件 drivers/mtd/nand/nand_base.c 中的 nand_oob_16 结构。

Linux 2.6.22.6 内核使用新的 oob layout，格式如下。它表示 ECC 码存放的位置是 oob 区中 0、1、…、7 这 8 个字节；剩下的空间称为可用空间，供文件系统使用，代码中将这些数据称为标记（tag）：

```
static struct nand_ecclayout nand_oob_16 = {
    .eccbytes = 6,
    .eccpos = {0, 1, 2, 3, 6, 7},
    .oobfree = {
        {.offset = 8,
         . length = 8}}
};
```

以前的内核使用老的 oob layout，格式如下，ECC 码的位置不一样，标记的位置也不一样。

```
static struct nand_ecclayout nand_oob_16 = {
    .eccbytes = 6,
    .eccpos = { 8, 9, 10, 13, 14, 15 },
    .oobavail = 9,
    .oobfree = { { 0, 4 }, { 6, 2 }, { 11, 2 }, { 4, 1 } }
};
```

如果要使用老格式的 YAFFS1 映象文件，定义 CONFIG_YAFFS_9BYTE_TAGS 配置项，并且修改 nand_oob_16 结构为老的格式。

本书使用新格式的 YAFFS1 映象文件。

⑥ CONFIG_YAFFS_DOES_ECC：使用 YAFFS 本身的 ECC 校验函数。

一般使用 MTD 设备层的 ECC 校验函数，这个配置项不用设置。

了解各个配置项的意义后，就可以配置内核，选上对 YAFFS 的支持了。在内核配置界面中选中“YAFFS2 file system support”即可，其他配置项使用默认值。

```
File systems  --->
    Miscellaneous filesystems  --->
        <*> YAFFS2 file system support
```

最后执行“make uImage”编译内核。

16.3.5 编译、烧写、启动内核

到本节为止，内核已经同时支持 S3C2410 和 S3C2440，修改了 NAND Flash 的分区，增加了对 YAFFS 文件系统的支持。另外，内核原来已经支持 JFFS2 文件系统。现在的内核，已经基本可用，可以将它烧入 NAND Flash 中了。

1. 编译内核

本章中所做修改比较分散，为方便读者，将这些修改制作成补丁文件 linux 2.6.22.6_

100ask24x0.patch，它位于/work/system 目录下。

所以，读者可以按照前面的章节一个个地修改涉及的文件，也可以使用以下命令直接打补丁。

```
$ cd /work/system
$ tar xjf linux-2.6.22.6.tar.bz2
$ cd /work/system/linux-2.6.22.6
$ patch -p1 < ../linux2.6.22.6_100ask24x0.patch
```

内核根目录下的 config_ok 文件是本章所用的配置文件，直接使用它即可。执行以下命令编译内核，它将在 arch/arm/boot 目录下生成 uImage 文件。

```
$ cp config_ok .config
$ make uImage
```

2. 烧写内核

将上面生成的 uImage 放入 tftp 服务器目录或 nfs 目录（/work/nfs_root），然后在 U-Boot 中执行以下命令下载、烧写。

```
tftp  0x32000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
nand erase 0 0x200000                          /* 擦除 NAND Flash 前 2MB */
nand write.jffs2 0x32000000 0 $(filesize)    /* 烧写 uImage */
```

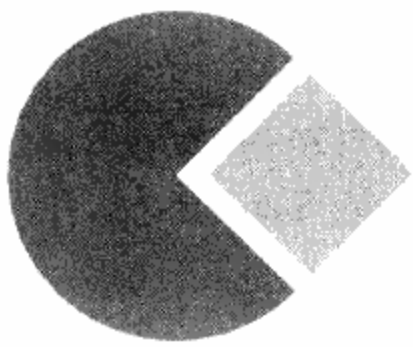
3. 启动内核

可以使用以下命令启动 NAND Flash 上的内核。

```
nboot 0x32000000 0 0
bootm 0x32000000
```

要想开发板上电后内核自动启动，可以设置 bootcmd 环境变量。

```
set bootcmd 'nboot 0x32000000 0 0; bootm 0x32000000'
saveenv
```



第 17 章　构建 Linux 根文件系统

本章目标

- 了解 Linux 的文件系统层次标准（FHS）
- 了解根文件系统下各目录的作用
- 掌握构建根文件系统的方法：移植 Busybox、构造各个目录、文件等
- 掌握制作 yaffs、jffs2 文件系统映象文件的方法

17.1　Linux 文件系统概述

17.1.1　Linux 文件系统的特点

类似于 Windows 下的 C、D、E 等各个盘，Linux 系统也可以将磁盘、Flash 等存储设备划分为若干个分区，在不同分区存放不同类别的文件。与 Windows 的 C 盘类似，Linux 一样要在一个分区上存放系统启动所必需的文件，比如内核映象文件（在嵌入式系统中，内核一般单独存放在一个分区中）、内核启动后运行的第一个程序(init)、给用户提供操作界面的 shell 程序、应用程序所依赖的库等。这些必需、基本的文件合称为根文件系统，它们存放在一个分区中。Linux 系统启动后首先挂接这个分区，称为挂接（mount）根文件系统。其他分区上所有目录、文件的集合，也称为文件系统。

Linux 中并没有 C、D、E 等盘符的概念，它以树状结构管理所有目录、文件，其他分区挂接在某个目录上，这个目录被称为挂接点或安装点（mount point），然后就可以通过这个目录来访问这个分区上的文件了。比如根文件系统被挂接在根目录“/”上后，在根目录下就有根文件系统的各个目录、文件：/bin、/sbin、/mnt 等；再将其他分区挂接到/mnt 目录上，/mnt 目录下就有这个分区的各个目录、文件。

在一个分区上存储文件时，需要遵循一定的格式，这种格式称为文件系统类型，比如 fat16、fat32、ntfs、ext2、ext3、jffs2、yaffs 等。除这些拥有实实在在的存储分区的文件系统类型外，Linux 还有几种虚拟的文件系统类型，比如 proc、sysfs 等，它们的文件并不存储在实际的设备上，而是在访问它们时由内核临时生成。比如 proc 文件系统下的 uptime 文件，读取它时可以得到两个时间值（用来表示系统启动后运行的秒数、空闲的秒数），每次读取时

都由内核即刻生成，每次读取结果都不一样。

“文件系统类型”常被简称为“文件系统”，比如“硬盘第二个分区上的文件系统是EXT2”指的就是文件系统类型。所以“文件系统”这个术语，有时候指的是分区上的文件集合，有时候指的是文件系统类型，需要根据语境分辨，读者在阅读各类文献时需要注意这点。

17.1.2 Linux根文件系统目录结构

为了在安装软件时能够预知文件、目录的存放位置，为了让用户方便地找到不同类型的文件，在构造文件系统时，建议遵循FHS标准（Filesystem Hierarchy Standard，文件系统层次标准）。它定义了文件系统中目录、文件分类存放的原则，定义了系统运行所需的最小文件、目录的集合，并列举了不遵循这些原则的例外情况及其原因。FHS并不是一个强制的标准，但是大多的Linux、UNIX发行版本都遵循FHS。

本节根据FHS标准描述Linux根文件系统的目录结构，并不深入描述各个子目录的结构，读者可以自行阅读FHS标准，FHS文档可以从网站http://www.pathname.com/fhs/下载。

Linux根文件系统中一般有如图17.1所示的几个目录。

下面依次讲述这几个目录的作用。

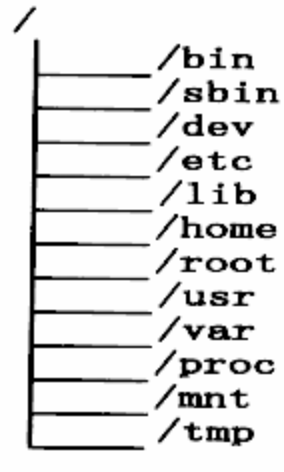


图17.1 Linux根文件系统结构

1. /bin目录

该目录下存放所有用户（包括系统管理员和一般用户）都可以使用的、基本的命令，这些命令在挂接其他文件系统之前就可以使用，所以/bin目录必须和根文件系统在同一个分区中。

/bin目录下常用的命令有：cat、chgrp、chmod、cp、ls、sh、kill、mount、umount、mkdir、mknod、[、test等。“[”命令其实就是test命令，在脚本文件中“[expr]”就等价于“test expr”。

2. /sbin目录

该目录下存放系统命令，即只有管理员能够使用的命令，系统命令还可以存放在/usr/sbin、/usr/local/sbin目录下。/sbin目录中存放的是基本的系统命令，它们用于启动系统、修复系统等。与/bin目录相似，在挂接其他文件系统之前就可以使用/sbin，所以/sbin目录必须和根文件系统在同一个分区中。

/sbin目录下常用的命令有：shutdown、reboot、fdisk、fsck等。

不是急迫需要使用的系统命令存放在/usr/sbin目录下。本地安装的（Locally- installed）的系统命令存放在/usr/local/sbin目录下。

3. /dev目录

该目录下存放的是设备文件。设备文件是Linux中特有的文件类型，在Linux系统下，以文件的方式访问各种外设，即通过读写某个设备文件操作某个具体硬件。比如通过“/dev/ttySAC0”文件可以操作串口0，通过“/dev/mtdblock1”可以访问MTD设备（NAND Flash、NOR Flash等）的第2个分区。

设备文件有两种：字符设备和块设备。在PC上执行命令“ls /dev/ttySAC0 /dev/hda1 -l”

可以看到如下结果。

```
brwxrwxr-x   1 root    49        3,   1 Oct  9  2005 /dev/hda1
crwxrwxr-x   1 root    root      4,  64 Sep 24  2007 /dev/ttySAC0
```

其中字母“b”、“c”表示这是一个块设备文件或字符设备文件；“3, 1”、“4, 64”表示设备文件的主、次设备号；主设备号用来表示这是哪类设备，次设备号用来表示这是这类设备中的哪个。

设备文件可以使用 mknod 命令创建，比如：

```
mknod /dev/ttySAC0 c 4 64
mknod /dev/hda1 b 3 1
```

/dev 的创建有 3 种方法。

（1）手动创建。

在制作根文件系统的时候，就在/dev 目录下创建好要使用的设备文件，比如 ttySAC0 等。系统挂接根文件系统后，就可以使用/dev 目录下的设备文件了。

（2）使用 devfs 文件系统：这种方法已经过时。

在以前的内核中，有一个配置选项 CONFIG_DEVFS_FS，它用来将虚拟文件系统 devfs 挂接在/dev 目录上，各个驱动程序注册时会在/dev 目录下自动生成各种设备文件。这就免去了手动创建设备文件的麻烦，在制作根文件系统时，/dev 目录可以为空。

使用 devfs 比手动创建设备节点更便利，但是它仍有一些无法克服的缺点。

① 不确定的设备映射。

比如 USB 接口连接两台打印机 A 和 B，在都开机的情况下以/dev/usb/lp0 访问 A，以/dev/usb/lp1 访问 B。但是假如 A 没有上电，则系统启动时会根据扫描到的设备的顺序，以/dev/usb/lp0 访问 B。

② 没有足够的主/次设备号。

主次设备号是两个 8 位的数字，它们并不足以与日益增加的外设一一对应。

③ 命名不够灵活。

由于 devfs 由内核创建设备节点，当想重新修改某个设备的名字时需要修改、编译内核。

④ devfs 消耗大量的内存。

由于这些缺点，在 Linux 2.3.46 引入 devfs 之后，又在 Linux 2.6.13 后面的版本中移除了 devfs，而使用 udev 机制代替。

（3）udev。

udev 是个用户程序（u 是指 user space，dev 是指 device），它能够根据系统中硬件设备的状态动态地更新设备文件，包括设备文件的创建，删除等。

使用 udev 机制也不需要在/dev 目录下创建设备节点，它需要一些用户程序的支持，并且内核要支持 sysfs 文件系统。它的操作相对复杂，但是灵活性很高。

在 busybox 中有一个 mdev 命令，它是 udev 命令的简化版本。

4. /etc 目录

如表 17.1、17.2 所示，该目录下存放各种配置文件。对于 PC 上的 Linux 系统，/etc 目录

下目录、文件非常多。这些目录、文件都是可选的，它们依赖于系统中所拥有的应用程序，依赖于这些程序是否需要配置文件。在嵌入系统中，这些内容可以大为精减。

表 17.1 /etc 目录下的子目录

目　录	描　述
opt	用来配置/opt 下的程序（可选)
X11	用来配置 X Window（可选）
sgml	用来配置 SGML（可选）
xml	用来配置 XML（可选）

表 17.2 /etc 目录下的文件

文　件	描　述
export	用来配置 NFS 文件系统（可选）
fstab	用来指明当执行“mount -a”时，需要挂接的文件系统（可选）
mtab	用来显示已经加载的文件系统，通常是/proc/mounts 的链接文件(可选)
ftpusers	启动 FTP 服务时，用来配置用户的访问权限（可选）
group	用户的组文件（可选）
inittab	init 进程的配置文件（可选）
ld.so.conf	其他共享库的路径（可选）
passwd	密码文件（可选）

5. /lib 目录

该目录下存放共享库和可加载模块（即驱动程序），共享库用于启动系统、运行根文件系统中的可执行程序，比如/bin、/sbin 目录下的程序。其他不是根文件系统所必需的库文件可以放在其他目录，比如/usr/lib、/usr/X11R6/lib、/var/lib 等。

表 17.3 所示是/lib 目录中的内容。

表 17.3 /lib 目录中的内容

目录/文件	描　述
libc.so.*	动态连接 C 库（可选）
ld*	连接器、加载器（可选）
modules	内核可加载模式存放的目录（可选）

6. /home 目录

用户目录，它是可选的。对于每个普通用户，在/home 目录下都有一个以用户名命名的子目录，里面存放用户相关的配置文件。

7. /root 目录

根用户（用户名为 root）的目录，与此对应，普通用户的目录是/home 下的某个子目录。

8. /usr 目录

/usr 目录的内容可以存在另一个分区中，在系统启动后再挂接到根文件系统中的/usr 目录下。里面存放的是共享、只读的程序和数据，这表明/usr 目录下的内容可以在多个主机间共享，这些主机也是符合 FHS 标准的，/usr 中的文件应该是只读的，其他主机相关、可变的文件应该保存在其他目录下，比如/var。

/usr 目录通常包含如下内容，嵌入式系统中，这些内容可以进一步精减。/usr 目录中的内容如表 17.4 所示。

表 17.4　　/usr 目录中的内容

目　录	描　述
bin	很多用户命令存放在这个目录下
include	C 程序的头文件，这在 PC 上进行开发时才用到，在嵌入式系统中不需要
lib	库文件
local	本地目录
sbin	非必需的系统命令（必需的系统命令放在/sbin 目录下）
share	架构无关的数据
X11R6	XWindow 系统
games	游戏
src	源代码

9. /var 目录

与/usr 目录相反，/var 目录中存放可变的数据，比如 spool 目录（mail、news、打印机等用的），log 文件、临时文件。

10. /proc 目录

这是一个空目录，常作为 proc 文件系统的挂接点。proc 文件系统是个虚拟的文件系统，它没有实际的存储设备，里面的目录、文件都是由内核临时生成的，用来表示系统的运行状态，也可以操作其中的文件控制系统。

系统启动后，使用以下命令挂接 proc 文件系统（常在/etc/fstab 进行设置以自动挂接）。

```
# mount -t proc none /proc
```

11. /mnt 目录

用于临时挂接某个文件系统的挂接点，通常是空目录；也可以在里面创建一些空的子目录，比如/mnt/cdram、/mnt/hda1 等，用来临时挂接光盘、硬盘。

12. /tmp 目录

用于存放临时文件，通常是空目录。一些需要生成临时文件的程序要用到/tmp 目录，所

以/tmp目录必须存在并可以访问。

为减少对Flash的操作，当在/tmp目录上挂接内存文件系统时，如下所示：

```
# mount -t tmpfs none /tmp
```

17.1.3　Linux文件属性介绍

Linux系统有如表17.5所示的几种文件类型。

表17.5　　Linux文件类型

文件类型	描　述
普通文件	这是最常见的文件类型
目录文件	目录也是一种文件
字符设备文件	用来访问字符设备
块设备文件	用来访问块设备
FIFO	用于进程间的通信，也称为命名管道
套接口	用于进程间的网络通信
连接文件	它指向另一个文件，有软连接、硬连接

使用“ls -lih”命令可以看到各个文件的具体信息，下面选取这几种文件，列出它们的信息。

```
 228883 -rw-r--r--   2 root     root         6 Sep 27 22:10 readme.txt
 228884 lrwxrwxrwx   1 root     root        10 Sep 27 22:11 ln_soft -> readme.txt
 228883 -rw-r--r--   2 root     root         6 Sep 27 22:10 ln_hard
 228882 drwxr-xr-x   2 root     root      4.0K Sep 27 22:10 tmp_dir
 228880 crw-r--r--   1 root     root    4,  64 Sep 27 22:09 ttySAC0
 228881 brw-r--r--   1 root     root   31,  0 Sep 27 22:09 mtdblock0
 228885 prw-r--r--   1 root     root         0 Sep 27 22:16 my_fifo
 343929 srwxr-xr-x   1 root     root         0 May 20  2006 klaunchertIdhOa.
slave-socket
```

除设备文件ttySAC0、mtdblock0外，这些信息都分为8个字段，比如：

```
228883 -rw-r--r--   2 root     root         6 Sep 27 22:10 readme.txt
字段1    2         3 4        5            6     7        8
```

它们的意义如下。

（1）字段1：文件的索引节点inode。

索引节点里存放一个文件的上述信息，比如文件大小、属主、归属的用户组、读写权限等，并指明文件的实际数据存放的位置。

（2）字段2：文件种类和权限。

这字段共分10位，格式如下。

文件类型有7种，“-”表示普通文件，“d”表示目录，“c”表示字符设备，“b”表示块

设备，“p”表示 FIFO（即管道），“l”表示软连接（也称符号连接），“s”表示套接口（socket）。

没有专门的符号来表示“硬连接”类型，硬连接也是普通文件，只不过文件的实际内容只有一个副本，连接文件、被连接文件都指向它。比如上面的 ln_hard 文件是使用命令“ln readme.txt ln_hard”创建出来的到 readme.txt 文件的硬连接，readme.txt 和 ln_hard 的地位完全一致，它们都指向文件系统中的同一个位置，它们的“硬连接个数”都是 2，表示这个文件的实际内容被引用两次，可以看到这两个文件的 inode 都是 228883。

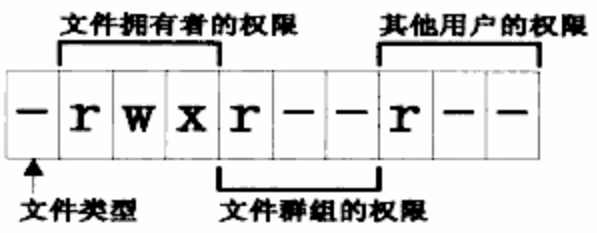


图 17.2 文件类型及属性

硬连接文件的引入的另一个作用是使得可以用别名来引用一个文件，避免文件被误删除——只有当硬连接个数为 1 时，对一个文件执行删除操作才会真正删除文件的副本。它的缺点是不能创建到目录的连接，被连接文件和连接文件必须在同一个文件系统中。对此，引入软连接，也称符号连接，软连接只是简单地指向一个文件（可以是目录），并不增加它的硬连接个数。比如上面的 ln_soft 文件就是使用命令“ln -s readme.txt ln_hard”创建出来的到 readme.txt 文件的软连接，它使用另一个 inode。

剩下的 9 位分为 3 组，分别用来表示文件拥有者、同一个群组的用户、其他用户对这个文件的访问权限。每组权限由 rwx 三位组成，表示可读、可写、可执行。如果某一位被设为“-”，则表示没有相应的权限，比如“rw-”表示只有读写权限，没有执行权限。

（3）字段 3：硬连接个数。
（4）字段 4：文件拥有者。
（5）字段 5：所属群组。
（6）字段 6：文件或目录的大小。
（7）字段 7：最后访问或修改时间。
（8）字段 8：文件名或目录名。

对于设备文件，字段 6 表示主设备号，字段 7 表示次设备号。

17.2 移植 Busybox

所谓制作根文件系统，就是创建各种目录，并且在里面创建各种文件。比如在/bin、/sbin 目录下存放各种可执行程序，在/etc 目录下存放配置文件，在/lib 目录下存放库文件。这节讲述如何使用 Busybox 来创建/bin、/sbin 等目录下的可执行文件。

17.2.1 Busybox 概述

Busybox 是一个遵循 GPL v2 协议的开源项目。Busybox 将众多的 UNIX 命令集合进一个很小的可执行程序中，可以用来替换 GNU fileutils、shellutils 等工具集。Busybox 中各种命令与相应的 GNU 工具相比，所能提供的选项较少，但是能够满足一般应用。Busybox 为各种小型的或者嵌入式系统提供了一个比较完全的工具集。

Busybox 在编写过程中对文件大小进行优化，并考虑了系统资源有限（比如内存等）的

情况。与一般的 GNU 工具集动辄几 MB 的体积相比，动态连接的 Busybox 只有几百 KB，即使静态连接也只有 1MB 左右。Busybox 按模块进行设计，可以很容易地加入、去除某些命令，或增减命令的某些选项。

在创建一个最小的根文件系统时，使用 Busybox 的话，只需要在/dev 目录下创建必要的设备节点、在/etc 目录下创建一些配置文件就可以了，如果 Busybox 使用动态连接，还要在/lib 目录下包含库文件。

Busybox 支持 uClibc 库和 glibc 库，对 Linux 2.2.x 之后的内核支持良好。

Busybox 的官方网站是 http://www.busybox.net/，源码可以从 http://www. busybox.net/downloads/下载，本书使用 busybox-1.7.0.tar.bz2。

17.2.2　init 进程介绍及用户程序启动过程

init 进程是由内核启动的第一个（也是惟一的一个）用户进程（进程 ID 为 1），它根据配置文件决定启动哪些程序，比如执行某些脚本、启动 shell、运行用户指定的程序等。init 进程是后续所有进程的发起者，比如 init 进程启动/bin/sh 程序后，才能够在控制台上输入各种命令。

init 进程的执行程序通常是/sbin/init，上面讲述的 init 进程的作用只不过是/sbin/init 这个程序的功能。我们完全可以编写自己的/sbin/init 程序，或者传入命令行参数“init=xxxxx”指定某个程序作为 init 进程运行。

一般而言，在 Linux 系统有两种 init 程序：BSD init 和 System V init。BSD 和 System V 是两种版本的 UNIX 系统。这两种 init 程序各有优缺点，现在大多 Linux 的发行版本使用 System V init。但是在嵌入式领域，通常使用 Busybox 集成的 init 程序，下面基于它进行讲解。

1. 内核如何启动 init 进程

内核启动的最后一步就是启动 init 进程，代码在 init/main.c 文件中，如下所示：

```
748 static int noinline init_post(void)
749 {
…
756     if (sys_open((const char _ _user *) "/dev/console", O_RDWR, 0) < 0)
757       printk(KERN_WARNING "Warning: unable to open an initial console.\n");
758
759     (void) sys_dup(0);
760     (void) sys_dup(0);
761
762     if (ramdisk_execute_command) {
763         run_init_process(ramdisk_execute_command);
764         printk(KERN_WARNING "Failed to execute %s\n",
765                 ramdisk_execute_command);
766     }
```

```
...
774     if (execute_command) {
775         run_init_process(execute_command);
776         printk(KERN_WARNING "Failed to execute %s.  Attempting "
777                     "defaults...\n", execute_command);
778     }
779     run_init_process("/sbin/init");
780     run_init_process("/etc/init");
781     run_init_process("/bin/init");
782     run_init_process("/bin/sh");
783
784     panic("No init found.  Try passing init= option to kernel.");
785 }
786
```

代码并不复杂，其中的 run_init_process 函数使用它的参数所指定的程序来创建一个用户进程。需要注意，一旦 run_init_process 函数创建进程成功，它将不会返回。

内核启动 init 进程的过程如下。

（1）打开标准输入、标准输出、标准错误设备。

Linux 中最先打开的 3 个文件分别称为标准输入（stdin）、标准输出（stdout）、标准错误（stderr），它们对应的文件描述符分别为 0、1、2。所谓标准输入就是在程序中使用 scanf()、fscanf(stdin,…)获取数据时，从哪个文件（设备）读取数据；标准输出、标准错误都是输出设备，前者对应 printf()、fprintf(stdout,…)，后者对应 fprintf(stderr,…)。

第 756 行尝试打开/dev/console 设备文件，如果成功，它就是 init 进程标准输入设备。

第 759、760 将文件描述符 0 复制给文件描述符 1、2，所以标准输入、标准输出、标准错误都对应同一个文件（设备）。

在移植 Linux 内核时，如果发现打印出“Warning: unable to open an initial console.”，其原因大多是：根文件系统虽然被正确挂接了，但是里面的内容不正确，要么没有/dev/console 这个文件，要么它没有对应的设备。

（2）如果 ramdisk_execute_command 变量指定了要运行的程序，启动它。

ramdisk_execute_command 的取值（代码也在 init/main.c 中）分 3 种情况。

① 如果命令行参数中指定了“rdinit=…”，则 ramdisk_execute_command 等于这个参数指定的程序。

② 否则，如果/init 程序存在，ramdisk_execute_command 就等于“/init”。

③ 否则，ramdisk_execute_command 为空。

本书所用的命令行没有设定“rdinit=…”，根文件系统中也没有/init 程序，所以 ramdisk_execute_command 为空，第 763～765 这几行的代码不执行。

（3）如果 execute_command 变量指定了要运行的程序，启动它。

如果命令行参数中指定了“init=…”，则 execute_command 等于这个参数指定的程序，否则为空。

本书所用的命令行没有设定“init=…”，所以第775～777行代码不执行。

（4）依次尝试执行/sbin/init、/etc/init、/bin/init、/bin/sh。

第779行执行/sbin/init程序，这个程序在我们的根文件系统中是存在的，所以init进程所用的程序就是/sbin/init。从此系统的控制权交给/sbin/init，不再返回init_post函数中。

run_init_process函数也在init/main.c中，代码如下：

```
184 static char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
185 char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, };
…
739 static void run_init_process(char *init_filename)
740 {
741     argv_init[0] = init_filename;
742     kernel_execve(init_filename, argv_init, envp_init);
743 }
744
```

所以执行/sbin/init程序时，它的环境参数为“"HOME=/", "TERM=linux"”。

2．Busybox init进程的启动过程

Busybox init程序对应的代码在init/init.c文件中，下面以busybox-1.7.0为例进行讲解。先概述其流程，再结合一个/etc/inittab文件讲述init进程的启动过程。

（1）Busybox init程序流程。

流程图如图17.3所示，其中与构建根文件系统关系密切的是控制台的初始化、对inittab文件的解释及执行。

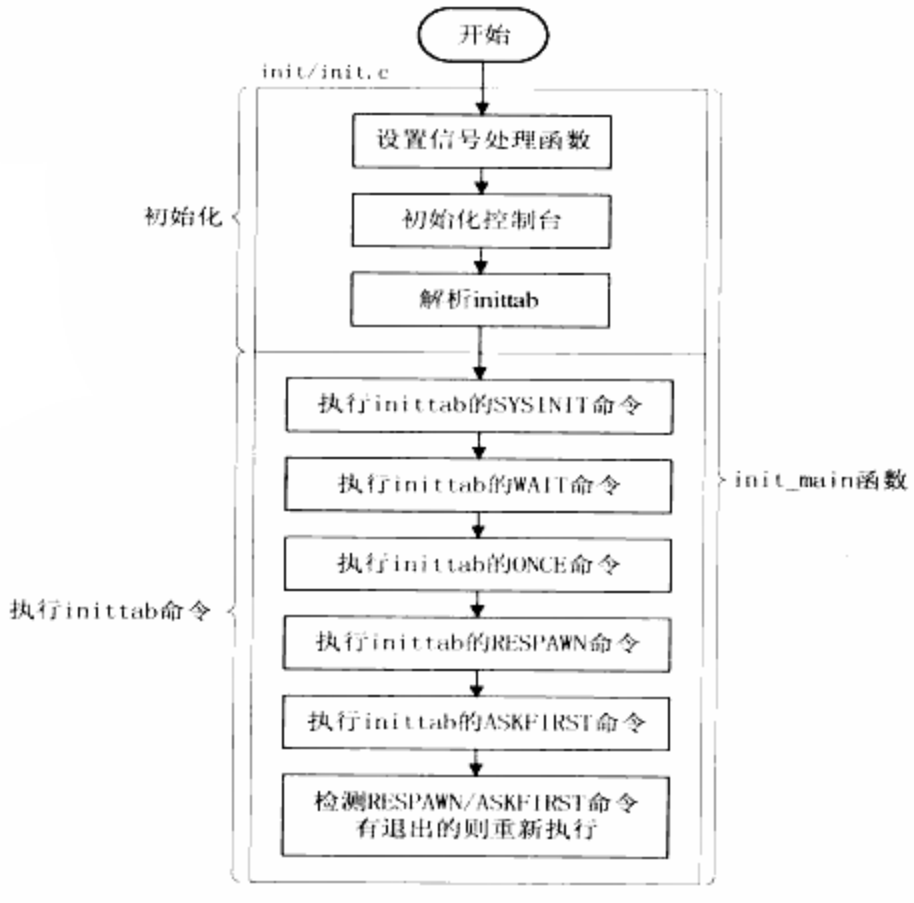


图17.3　Busybox init程序流程

内核启动 init 进程时已经打开“/dev/console”设备作为控制台，一般情况下 Busybox init 程序就使用/dev/console。但是如果内核启动 init 进程的同时设置了环境变量 CONSOLE 或 console，则使用环境变量所指定的设备。在 Busybox init 程序中，还会检查这个设备是否可以打开，如果不能打开则使用“/dev/null”。

Busybox init 进程只是作为其他进程的发起者和控制者，并不需要控制台与用户交互，所以 init 进程会把它关掉，系统启动后运行命令“ls /proc/1/fd/”可以看到该目录为空。init 进程创建其他子进程时，如果没有在/etc/inittab 中指明它的控制台，则使用前面确定的控制台。

/etc/inittab 文件的相关文档和示例代码都在 Busybox 的 examples/inittab 文件中。

如果存在/etc/inittab 文件，Busybox init 程序解析它，然后按照它的指示创建各种子进程；否则使用默认的配置创建子进程。

/etc/inittab 文件中每个条目用来定义一个子进程，并确定它的启动方法，格式如下：

```
<id>:<runlevels>:<action>:<process>
```

例如：

```
ttySAC0::askfirst:-/bin/sh
```

对于 Busybox init 程序，上述各个字段作用如下。

① <id>：表示这个子进程要使用的控制台（即标准输入、标准输出、标准错误设备）。如果省略，则使用与 init 进程一样的控制台。

② <runlevels>：对于 Busybox init 程序，这个字段没有意义，可以省略。

③ <action>：表示 init 进程如何控制这个子进程，有如表 17.6 所示的 8 种取值。

表 17.6　　/etc/inittab 文件中<action>字段的意义

action 名称	执 行 条 件	说　明
sysinit	系统启动后最先执行	只执行一次，init 进程等待它结束才继续执行其他动作
wait	系统执行完 sysinit 进程后	只执行一次，init 进程等待它结束才继续执行其他动作
once	系统执行完 wait 进程后	只执行一次，init 进程不等待它结束
respawn	启动完 once 进程后	init 进程监测发现子进程退出时，重新启动它
askfirst	启动完 respawn 进程后	与 respawn 类似，不过 init 进程先输出“Please press Enter to activate this console.”，等用户输入回车键之后才启动子进程。
shutdown	当系统关机时	即重启、关闭系统命令时
restart	Busybox 中配置了 CONFIG_FEATURE_USE_INITTAB，并且 init 进程接收到 SIGHUP 信号时	先重新读取、解析/etc/inittab 文件，再执行 restart 程序
ctrlaltdel	按下 Ctrl+Alt+Del 组合键时	—

④ <process>：要执行的程序，它可以是可执行程序，也可以是脚本。

如果<procss>字段前有“-”字符，这个程序被称为“交互的”。

在/etc/inittab 文件的控制下，init 进程的行为总结如下。

① 在系统启动前期，init 进程首先启动<action>为 sysinit、wait、once 的 3 类子进程。

② 在系统正常运行期间，init 进程首先启动<action>为 respawn、askfirst 的两类子进程，并监视它们，发现某个子进程退出时重新启动它。

③ 在系统退出时，执行<action>为 shutdown、restart、ctrlaltdel 的 3 类子进程（之一或全部）。

如果根文件系统中没有/etc/inittab 文件，Busybox init 程序将使用如下默认的 inittab 条目。

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
tty2::askfirst:/bin/sh
tty3::askfirst:/bin/sh
tty4::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
::restart:/sbin/init
```

（2）/etc/inittab 实例。

仿照 Busybox 的 examples/inittab 文件，创建一个 inittab 文件，内容如下：

```
# /etc/inittab
# 这是 init 进程启动的第一个子进程，它是一个脚本，可以在里面指定用户想执行的操作
# 比如挂接其他文件系统、配置网络等
::sysinit:/etc/init.d/rcS

# 启动 shell，以/dev/ttySAC0 作为控制台
ttySAC0::askfirst:-/bin/sh

# 按下 Ctrl+Alt+Del 之后执行的程序，不过在串口控制台中无法输入 Ctrl+Alt+Del 组合键
::ctrlaltdel:/sbin/reboot
# 重启、关机前执行的程序
::shutdown:/bin/umount -a -r
```

17.2.3 编译/安装 Busybox

从 http://www.busybox.net/downloads/下载 busybox-1.7.0.tar.bz2。

使用如下命令解压得到 busybox-1.7.0 目录，里面就是所有的源码。

```
$ tar xjf busybox-1.7.0.tar.bz2
```

Busybox 集合了几百个命令，在一般系统中并不需要全部使用。可以通过配置 Busybox 来选择这些命令、定制某些命令的功能（选项）、指定 Busybox 的连接方法（动态连接还是

静态连接）、指定 Busybox 的安装路径。

1. 配置 Busybox

在 busybox-1.7.0 目录下执行“make menuconfig”命令即可进入配置界面。Busybox 将所有配置项分类存放，表 17.7 列出了这些类别，其中的“说明”是针对嵌入式系统而言的。

表 17.7　　Busybox 配置选项分类

配置项类型	说　明
Busybox Settings ---> General Configuration	一些通用的设置，一般不需要理会
Busybox Settings ---> Build Options	连接方式、编译选项等
Busybox Settings ---> Debugging Options	调试选项，使用 Busybox 时将打印一些调试信息，一般不选
Busybox Settings ---> Installation Options	Busybox 的安装路径，不需设置，可以在命令行中指定
Busybox Settings ---> Busybox Library Tuning	Busybox 的性能微调，比如设置在控制台上可以输入的最大字符个数，一般使用默认值即可
Archival Utilities	各种压缩、解压缩工具，根据需要选择相关命令
Coreutils	核心的命令，比如 ls、cp 等
Console Utilities	控制台相关的命令，比如清屏命令 clear 等。只是提供一些方便而已，可以不理会
Debian Utilities	Debian 命令（Debian 是 Linux 的一种发行版本），比如 which 命令可以用来显示一个命令的完整路径
Editors	编辑命令，一般都选中 vi
Finding Utilities	查找命令，一般不用
Init Utilities	init 程序的配置选项，比如是否读取 inittab 文件，使用默认配置即可
Login/Password Management Utilities	登录、用户账号/密码等方面的命令
Linux Ext2 FS Progs	Ext2 文件系统的一些工具
Linux Module Utilities	加载/卸载模块的命令，一般都选中
Linux System Utilities	一些系统命令，比如显示内核打印信息的 dmesg 命令、分区命令 fdisk 等
Miscellaneous Utilities	一些不好分类的命令
Networking Utilities	网络方面的命令，可以选择一些可以方便调试的命令，比如 telnetd、ping、tftp 等
Process Utilities	进程相关的命令，比如查看进程状态的命令 ps、查看内存使用情况的命令 free、发送信号的命令 kill、查看最消耗 CPU 资源的前几个进程的命令 top 等。为方便调试，可以都选中

续表

配置项类型	说　明
Shells	有多种 shell，比如 msh、ash 等，一般选择 ash
System Logging Utilities	系统记录（log）方面的命令
Runit Utilities	本书没有用到
ipsvd utilities	监听 TCP、DPB 端口，发现有新的连接时启动某个程序

本节使用默认配置，执行“make menuconfig”后退出、保存配置即可。

下面只讲述一些常用的选项，以便读者参考。Busybox 的配置过程大多是选择、去除各种命令，一目了然。

（1）Busybox 的性能微调。

设置“TAB”键补全，比如在控制给上输入一个“ifc”后按“TAB”键，它会补全为“ifconfig”。如下配置：

```
Busybox Settings  --->
    Busybox Library Tuning  --->
        [*]   Tab completion
```

（2）连接/编译选项。

以下选项指定是否使用静态连接：

```
Build Options  --->
    [ ] Build BusyBox as a static binary (no shared libs)
```

使用 glibc 时，如果静态编译 Buxybox 会提示以下警告信息，表示会出现一些莫名其秒的问题。

```
#warning Static linking against glibc produces buggy executables
```

所以，本书使用动态连接的 Busybos，在构造根文件系统时需要在/lib 目录下放置 glibc 库文件。

（3）Archival Utilities 选项。

选择 tar 命令：

```
Archival Utilities  --->
    [*] tar
    [*]     Enable archive creation
    [*]     Enable -j option to handle .tar.bz2 files
    [*]     Enable -X (exclude from) and -T (include from) options)
    [*]     Enable -z option
    [*]     Enable -Z option
    [*]     Enable support for old tar header format
    [*]     Enable support for some GNU tar extensions
    [*]     Enable long options
```

（4）Linux Module Utilities 选项。

要使用可加载模块，下面的配置要选上。

```
Linux Module Utilities  --->
    [*] insmod
    [*]   Module version checking
    [*]   Add module symbols to kernel symbol table
    [*]   In kernel memory optimization (uClinux only)
    [*]   Enable load map (-m) option
    [*]     Symbols in load map
    [*] rmmod
    [*] lsmod
    [*] Support version 2.6.x Linux kernels
```

（5）Linux System Utilities 选项。

支持 mdev，这可以很方便地构造/dev 目录，并且可以支持热拔插设备。另外，为方便调试，选中 mount、umount 命令，并让 mount 命令支持 NFS（网络文件系统）。

```
Linux System Utilities  --->
    [*] mdev
    [*]   Support /etc/mdev.conf
    [*]     Support command execution at device addition/removal
    [*] mount
    [*]     Support mounting NFS file systems
    [*] umount
    [*]   umount -a option
```

（6）Networking Utilities 选项。

除其他默认配置外，增加 ifconfig 命令。

```
Networking Utilities  --->
    [*] ifconfig
    [*]     Enable status reporting output (+7k)
    [ ]     Enable slip-specific options "keepalive" and "outfill"
    [ ]     Enable options "mem_start", "io_addr", and "irq"
    [*]     Enable option "hw" (ether only)
    [*]     Set the broadcast automatically
```

2. 编译和安装 Busybox

编译之前，先修改 Busybox 根目录的 Makefile，使用交叉编译器。

修改前：

```
175 ARCH            ?= $(SUBARCH)
176 CROSS_COMPILE ?=
```

修改后：

```
175 ARCH          ?= arm
176 CROSS_COMPILE   ?= arm-linux-
```

然后可执行“make”命令编译Busybox。

最后是安装，执行“make CONFIG_PREFIX=dir_path install”就可以将Busybox安装在dir_name指定的目录下。执行以下命令在/work/nfs_root/fs_mini目录下安装Busybox。

```
$ make CONFIG_PREFIX=/work/nfs_root/fs_mini install
```

一切完成后，将在/work/nfs_root/fs_mini目录下生成如下文件、目录。

```
drwxr-xr-x 2 book book 4096 2008-01-22 06:56 bin
lrwxrwxrwx 1 book book   11 2008-01-22 06:56 linuxrc -> bin/busybox
drwxr-xr-x 2 book book 4096 2008-01-22 06:56 sbin
drwxr-xr-x 4 book book 4096 2008-01-22 06:56 usr
```

其中linuxrc和上面分析的/sbin/init程序功能完全一样；其他目录下是各种命令，不过它们都是到/bin/busybox的符号连接，比如/work/nfs_root/fs_mini/sbin目录下：

```
lrwxrwxrwx 1 book book 14 2008-01-22 06:56 halt -> ../bin/busybox
lrwxrwxrwx 1 book book 14 2008-01-22 06:56 ifconfig -> ../bin/busybox
lrwxrwxrwx 1 book book 14 2008-01-22 06:56 init -> ../bin/busybox
lrwxrwxrwx 1 book book 14 2008-01-22 06:56 insmod -> ../bin/busybox
lrwxrwxrwx 1 book book 14 2008-01-22 06:56 klogd -> ../bin/busybox
...
```

除bin/busybox外，其他文件都是到bin/busybox的符号连接。busybox是所有命令的集合体，这些符号连接文件可以直接运行。比如在开发板上，运行“ls”命令和“busybox ls”命令是一样的。

17.3 使用glibc库

在第2章制作交叉编译工具链时，已经生成了glibc库，可以直接使用它来构建根文件系统。

17.3.1 glibc库的组成

glibc库的位置是/work/tools/gcc-3.4.5- glibc-2.3.6/arm-linux/lib。

这个目录下的文件并非都属于glibc库，比如crt1.o、libstdc++.a等文件是GCC工具本身生成的。本书不区分它们的来源，统一处理。

里面的目录、文件可以分为8类。

① 加载器ld-2.3.6.so、ld-linux.so.2。

动态程序启动前，它们被用来加载动态库。

② 目标文件（.o）。

比如 crt1.o、crti.o、crtn.o、gcrt1.o、Mcrt1.o、Scrt1.o 等。在生成应用程序时，这些文件像一般的目标文件一样被连接。

③ 静态库文件（.a）。

比如静态数学库 libm.a、静态 C++库 libstdc++.a 等，编译静态程序时会连接它们。

④ 动态库文件（.so、.so.[0-9]*）。

比如动态数学库 libm.so、动态 C++库 libstdc++.so 等，它们可能是一个链接文件。编译动态库时会用到这些文件，但是不会连接它们，运行时才连接。

⑤ libtool 库文件（.la）。

在连接库文件时，这些文件会被用到，比如它们列出了当前库文件所依赖的其他库文件。程序运行时无需这些文件。

⑥ gconv 目录。

里面是有头字符集的动态库，比如 ISO8859-1.so、GB18030.so 等。

⑦ ldscripts 目录。

里面是各种连接脚本，在编译应用程序时，它们被用于指定程序的运行地址、各段的位置等。

⑧ 其他目录及文件。

17.3.2 安装 glibc 库

在开发板上只需要加载器和动态库，假设要构建的根文件系统目录为/work/nfs_root/fs_mini，操作如下。

```
$ mkdir -p /work/nfs_root/fs_mini/lib
$ cd /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
$ cp *.so* /work/nfs_root/fs_mini/lib -d
```

上面复制的库文件不是每个都会被用到，可以根据应用程序对库的依赖关系保留需要用到的。通过 ldd 命令可以查看一个程序会用到哪些库，主机自带的 ldd 命令不能查看交叉编译出来的程序，有以下两种替代方法。

① 如果有 uClibc-0.9.28 的代码，可以进入 utils 子目录生成 ldd.host 工具。

```
$ cd uClibc-0.9.28/utils
$ make ldd.host
```

然后将生成的 ldd.host 放到主机/usr/local/bin 目录下即可使用。

比如对于动态连接的 Busybox，它的库依赖关系如下：

```
$ ldd.host busybox
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x00000000)
        libm.so.6 => /lib/libm.so.6 (0x00000000)
        libc.so.6 => /lib/libc.so.6 (0x00000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00000000)
```

这表示 Busybox 要使用的库文件有 libcrypt.so.1、libm.so.6、libc.so.6，加载器为/lib/ld-linux.so.2（实际上在交叉工具链目录下，加载器为 ld-linux.so.2）。上面的“not found”表示主机上没有这个文件，这没关系，开发板的根文件系统上有就行。

② 可以使用以下命令：

```
$ arm-linux-readelf -a "your binary" | grep "Shared"
```

比如对于动态连接的 Busybox，它的库依赖关系如下：

```
$ arm-linux-readelf -a ./busybox | grep "Shared"
 0x00000001 (NEEDED)                     Shared library: [libcrypt.so.1]
 0x00000001 (NEEDED)                     Shared library: [libm.so.6]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
```

里面没有列出加载器，构造根文件系统时，它也要复制进去。

17.4　构建根文件系统

上面两节在介绍了如何安装 Busbybox、C 库，建立了 bin/、sbin/、usr/bin/、usr/sbin/、lib/等目录，最小根文件系统的大部分目录、文件已经建好。本节介绍剩下的部分，假设开发板的根文件系统在主机上的目录为/work/nfs_root/fs_mini。

17.4.1　构建 etc 目录

init 进程根据/etc/inittab 文件来创建其他子进程，比如调用脚本文件配置 IP 地址、挂接其他文件系统，最后启动 shell 等。

etc 目录下的内容取决于要运行的程序，本节只需要创建 3 个文件：etc/inittab、etc/init.d/rcS、etc/fstab。

1. 创建 etc/inittab 文件

仿照 Busybox 的 examples/inittab 文件，在/work/nfs_root/fs_mini/etc 目录下创建一个 inittab 文件，内容如下。

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
ttySAC0::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

2. 创建 etc/init.d/rcS 文件

这是一个脚本文件，可以在里面添加想自动执行命令。以下命令配置 IP 地址、挂接/etc/fstab 指定的文件系统。

```
#!/bin/sh
ifconfig eth0 192.168.1.17
mount -a
```

第一行表示这是一个脚本文件，运行时使用/bin/sh 解析。

第二行用来配置 IP 地址。

第三行挂接/etc/fstab 文件指定的所有文件系统。

最后，还要改变它的属性，使它能够执行。

```
chmod +x etc/init.d/rcS
```

3. 创建 etc/fstab 文件

内容如下，表示执行“mount -a”命令后将挂接 proc、tmpfs 文件系统。

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

/etc/fstab 文件被用来定义文件系统的“静态信息”，这些信息被用来控制 mount 命令的行为。文件中各字段意义如下。

① device：要挂接的设备。

比如/dev/hda2、/dev/mtdblock1 等设备文件；也可以是其他格式，比如对于 proc 文件系统这个字段没有意义，可以是任意值；对于 NFS 文件系统，这个字段为<host>:<dir>。

② mount-point：挂接点。

③ type：文件系统类型。

比如 proc、jffs2、yaffs、ext2、nfs 等；也可以是 auto，表示自动检测文件系统类型。

④ options：挂接参数，以逗号隔开。

/etc/fstab 的作用不仅仅是用来控制“mount -a”的行为，即使是一般的 mount 命令也受它控制，这可以从表 17.8 的参数看到。除与文件系统类型相关的参数外，常用的有以下几种取值。

表 17.8　　/etc/fstab 参数字段常用的取值

参 数 名	说 明	默 认 值
auto noauto	决定执行“mount -a”时是否自动挂接。 auto：挂接；noauto：不挂接	auto
user nouser	user：允许普通用户挂接设备； nouser：只允许 root 用户挂接设备	nouser
exec noexec	exec：允许运行所挂接设备上的程序 noexec：不允许运行所挂接设备上的程序	exec
Ro	以只读方式挂接文件系统	-
rw	以读写方式挂接文件系统	-
sync async	sync：修改文件时，它会同步写入设备中； async：不会同步写入	sync
defaults	rw、suid、dev、exec、auto、nouser、async 等的组合	-

⑤ dump 和 fsck order：用来决定控制 dump、fsck 程序的行为。

dump 是一个用来备份文件的程序，fsck 是一个用来检查磁盘的程序。要想了解更多信息，请阅读它们的 man 手册。

dump 程序根据 dump 字段的值来决定这个文件系统是否需要备份，如果没有这个字段，或其值为 0，则 dump 程序忽略这个文件系统。

fsck 程序根据 fsck order 字段来决定磁盘的检查顺序，一般来说对于根文件系统这个字段设为 1，其他文件系统设为 2。如果设为 0，则 fsck 程序忽略这个文件系统。

17.4.2 构建 dev 目录

本节使用两种方法构建 dev 目录。

1. 静态创建设备文件

为简单起见，本书先使用最原始的方法处理设备：在/dev 目录下静态创建各种节点（即设备文件）。

从系统启动过程可知，涉及的设备有：/dev/mtdblock*(MTD 块设备)、/dev/ttySAC*（串口设备）、/dev/console、/dev/null，只要建立以下设备就可以启动系统。

```
$ mkdir -p /work/nfs_root/fs_mini/dev
$ cd /work/nfs_root/fs_mini/dev
$ sudo mknod console c 5 1
$ sudo mknod null c 1 3
$ sudo mknod ttySAC0 c 204 64
$ sudo mknod mtdblock0 b 31 0
$ sudo mknod mtdblock1 b 31 1
$ sudo mknod mtdblock2 b 31 2
```

注意 在一般系统中，ttySAC0 的主设备号为 4，但是在 S3C2410、S3C2440 所用的 Linux 2.6.22.6 上，它们的串口主设备号为 204。

其他设备文件可以当系统启动后，使用“cat /proc/devices”命令查看内核中注册了哪些设备，然后一一创建相应的设备文件。

实际上，各个 Linux 系统中 dev 目录的内容很相似，本书最终使用的 dev 目录就是从其他系统中复制过来的。

2. 使用 mdev 创建设备文件

mdev 是 udev 的简化版本，它也是通过读取内核信息来创建设备文件。

mdev 的用法请参考 busybox-1.7.0/doc/mdev.txt 文件。mdev 的用途主要有两个：初始化/dev 目录、动态更新。动态更新不仅是更新/dev 目录，还支持热拔插，即接入、卸下设备时执行某些动作。

要使用 mdev，需要内核支持 sysfs 文件系统，为了减少对 Flash 的读写，还要支持 tmpfs

文件系统。先确保内核已经设置了 CONFIG_SYSFS、CONFIG_TMPFS 配置项。

使用 mdev 的命令如下，请参考它们的注释以了解其作用：

```
$ mount -t tmpfs mdev /dev                    /* 使用内存文件系统，减少对 Flash 的读写 */
$ mkdir /dev/pts                    /* devpts 用来支持外部网络连接(telnet)的虚拟终端 */
$ mount -t devpts devpts /dev/pts
$ mount -t sysfs sysfs /sys                   /* mdev 通过 sysfs 文件系统获得设备信息 */
$ echo /bin/mdev > /proc/sys/kernel/hotplug   /* 设置内核，当有设备拔插时调用/bin/mdev
程序 */
$ mdev -s                                     /* 在/dev 目录下生成内核支持的所有设备的结点 */
```

要在内核启动时，自动运行 mdev。这要修改/work/nfs_root/fs_mini 中的两个文件：修改 etc/fstab 来自动挂载文件系统、修改 etc/init.d/rcS 加入要自动运行的命令。修改后的文件如下。

① etc/fstab。

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

② etc/init.d/rcS：加入下面几行。

```
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

需要注意的是，开发板上通过 mdev 生成的/dev 目录中，S3C2410、S3C2440 是串口名是 s3c2410_serial 0、s3c2410_serial 1、s3c2410_serial 2，不是 ttySAC0、ttySAC1、ttySAC2。需要修改 etc/inittab 文件。

修改前：

```
ttySAC0::askfirst:-/bin/sh
```

修改后：

```
s3c2410_serial0::askfirst:-/bin/sh
```

另外，mdev 是通过 init 进程来启动的，在使用 mdev 构造/dev 目录之前，init 进程至少要用到设备文件/dev/console、/dev/null，所以要建立这两个设备文件。

```
$ mkdir -p /work/nfs_root/fs_mini/dev
$ cd /work/nfs_root/fs_mini/dev
$ sudo mknod console c 5 1
$ sudo mknod null c 1 3
```

17.4.3　构建其他目录

其他目录可以是空目录，比如 proc、mnt、tmp、sys、root 等，如下创建：

```
# cd /work/nfs_root/fs_mini
# mkdir proc mnt tmp sys root
```

现在，/work/nfs_root/fs_mini 目录下就是一个非常小的根文件系统。开发板可以将它作为网络根文件系统直接启动。如果要烧入开发板，还要将它制作为一个文件，称为映象文件。

17.4.4　制作/使用 yaffs 文件系统映象文件

按照前面的方法，在/work/nfs_root 目录下构造了两个根文件系统：fs_mini、fs_mini_mdev。前者使用 dev/目录中事先建立好的设备文件，后者使用 mdev 机制来生成 dev/目录，它们的差别只在于 3 点：etc/inittab 文件、etc/init.d/rcS 文件、dev/目录。下面以/work/nfs_root/fs_mini 为例制作根文件系统映象。

所谓制作文件系统映象文件，就是将一个目录下的所有内容按照一定的格式存放到一个文件中，这个文件可以直接烧写到存储设备上去。当系统启动后挂接这个设备，就可以看到与原来目录一样的内容。

制作不同类型的文件系统映象文件需要使用不同的工具。

1．修改制作 yaffs 映象文件的工具

在 yaffs 源码中有个 utils 目录（假设这个目录为/work/system/Development/ yaffs2/utils），里面是工具 mkyaffsimage 和 mkyaffs2image 的源代码。前者用来制作 yaffs1 映象文件，后者用来制作 yaffs2 映象文件。

目前 mkyaffsimage 工具只能生成老格式的 yaffs1 映象文件，需要修改才能支持新格式。对 mkyaffsimage 代码的修改都在补丁文件 yaffs_util_mkyaffsimage.patch 中，读者可以直接打补丁，也可以根据本小节进行修改。

yaffs1 新、老格式的不同在于 oob 区的使用发生了变化：一是 ECC 检验码的位置发生了变化，二是可用空间即标记（tag）的数据结构定义发生了变化。

另外，由于配置内核时没有设置 CONFIG_YAFFS_DOES_ECC，yaffs 文件系统将使用 MTD 设备层的 ECC 校验方法，制作映象文件时也使用与 MTD 设备层相同的函数计算 ECC 码。

① oob 区中检验码的位置变化：

oob 区中使用 6 个字节来存放 ECC 校验码，前 3 个字节对应上半页，后 3 个字节对应下半页。

由 nand_oob_16 结构可知，以前的校验码在 oob 区中存放的位置为 8、9、10、13、14 和 15，现在改为 0、1、2、3、6 和 7。

② oob 区中可用空间的数据结构定义变化。

oob 区中可用的空间有 8 个字节，它用来存放文件系统的数据，代码中这些数据被称为标记（tag）。

老格式的 yaffs1 中，这 8 个字节的数据结构定义如下（在 yaffs_guts.h 文件中）所示：

```
typedef struct {
    unsigned chunkId:20;
    unsigned serialNumber:2;
    unsigned byteCount:10;
    unsigned objectId:18;
    unsigned ecc:12;
    unsigned unusedStuff:2;
} yaffs_Tags;
```

新格式的 yaffs1 中，定义如下（在 yaffs_packedtags1.h 文件中）所示：

```
typedef struct {
    unsigned chunkId:20;
    unsigned serialNumber:2;
    unsigned byteCount:10;
    unsigned objectId:18;
    unsigned ecc:12;
    unsigned deleted:1;
    unsigned unusedStuff:1;
    unsigned shouldBeFF;      /* 新格式中，这个字节没有使用，yaffs_PackedTags1 还是 8
个字节 */
} yaffs_PackedTags1;
```

新、老结构有细微差别：老结构中有两位没有使用（unusedStuff）；新结构中只有一位没有使用，另一位（deleted）被用来表示当前页是否已经删除。

③ oob 区中 ECC 码的计算。

如果配置内核时设置了 CONFIG_YAFFS_DOES_ECC，则 yaffs 文件系统将使用 yaffs2/yaffs_ecc.c 文件中的 yaffs_ECCCalculate 函数来计算 ECC 码；否则使用 drivers/mtd/ nand/nand_ecc.c 文件中的 nand_calculate_ecc 函数。

mkyaffsimage 工具原来的代码中使用 yaffs_ECCCalculate 函数。由于上面配置内核时，没有选择 CONFIG_YAFFS_DOES_ECC，为了使映象文件与内核保持一致，要修改 mkyaffsimage 源码，使用 nand_calculate_ecc 函数。

对 mkyaffsimage 的修改如下所示。

① 增加头文件。

修改文件 mkyaffsimage.c，加上下面这行，里面定义了 yaffs_PackedTags1 结构。

```
#include "yaffs_packedtags1.h"
```

② 修改 mkyaffsimage.c 文件的 write_chunk 函数。

代码如下：

```
231 static int write_chunk(__u8 *data, __u32 objId, __u32 chunkId, __u32 nBytes)
232 {
233 #ifdef CONFIG_YAFFS_9BYTE_TAGS   /* 如果要生成老格式的 yaffs1 映象文件，定义这个宏 */
... /* 原来的代码 */
```

```
260 #else
261     yaffs_PackedTags1 pt1;
262     yaffs_ExtendedTags  etags;
263     __u8 ecc_code[6];
264     __u8 oobbuf[16];
265
266     /* 写页数据，512个字节 */
267     error = write(outFile,data,512);
268     if(error < 0) return error;
269
270     /* 构造tag */
271     etags.chunkId       = chunkId;
272     etags.serialNumber  = 0;
273     etags.byteCount     = nBytes;
274     etags.objectId      = objId;
275     etags.chunkDeleted  = 0;
276
277     /*
278      * 重定位oob区中的可用数据(称为tag)
279      */
280     yaffs_PackTags1(&pt1, &etags);
281
282     /* 计算tag本身的ECC码 */
283     yaffs_CalcTagsECC((yaffs_Tags *)&pt1);
284
285     memset(oobbuf, 0xff, 16);
286     memcpy(oobbuf+8, &pt1, 8);
287
288     /*
289      * 使用与内核MTD层相同的方法计算一页数据(5124字节)的ECC码
290      * 并把它们填入oob
291      */
292     nand_calculate_ecc(data, &ecc_code[0]);
293     nand_calculate_ecc(data+256, &ecc_code[3]);
294
295     oobbuf[0] = ecc_code[0];
296     oobbuf[1] = ecc_code[1];
297     oobbuf[2] = ecc_code[2];
298     oobbuf[3] = ecc_code[3];
```

```
299     oobbuf[6] = ecc_code[4];
300     oobbuf[7] = ecc_code[5];
301
302     nPages++;
303
304     /* 写 oob 数据，169 字节 */
305     return write(outFile, oobbuf, 16);
306 #endif
307 }
308
```

值得注意的是：第 275 行设置新 tag 结构中增加的 chunkDeleted 成员；第 292～300 行将计算出来的 ECC 码填入新的 ECC 位置，它正是 nand_oob_16 结构的 eccpos 数组定义的位置。

其中第 292、293 行的 nand_calculate_ecc 函数是从内核源文件 drivers/mtd/nand/nand_ ecc.c 修改而来：在/work/system/Development/yaffs2/utils 目录下新建一个同名文件 nand_ecc.c，把内核文件 nand_ecc.c 的 nand_calculate_ecc 函数、函数中用到的 nand_ecc_ precalc_table 数组摘出来；并去除函数中的第一个形参“struct mtd_info *mtd”。

③ 添加文件，修改 Makefile。

第 280 行的 yaffs_PackTags1 函数在上一层目录 yaffs_packedtags1.c 中定义，先将这个文件复制到当前目录。

```
$ cp ../yaffs_packedtags1.c ./
```

另外，and_calculate_ecc 函数是在新加的 nand_ecc.c 中定义的，所以要修改 Makefile，把 yaffs_packedtags1.c 和 nand_ecc.c 也编译进 mkyaffsimage 工具中。

修改前：

```
31 MKYAFFSSOURCES = mkyaffsimage.c
```

修改后：

```
31 MKYAFFSSOURCES = mkyaffsimage.c yaffs_packedtags1.c nand_ecc.c
```

现在，在/work/system/Development/yaffs2/utils 目录下执行“make”命令生成 mkyaffsimage 工具，将它复制到/usr/local/bin 目录。

```
$ sudo cp mkyaffsimage /usr/local/bin
$ sudo chmod +x /usr/local/bin/mkyaffsimage
```

2. 制作/烧写 yaffs 映象文件

使用如下命令将/work/nfs_root/fs_mini 目录制作为 fs_mini.yaffs 文件。

```
# cd /work/nfs_root
# mkyaffsimage fs_mini fs_mini.yaffs
```

将 fs_mini.yaffs 放入 tftp 目录或 nfs 目录后，在 U-Boot 控制界面就可以下载、烧入 NAND

Flash中，操作命令如下：

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

现在可以修改命令行参数以MTD2分区作为根文件系统，比如在U-Boot控制界面如下设置：

```
# set bootargs noinitrd console=ttySAC0 root=/dev/mtdblock2 rootfstype=yaffs
# saveenv
```

17.4.5 制作/使用jffs2文件系统映象文件

1. 编译制作jffs2映象文件的工具

/work/tools/mtd-utils-05.07.23.tar.bz2是MTD设备的工具包，编译它生成mkfs.jffs2工具，用它来将一个目录制作成jffs2文件系统映象文件。

这个工具包需要zlib压缩包，先安装zlib。在/work/GUI/xwindow/X/deps下有zlib源码zlib-1.2.3.tar.gz，执行以下命令进行安装。

```
$ cd /work/GUI/xwindow/X/deps
$ tar xzf zlib-1.2.3.tar.gz
$ cd zlib-1.2.3
$ ./configure --shared --prefix=/usr
$ make
$ sudo make install
```

然后编译mkfs.jffs2。

```
$ cd /work/tools
$ tar xjf mtd-utils-05.07.23.tar.bz2
$ cd mtd-utils-05.07.23/util
$ make
$ sudo make install
```

2. 制作/烧写jffs2映象文件

使用如下命令将/work/nfs_root/fs_mini目录制作为fs_mini.jffs2文件：

```
$ cd /work/nfs_root
$ mkfs.jffs2 -n -s 512 -e 16KiB -d fs_mini -o fs_mini.jffs2
```

上面命令中，“-n”表示不要在每个擦除块上都加上清除标志，“-s 512”指明一页大小为512字节，“-e 16KiB”指明一个擦除块大小为16KB，“-d”表示根文件系统目录，“-o”表示

输出文件。

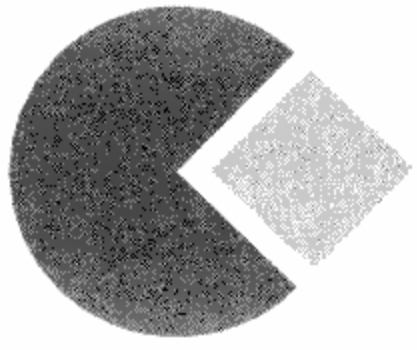
将 fs_mini.jffs2 放入 tftp 目录或 nfs 目录后,在 U-Boot 控制界面就可以将下载、烧入 NAND Flash 中，操作命令如下：

```
① tftp 0x30000000 fs_mini.jffs2 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/fs_mini.jffs2
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。

也可以修改命令行参数以 MTD1 分区作为根文件系统，比如在 U-Boot 控制界面如下设置：

```
# set bootargs noinitrd console=ttySAC0 root=/dev/mtdblock1 rootfstype=jffs2
# saveenv
```



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。调试完华后，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“tty

SAC”，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 {
```

```
    /* 等待，直到发送缓冲区中的数据已经全部发送出去 */
    while (!(rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_UTRSTAT) & S3C2410_UTRSTAT_TXE));

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

```

③ 接收单字符函数。

```
static int kgdb_serial_getchar(void)
{
    /* 等待，直到接收缓冲区中有数据 */
    while (!(rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_UTRSTAT) & S3C2410_
UTRSTAT_RXDR));

    /* 直接读取 URXH 寄存器，即可获得接收到的数据 */
    return rd_regb(CONFIG_KGDB_PORT_NUM, S3C2410_URXH);
}

```

④ 使用这些函数构建 kgdb_io_ops 结构。

```
struct kgdb_io kgdb_io_ops = {
    .init = kgdb_serial_init,
    .read_char = kgdb_serial_getchar,
    .write_char = kgdb_serial_putchar,
};
```

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。

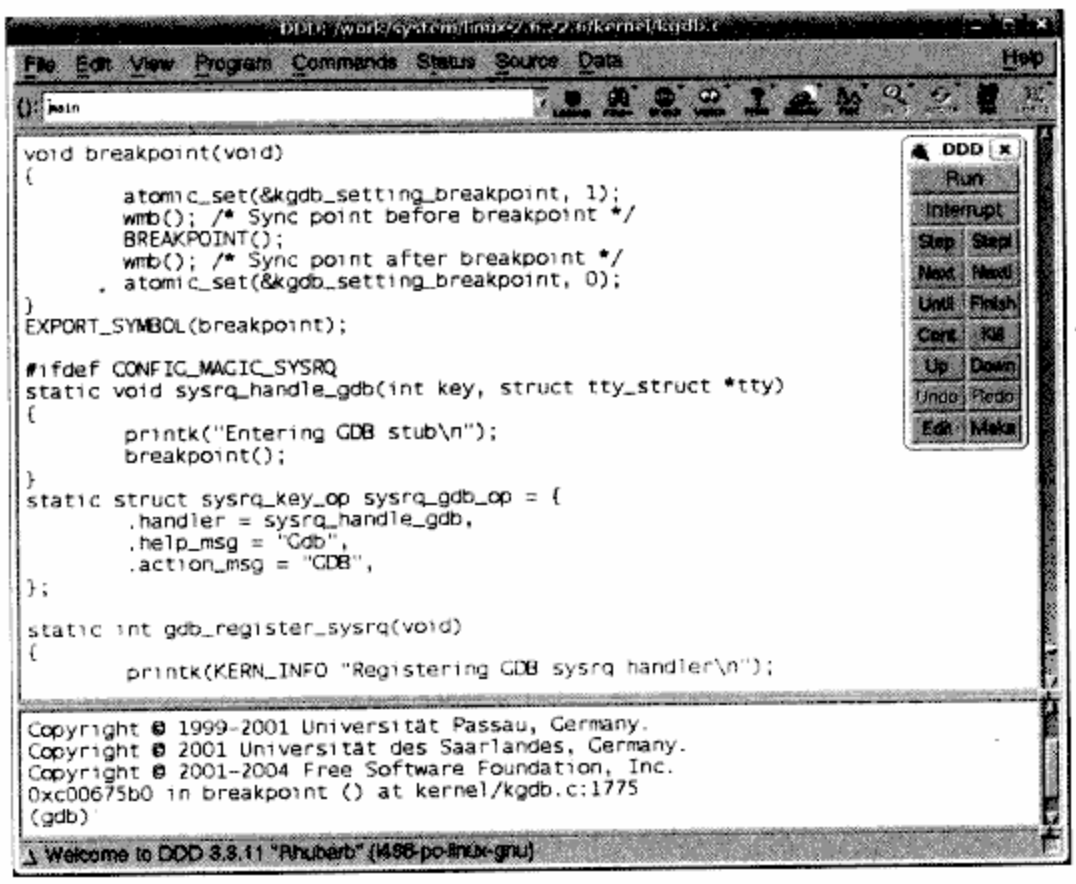


图 18.1 DDD 调用 arm-linux-gdb 来调试内核的启动界面

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：

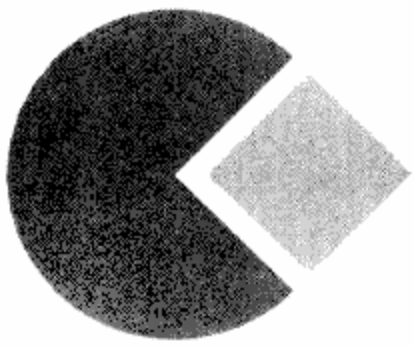
```
c01bf4c8 <platform_drv_probe>:
c01bf4c8:   e1a0c00d    mov ip, sp
c01bf4cc:   e92dd800    stmdb   sp!, {fp, ip, lr, pc}
...
c01bf4e8:   e89da800    ldmia   sp, {fp, sp, pc}    // lr 值(c01bf4e8)对应的指令
```

{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。

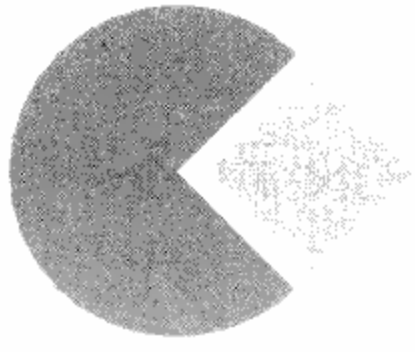
有些函数很简单，没有使用栈（sp值在这个函数中没有变化），或者没有在栈中保存lr值。这些情况需要读者灵活处理，较强的汇编程序阅读能力是关键。



第 4 篇　嵌入式 Linux 设备驱动开发篇

- 字符设备驱动程序
- Linux 异常处理体系结构
- 扩展串口驱动程序移植
- 网卡驱动程序移植
- IDE 接口和 SD 卡驱动程序移植
- LCD 和 USB 驱动程序移植

.....



第 19 章　字符设备驱动程序

本章目标

- 了解 Linux 系统中驱动程序的地位和作用
- 了解驱动程序开发的一般流程
- 掌握简单的字符设备驱动程序的开发方法

19.1　Linux 驱动程序开发概述

19.1.1　应用程序、库、内核、驱动程序的关系

从上到下，一个软件系统可以分为：应用程序、库、操作系统（内核）、驱动程序。开发人员可以专注于自己熟悉的部分，对于相邻层，只需要了解它的接口，无需关注它的实现细节。

以点亮一个 LED 为例，这 4 层软件的协作关系如下，如图 19.1 所示。

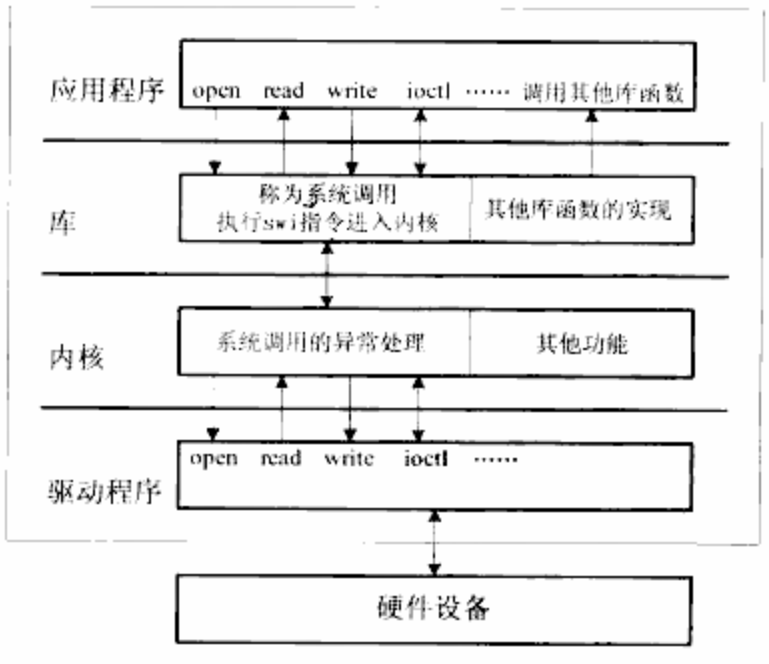


图 19.1　Linux 软件系统的层次关系（swi 是 ARM 指令）

（1）应用程序使用库提供的 open 函数打开代表 LED 的设备文件。

（2）库根据 open 函数传入的参数执行“swi”指令，这条指令会引起 CPU 异常，进入内核。

（3）内核的异常处理函数根据这些参数找到相应的驱动程序，返回一个文件句柄给库，进而返回给应用程序。

（4）应用程序得到文件句柄后，使用库提供的 write 或 ioclt 函数发出控制命令。

（5）库根据 write 或 ioclt 函数传入的参数执行“swi”指令，这条指令会引起 CPU 异常，进入内核。

（6）内核的异常处理函数根据这些参数调用驱动程序的相关函数，点亮 LED。

库（比如 glibc）给应用程序提供的 open、read、write、ioctl、mmap 等接口函数被称为系统调用，它们都是设置好相关寄存器后，执行某条指令引发异常进入内核。对于 ARM 架构的 CPU，这条指令为 swi。除系统调用接口外，库还提供其他函数，比如字符串处理函数（strcpy、strcmp 等）、输入/输出函数（scanf、printf 等）、数学库，还有应用程序的启动代码等。

在异常处理函数中，内核会根据传入的参数执行各种操作，比如根据设备文件名找到对应的驱动程序，调用驱动程序的相关函数等。

一般来说，当应用程序调用 open、read、write、ioctl、mmap 等函数后，将会使用驱动程序中的 open、read、write、ioctl、mmap 函数来进行相关操作，比如初始化、读、写等。

实际上，内核和驱动程序之间并没有界线，因为驱动程序最终是要编进内核去的：通过静态链接或动态加载。

从上面操作 LED 的过程可以知道，与应用程序不同，驱动程序从不主动运行，它是被动的：根据应用程序的要求进行初始化，根据应用程序的要求进行读写。驱动程序加载进内核时，只是告诉内核“我在这里，我能做这些工作”，至于这些“工作”何时开始，取决于应用程序。当然，这不是绝对的，比如用户完全可以写一个由系统时钟触发的驱动程序，让它自动点亮 LED。

在 Linux 系统中，应用程序运行于“用户空间”，拥有 MMU 的系统能够限制应用程序的权限（比如将它限制于某个内存块中），这可以避免应用程序的错误使得整个系统崩溃。而驱动程序运行于“内核空间”，它是系统“信任”的一部分，驱动程序的错误有可能导致整个系统崩溃。

19.1.2 Linux 驱动程序的分类和开发步骤

1. Linux 驱动程序分类

Linux 的外设可以分为 3 类：字符设备（character device）、块设备（block device）和网络接口（network interface）。

字符设备是能够像字节流（比如文件）一样被访问的设备，就是说对它的读写是以字节为单位的。比如串口在进行收发数据时就是一个字节一个字节进行的，我们可以在驱动程序内部使用缓冲区来存放数据以提高效率，但是串口本身对这并没有要求。字符设备的驱动程序中实现了 open、close、read、write 等系统调用，应用程序可以通过设备文件（比如/dev/ttySAC0 等）来访问字符设备。

块设备上的数据以块的形式存放，比如 NAND Flash 上的数据就是以页为单位存放的。块设备驱动程序向用户层提供的接口与字符设备一样，应用程序也可以通过相应的设备文件（比如/dev/mtdblock0、/dev/hda1 等）来调用 open、close、read、write 等系统调用，与块设备传送任意字节的数据。对用户而言，字符设备和块设备的访问方式没有差别。块设备驱动程序的特别之处如下。

（1）操作硬件的接口实现方式不一样。

块设备驱动程序先将用户发来的数据组织成块，再写入设备；或从设备中读出若干块数据，再从中挑出用户需要的。

（2）数据块上的数据可以有一定的格式。

通常在块设备中按照一定的格式存放数据，不同的文件系统类型就是用来定义这些格式的。内核中，文件系统的层次位于块设备块驱动程序上面，这意味着块设备驱动程序除了向用户层动提供与字符设备一样的接口外，还要向内核其他部件提供一些接口，这些接口用户是看不到的。这些接口使得可以在块设备上存放文件系统，挂接（mount）块设备。

网络接口同时具有字符设备、块设备的部分特点，无法将它归入这两类中：如果说它是字符设备，它的输入/输出却是有结构的、成块的（报文、包、帧）；如果说它是块设备，它的“块”又不是固定大小的，大到数百甚至数千字节，小到几字节。UNIX 式的操作系统访问网络接口的方法是给它们分配一个惟一的名字（比如 eth0），但这个名字在文件系统中（比如/dev 目录下）不存在对应的节点项。应用程序、内核和网络驱动程序间的通信完全不同于字符设备、块设备，库、内核提供了一套和数据包传输相关的函数，而不是 open、read、write 等。

2．Linux 驱动程序开发步骤

Linux 内核就是由各种驱动组成的，内核源码中有大约 85%是各种驱动程序的代码。内核中驱动程序种类齐全，可以在同类型驱动的基础上进行修改以符合具体单板。

编写驱动程序的难点并不是硬件的具体操作，而是弄清楚现有驱动程序的框架，在这个框架中加入这个硬件。比如，x86 架构的内核对 IDE 硬盘的支持非常完善：首先通过 BIOS 得到硬盘的信息，或者使用默认 I/O 地址去枚举硬盘，然后识别分区、挂接文件系统。对于其他架构的内核，只要指定了硬盘的访问地址和中断号，后面的枚举、识别和挂接的过程完全是一样的。也许修改的代码不超过 10 行，花费精力的地方在于：了解硬盘驱动的框架，找到修改的位置。

编写驱动程序还有很多需要注意的地方，比如：驱动程序可能同时被多个进程使用，这需要考虑并发的问题；尽可能发挥硬件的作用以提高性能。比如在硬盘驱动程序中既可以使用 DMA 也可以不用，使用 DMA 时程序比较复杂，但是可以提高效率；处理硬件的各种异常情况（即使概率很低），否则出错时可能导致整个系统崩溃。

一般来说，编写一个 Linux 设备驱动程序的大致流程如下。

（1）查看原理图、数据手册，了解设备的操作方法。

（2）在内核中找到相近的驱动程序，以它为模板进行开发，有时候需要从零开始。

（3）实现驱动程序的初始化：比如向内核注册这个驱动程序，这样应用程序传入文件名时，内核才能找到相应的驱动程序。

（4）设计所要实现的操作，比如 open、close、read、write 等函数。

（5）实现中断服务（中断并不是每个设备驱动所必须的）。
（6）编译该驱动程序到内核中，或者用 insmod 命令加载。
（7）测试驱动程序。

19.1.3 驱动程序的加载和卸载

可以将驱动程序静态编译进内核中，也可以将它作为模块在使用时再加载。在配置内核时，如果某个配置项被设为 m，就表示它将会被编译成一个模块。在 2.6 的内核中，模块的扩展名为.ko，可以使用 insmod 命令加载，使用 rmmod 命令卸载，使用 lsmod 命令查看内核中已经加载了哪些模块。

当使用 insmod 加载模块时，模块的初始化函数被调用，它用来向内核注册驱动程序；当使用 rmmod 卸载模块时，模块的清除函数被调用。在驱动代码中，这两个函数要么取固定的名字：init_module 和 cleanup_module，要么使用以下两行来标记它们（假设初始化函数、清除函数为 my_init 和 my_cleanup）。

```
module_init(my_init);
module_exit(my_cleanup);
```

注意 模块有多种，比如文件系统也可以编译为模块，并不是只有驱动程序。

19.2 字符设备驱动程序开发

19.2.1 字符设备驱动程序中重要的数据结构和函数

Linux 操作系统将所有的设备（而不仅是存储器里的文件）都看成文件，以操作文件的方式访问设备。应用程序不能直接操作硬件，而是使用统一的接口函数调用硬件驱动程序。这组接口被称为系统调用，在库函数中定义。可以在 glibc 的 fcntl.h、unistd.h、sys/ioctl.h 等文件中看到如下定义，这些文件也可以在交叉编译工具链的/usr/local/arm/3.4.1/ include 目录下找到。

```
extern int open (__const char *__file, int __oflag, ...) __nonnull ((1));
extern ssize_t read (int __fd, void *__buf, size_t __nbytes);
extern ssize_t write (int __fd, __const void *__buf, size_t __n);
extern int ioctl (int __fd, unsigned long int __request, ...) __THROW;
...
```

对于上述每个系统调用，驱动程序中都有一个与之对应的函数。对于字符设备驱动程序，这些函数集合在一个 file_operations 类型的数据结构中。file_operations 结构在 Linux 内核的 include/linux/fs.h 文件中定义。

```
struct file_operations {
    struct module *owner;
```

```
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t
*, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
    };
```

当应用程序使用 open 函数打开某个设备时，设备驱动程序的 file_operations 结构中的 open 成员就会被调用；当应用程序使用 read、write、ioctl 等函数读写、控制设备时，驱动程序的 file_operations 结构中的相应成员（read、write、ioctl 等）就会被调用。从这个角度来说，编写字符设备驱动程序就是为具体硬件的 file_operations 结构编写各个函数(并不需要全部实现 file_operations 结构中的成员)。

那么，当应用程序通过 open、read、write 等系统调用访问某个设备文件时，Linux 系统怎么知道去调用哪个驱动程序的 file_operations 结构中的 open、read、write 等成员呢？

（1）设备文件有主/次设备号。

设备文件分为字符设备、块设备，比如 PC 机上的串口属于字符设备，硬盘属于块设备。在 PC 上运行命令“ls /dev/ttyS0 /dev/hda1-l”可以看到：

```
brw-rw----    1 root     disk       3,   1 Jan 30  2003 /dev/hda1
crw-rw----    1 root     uucp       4,  64 Jan 30  2003 /dev/ttyS0
```

“brw-rw----”中的“b”表示/dev/hda1 是个块设备，它的主设备号为 2，次设备号为 1；“crw-rw----”中的“c”表示/dev/ttyS0 是个块设备，它的主设备号为 4，次设备号为 64。

（2）模块初始化时，将主设备号与 file_operations 结构一起向内核注册。

驱动程序有一个初始化函数，在安装驱动程序时会调用它。在初始化函数中，会将驱动程序的 file_operations 结构连同其主设备号一起向内核进行注册。对于字符设备使用如下以下函数进行注册：

```
int register_chrdev(unsigned int major, const char * name, struct file_
operations *fops);
```

这样，应用程序操作设备文件时，Linux 系统就会根据设备文件的类型（是字符设备还是块设备）、主设备号找到在内核中注册的 file_operations 结构（对于块设备为 block_device_operations 结构），次设备号供驱动程序自身用来分辨它是同类设备中的第几个。

编写字符驱动程序的过程大概如下。

（1）编写驱动程序初始化函数。

进行必要的初始化，包括硬件初始化（也可以放其他地方）、向内核注册驱动程序等。

（2）构造 file_operations 结构中要用到的各个成员函数。

实际的驱动程序当然比上述两个步骤复杂，但这两个步骤已经可以让我们编写比较简单的驱动程序，比如 LED 控制。其他比较高级的技术，比如中断、select 机制、fasync 异步通知机制，将在其他章节的例子中介绍。

19.2.2 LED 驱动程序源码分析

本节以一个简单的 LED 驱动程序作为例子，让读者初步了解驱动程序的开发。

本书的开发板使用引脚 GPB5～8 外接 4 个 LED，它们的操作方法在第 5 章已经做了细致的说明。

（1）引脚功能设为输出。

（2）要点亮 LED，令引脚输出 0。

（3）要熄灭 LED，令引脚输出 1。

硬件连接方式如图 19.2 所示。

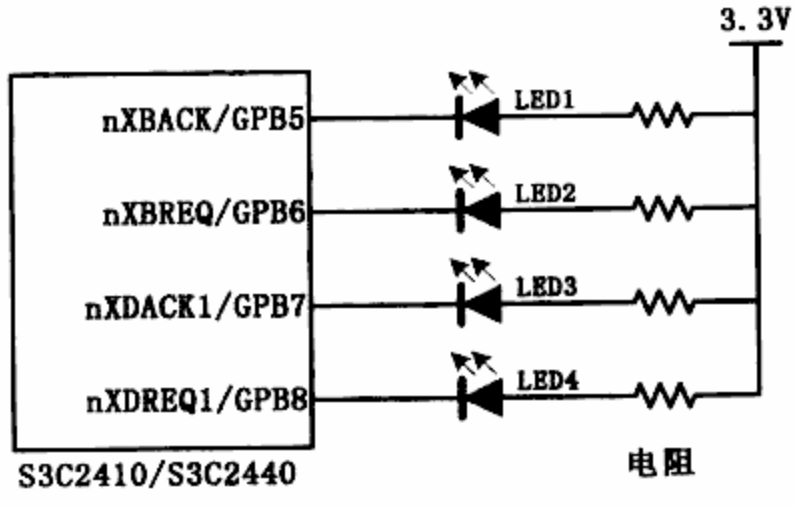


图 19.2 LED 连线图

1. LED 驱动程序代码分析

LED 驱动程序就是光盘上的 drivers_and_test/leds/s3c24xx_leds.c 文件，下面按照函数调用的顺序进行讲解。

模块的初始化函数和卸载函数如下：

```
86 /*
87  * 执行“insmod s3c24xx_leds.ko”命令时就会调用这个函数
88  */
89 static int __init s3c24xx_leds_init(void)
90 {
91     int ret;
92
93     /* 注册字符设备驱动程序
94      * 参数为主设备号、设备名字、file_operations 结构;
95      * 这样，主设备号就和具体的 file_operations 结构联系起来了，
96      * 操作主设备为 LED_MAJOR 的设备文件时，就会调用 s3c24xx_leds_fops 中的相关成
员函数
97      * LED_MAJOR 可以设为 0，表示由内核自动分配主设备号
98      */
99     ret = register_chrdev(LED_MAJOR, DEVICE_NAME, &s3c24xx_leds_fops);
100     if (ret < 0) {
101         printk(DEVICE_NAME " can't register major number\n");
102         return ret;
103     }
104
105     printk(DEVICE_NAME " initialized\n");
106     return 0;
107 }
108
109 /*
110  * 执行“rmmod s3c24xx_leds.ko”命令时就会调用这个函数
111  */
112 static void __exit s3c24xx_leds_exit(void)
113 {
114     /* 卸载驱动程序 */
115     unregister_chrdev(LED_MAJOR, DEVICE_NAME);
116 }
117
118 /* 这两行指定驱动程序的初始化函数和卸载函数 */
119 module_init(s3c24xx_leds_init);
120 module_exit(s3c24xx_leds_exit);
121
```

第119、120两行用来指明装载、卸载模块时所调用的函数。也可以不使用这两行，但是需要将这两个函数的名字改为init_module、cleanup_module。

执行“insmod s3c24xx_leds.ko”命令时就会调用 s3c24xx_leds_init 函数，这个函数核心的代码只有第 99 行。它调用 register_chrdev 函数向内核注册驱动程序：将主设备号 LED_MAJOR 与 file_operations 结构 s3c24xx_leds_fops 联系起来。以后应用程序操作主设备号为 LED_MAJOR 的设备文件时，比如 open、read、write、ioctl，s3c24xx_leds_fops 中的相应成员函数就会被调用。但是，s3c24xx_leds_fops 中并不需要全部实现这些函数，用到哪个就实现哪个。

执行“rmmod s3c24xx_leds.ko”命令时就会调用 s3c24xx_leds_exit 函数，它进而调用 unregister_chrdev 函数卸载驱动程序，它的功能与 register_chrdev 函数相反。

s3c24xx_leds_init、s3c24xx_leds_exit 函数前的“__init”、“__exit”只有在将驱动程序静态链接进内核时才有意义。前者表示 s3c24xx_leds_init 函数的代码被放在“.init.text”段中，这个段在使用一次后被释放（这可以节省内存）；后者表示 s3c24xx_leds_exit 函数的代码被放在“.exit.data”段中，在连接内核时这个段没有使用，因为不可能卸载静态键接的驱动程序。

下面来看看 s3c24xx_leds_fops 的组成。

```
76 /* 这个结构是字符设备驱动程序的核心
77  * 当应用程序操作设备文件时所调用的 open、read、write 等函数，
78  * 最终会调用这个结构中的对应函数
79  */
80 static struct file_operations s3c24xx_leds_fops = {
81     .owner  =   THIS_MODULE,    /* 这是一个宏，指向编译模块时自动创建的__this_
module 变量 */
82     .open   =   s3c24xx_leds_open,
83     .ioctl  =   s3c24xx_leds_ioctl,
84 };
85
```

第 81 行的宏 THIS_MODULE 在 include/linux/module.h 中定义如下，__this_module 变量在编译模块时自动创建，无需关注这点。

```
#define THIS_MODULE (&__this_module)
```

file_operations 类型的 s3c24xx_leds_fops 结构是驱动中最重要的数据结构，编写字符设备驱动程序的主要工作也是填充其中的各个成员。比如本驱动程序中用到 open、ioctl 成员被设为 s3c24xx_leds_open、s3c24xx_leds_ioctl 函数，前者用来初始化 LED 所用的 GPIO 引脚，后者用来根据用户传入的参数设置 GPIO 的输出电平。

s3c24xx_leds_open 函数的代码如下：

```
33 /* 应用程序对设备文件/dev/leds 执行 open()时，
34  * 就会调用 s3c24xx_leds_open 函数
35  */
36 static int s3c24xx_leds_open(struct inode *inode, struct file *file)
37 {
38     int i;
39
```

```
40      for (i = 0; i < 4; i++) {
41          // 设置 GPIO 引脚的功能：本驱动中 LED 所涉及的 GPIO 引脚设为输出功能
42          s3c2410_gpio_cfgpin(led_table[i], led_cfg_table[i]);
43      }
44      return 0;
45  }
46
```

在应用程序执行 open("/dev/leds",...)系统调用时，s3c24xx_leds_open 函数将被调用。它用来将 LED 所涉及的 GPIO 引脚设为输出功能。不在模块的初始化函数中进行这些设置的原因是：虽然加载了模块，但是这个模块却不一定会被用到，就是说这些引脚不一定用于这些用途，它们可能在其他模块中另作他用。所以，在使用时才去设置它，我们把对引脚的初始化放在 open 操作中。

第 42 行的 s3c2410_gpio_cfgpin 函数是内核里实现的，它被用来选择引脚的功能。其实现原理就是设置 GPIO 的控制寄存器，这在第 5 章已经讲过。

s3c24xx_leds_ioctl 函数的代码如下：

```
47  /* 应用程序对设备文件/dev/leds 执行 ioclt()时，
48   * 就会调用 s3c24xx_leds_ioctl 函数
49   */
50  static int s3c24xx_leds_ioctl(
51      struct inode *inode,
52      struct file *file,
53      unsigned int cmd,
54      unsigned long arg)
55  {
56      if (arg > 4) {
57          return -EINVAL;
58      }
59
60      switch(cmd) {
61      case IOCTL_LED_ON:
62          // 设置指定引脚的输出电平为 0
63          s3c2410_gpio_setpin(led_table[arg], 0);
64          return 0;
65
66      case IOCTL_LED_OFF:
67          // 设置指定引脚的输出电平为 1
68          s3c2410_gpio_setpin(led_table[arg], 1);
69          return 0;
```

```
70
71      default:
72          return -EINVAL;
73      }
74 }
75
```

应用程序执行系统调用 ioclt(fd, cmd, arg)时（fd 是前面执行 open 系统调用时返回的文件句柄），s3c24xx_leds_ioctl 函数将被调用。第 63、68 行根据传入的 cmd、arg 参数调用 s3c2410_gpio_setpin 函数，来设置引脚的输出电平：输出 0 时点亮 LED，输出 1 时熄灭 LED。

s3c2410_gpio_setpin 函数也是内核中实现的，它通过 GPIO 的数据寄存器来设置输出电平。

注意 应用程序执行的 open、ioctl 等系统调用，它们的参数和驱动程序中相应函数的参数不是一一对应的，其中经过了内核文件系统层的转换。

系统调用函数原型如下：

```
int open(const char *pathname, int flags);
int ioctl(int d, int request, ...);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
...
```

file_operations 结构中的成员如下：

```
int (*open) (struct inode *inode, struct file *filp);
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
ssize_t (*read) (struct file *filp, char __user *buff, size_t count, loff_t *offp);
ssize_t (*write) (struct file *filp, const char __user *buff, size_t count,
loff_t *offp);
...
```

可以看到，这些参数有很大一部分非常相似。

（1）系统调用 open 传入的参数已经被内核文件系统层处理了，在驱动程序中看不出原来的参数了。

（2）系统调用 ioclt 的参数个数可变，一般最多传入 3 个：后面两个参数与 file_operations 结构中 ioctl 成员的后两个参数对应。

（3）系统调用 read 传入的 buf、count 参数，对应 file_operations 结构中 read 成员的 buf、count 参数。而参数 offp 表示用户在文件中进行存取操作的位置，当执行完读写操作后由驱动程序设置。

（4）系统调用 write 与 file_operations 结构中 write 成员的参数关系，与第（3）点

相似。

在驱动程序的最后，有如下描述信息，它们不是必须的。

```
122 /* 描述驱动程序的一些信息，不是必须的 */
123 MODULE_AUTHOR("http://www.100ask.net");                    // 驱动程序的作者
124 MODULE_DESCRIPTION("S3C2410/S3C2440 LED Driver");  // 一些描述信息
125 MODULE_LICENSE("GPL");                                     // 遵循的协议
126
```

2．驱动程序编译

将光盘上的 drivers_and_test/leds/s3c24xx_leds.c 文件放入内核 drivers/char 子目录下，在 drivers/char/Makefile 中增加下面一行：

```
obj-m    += s3c24xx_leds.o
```

然后在内核根目录下执行“make modules”，就可以生成模块 drivers/char/s3c24xx_leds.ko。把它放到单板根文件系统的/lib/modules/2.6.22.6/目录下，就可以使用“insmod s3c24xx_leds”、“rmmod s3c24xx_leds”命令进行加载、卸载了。

3．驱动程序测试

首先要编译测试程序 drivers_and_test/leds/led_test.c，它的代码很简单，关键部分如下：

```
06 #define IOCTL_LED_ON    0
07 #define IOCTL_LED_OFF   1
…
16 int main(int argc, char **argv)
17 {
…
24      fd = open("/dev/leds", 0);                  // 打开设备
…
30      led_no = strtoul(argv[1], 0, 0) - 1;        // 操作哪个 LED?
…
34      if (!strcmp(argv[2], "on")) {
35          ioctl(fd, IOCTL_LED_ON, led_no);        // 点亮它
36      } else if (!strcmp(argv[2], "off")) {
37          ioctl(fd, IOCTL_LED_OFF, led_no);       // 熄灭它
38      } else {
39          goto err;
40      }
…
50 }
51
```

其中的 open、ioclt 最终会调用驱动程序中的 s3c24xx_leds_open、s3c24xx_leds_ioctl 函数。

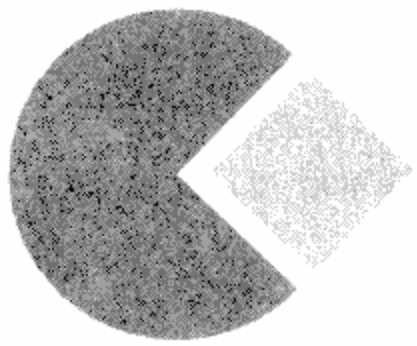
在 drivers_and_test/leds/目录下执行“make”命令生成可执行程序 led_test，将它放入单板根文件系统/usr/bin/目录下后。

然后，在单板根文件系统中如下建立设备文件：

```
# mknod /dev/leds c 231 0
```

现在就可以参照 led_test 的使用说明（直接运行 led_test 命令即可看到）操作 LED 了，以下两条命令点亮、熄灭 LED1。

```
# led_test 1 on
# led_test 1 off
```



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。

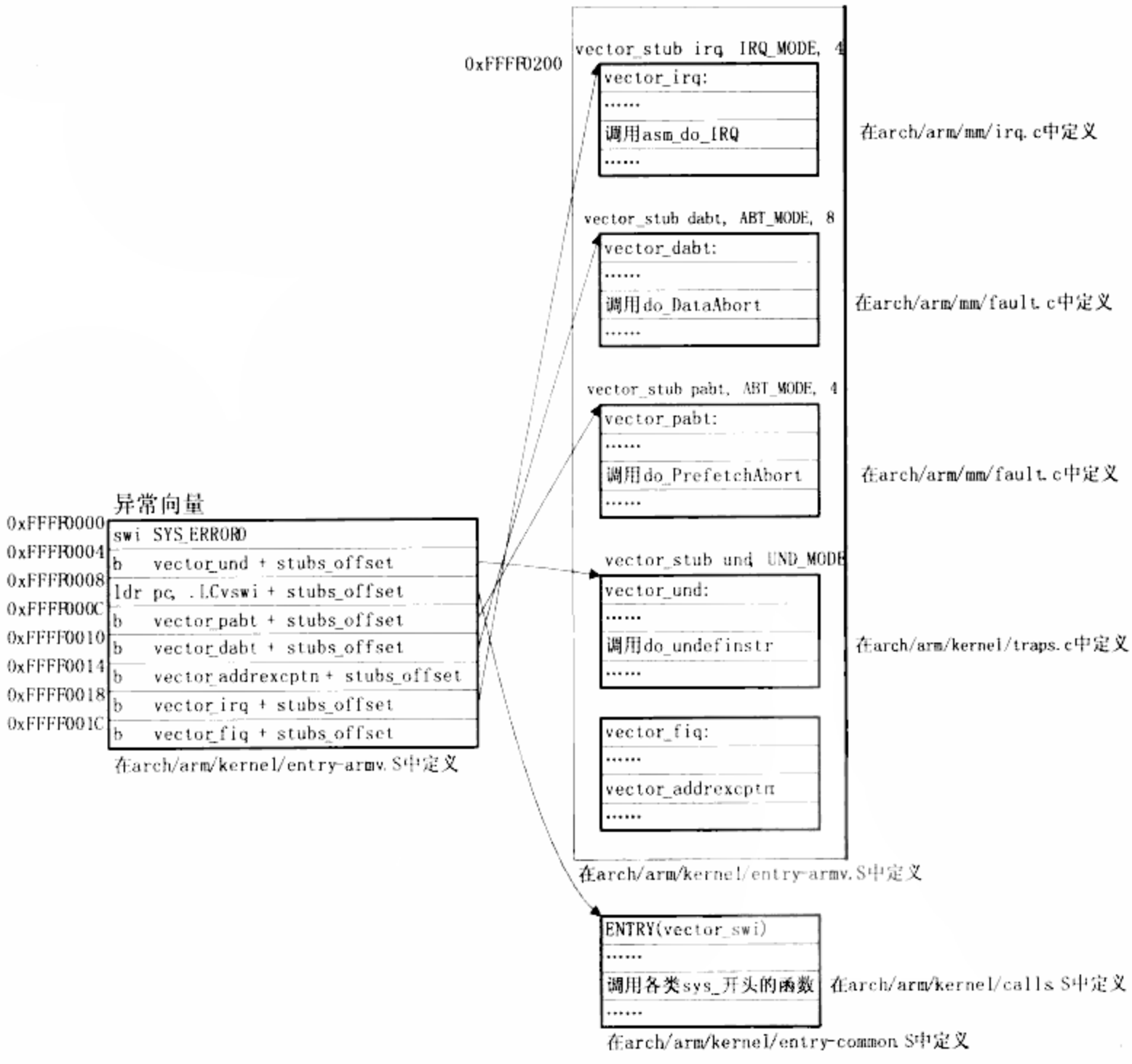


图 20.1 ARM 架构 Linux 内核的异常处理体系结构

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：

```
98  struct irq_chip {
99   const char  *name;
100  unsigned int   (*startup)(unsigned int irq); /* 启动中断,如果不设置,缺省为
"enable" */
101  void    (*shutdown)(unsigned int irq);     /* 关闭中断，如果不设置，缺省为
"disable" */
102  void    (*enable)(unsigned int irq);       /* 使能中断，如果不设置，缺省为
"unmask" */
103  void    (*disable)(unsigned int irq);    /* 禁止中断，如果不设置，缺省为"mask" */
104
105  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*/
106  void    (*mask)(unsigned int irq);       /* 屏蔽中断源 */
107  void    (*mask_ack)(unsigned int irq);   /* 屏蔽和响应中断 */
108  void    (*unmask)(unsigned int irq);     /* 开启中断源 */
...
126 }
```

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：

```
84 struct irqaction {
85     irq_handler_t handler;    /* 用户注册的中断处理函数 */
86    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 */
87     cpumask_t mask;           /* 用于 SMP(对称多处理器系统) */
88     const char *name;         /* 用户注册的中断名字，"cat /proc/interrupts"时
可以看到 */
89     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 */
90     struct irqaction *next;
91     int irq;                  /* 中断号 */
```

```
92      struct proc_dir_entry *dir;
93  };
```

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。

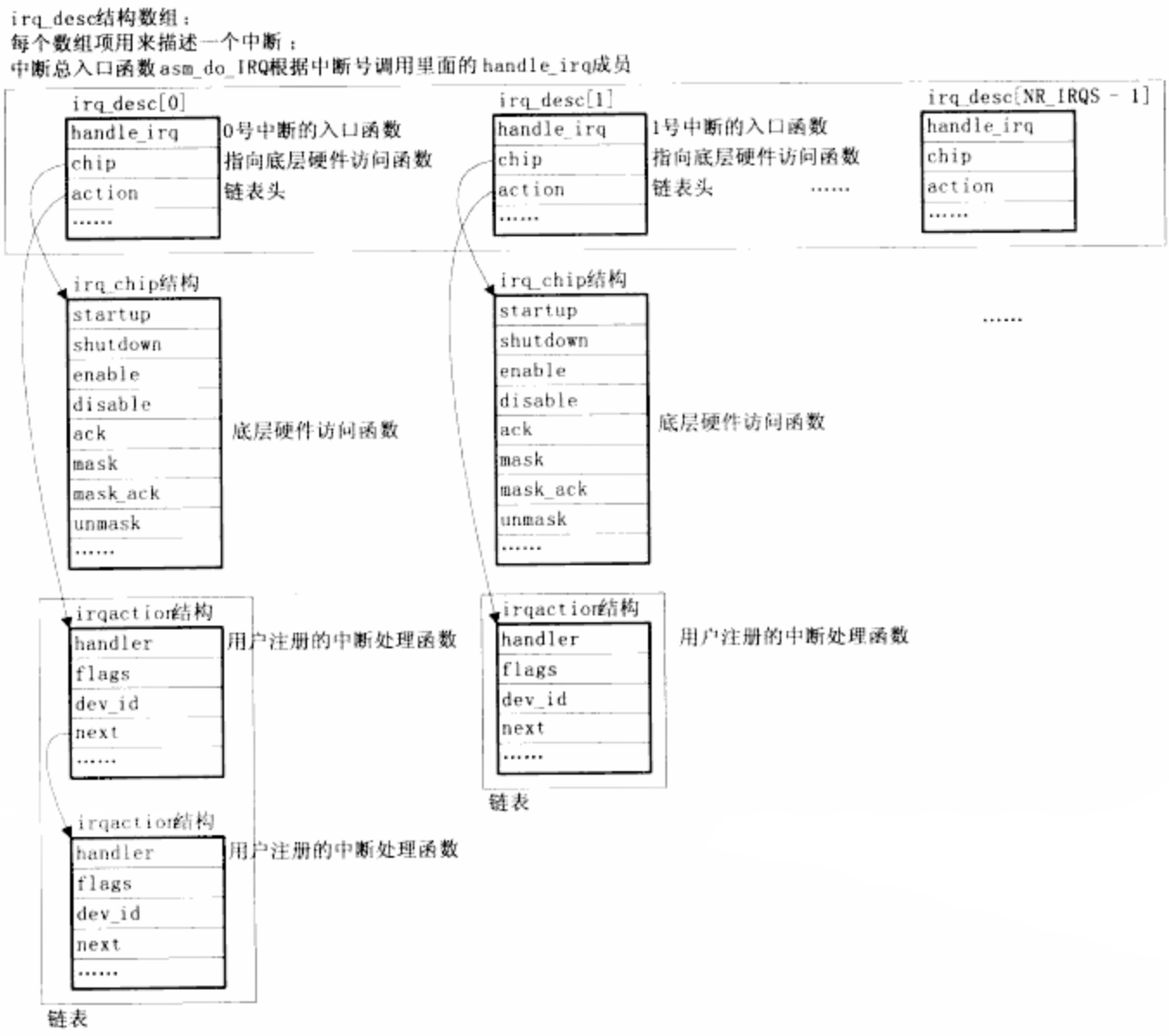


图 20.2 Linux 中断处理体系结构

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：

```
445 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。

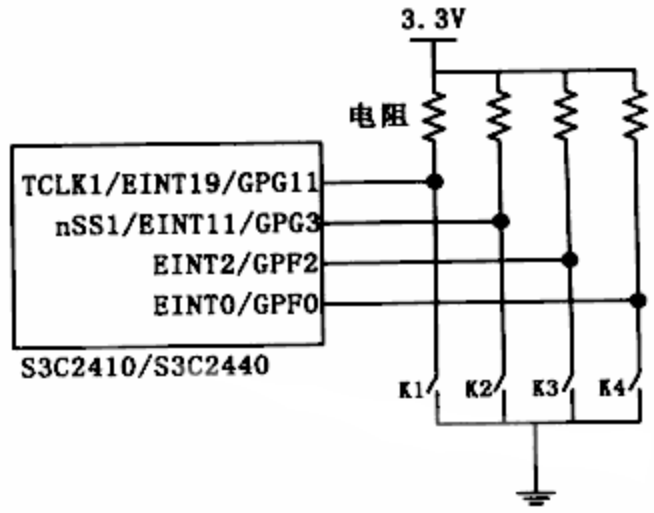


图 20.3　按键连线图

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

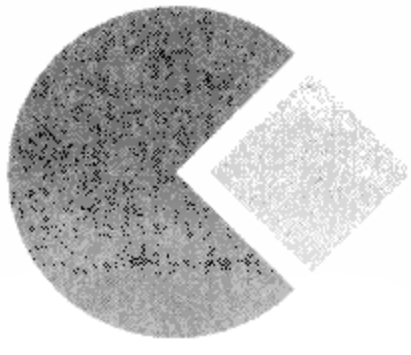
读取数据的代码如下：

```
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。



第 21 章　扩展串口驱动程序移植

本章目标

了解串口终端设备驱动程序的层次结构
掌握移植标准串口驱动程序的方法

21.1　串口驱动程序框架概述

21.1.1　串口驱动程序术语介绍

在 Linux 中经常碰到“控制台”、“终端”、“console”、“tty”、“terminal”等术语，也经常使用到这些设备文件：/dev/console、/dev/ttySAC0、/dev/tty0 等。要理解这些术语，需要从以前的计算机说起。

最初的计算机价格昂贵，一台计算机通常连接上多套键盘和显示器供多人使用。在以前专门有这种可以连上一台电脑的设备，它只有显示器和键盘，外加简单的处理电路，本身不具有处理计算机信息的能力。用户通过它连接到计算机上（通常是通过串口），然后登录系统，并对计算机进行操作。这样一台只有输入、显示部件（比如键盘和显示器）并能够连接到计算机的设备就叫做终端。tty 是 Teletype 的缩写，Teletype 是最早出现的一种终端设备，很像电传打字机。在 Linux 中，就用 tty 来表示“终端”，比如内核文件 tty_io.c、tty_ioctl.c 等都是与“终端”相关的驱动程序；设备文件/dev/ttySAC0、/dev/tty0 等也表示某类终端设备。

“console”的意思即为“控制台”，顾名思义，控制台就是用户与系统进行交互的设备，这和终端的作用相似。实际上，控制台与终端相比，也只是多了一项功能：它可以显示系统信息，比如内核消息、后台服务消息。从硬件上看，控制台与终端都是具备输入、显示功能的设备，没有区别。“控制台”、“终端”、“控制终端”这些名词经常混着用，表示的是同一个意思。

控制台与终端的区别体现在软件上，Linux 内核从很早以前发展而来，代码中仍保留了“控制台”、“终端”的概念。启动 Linux 内核前传入的命令行参数“console=…”就是用来指定“控制台”的。控制台在 tty 驱动初始化之前就可以使用了，它最开始的时候被用来显示内核消息（比如 printk 函数输出的消息）。

当 tty 驱动初始化完毕之后，用户程序就可以通过 tty 驱动的接口来操作各类终端设备，包括控制台。从这个意义上来说，控制台也是一种终端，只不过它还能显示内核消息。

从命令行参数“console=ttySAC0”、“console=tty0”可以了解到：系统中有很多终端设备，可以从它们之间选取一个或多个用作控制台。设备文件“/dev/console”对应的设备就是命令行参数“console=…”中指定的、用作控制台的终端设备。

21.1.2 串口驱动程序的 4 层结构

终端设备种类有很多，比如串行终端、键盘和显示器、通过网络实现的终端等。串口也属于一种终端设备，它的驱动程序并不仅仅是简单的初始化硬件、接收/发送数据。在基本硬件操作的基础上，还增加了很多软件的功能，这是一个多层次的驱动程序。

串口驱动程序从上到下分为 4 层：终端设备层、行规程、串口抽象层、串口芯片层。这种分法不是绝对的，只是为了方便理解程序。图 21.1 形象地表示了这些层次结构。

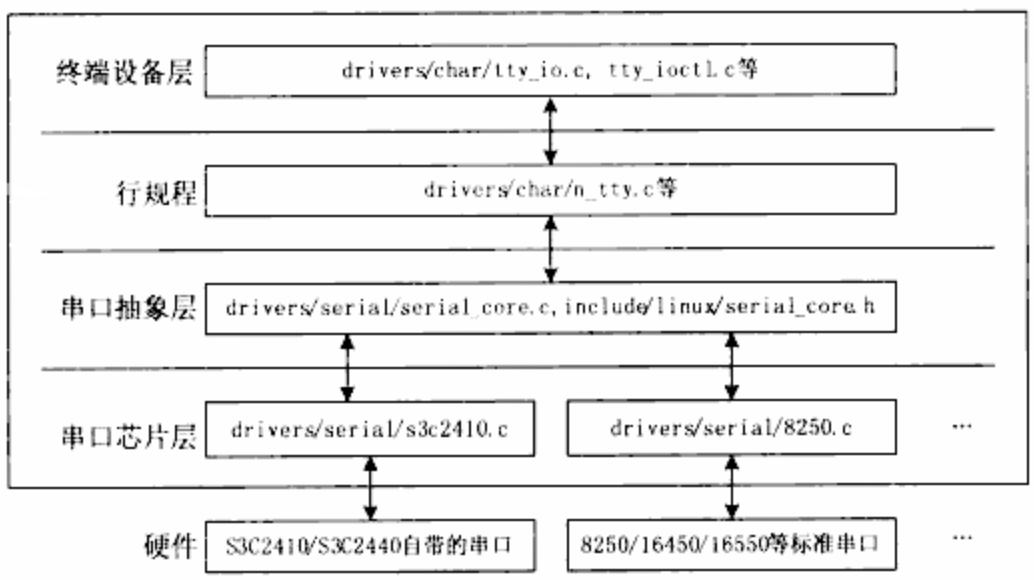


图 21.1 串口驱动程序层次结构

终端设备层和行规程下面还有其他类型的层次与串口的层次并列，比如键盘/显示器等，本章只关注串口。

终端设备层向上提供统一的访问接口，使得用户不必关注具体终端的类型。

行规程的作用是指定数据交互的“规据”，比如流量控制、对输入的数据进行变换处理等。常见的用途有：将 TAB 字符转换为 8 个空格，当接收到删除键（Backspace）时删除前面输入的字符，当接收到“Ctrl+C”字符时发送 SIGINT 信号等。

串口抽象层和串口芯片层都属于低层的驱动程序，它们用来操作硬件。串口抽象层将各类串口的共性概括出来，它也是低层串口驱动的核心部分，比如根据串口芯片层提供的地址识别串口类型、设置串口波特率等。

串口芯片层与具体芯片相关，主要是向串口抽象层提供串口芯片所用的资源（比如访问地址、中断号），还进行一些与芯片相关的设置。对于标准串口，移植的工作主要是在这一层。

下面以几种情景为例，通过驱动程序中各主要函数的调用关系来说明这些层次关系。

1. 串口接收到“Ctrl+C”时

在串口控制台的前台运行一个程序时，如果要手动结束它，可以输入“Ctrl+C”，处理流程如下。

（1）串口接收到字符“Ctrl+C”（ASCII 码为 0x03）后触发中断。假设中断处理函数是

drivers/serial/8250.c 中的 serial8250_interrupt，它属于最低层的函数。

（2）中断处理函数最终会将这个字符放入 tty 层的缓冲区中，每个终端设备都有一个接收缓冲区，里面保存的是原始数据。这一步的函数调用顺序如下：

```
serial8250_interrupt (串口芯片层) ->
    serial8250_handle_port (串口芯片层) ->
        receive_chars (串口芯片层) ->
            uart_insert_char (串口抽象层) ->
                tty_insert_flip_char (终端设备)
```

（3）中断处理函数还要调用其他函数进一步处理原始数据，它最终会向当前进程发送 SIGINT 信号，让它退出。这一步的函数调用顺序如下：

```
serial8250_interrupt (串口芯片层) ->
    serial8250_handle_port (串口芯片层) ->
        receive_chars (串口芯片层) ->
            uart_insert_char (串口抽象层) ->
                tty_insert_flip_char (终端设备层) // 保存接收到的数据及它的标志(是否有错误)
            tty_flip_buffer_push (终端设备层) ->
                flush_to_ldisc (终端设备层) ->
                    disc->receive_buf，即 n_tty_receive_buf (行规程) ->
                        n_tty_receive_char (行规程) ->
                            n_tty_receive_char (终端设备层) ->  /* 根据字符进行不同
的处理*/
                                发送 SIGINT 信号: isig (行规程) // 对于“Ctrl+C”，发信号
```

2. 串口接收普通数据时

串口的接口简单，它的驱动程序相对于 USB、IDE 等接口的驱动程序而言比较容易掌握。但是串口驱动程序中的分层思想、通过中断处理函数或定时器处理函数来完成硬件的操作以释放 CPU 资源的技巧等，这些技术在内核中普遍使用。

以串口接收到字符为例，在控制台上输入“ls”并按回车键时，发生如下事情。

（1）shell 程序一直在休眠，等待接收到“足够”或“合适”的字符。

（2）串口接收到字符“l”，把它保存起来。

（3）串口输出字符“l”，这样在控制台上就可以看见“l”字样了。

（4）类似的，串口接收到字符“s”，保存、输出（这被称为“回显”，echo）。

（5）串口接收到回车符，唤醒 shell 进程。

（6）shell 进程就会读取这些字符决定做什么事。在本例中，它会打印出当前目录下的内容。

这些过程涉及的函数调用与上面对“Ctl+C”的处理类似，只是在 n_tty_receive_char 函数中，对于普通字符将调用 echo_char 函数将它回显；对于回车符，回显之后还要调用 waitqueue_active 函数唤醒等待数据的进程。

3. 串口发送数据时

往串口上发送数据时，在 U-Boot 中是发送一个字符后，循环查询串口状态，当串口再次就绪时，发送下一个字符。如此循环，直到发送完所有字符。在查询状态的过程中，耗费了 CPU 资源，效率低下。

在 Linux 中，串口字符的发送也是通过中断来驱动的。比如在串口控制台上运行一个程序，里面有“printf("hello, world! ")”字样的语句，它的函数调用关系如下：

```
tty_write (终端设备层) ->
    do_tty_write (终端设备层) ->
        write_chan (行规程) ->
            add_wait_queue(&tty->write_wait, &wait);   // 加入等待队列
            tty->driver->write, 即 uart_write (串口抽象层) ->
                // 数据先被保存在串口端口(port)的缓冲区中，然后启动发送
                uart_start (串口抽象层) ->
                    __uart_start (串口抽象层) ->
                        port->ops->start_tx, 即 serial8250_start_tx (串口芯片层) ->
                            up->ier |= UART_IER_THRI;          // 这两行使能串口发送中断
                            serial_out(up, UART_IER, up->ier);  // 字符的发送在中
断函数中进行
            schedule( )  // 假如 uart_write 没“立刻”发送完数据，进程休眠
```

可见，即使是发送数据，也没有使用循环查询的方法，它只是把数据保存起来，然后开启发送中断。当串口芯片内部的发送缓冲区可以再次存入数据时，这个中断被触发；在中断处理函数中将数据一点点地发送给串口芯片。

仍以 drivers/serial/8250.c 中的 serial8250_interrupt 函数为例，前面讲述了它在接收数据时的调用关系，发送数据时的调用关系如下：

```
serial8250_interrupt (串口芯片层) ->
    serial8250_handle_port (串口芯片层) ->
        transmit_chars (串口芯片层) ->
            serial_out (串口芯片层)   // 将数据写入给串口芯片
            // 如果已经发送完毕，唤醒进程
            uart_write_wakeup, 将调用 uart_tasklet_action (串口抽象层) ->
                tty_wakeup (终端设备层) ->
                    /* 与上面 write_chan 中的“ add_wait_queue(&tty->write_wait,
&wait) ”对应*/
                    wake_up_interruptible(&tty->write_wait);    /* 唤醒“等待发送
完毕”的进程*/
            // 如果已经发送完毕，则禁止发送中断
            __stop_tx (串口芯片层)
```

建议读者沿着这些函数的调用关系，深入了解串口驱动程序，比起以后碰到的更复杂的驱动程序，它是个简单的入口点。

21.2 扩展串口驱动程序移植

21.2.1 串口驱动程序低层代码分析

扩展串口在开发板上的连线如图 21.2 所示，中间的缓冲器用来提高电路的驱动能力。

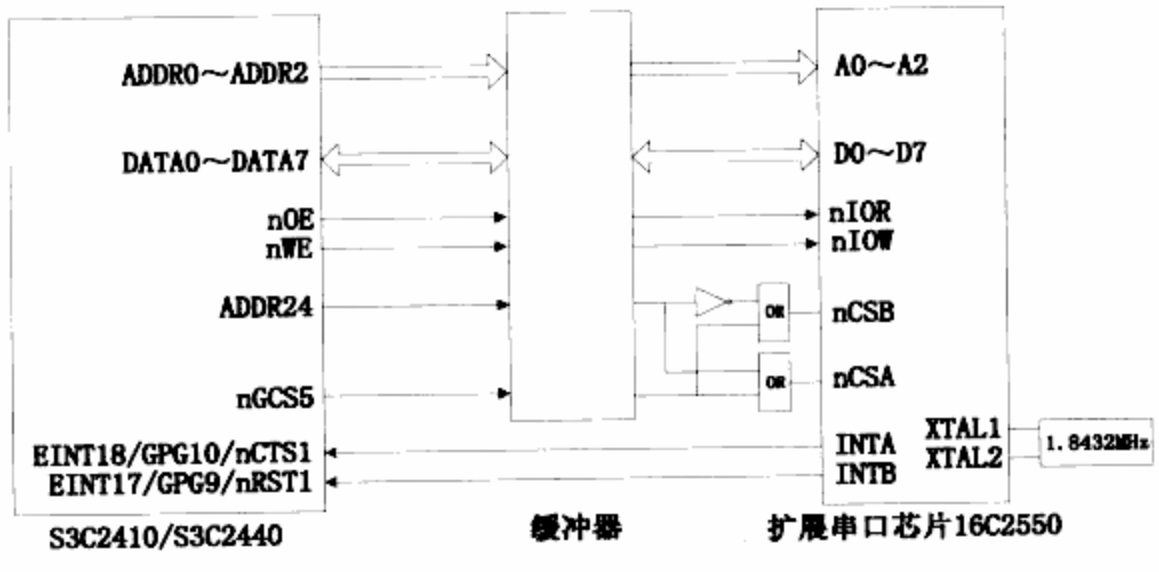


图 21.2 扩展串口连线图

扩展串口芯片 16C2550 属于标准串口，内核的串口驱动程序对它支持良好。可以大胆假设，移植的工作只有一点：告诉这些驱动程序这个扩展芯片所使用的资源，即访问地址和中断号。

与具体芯片相关的驱动代码在“串口芯片层”。对于 16C2550，它就是 drivers/serial/ 8250.c。入口函数为 serial8250_init，它被用来向上层驱动程序注册串口的物理信息，只要弄清楚了这个函数就知道怎么增加对扩展串口的支持了。

serial8250_init 函数代码如下：

```
2848 static int __init serial8250_init(void)
2849 {
...
2862     ret = uart_register_driver(&serial8250_reg); /*注册串口终端设备，未和具体串口挂钩*/
...
2866     serial8250_isa_devs = platform_device_alloc("serial8250", /*分配 platform_device 结构*/
2867                                 PLAT8250_DEV_LEGACY);
...
2873     ret = platform_device_add(serial8250_isa_devs);          // 加入内核设备层
...
```

```
        // 枚举 old_serial_port 中定义的串口
2877    serial8250_register_ports(&serial8250_reg, &serial8250_isa_devs->dev);
2878
2879    ret = platform_driver_register(&serial8250_isa_driver);  // 枚举内核
                                                                 // 设备层中的串口
...
2890 }
2891
```

上面这 5 个函数是关键，其中第 2879 行的 platform_driver_register 是重点。

第 2866 行使用 uart_register_driver 函数向“终端设备层”注册驱动 serial8250_reg，它指定了终端设备的名称、主/次设备号等。serial8250_reg 的内容如下：

```
2574 static struct uart_driver serial8250_reg = {
2575     .owner          = THIS_MODULE,
2576     .driver_name    = "serial",     // 驱动名称
                                        // 可以使用 “cat /proc/tty/driver/serial” 来查看
2577     .dev_name       = "ttyS",       // 设备名称，可以使用"cat /proc/devices"来查看
2578     .major          = TTY_MAJOR,    // 主设备号为 4
2579     .minor          = 64,           // 次设备号
2580     .nr             = UART_NR,      // 支持的最大串口数，默认为 8
2581     .cons           = SERIAL8250_CONSOLE, // 控制台，如果非空，可以用作控制台，比如
                                              // 命令行参数可以传入 “console=ttyS0” 等
2582 };
2583
```

第 2862 行只是注册了主、次设备号为 4 和 64 的终端设备，它还没有和具体的硬件挂钩。

第 2866～2877 行的 3 个函数被用来枚举“用老方法定义的”串口设备，在后面修改代码时不使用这种方法，读者可以略过这几行代码。所谓“用老方法定义的”串口设备就是使用 old_serial_port 结构指定物理信息（访问地址、中断号等）的串口，这是为了与以前的串口驱动兼容而遗留下的数据结构。在 drivers/serial/8250.c 中有如下几行，其中的 SERIAL_PORT_DFNS 宏在本书所用内核中被定义为 NULL：

```
106 static const struct old_serial_port old_serial_port[] = {
107     SERIAL_PORT_DFNS /* defined in asm/serial.h */
108 };
109
```

第 2879 行中的 platform_driver_register 函数向内核注册一个平台驱动 serial8250_isa_driver，它用来枚举名称为“serial8250”的平台设备。

内核根据其他方式确定了很多设备的信息，这些设备被称为平台设备；加载平台驱动程序时将驱动程序与平台设备逐个比较，如果两者匹配，就使用这个驱动来进一步处理（枚举）。

是否匹配的判断方法是：设备名称和驱动名称是否一样。serial8250_isa_driver 结构如下定义：

```
2718 static struct platform_driver serial8250_isa_driver = {
2719     .probe      = serial8250_probe,
2720     .remove     = __devexit_p(serial8250_remove),
2721     .suspend    = serial8250_suspend,
2722     .resume     = serial8250_resume,
2723     .driver     = {
2724         .name   = "serial8250",
2725         .owner  = THIS_MODULE,
2726     },
2727 };
2728
```

可见，serial8250_isa_driver 中驱动名称为“serial8250”，只要内核中有相同名称的平台设备，platform_driver_register 函数最终会调用第 2719 行的 serial8250_probe 函数来枚举它。

serial8250_probe 函数也是在 drivers/serial/8250.c 中定义，只要内核中名为“serial8250”的平台设备定义了正确的串口物理信息，这个函数就能够自动地检测到串口，并将它和前面注册的终端设备联系起来。

总之，移植扩展串口的工作主要是构建一个平台设备的数据结构，在里面指定串口的物理信息。

21.2.2 修改代码以支持扩展串口

串口的物理信息主要有两类：访问地址、中断号。所以只要指明了这两点，并使它们可用，就可以驱动串口了。“使它们可用”的意思是：设置相关的存储控制器以适当的位宽访问这些地址，注册中断时指明合适的触发方式。在移植代码的过程中，这些要点都会一一讲述。

1. 构建串口平台设备的数据结构

在内核代码中查找字符“serial8250”，可以在 arch/arm/mach-s3c2410/mach-bast.c 中看到如下代码：

```
static struct plat_serial8250_port bast_sio_data[] = {
...
};

static struct platform_device bast_sio = {
    .name           = "serial8250",
    .id             = PLAT8250_DEV_PLATFORM,
    .dev            = {
        .platform_data  = &bast_sio_data,
    },
```

```
};
...
static struct platform_device *bast_devices[] __initdata = {
...
    &bast_sio,
};
```

在 arch/arm/plat-s3c24xx/common-smdk.c 中仿照 mach-bast.c 文件增加如下 3 段代码。增加的代码如下，它们都使用宏 CONFIG_SERIAL_EXTEND_S3C24xx 包含起来：

（1）增加要包含的头文件。

```
47 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
48 #include <linux/serial_8250.h>
49 #endif
50
```

（2）增加平台设备数据结构。

```
152 /* for extend serial chip*/
153 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
154 static struct plat_serial8250_port s3c_device_8250_data[] = {
155     [0] = {
156         .mapbase    = 0x28000000,
157         .irq        = IRQ_EINT18,
158         .flags      = (UPF_BOOT_AUTOCONF | UPF_IOREMAP | UPF_SHARE_IRQ),
159         .iotype     = UPIO_MEM,
160         .regshift   = 0,
161         .uartclk    = 115200*16,
162     },
163     [1] = {
164         .mapbase    = 0x29000000,
165         .irq        = IRQ_EINT17,
166         .flags      = (UPF_BOOT_AUTOCONF | UPF_IOREMAP | UPF_SHARE_IRQ),
167         .iotype     = UPIO_MEM,
168         .regshift   = 0,
169         .uartclk    = 115200*16,
170     },
171     { }
172 };
173
174 static struct platform_device s3c_device_8250 = {
175     .name           = "serial8250",
```

```
176     .id             = 0,
177     .dev            = {
178         .platform_data  = &s3c_device_8250_data,
179     },
180 };
181
182 #endif
183
```

第 154 行的 s3c_device_8250_data 结构定义了两个数组项，表示 16C2550 芯片中的两个串口。数组项 0 表示扩展串口 A，数组项 1 表示扩展串口 B。

第 156 行的“.mapbase = 0x28000000”表示串口 A 的访问基址，这是物理地址。

第 157 行指定串口 A 使用的中断号为 IRQ_EINT18，从图 21.2 可以知道这点。

第 158 行指定串口 A 的标志，其中 UPF_BOOT_AUTOCONF 表示自动配置串口，即自动检测它的类型、FIFO 大小等；UPF_IOREMAP 表示需要将前面使用“.mapbase = 0x28000000”指定的物理地址映射为虚拟地址，然后才能使用这个虚拟地址来访问串口 A；UPF_SHARE_IRQ 表示 IRQ_EINT18 是个共享中断，在本书所用开发板中 IRQ_EINT18 只有串口 A 用到，可以不设置它。

第 159 行的“.iotype = UPIO_MEM”表示使用“内存地址”（就是 mapbase 映射后的地址）来访问串口 A，与之对应的有 UPIO_HUB6、UPIO_RM9000 等，它们读写串口芯片的方式有所不同。

第 160 行的“.regshift = 0”用来计算串口的寄存器地址。串口的寄存器都有特定的序号，比如发送/接收寄存器（TX/RX）序号为 0，中断使能寄存器（IER）序号为 1。假设 mapbase 映射后的地址为 membase，寄存器序号为 index，则它的访问地址为：membase + (index << regshift)。从图 21.2 可知，S3C2410/3SC2440 与 16C2550 的连接的总线宽度为 8，所以 regshift 为 0；如果总线宽度为 16，则 regshift 为 1；如果总线宽度为 32，则 regshift 为 2。

第 161 行的“.uartclk = 115200*16”表示串口 A 的时钟。此值计算方法为：假设为了设置串口波特率为 baud，需要往串口的商数寄存器中写入数值 quot=uartclk/(baud×16)。从 16C2550 的芯片手册可以知道 quot 的计算公式为：

```
divisor (decimal) = (XTAL1 clock frequency) / (serial data rate x 16)
```

从图 21.2 可知，晶振频率为 1.8432MHz，所以：

```
uartclk = (XTAL1 clock frequency) = 1.8432M = 115200 * 16.
```

第 163～170 行用于串口 B，意义与串口 A 相似。

第 174～180 行定义了一个平台设备 s3c_device_8250(platform_device 类型的数据结构)，它的名字为“serial8250”，这与 drivers/serial/8250.c 中的平台驱动程序 serial8250_isa_driver 相对应。

（3）加入内核设备列表中。

把平台设备 s3c_device_8250 加入 smdk_devs 数组后，系统启动时会把这个数组中的设备注册进内核中。增加的代码如下（第 193～195 行）：

```
187 static struct platform_device __initdata *smdk_devs[] = {
188     &s3c_device_nand,
...
193 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
194     &s3c_device_8250,
195 #endif
196 };
197
```

现在，平台设备的数据结构已经设备好，就只剩下下面两步了。

2. 增加开发板相关的代码使得串口可用

如前所述，这步需要实现两点：设置相关的存储控制器以适当的位宽访问串口芯片，注册中断时指明合适的触发方式。这需要在 drivers/serial/8250.c 中增加代码。

（1）增加头文件。

设置存储控制器的 BANK5 时需要用到这个头文件，代码如下：

```
47 /* for extend serial chip, www.100ask.net */
48 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
49 #include <asm/arch-s3c2410/regs-mem.h>
50 #endif
51
```

（2）设置存储控制器的 BANK5 的位宽。

从图 21.2 可知，16C2550 扩展串口芯片需要以 8 位的总线宽度进行访问，我们在 drivers/serial/8250.c 的初始化函数前面进行设置（第 2867～2871 行），如下所示：

```
2856 static int __init serial8250_init(void)
2857 {
2858     int ret, i;
2859
2860     if (nr_uarts > UART_NR)
2861         nr_uarts = UART_NR;
2862
2863     printk(KERN_INFO "Serial: 8250/16550 driver $Revision: 1.90 $ "
2864         "%d ports, IRQ sharing %sabled\n", nr_uarts,
2865         share_irqs ? "en" : "dis");
2866
2867 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
2868     /* 设置 BANK5 的位宽为 8, */
2869     *((volatile unsigned int *)S3C2410_BWSCON) =
```

```
    2870        ((*((volatile unsigned int *)S3C2410_BWSCON)) & ~(3<<20)) | S3C2410_
BWSCON_DW5_8;
    2871 #endif
    ...
```

（3）注册中断处理程序时，指定触发方式。

从图 21.2 可以看出，16C2550 扩展串口芯片的 INTA、INTB 中断信号为高电平有效。低电平有效的信号在电路原理图中一般都在前面加上字母“n”，或者加上上划线，比如图 21.2 中 nIOR、nIOW 等信号表示低电平有效。

所以需要将 INTA、INTB 指定为上升沿触发（指定为高电平触发也可以），在 drivers/serial/8250.c 文件中调用 request_irq 函数之前增加如下代码（第 1552～1554 行）。

```
1535 static int serial_link_irq_chain(struct uart_8250_port *up)
1536 {
...
1552 #ifdef CONFIG_SERIAL_EXTEND_S3C24xx
1553         irq_flags |= IRQF_TRIGGER_RISING;   // 中断触发方式为上升沿触发
1554 #endif
1555         ret = request_irq(up->port.irq, serial8250_interrupt,
1556                   irq_flags, "serial", i);
...
```

3. 增加内核配置项 CONFIG_SERIAL_EXTEND_S3C24xx

在内核文件 drivers/serial/Kconfig 中增加如下几行：

```
# www.100ask.net for extend UART
config SERIAL_EXTEND_S3C24xx
        bool "Extend UART for S3C24xx DEMO Board
        depends on SERIAL_8250=y
        ---help---
          Say Y here to use the extend UART
```

现在，所有的修改都完成了，以下是配置/编译内核，测试扩展串口。

21.2.3 测试扩展串口

1. 准备工作

首先配置内核，选中配置项 CONFIG_SERIAL_EXTEND_S3C24xx。执行“make menuconfig”后，如下选择：

```
Device Drivers  --->
    Character devices  --->
```

```
Serial drivers  --->
    <*> 8250/16550 and compatible serial support
    …
    [*]   Extend UART for S3C24xx DEMO Board
```

然后执行“make uImae”编译内核，这将在内核 arch/arm/boot 目录下生成内核映象文件 uImage。

最后修改开发板根文件系统，步骤如下。

（1）如果不使用 mdev，如下增加 ttyS0、ttyS1 设备文件；如果使用 mdev，这步可以省略。

```
# mknod /dev/ttyS0 c 4 64
# mknod /dev/ttyS1 c 4 65
```

（2）修改/etc/inittab 文件，增加如下代码。

```
ttyS0::askfirst:-/bin/sh
```

2. 测试扩展串口

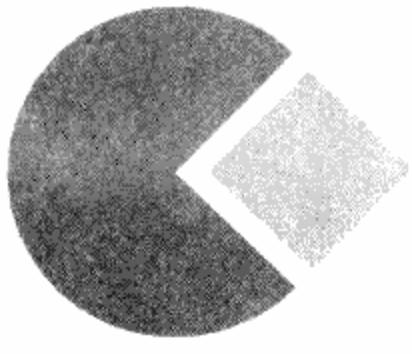
使用新内核、新的根文件系统启动系统，然后原来的控制台下执行如下命令，可以看到检测到了两个串口（0、1 开头的两行）。

```
# cat /proc/tty/driver/serial
serinfo:1.0 driver revision:
0: uart:16550A mmio:0x28000000 irq:62 membase 0xC486A000 tx:0 rx:0
1: uart:16550A mmio:0x29000000 irq:61 membase 0xC486C000 tx:0 rx:0
2: uart:unknown port:00000000 irq:0
3: uart:unknown port:00000000 irq:0
```

将第一个扩展串口连接到主机上、将主机的串口设为（9600, 8N1）后，就可以通过这个扩展串口来控制系统了。注意，这个串口不能看到 U-Boot 的信息和内核的打印信息。

另外，如果想设置扩展串口的默认波特率为 115200，可以如下修改内核文件 drivers/serial/serial_core.c。

```
2167 int uart_register_driver(struct uart_driver *drv)
2168 {
…
2197     normal->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
改为:
2197     normal->init_termios.c_cflag = B115200 | CS8 | CREAD | HUPCL | CLOCAL;
```



第 22 章　网卡驱动程序移植

本章目标

- 了解 Linux 系统的网络栈结构
- 掌握移植网卡驱动程序的一般方法
- 掌握 CS8900A、DM9000 两类网卡驱动程序的移植

22.1　CS8900A 网卡驱动程序移植

22.1.1　CS8900A 网卡特性

CS8900A 是一款针对嵌入式应用的低成本局域以太网控制器。与其他以太网控制器不同，该款产品采用高集成度的设计，因此无需使用昂贵的外部元件。

CS8900A 包括片上 RAM、10Bast-T 发送和接收滤波器，以及一个有 24mA 驱动器的直接 ISA-Bus 接口。

除了高集成度，CS8900A 还具有众多性能特点，并可采用不同的配置。其独特的 PacketPage 架构可以自动适应网络流量模式和可用系统资源的变化。因此可以使系统的效率大大提高。

CS8900A 采用 100 引脚 TQFP 封装，是小型化及对成本敏感的以太网应用的理想选择。采用 CS8900A，用户可以设计出完整的以太网电路。这些电路仅占用不到 $10cm^2$ 的板上空间。

CS8900A 有如下特点。

- 单芯片的 IEEE802.3 以太网解决方案。
- 拥有完整的软件驱动程序。
- 高效的 PacketPage 架构可以采用 DMA 从模式在 I/O 及存储空间运行。
- 全双工操作。
- 片上 RAM 缓冲器发送和接收架构。
- 10Base-T 端口和滤波器（极性检测及纠错）。
- 10Base-2、10Base-5 和 10Base-F 全部采用 AUI 端口。
- 冲突自动再发送、填充及 CRC（循环冗余校验）功能。

- 可编程接收功能。
- 流传输可降低 CPU 负荷。
- DMA 和片上存储器间的自动切换。
- 可早期中断结构先置处理。
- 自动抑制错误信息包。
- EEPROM 支持无跳线配置。
- Boot PROM 支持无盘系统。
- 边界扫描和循环测试。
- LED 驱动器支持链接状态及局域网活动。
- 待机及休眠模式。
- 工作电压为 3V ~ 5V，满足商业及工业应用温度要求。
- 5V 最大功耗为 120mA，5V 典型功耗为 90mA。
- 采用 100 引脚无铅 TQFP 封装。

22.1.2　CS8900A 网卡驱动程序修改

1. Linux 系统网络架构概述

与串口驱动程序类似，网络驱动程序也是分为多个层次的。Linux 系统网络栈的架构如图 22.1 所示。

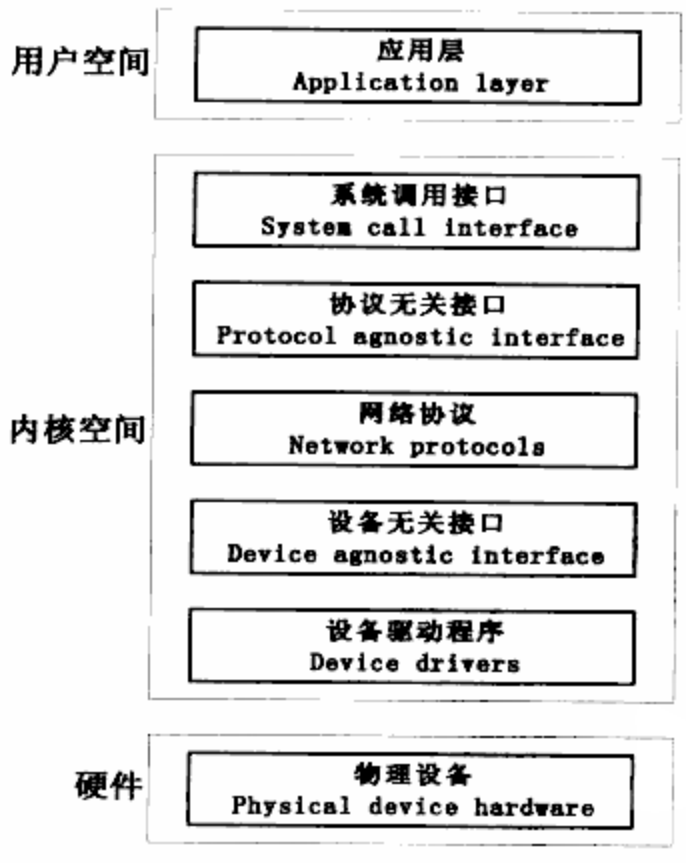


图 22.1　Linux 系统的网络栈架构

最上面是用户空间层，或称为应用层，它通常是一个语义层，能够理解要传输的数据。例如，超文本传输协议（HTTP）就负责传输服务器和客户机之间对 Web 内容的请求与响应，电子邮件协议 SMTP（Simple Mail Transfer Protocol）向用户提供高效、可靠的邮件传输。

最下面是物理设备，提供了对网络的连接能力（串口或诸如以太网之类的高速网络）。

中间是内核空间，即网络子系统，它是驱动移植的重点所在。顶部是系统调用接口，它简单地为用户空间的应用程序提供了一种访问内核网络子系统的方法。位于它下面的是一个协议无关层，它提供了一种通用方法来使用底层传输层协议。然后是实际协议，在 Linux 中包括内嵌的协议 TCP、UDP，当然还有 IP。然后是另外一个设备无关层，提供了与各个设备驱动程序通信的通用接口。设备驱动程序本身是本章移植工作的重点。

2. CS8900A 驱动程序代码修改

Linux 内核中已经有 CS8900A 网卡驱动程序，源文件为 drivers/net/cs89x0.c。与移植扩展串口驱动程序类似，所要做的工作也是："告诉内核" CS8900A 芯片使用的资源（访问地址、中断号等），使得这些资源可用。

CS8900A 在开发板上的连线如图 22.2 所示。

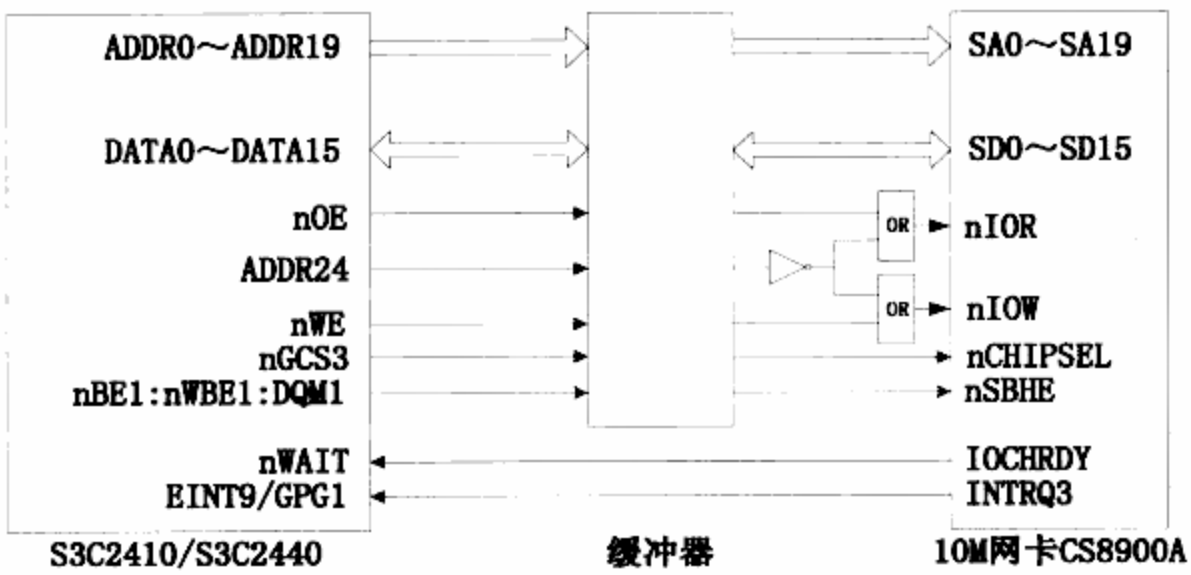


图 22.2 CS8900A 网卡连线图

从上图可以确定以下几点。

（1）CS8900A 的访问基址为 0x19000000（由 BANK3 的基址为 0x18000000 并且 ADDR24 为高可以确定）。

（2）总线位宽为 16，用到 nWAIT、nBE1（字节使能）信号。在 CS8900A 芯片手册中，nSBHE 引脚被称为 "System Bus High Enable"，它为低电平时表示系统数据总线上高字节（SD8～SD15）的数据有效。所以 S3C2410/S3C2440 中，"nBE1:nWBE1:DQM1" 引脚的功能应该设为 "nBE1"。

（3）中断引脚为 EINT9。

驱动文件 drivers/net/cs89x0.c 既可以编进内核，也可以编译为一个可加载模块。编译进内核时，它的入口函数为 cs89x0_probe；编译为模块时，它的入口函数为 init_module。这两个函数最终都会调用 cs89x0_probe1 函数来枚举 CS8900A。需要在调用 cs89x0_probe1 函数之前，指明 CS8900A 芯片使用的资源。

drivers/net/cs89x0.c 被编译进内核时，入口函数 cs89x0_probe 在 drivers/net/ space.c 文件中被调用了 8 次，对于本书所用开发板只需要调用一次（修改代码时会看到）。调用过程如下：

```
net_olddevs_init ->
    ethif_probe2(被调用 8 次) ->
        probe_list2 ->
            cs89x0_probe
```

下面修改驱动文件 drivers/net/cs89x0.c。

（1）指定 CS8900A 使用的资源。

在文件的开头增加以下几行，它们在宏 CONFIG_ARCH_S3C2410 被定义时起作用，表示用于 S3C32410/S3C2440 开发板。

```
197 #elif defined(CONFIG_ARCH_S3C2410)
198 #include <asm/irq.h>
199 #include <asm/arch-s3c2410/regs-mem.h>
200 #define S3C24XX_PA_CS8900   0x19000000  /* 物理基地址 */
201 static unsigned int netcard_portlist[] __initdata = {0, 0};     /* 在下面
进行设置 */
202 static unsigned int cs8900_irq_map[] = {IRQ_EINT9, 0, 0, 0};   /* 中断号 */
203 #else
...
```

第 200 行定义了一个宏 S3C24XX_PA_CS8900，它表示访问 CS8900A 时使用的物理基址，在后面需要将它映射为虚拟地址。

第 201 行的 netcard_portlist 用来指定网卡的访问地址（现在没有设置），它是虚拟地址或 I/O 地址，可以直接使用来访问网卡。后面将 200 行指定的物理地址映射为虚拟地址后，存入 netcard_portlist。

第 202 行指定 CS8900A 使用的中断号。

（2）修改入口函数 cs89x0_probe。

以下使用宏 CONFIG_ARCH_S3C2410 包括起来的代码是新加的。

```
317 struct net_device * __init cs89x0_probe(int unit)
318 {
...
325 #if defined(CONFIG_ARCH_S3C2410)
326     unsigned int oldval_bwscon;     /* 用来保存 BWSCON 寄存器的值 */
327     unsigned int oldval_bankcon3;   /* 用来保存 S3C2410_BANKCON3 寄存器的值 */
328 #endif
...
335     io = dev->base_addr;
336     irq = dev->irq;
337
338 #if defined(CONFIG_ARCH_S3C2410)
339     // cs89x0_probe 会被调用多次，我们只需要 1 次，根据 netcard_portlist[0]的值
忽略后面的调用
340     if (netcard_portlist[0])
341         return -ENODEV;
342
```

```
343     // 将 CS8900A 的物理地址转换为虚拟地址，0x300 是 CS8900A 内部的 I/O 空间的偏移地址
344     netcard_portlist[0] = (unsigned int)ioremap(S3C24XX_PA_CS8900, SZ_1M) +
0x300;
345
346     /* 设置默认 MAC 地址，
347      * MAC 地址可以由 CS8900A 外接的 EEPROM 设定(有些开发板没接 EEPROM)，
348      * 或者启动系统后使用 ifconfig 修改
349      */
350     dev->dev_addr[0] = 0x08;
351     dev->dev_addr[1] = 0x89;
352     dev->dev_addr[2] = 0x89;
353     dev->dev_addr[3] = 0x89;
354     dev->dev_addr[4] = 0x89;
355     dev->dev_addr[5] = 0x89;
356
357     /* 设置 Bank3: 总线宽度为 16，使能 nWAIT，使能 UB/LB。by www.100ask.net */
358     oldval_bwscon = *((volatile unsigned int *)S3C2410_BWSCON);
359     *((volatile unsigned int *)S3C2410_BWSCON) = (oldval_bwscon & ~(3<<12)) \
360         | S3C2410_BWSCON_DW3_16 | S3C2410_BWSCON_WS3 | S3C2410_BWSCON_ST3;
361
362     /* 设置 BANK3 的时间参数*/
363     oldval_bankcon3 = *((volatile unsigned int *)S3C2410_BANKCON3);
364     *((volatile unsigned int *)S3C2410_BANKCON3) = 0x1f7c;
365 #endif
…
375         for (port = netcard_portlist; *port; port++) {
376             if (cs89x0_probe1(dev, *port, 0) == 0)
…
386 out:
387 #if defined(CONFIG_ARCH_S3C2410)
388     iounmap(netcard_portlist[0]);
389     netcard_portlist[0] = 0;
390
391     /* 恢复寄存器原来的值 */
392     *((volatile unsigned int *)S3C2410_BWSCON) = oldval_bwscon;
393     *((volatile unsigned int *)S3C2410_BANKCON3) = oldval_bankcon3;
394 #endif
…
398 }
```

前面讲过，cs89x0_probe 会被调用 8 次，第 340 行用来略过后面的 7 次。

第 344 行将 CS8900A 的访问地址存在 netcard_portlist[0]中，这是虚拟地址，在 Linux 内核空间访问硬件时都使用虚拟地址。它将 CS8900A 的物理基址转换为虚拟地址，再加上 0x300（0x300 是 CS8900A 内部的 I/O 空间的偏移地址）。从图 22.2 的 nIOR、nIOW 信号可知，本书的开发板通过它的 I/O 空间来使用 CS8900A。

第 350～355 行设置 CS8900A 的 MAC 地址。在后面，还会尝试从 CS8900A 外接的 EEPROM 读取 MAC 地址，本书的开发板没有接 EEPROM。也可以在系统启动后通过 ifconfig 命令修改 MAC 地址。

第 358～360 行用来设置 BWSCON 寄存器，将 BANK3 设为：总线宽度为 16，使能 nWAIT 信号，使能 UB/LB 信号（即图 22.2 中的 nBE1 信号）。

第 363～364 行用来设置 BANK3 的时间参数，本书使用最宽松的值，几乎都取最大值。读者可以根据 CS8900A 数据手册进行调整。

第 375～376 行就是实际的枚举函数了。

第 387～394 行用来处理出错情况，它将前面第 338 行映射的虚拟地址释放掉，设置 netcard_portlist[0]为 0，将 BWSCON、BANKCON3 寄存器设为原来的值。

（3）修改模块入口函数 init_module、卸载函数 cleanup_module。

init_module 函数的修改与上述 cs89x0_probe 函数相似，使用宏 CONFIG_ARCH_S3C2410 包括起来的代码是新加的。它们的作用可以参考上面对 cs89x0_probe 函数的描述，不再重复。代码如下：

```
1958 int __init init_module(void)
1959 {
1960     struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
...
1964 #if defined(CONFIG_ARCH_S3C2410)
1965     unsigned int oldval_bwscon;     /* 用来保存 BWSCON 寄存器的值 */
1966     unsigned int oldval_bankcon3;   /* 用来保存 S3C2410_BANKCON3 寄存器的值 */
1967 #endif
1968
...
1974     if (!dev)
1975         return -ENOMEM;
1976
1977 #if defined(CONFIG_ARCH_S3C2410)
1978     // 将 CS8900A 的物理地址转换为虚拟地址，0x300 是 CS8900A 内部的 I/O 空间的偏移地址
1979     dev->base_addr = io = (unsigned int)ioremap(S3C24XX_PA_CS8900, SZ_1M)
+ 0x300;
1980     dev->irq = irq = cs8900_irq_map[0]; /* 中断号 */
1981
1982     /* 设置默认 MAC 地址，
```

```
1983     * MAC 地址可以由 CS8900A 外接的 EEPROM 设定(有些开发板没接 EEPROM),
1984     * 或者启动系统后使用 ifconfig 修改
1985     */
1986    dev->dev_addr[0] = 0x08;
1987    dev->dev_addr[1] = 0x89;
1988    dev->dev_addr[2] = 0x89;
1989    dev->dev_addr[3] = 0x89;
1990    dev->dev_addr[4] = 0x89;
1991    dev->dev_addr[5] = 0x89;
1992
1993    /* 设置 Bank3: 总线宽度为 16, 使能 nWAIT, 使能 UB/LB */
1994    oldval_bwscon = *((volatile unsigned int *)S3C2410_BWSCON);
1995   *((volatile unsigned int *)S3C2410_BWSCON) = (oldval_bwscon & ~(3<<12)) \
1996        | S3C2410_BWSCON_DW3_16 | S3C2410_BWSCON_WS3 | S3C2410_ BWSCON_ST3;
1997
1998    /* 设置 BANK3 的时间参数*/
1999    oldval_bankcon3 = *((volatile unsigned int *)S3C2410_BANKCON3);
2000    *((volatile unsigned int *)S3C2410_BANKCON3) = 0x1f7c;
2001 #else
2002     dev->irq = irq;
2003     dev->base_addr = io;
2004 #endif
...
2030         if (io == 0) {
...
2034         goto out;
2035         } else if (io <= 0x1ff) {
2036         ret = -ENXIO;
2037         goto out;
2038     }
...
2047     ret = cs89x0_probe1(dev, io, 1);
...
2053 out:
2054 #if defined(CONFIG_ARCH_S3C2410)
2055     iounmap(dev->base_addr);
2056
2057     /* 恢复寄存器原来的值 */
2058     *((volatile unsigned int *)S3C2410_BWSCON) = oldval_bwscon;
```

```
2059     *((volatile unsigned int *)S3C2410_BANKCON3) = oldval_bankcon3;
2060 #endif
2061     free_netdev(dev);
2062     return ret;
2063 }
```

需要注意的是第2035行的判断语句“io <= 0x1ff”，io变量本来的类型为int，需要将它改为unsigned int。因为前面第1979行映射得到的地址为在0x80000000之上，使用int类型的话，这是一个负数。

卸载驱动时，要将前面1979行映射的虚拟地址释放掉，这需要修改cleanup_module函数。下面使用宏CONFIG_ARCH_S3C2410包括起来的代码是新加的。

```
2065 void __exit
2066 cleanup_module(void)
2067 {
2068     unregister_netdev(dev_cs89x0);
2069     writeword(dev_cs89x0->base_addr, ADD_PORT, PP_ChipID);
2070     release_region(dev_cs89x0->base_addr, NETCARD_IO_EXTENT);
2071 #if defined(CONFIG_ARCH_S3C2410)
2072     iounmap(dev_cs89x0->base_addr);
2073 #endif
2074     free_netdev(dev_cs89x0);
2075 }
```

（4）注册中断处理程序时，指定中断触发方式。

驱动程序中，在net_open函数使用reques_irq函数注册中断处理函数。如下修改，使用宏CONFIG_ARCH_S3C2410包括起来的代码是新加的，CS8900A的中断触发方式为上升沿触发。

```
1369 /* And 2.3.47 had this: */
1370 #if 0
1371         writereg(dev, PP_BusCTL, ENABLE_IRQ | MEMORY_ON);
1372 #endif
1373         write_irq(dev, lp->chip_type, dev->irq);
1374 #if defined(CONFIG_ARCH_S3C2410)
1375         ret = request_irq(dev->irq, &net_interrupt, IRQF_TRIGGER_RISING,
dev->name, dev);
1376 #else
1377         ret = request_irq(dev->irq, &net_interrupt, 0, dev->name, dev);
1378 #endif
```

（5）其他修改。

最后剩下两个要修改的地方。

① 在 drivers/net/cs89x0.c 中适当的位置加上 CONFIG_ARCH_S3C2410 宏的编译开关，这可以在用到宏 CONFIG_ARCH_PNX010X 的一些地方，仿照它加上宏 CONFIG_ARCH_S3C2410。

② 全局变量“static int io;”改为“static unsigned int io;”。

第一点的修改结果如下，共修改了 3 个位置（注意，下面代码中也给出了所修改行的邻近行，它们用于帮助读者定位）。

① 第一个位置。

修改前

```
1320 static int
1321 net_open(struct net_device *dev)
1322 {
1323     struct net_local *lp = netdev_priv(dev);
1324     int result = 0;
1325     int i;
1326     int ret;
1327
1328 #if !defined(CONFIG_SH_HICOSH4) && !defined(CONFIG_ARCH_PNX010X) /* uses
irq#1, so this won't work */
```

修改后

```
1328  #if  !defined(CONFIG_SH_HICOSH4)  &&  !defined(CONFIG_ARCH_PNX010X)
&& !defined(CONFIG_ARCH_S3C2410) /* uses irq#1, so this won't work */
```

② 第二个位置。

修改前

```
1359 #if !defined(CONFIG_MACH_IXDP2351) && !defined(CONFIG_ARCH_IXDP2X01)
&& !defined(CONFIG_ARCH_PNX010X)
```

修改后

```
1359 #if !defined(CONFIG_MACH_IXDP2351) && !defined(CONFIG_ARCH_IXDP2X01)
&& !defined(CONFIG_ARCH_PNX010X) && !defined(CONFIG_ARCH_S3C2410)
1360         if (((1 << dev->irq) & lp->irq_map) == 0) {
```

③ 第三个位置。

修改前

```
1448 #if defined(CONFIG_ARCH_PNX010X)
```

修改后

```
1448 #if defined(CONFIG_ARCH_PNX010X) || defined(CONFIG_ARCH_S3C2410)
1449     result = A_CNF_10B_T;
```

3. 内核配置文件修改

要使用驱动文件 drivers/net/cs89x0.c，需要设置配置项 CONFIG_CS89x0，它在配置文件 drivers/net/Kconfig 中描述。修改前代码如下：

```
config CS89x0
    tristate "CS89x0 support"
    depends on NET_PCI && (ISA || MACH_IXDP2351 || ARCH_IXDP2X01 || ARCH_PNX010X)
```

在本书开发板中，CS8900A 并不需要使用 PCI，所以要修改它的依赖条件。修改后的代码如下：

```
config CS89x0
        tristate "CS89x0 support"
        depends on (NET_PCI && (ISA || MACH_IXDP2351 || ARCH_IXDP2X01 || ARCH_
PNX010X)) || ARCH_S3C2410
```

4. 使用 CS8900A 网卡

在内核根目录下执行“make menuconfig”后，如下配置将 CS8900A 编入内核（也可以配置为模块）。

```
Device Drivers  --->
    Network device support  --->
        [*] Network device support
            Ethernet (10 or 100Mbit)  --->
                <*> CS89x0 support
```

另外，增加对 NFS 的支持，如下配置：

```
File systems  --->
    Network File Systems  --->
        <*> NFS file system support
        [*]   Provide NFSv3 client support
        [*]     Provide client support for the NFSv3 ACL protocol extension
        [*]   Provide NFSv4 client support (EXPERIMENTAL)
        [*] Root file system on NFS
```

然后编译内核：执行“make uImage”命令即可生成 arch/arm/boot/uImamge。

新内核具备了网络功能，这时可以通过 NFS 启动系统，或者从 NAND Flash 上启动系统后挂接 NFS 文件系统，可以 telnet 登录到开发板上等，这时候调试程序就非常方便了，不用每次都将程序烧到开发板上。

按照 U-Boot 的使用说明烧写新内核，在 Linux 主机上启动 NFS 服务，现在就可以在 U-Boot 控制界面修改命令行参数通过 NFS 启动系统了。假设主机 IP 为 192.168.1.57，NFS

目录为/work/my_root_fs，就可以如下设置命令行参数后，启动内核。

```
set bootargs noinitrd root=/dev/nfs console=ttySAC0 nfsroot= 192.168.1.57:/ work/my_root_fs ip=192.168.1.17:192.168.1.57:192.168.1.2:255.255.255.0::eth0:off
```

上面“ip=…”的格式如下，请参考内核文档 Documentation/nfsroot.txt。

```
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>
```

当系统启动后，还可以在控制台使用以下命令挂接 NFS 文件系统。

```
# ifconfig eth0 192.168.1.17
# mount -t nfs -o nolock 192.168.1.57:/work/nfs_root/fs_mini_mdev /mnt
```

要从主机上通过 telnet 登录开发板，首先要在开发板上启动 telnetd 服务。

```
# ifconfig eth0 192.168.1.17
# telnetd -l /bin/sh
```

telnetd 的参数“-l /bin/sh”表示连接时运行程序“/bin/sh”，否则需要验证密码。本书构建的根文件系统中没有设置用户和密码，无法登录。

如果将 CS8900A 的驱动配置为模块，在内核根目录下执行“make modules”命令后，会在 drivers/net 下生成可加载模块 cs89x0.ko。将它放到开发板根文件系统中，加载之后再设置 IP，就可以挂接 NFS 文件系统、启动 telnetd 服务了。

22.2 DM9000 网卡驱动程序移植

22.2.1 DM9000 网卡特性

DM9000 是一款高度集成的、低成本的的单片快速以太网 MAC 控制器，含有带有通用处理器接口、10M/100M 物理层和 16kB 的 SRAM。

DM9000 有如下特点。

- 支持的处理器接口类型：以字节/字/双字的 I/O 指令访问 DM9000 内部数据。
- 集成的 10M/100M 收发器。
- 支持 MII/RMII 接口。
- 支持半双工背压流量控制模式。
- IEEE802.3x 全双工流量控制模式。
- 支持远端唤醒和连接状态变化。
- 集成 4KB 的双字 SRAM。
- 支持从 EEPROM 中自动获取厂商 ID（vendor ID）和产品 ID（product ID）。
- 支持 4 个 GPIO 管脚。
- 可以使用 EEPROM 来配置（可选）。
- 低功耗模式。
- I/O 管脚 3.3V 和 5V 兼容。

- 100-pin CMOS 工艺 LQFP 封装。

22.2.2　DM9000 网卡驱动程序修改

DM9000 在开发板上的连线如图 22.3 所示。

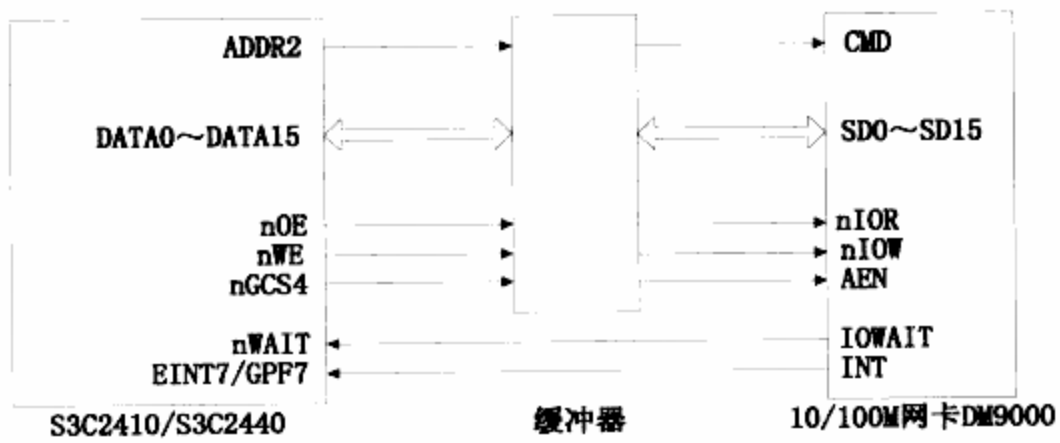


图 22.3　DM9000 网卡连线图

从上图可以确定以下几点。

（1）DM9000 的访问基址为 0x20000000（BANK4 的基址），这是物理地址。

（2）只用到一条地址线：ADDR2。这是由 DM9000 的特性决定的：DM9000 的地址信号和数据信号复用，使用 CMD 引脚来区分它们（CMD 为低时数据总线上传输的是地址信号，CMD 为高时传输的是数据信号）。访问 DM9000 内部寄存器时，需要先将 CMD 置为低电平，发出地址信号；然后将 CMD 置为高电平，读写数据。

（3）总线位宽为 16，用到 nWAIT 信号。

（4）中断引脚为 EINT7。

Linux 内核中已经有 DM9000 网卡驱动程序，源文件为 drivers/net/dm9000.c。它既可以编译进内核，也可以编译为一个模块。入口函数都是 dm9000_init，代码如下：

```
1248 static int __init
1249 dm9000_init(void)
1250 {
1251     printk(KERN_INFO "%s Ethernet Driver\n", CARDNAME);
1252
1253     return platform_driver_register(&dm9000_driver);    /* search board
and register */
1254 }
1255
```

第 1253 行向内核注册平台驱动 dm9000_driver。dm9000_driver 结构的名称为“dm9000”，如果内核中有相同名称的平台设备，则调用 dm9000_probe 函数（下面第 1242 行）。dm9000_driver 结构如下定义：

```
1237 static struct platform_driver dm9000_driver = {
1238     .driver = {
```

```
1239        .name    = "dm9000",
1240        .owner   = THIS_MODULE,
1241    },
1242    .probe   = dm9000_probe,
1243    .remove  = dm9000_drv_remove,
1244    .suspend = dm9000_drv_suspend,
1245    .resume  = dm9000_drv_resume,
1246 };
1247
```

所以，首先要为 DM9000 定义一个平台设备的数据结构，然后修改 drivers/net/dm9000.c，增加一些开发板相关的代码。

1. 增加 DM9000 平台设备

增加平台设备的方法在移植串口驱动程序时已经介绍过，过程相似。这需要修改 arch/arm/plat-s3c24xx/common-smdk.c 文件。

（1）添加要包含的头文件，增加以下代码：

```
46 #if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
47 #include <linux/dm9000.h>
48 #endif
```

（2）添加 DM9000 的平台设备结构，增加以下代码：

```
154 #if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
155 /* DM9000 */
156 static struct resource s3c_dm9k_resource[] = {
157     [0] = {
158         .start = S3C2410_CS4,       /* ADDR2=0，发送地址时使用这个地址 */
159         .end   = S3C2410_CS4 + 3,
160         .flags = IORESOURCE_MEM,
161     },
162     [1] = {
163         .start = S3C2410_CS4 + 4,   /* ADDR2=1，传输数据时使用这个地址 */
164         .end   = S3C2410_CS4 + 4 + 3,
165         .flags = IORESOURCE_MEM,
166     },
167     [2] = {
168         .start = IRQ_EINT7,         /* 中断号 */
169         .end   = IRQ_EINT7,
170         .flags = IORESOURCE_IRQ,
```

```
171     }
172
173 };
174
175 /* for the moment we limit ourselves to 16bit IO until some
176  * better IO routines can be written and tested
177 */
178
179 static struct dm9000_plat_data s3c_dm9k_platdata = {
180     .flags      = DM9000_PLATF_16BITONLY,  /* 数据总线宽度为 16 */
181 };
182
183 static struct platform_device s3c_device_dm9k = {
184     .name       = "dm9000",
185     .id     = 0,
186     .num_resources  = ARRAY_SIZE(s3c_dm9k_resource),
187     .resource   = s3c_dm9k_resource,
188     .dev        = {
189         .platform_data = &s3c_dm9k_platdata,
190     }
191 };
192 #endif /* CONFIG_DM9000 */
```

以上代码是仿照 arch/arm/mach-s3c2410/mach-bast.c 增加的，主要修改了 DM9000 所使用的资源，即第 156 行的 s3c_dm9k_resource 结构。s3c_dm9k_resource 结构中定义了 3 个资源：两个内存空间、中断号。数组项 0、1 定义了访问 DM9000 时使用的地址，前一个地址的 ADDR2 为 0，用来传输地址；后一个地址的 ADDR2 为 1，用来传输数据。数组项 2 定义了 DM9000 使用的中断号。

第 180 行指定访问 DM9000 时，数据位宽为 16。DM9000 支持 8/16/32 位的访问方式。

（3）加入内核设备列表中。

把平台设备 s3c_device_dm9k 加入 smdk_devs 数组中即可，系统启动时会把这个数组中的设备注册进内核中。增加的代码如下（第 235～237 行）：

```
229 static struct platform_device __initdata *smdk_devs[] = {
...
235 #if defined(CONFIG_DM9000) || defined(CONFIG_DM9000_MODULE)
236     &s3c_device_dm9k,
237 #endif
...
241 };
```

2. 修改 drivers/net/dm9000.c

对 DM9000 的枚举最终由 dm9000_probe 函数来完成，首先从它分析这个驱动是如何使用上面定义的两个内存空间地址和中断号的，然后再给出修改方法。

（1）驱动源码简要分析。

从 dm9000_probe 函数就可以看出前面定义的资源是如何被使用的，代码如下：

```
391 static int
392 dm9000_probe(struct platform_device *pdev)
393 {
...
437     if (pdev->num_resources < 2) {
...
440     } else if (pdev->num_resources == 2) {
...
453     } else {
454         db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);   //
S3C2410_CS4
455         db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);  //
S3C2410_CS4 + 4
456         db->irq_res  = platform_get_resource(pdev, IORESOURCE_IRQ, 0);  //
IRQ_EINT7
...
465         i = res_size(db->addr_res);
...
475         db->io_addr = ioremap(db->addr_res->start, i); // S3C2410_CS4 对
应的虚拟地址
...
483         iosize = res_size(db->data_res);
...
493                             db->io_data    =    ioremap(db->data_res->start,
iosize);//(S3C2410_CS4+4)对应的虚拟地址
...
503         ndev->base_addr = (unsigned long)db->io_addr;
504         ndev->irq   = db->irq_res->start;  // IRQ_EINT7
...
508     }
...
511     if (pdata != NULL) {
...
```

```
518         if (pdata->flags & DM9000_PLATF_16BITONLY)
519             dm9000_set_io(db, 2);
...
535     }
536
537     dm9000_reset(db);
538
539     /* try two times, DM9000 sometimes gets the first read wrong */
540     for (i = 0; i < 2; i++) {
541         id_val  = ior(db, DM9000_VIDL);
542         id_val |= (u32)ior(db, DM9000_VIDH) << 8;
543         id_val |= (u32)ior(db, DM9000_PIDL) << 16;
544         id_val |= (u32)ior(db, DM9000_PIDH) << 24;
...
549     }
...
}
```

arch/arm/plat-s3c24xx/common-smdk.c 文件中的 s3c_dm9k_resource 结构有 3 个数组项，表示有“3 个资源”，所以 pdev->num_resources 数值为 3，将执行 453 行的分支。

参考第 454～504 行的代码，可以知道 s3c_dm9k_resource 结构中定义的两个内存空间经过映射后，它们的虚拟基地址保存在 db->io_addr 和 db->io_data 中，下面可以看到它们是如何使用的。ndev->irq 中保存了中断号。

第 518～519 行根据 arch/arm/plat-s3c24xx/common-smdk.c 文件中的 s3c_dm9k_platdata 结构指定的访问位宽，设置了相关的读写函数。

现在来看看程序中是如何使用 db->io_addr 和 db->io_data 来访问 DM9000 的。第 537 行的 dm9000_reset 函数如下定义，先往地址 db->io_addr 写入值 DM9000_NCR，再往地址 db->io_data 写入 NCR_RST 就可以复位 DM9000。

```
177 static void
178 dm9000_reset(board_info_t * db)
179 {
180     PRINTK1("dm9000x: resetting\n");
181     /* RESET device */
182     writeb(DM9000_NCR, db->io_addr);
183     udelay(200);
184     writeb(NCR_RST, db->io_data);
185     udelay(200);
186 }
```

第 541～544 行的 ior 函数用来读取 DM9000 的寄存器，它如下定义：

```
191 static u8
192 ior(board_info_t * db, int reg)
193 {
194     writeb(reg, db->io_addr);  // 先往地址 db->io_addr 写入寄存器地址
195     return readb(db->io_data); // 再从地址 db->io_data 读出数值
196 }
```

iow 函数用来写 DM9000 的寄存器，它如下定义：

```
202 static void
203 iow(board_info_t * db, int reg, int value)
204 {
205     writeb(reg, db->io_addr);   // 先往地址 db->io_addr 写入寄存器地址
206     writeb(value, db->io_data); // 再将数值写入地址 db->io_data
207 }
```

（2）驱动源码修改：drivers/net/dm9000.c。

① 添加要包含的头文件，增加以下代码：

```
73 #if defined(CONFIG_ARCH_S3C2410)
74 #include <asm/arch-s3c2410/regs-mem.h>
75 #endif
```

② 设置存储控制器使 BANK4 可用，设置默认 MAC 地址（这不是必需的）。

增加的代码如下，它们被宏 CONFIG_ARCH_S3C2410 包含起来。

```
391 static int
392 dm9000_probe(struct platform_device *pdev)
393 {
...
403 #if defined(CONFIG_ARCH_S3C2410)
404     unsigned int oldval_bwscon;      /* 用来保存 BWSCON 寄存器的值 */
405     unsigned int oldval_bankcon4;    /* 用来保存 S3C2410_BANKCON4 寄存器的值 */
406 #endif
...
418     PRINTK2("dm9000_probe()");
419
420 #if defined(CONFIG_ARCH_S3C2410)
421     /* 设置 Bank4: 总线宽度为 16, 使能 nWAIT */
422     oldval_bwscon = *((volatile unsigned int *)S3C2410_BWSCON);
```

```
423     *((volatile unsigned int *)S3C2410_BWSCON) = (oldval_bwscon & ~(3<<16)) \
424         | S3C2410_BWSCON_DW4_16 | S3C2410_BWSCON_WS4 | S3C2410_BWSCON_ST4;
425
426     /* 设置BANK3的时间参数*/
427     oldval_bankcon4 = *((volatile unsigned int *)S3C2410_BANKCON4);
428     *((volatile unsigned int *)S3C2410_BANKCON4) = 0x1f7c;
429 #endif
…
599     if (!is_valid_ether_addr(ndev->dev_addr)) {
600         printk("%s: Invalid ethernet MAC address.  Please "
601                "set using ifconfig\n", ndev->name);
602 #if defined(CONFIG_ARCH_S3C2410)
603         printk("Now use the default MAC address: 08:90:90:90:90:90\n");
604         ndev->dev_addr[0] = 0x08;
605         ndev->dev_addr[1] = 0x90;
606         ndev->dev_addr[2] = 0x90;
607         ndev->dev_addr[3] = 0x90;
608         ndev->dev_addr[4] = 0x90;
609         ndev->dev_addr[5] = 0x90;
610 #endif
611     }
…
626  out:
627     printk("%s: not found (%d).\n", CARDNAME, ret);
628 #if defined(CONFIG_ARCH_S3C2410)
629     /* 恢复寄存器原来的值 */
630     *((volatile unsigned int *)S3C2410_BWSCON) = oldval_bwscon;
631     *((volatile unsigned int *)S3C2410_BANKCON4) = oldval_bankcon4;
632 #endif
…
637 }
```

这些增加的代码本身没有什么难度，不再细述，读者可以参考上面移植CS8900A时的讲解。

本书使用的开发板上，DM9000也没有外接EEPROM，所以使用第602～610行的代码来设置默认MAC地址，否则MAC地址为全0。这些代码不是必需的，也可以在系统启动后使用ifconfig命令配置。

③ 注册中断时，指定触发方式。

在dm9000_open中使用request_irq函数注册中断处理函数，修改它即可。DM9000的中断触发方式为上升沿触发。修改的代码如下（第651行）：

```
643 static int
644 dm9000_open(struct net_device *dev)
645 {
646     board_info_t *db = (board_info_t *) dev->priv;
647
648     PRINTK2("entering dm9000_open\n");
649
650 #if defined(CONFIG_ARCH_S3C2410)
651         if (request_irq(dev->irq, &dm9000_interrupt, IRQF_SHARED|IRQF_
TRIGGER_RISING, dev->name, dev))
652 #else
653     if (request_irq(dev->irq, &dm9000_interrupt, IRQF_SHARED, dev->name, dev))
654 #endif
...
```

3. 使用网卡 DM9000

在内核根目录下执行“make menuconfig”命令后，如下配置内核将 DM9000 编译入内核（也可以配置为模块）。

```
Device Drivers  --->
    Network device support  --->
        [*] Network device support
            Ethernet (10 or 100Mbit)  --->
                <*> DM9000 support
```

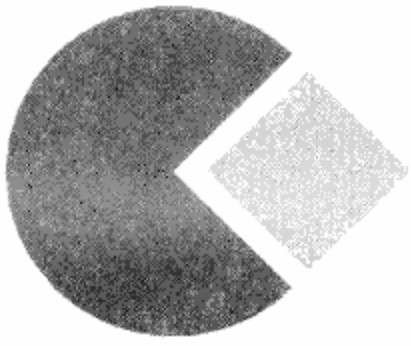
然后编译内核：执行“make uImage”命令即可生成 arch/arm/boot/uImamge。

它的使用方法与上面介绍的 CS8900A 一样，需要注意以下两点。

（1）如果内核中同时加载了 CS8900A 和 DM9000，分别使用 eth0、eth1 表示它们。如果它们都是编进内核的，则 eth0 表示 CS8900A，eth1 表示 DM9000。如果作为模块加载，则根据它们的加载顺序先后使用 eth0、eth1 来表示。

（2）如果要同时使用 CS8900A 和 DM9000，它们的 IP 地址不能是同一个网段。

从两款网卡芯片 CS8900A 和 DM9000 的移植过程，读者可以了解到移植、修改标准驱动程序的方法：了解驱动程序框架，确定外设使用的资源，然后将它们“告诉”驱动程序，并进行适当设置使它们“可用”。串口驱动移植、这章的网卡驱动程序移植都遵循这个步骤。



第 23 章　IDE 接口和 SD 卡驱动程序移植

本章目标

- 了解 IDE 接口驱动程序的框架，掌握移植方法
- 了解硬盘、光盘等块设备使用方法
- 了解 MMC/SD 卡驱动程序的框架
- 掌握通过补丁文件移植驱动程序的方法

23.1　IDE 接口驱动程序移植

23.1.1　IDE 接口相关概念介绍

IDE 的英文全称为“Integrated Drive Electronics”，即“电子集成驱动器”，它的本意是指把“硬盘控制器”与“盘体”（即存储部件）集成在一起的硬盘驱动器。IDE 代表着硬盘的一种类型，但在实际的应用中，人们也习惯用 IDE 来称呼最早出现 IDE 类型硬盘 ATA-1，这种类型的接口随着技术的发展已经被淘汰了，而其后发展出 ATA-2、ATA-3 等更高版本的接口规范。

IDE 硬盘又称为并口硬盘（与下面介绍的 SATA 接口硬盘相对），它的硬件接口有两种：台式机中使用的 40 脚 IDE 接口、笔记本中使用的 44 脚接口（其中的 40 个引脚是一样的）。从外形上看，台式机中的硬盘比较大，为 3.5 英寸，40 个引脚的旁边还有 4 个很大的引脚，它们用来接 12V、5V 电源；笔记本中的硬盘比较小，为 2.5 英寸，电源等就在多出来的 4 个引脚中。

ATA：ATA（AT Attachment）是一个 20 世纪 80 年代由一些软硬件厂家制订的 IDE 驱动器接口规范，AT 是指 IBM PC/AT 个人电脑及其总线结构。通常人们也把 ATA 接口称为 IDE 接口，但实际上两者有着细微的差别，ATA 主要是指硬盘驱动器与计算机的连接规范，而 IDE 则主要是指硬盘驱动器本身的技术规范。经过多年发展，ATA 规范逐渐升级，访问硬盘的速度逐渐提高。它们的特点简要介绍如下。

- ATA-1：这是最初的 IED 标准，ATA-1 主板上只有一个插口，支持 1 个主设备和 1 个从设备，每个设备的最大容量为 504MB。ATA-1 支持 PIO-0、PIO-1 和 PIO-2 共 3 种 PIO 模

式，传输速率只有 3.3MB/s；另外还支持 4 种 DMA 模式（没有得到实际应用）。这种标准的硬盘在市场上基本已经看不到。

- ATA-2：它是对 ATA-1 的扩展，也称为 EIDE（Enhanced IDE）或 Fast ATA。它在 ATA 的基础上增加了两种 PIO 模式和两种 DMA 模式，最高传输速率达到 16.7MB/s。同时引进了 LBA （Log ical Block Address）地址转换方式，原来的地址转换方式为 CHS（Cylinder, Head, Sec tor）。主板上有两个插口，每个插口可以连接 1 个主设备和 1 个从设备，共可以支持 4 个设备。
- ATA-3：它没有引入更高的传输模式，在传输速度上没有任何提升。最重要的是引入了一个划时代的技术——S.M.A.R.T 技术（Self-Monitoring, Analysis and Reporting Technology，自监测、分析和报告技术）。
- ATA-4：也称 Ultra DMA 33 或 ATA33，从它开始正式支持 Ultra DMA 数据传输模式，增加了 PIO-4 传输模式，传输速率达到 33MB/s。它首次采用了 Double Data Rate（双倍数据传输）技术，让接口在一个时钟周期内传输数据两次（上升沿和下降沿各一次），这样数据传输率从 16.7MB/s 提升到 33MB/s。
- ATA-5：也称 Ultra DMA 66 或 ATA66，传输速率达到 66.6MB/s。从 ATA-5 开始，为防止电磁干扰，硬盘的连线开始使用 40 针脚 80 芯的电缆，就是说其中的信号线仍是 40 根，这与以前的接口兼容，新增的 40 根都是地线。打开机箱，40 针脚 80 芯的电缆与原来的电缆相比，显得更细、更密，数一下排线的数目，会发现是 80 根。
- ATA-6：也称 ATA100，它也是使用 40 针脚 80 芯的电缆，传输速率达到了 100MB/s。这是目前市场上主流的 IDE 接口硬盘。
- ATA-7：也称 ATA133，它是 ATA 接口的最后一个版本，传输速率达到 133MB/s。只有迈拓公司推出了 ATA133 标准的硬盘，其他厂商则停止了对 IDE 接口的开发，转而生产 Serial ATA 接口标准的硬盘。

ATAPI：AT Attachment Packet Interface，AT 附加分组接口。ATA 可以使用户方便地在 PC 机上连接硬盘，但有时这样还不够。有些用户需要通过同样方便的手段连接 CDROM、磁带机、MO 驱动器等设备。ATAPI 标准就是为了解决在 IDE/EIDE 接口上连接多种设备而制定的。支持 ATAPI 的 IDE/EIDE 接口可以像连接硬盘一样连接 ATAPI 设备。目前几乎所有的 IDE/EIDE 接口都支持 ATAPI。ATAPI 是一个软件接口，它将 SCSI/ASPI 命令调整到 ATA 接口上，这使得光驱制造商能比较容易地将其高端的 CD/DVD 驱动器产品调整到 ATA 接口上。以光驱为例，它可以像硬盘一样接在任何一个 IDE 接口上，但是它的驱动程序与 IDE 硬盘不同。

SATA 和 PATA：两个都是 ATA 规范，PATA 的全称是 ParallelATA，就是并行 ATA 硬盘接口规范，即 ATA-1、ATA-2 等。PATA 硬盘接口规模已经具有相当的辉煌的历史了，而且从 ATA33/66 一直发展到 ATA100/133。而 SATA 硬盘全称则是 SerialATA，即串行 ATA 硬盘接口规范。当硬盘的访问速度进一步提高时，并行接口的电缆属性、连接器和信号协议都表现出了很大的技术瓶颈，在技术上突破这些瓶颈存在相当大的难度。SATA 的出现就是为了取代 PATA，第一代 SATA 硬盘的写入速度为 150MB/s，第二代 SATA 硬盘的写入速度则高达 300MB/s，比第一代的速度提高了一倍。SATA 除了速度更快外，另一个进步在于它的数据连线，它的体积更小，散热也更好，与硬盘的连接相当方便。与 PATA 相

比，SATA 的功耗更低，这对于笔记本而言是一个好消息，同时独有的 CRC 技术让数据传输也更为安全。

虽然 SATA 最终会取代 PATA，但是目前还有很多的 IDE 硬盘在使用，并且还存在其他使用 IDE 接口的设备（比如 CDROM 等），所以本章仍介绍 IDE 接口驱动程序，注意，它不仅能驱动硬盘，还可以驱动 CDROM 等设备。

23.1.2 IDE 接口驱动程序移植

1. IDE 接口驱动程序框架及源码分析

（1）IDE 接口驱动程序框架。

PC 机上最多可以有主、次（primary/secondary）两个 IDE 接口，每个 IDE 接口又可以支持主、从（master/slave）共两个 IDE 硬盘，所以最多可以有 4 个 IDE 硬盘（包括光盘/软盘等）。

内核有个数组 ide_hwifs[]，数组的每个元素都是一个 ide_hwif_t 数组结构，代表着系统中的一个可能的 IDE 接口。系统初始化时如果检测到一个 IDE 接口，就把相应表项中的 noprobe 字段设置成 0，表示后面要通过这个接口来检测上面是否连接了硬盘，这称为 IDE 枚举。

同时，ide_hwif_t 数据结构中又有个 ide_drive_t 结构数组 drives[]。IDE 枚举时如果检测到某个接口上有磁盘相连，就将相应 ide_drive_t 结构中的 present 字段也设成 1，并根据检测到或从系统的 CMOS 芯片中读到的各项参数设置这个数据结构。也就是说，ide_hwif_t 数据结构是对 IDE 接口的描述，而 ide_drive_t 数据结构是对连接在具体 IDE 接口上的“IDE 设备”的描述。

例如，如果在系统的主（primary）IDE 接口上检测到有主/从两个磁盘相连，就把这两个磁盘的参数分别填入 ide_hwifs[0]中的 drives[0]和 drives[1]，并把它们的 present 字段设置成 1。再例如，如果在次（secondary）IDE 接口上连接着一个 Mitsumi CDROM，那就把它的参数填入 ide_hwifs[1]里面的 drives[0]，并且把 ide_hwifs[1]中的字段 major 设置成 MITSUMI_CDROM _MAJOE。

在 ide_drive_t 结构中有个 void 指针 driver_data，可以指向不同的 ide_driver_t 数据结构（ide_driver_t 和 ide_drive_t 是两种不同的数据结构）。这个指针在系统初始化过程中根据枚举到的 IDE 设备的类型而设置成不同的数据结构。对于 IDE 硬盘，它指向一个 ide_driver_t 数据结构 idedisk_driver。同类的数据结构还有 idetape_driver、ide_cdrom_driver 以及 ide_floppy_driver，分别代表着连接到 IDE 接口上的不同类型的设备。如果 ide_drive_t 结构 drive s[]非空，但是它的 driver 指针却是 NULL，就说明初始化时虽然检测到了硬盘的存在，但是却因某种原因未能完成对设备以及数据结构的初始化。

上面涉及 3 种数据结构：ide_hwif_t、ide_drive_t 和 ide_driver_t，它们刚好表示了 IDE 驱动程序的 3 个层次。代码中，这 3 种变量的名称常写为 hwif、drive、driver，前面两个表示“硬件”，IDE 接口、磁盘；后面一个表示“软件”，表示这个磁盘的具体驱动程序，有 idedisk_driver、detape_driver 等。

为了形象，下面从软件人员的角度画图说明，如图 23.1 所示。

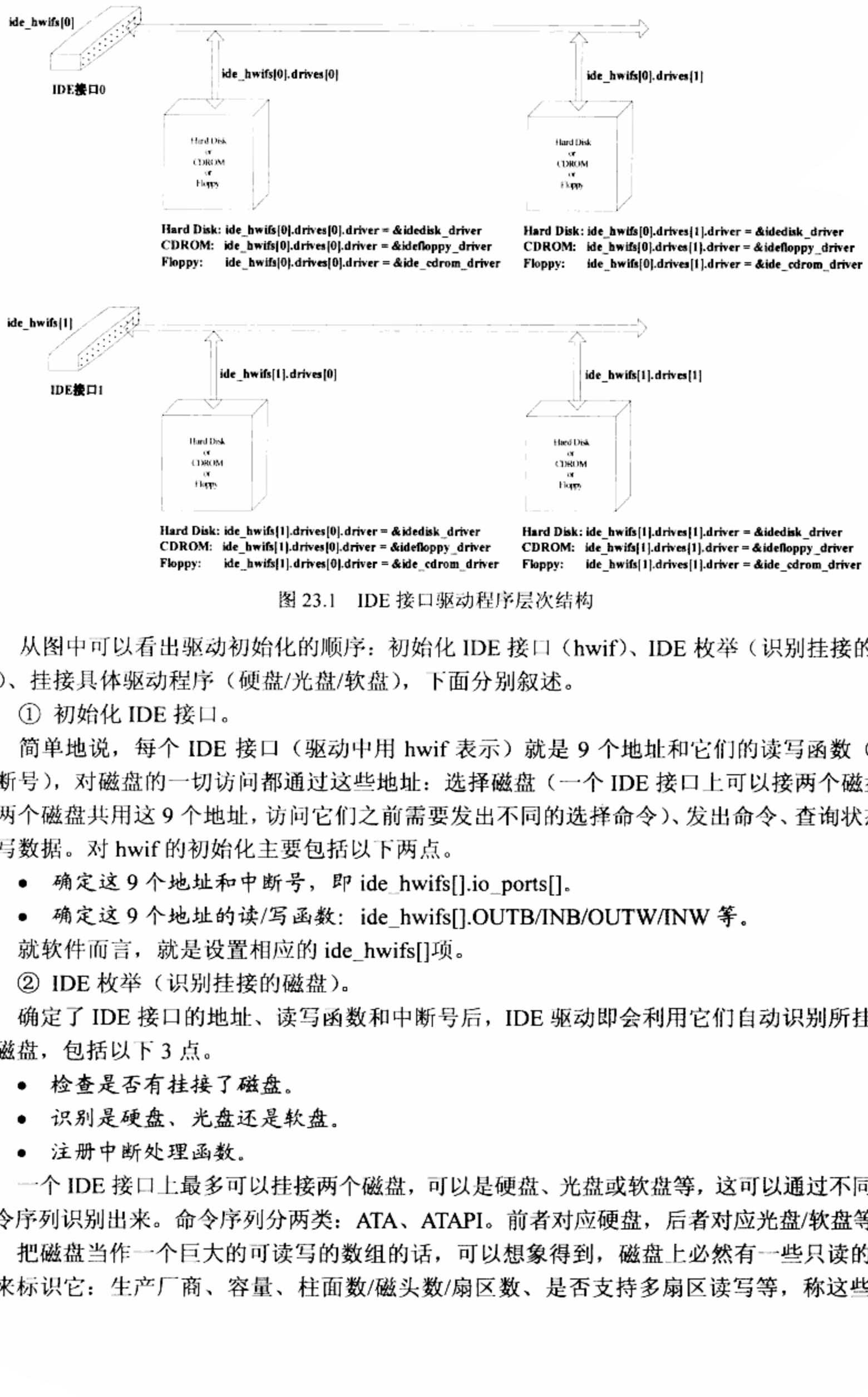


图 23.1 IDE 接口驱动程序层次结构

从图中可以看出驱动初始化的顺序：初始化 IDE 接口（hwif）、IDE 枚举（识别挂接的磁盘）、挂接具体驱动程序（硬盘/光盘/软盘），下面分别叙述。

① 初始化 IDE 接口。

简单地说，每个 IDE 接口（驱动中用 hwif 表示）就是 9 个地址和它们的读写函数（及中断号），对磁盘的一切访问都通过这些地址：选择磁盘（一个 IDE 接口上可以接两个磁盘，这两个磁盘共用这 9 个地址，访问它们之前需要发出不同的选择命令）、发出命令、查询状态、读写数据。对 hwif 的初始化主要包括以下两点。

- 确定这 9 个地址和中断号，即 ide_hwifs[].io_ports[]。
- 确定这 9 个地址的读/写函数：ide_hwifs[].OUTB/INB/OUTW/INW 等。

就软件而言，就是设置相应的 ide_hwifs[]项。

② IDE 枚举（识别挂接的磁盘）。

确定了 IDE 接口的地址、读写函数和中断号后，IDE 驱动即会利用它们自动识别所挂接的磁盘，包括以下 3 点。

- 检查是否有挂接了磁盘。
- 识别是硬盘、光盘还是软盘。
- 注册中断处理函数。

一个 IDE 接口上最多可以挂接两个磁盘，可以是硬盘、光盘或软盘等，这可以通过不同的命令序列识别出来。命令序列分两类：ATA、ATAPI。前者对应硬盘，后者对应光盘/软盘等。

把磁盘当作一个巨大的可读写的数组的话，可以想象得到，磁盘上必然有一些只读的数据来标识它：生产厂商、容量、柱面数/磁头数/扇区数、是否支持多扇区读写等，称这些信

息为“磁盘ID”，大小为512字节，这些信息读出后会保存起来。

就软件而言，就是设置数据结构 ide_hwifs[].drives[0]，对应主磁盘；设置数据结构 ide_hwifs[].drives[1]，它对应从磁盘：

如果检测到有磁盘相连，则将 ide_hwifs[].drives[].present 置1，然后使用 ATA/ATAPI 命令序列查询所接磁盘类型（硬盘/光盘/软盘），读取“磁盘ID”，保存在 ide_hwifs[].drives[].id 中；并设置 ide_hwifs[].drives[].media，取值有 ide_disk、ide_cdrom、ide_floppy 等。

注册中断：同一个 IDE 接口（hwif）下的主、从两个磁盘，它们共用一个中断线。识别出一个 IDE 接口下所有磁盘后，发现有磁盘时才注册中断。

③ 挂接具体驱动程序（硬盘/光盘/软盘）。

步骤②已经获取了磁盘的参数，并在 ide_hwifs[].drives[].media 中标明了磁盘类型（硬盘/光盘/软盘），不同的类型对应不同的驱动程序：硬盘驱动、光盘驱动、软盘驱动等。加载这些驱动时，它们会遍历每个磁盘，即每个 ide_hwifs[].drives[]。比对 ide_hwifs[].drives[].media 项，匹配的话就将 ide_hwifs[].drives[].driver_data 指向相应的结构。比如：对于 IDE 硬盘，它指向一个 ide_driver_t 数据结构 idedisk_driver。同类的数据结构还有 idetape_driver、ide_cdrom_driver 以及 ide_floppy_driver。以硬盘为例，以后就会利用 idedisk_driver 结构中提供的函数进行读写硬盘了。

然后，识别磁盘分区。磁盘分区表的表示方法并不属于 IDE 驱动的范畴，但是了解它有助于调试驱动，参见 23.2.4 小节。

综上所述，可以认为 IDE 驱动分为3层：IDE 接口层（hwif）、磁盘驱动器层（hwif.drives[]）、具体驱动程序（hwif.drives[].driver）。实际上，在 S3C2410/S3C2440 系统中移植 IDE 接口驱动程序，主要的工作也就是让操作系统能识别板上的 IDE 接口。

（2）IDE 接口驱动程序源码分析。

下面按照上述 3 个步骤分析驱动程序，代码都在 drivers/ide/目录下，分别是 ide.c、ide-generic.c、ide-disk.c（硬盘）/ide-cd.c（CDROM）/ide-floppy.c（软盘）。

① 初始化 IDE 接口。

入口在 drivers/ide/ide.c 的 ide_init 函数中，它初始化默认的 IDE 接口，或者调用体系结构相关的函数初始化 IDE 接口，即确定 IDE 接口的9个地址和它们的读写函数。主要的函数调用关系如下：

```
ide_init ->
    init_ide_data ->
        init_hwif_data ->
            init_hwif_data ->
                default_hwif_iops          // 确定IDE接口的默认读写函数，OUTB/INB/OUTW/INW等
                default_hwif_transport     // 也是一些默认的读写函数，会调用上面确定的函数
            init_hwif_default              // 确定默认的 IDE 接口地址，对于 x86 架构外的
CPU，通常没用
            ide_arm_init                   // 确定ARM架构相关的IDE接口地址
ide_init ->
```

```
    probe_for_hwifs ->
        //确定各种“已知的”IDE 接口，它们通常是架构相关的
```

再次提醒：本节只需要关心 IDE 接口的地址（即 ide_hwifs[].io_ports[]）和读写函数（OUTB/INB/OUTW/INW 等）。读者可以阅读上面提到的 default_hwif_iops、default_hwif_transport、init_hwif_default 这 3 个函数进一步了解，不再细述。

② IDE 枚举（识别挂接的磁盘）。

入口在 drivers/ide/ ide-generic.c 的 ide_generic_init 函数中，主要的函数调用关系如下：

```
ide_generic_init ->
    ideprobe_init ->
        probe_hwif ->              // 枚举磁盘
            probe_for_drive ->
                do_probe
        hwif_init                  // 将枚举到的磁盘作为块设备注册到内核中，并注册中断处
理函数等
```

函数 do_probe 在 drivers/ide/ide-probe.c 中定义，它利用前面确定的 IDE 接口的地址发出各类命令序列检测磁盘。摘取此函数里用到的一个函数，可以看到是它如何使用前面确定的 IDE 接口的：SELECT_DRIVE（hwif, drive）被用来发出选择命令，选择主/从磁盘，它在 drivers/ide/ide-iops.c 中的定义如下：

```
168 void SELECT_DRIVE (ide_drive_t *drive)
169 {
170     if (HWIF(drive)->selectproc)
171         HWIF(drive)->selectproc(drive);
172     HWIF(drive)->OUTB(drive->select.all, IDE_SELECT_REG);
173 }
```

第 172 行中，IDE_SELECT_REG 就是 hwif->io_ports[IDE_SELECT_OFFSET]）。这行使用 OUTB 向寄存器 IDE_SELECT_REG 输出一个字节，而 OUTB 通常就是前面的 default_hwif_iops 函数设置的 ide_outb，即 outb。

前面说过每个磁盘都有一些只读信息来标识自己——“磁盘 ID”，读取“磁盘 ID”的函数调用顺序如下：

```
ide_generic_init ->
    ideprobe_init ->
        probe_hwif ->
            probe_for_drive ->
                do_probe ->
                    try_to_identify ->
                        actual_try_to_identify ->
                            do_identify ->
```

```
hwif->ata_input_data ->
    INSW ->
        INW
```

它们最终还是通过前面确定的 OUTB/INB/OUTW/INW 等函数来完成，只要读出了“磁盘 ID”，磁盘的枚举基本就成功了。

③ 挂接具体驱动程序（硬盘/光盘/软盘）。

以硬盘驱动程序为例，代码在 drivers/ide/ide-disk.c 中，入口函数为 idedisk_init，代码如下：

```
1311 static int __init idedisk_init(void)
1312 {
1313     return driver_register(&idedisk_driver.gen_driver);
1314 }
```

第 1313 行向内核注册驱动后，最终会调用 drivers/ide/ide-disk.c 中的 ide_disk_probe 函数来识别每个磁盘。

对于所有磁盘，如果是硬盘（ide_hwifs[].drives[].driver_req 为“ide-disk”、ide_hwifs[].drives[].media 等于 ide_disk），则挂接硬盘驱动程序：ide_hwifs[].drives[].driver = &idedisk_driver。

为硬盘挂接具体驱动程序的主要函数调用关系如下：

```
idedisk_init ->
    ide_disk_probe ->
        idkp->driver = &idedisk_driver;    // 挂接硬盘驱动程序
        idedisk_setup // 根据磁盘 ID 作一些设置，
                      // 比如获取磁盘容量、确定能否多扇区操作、确定读写能否以 32 位进行等
        set_capacity   // 设置容量
        g->fops = &idedisk_ops;    // 确定文件处理函数(用户调用 open/read/write 等
时对应的函数)
        add_disk(g);               // 识别分区
```

假如一切正常，那么系统启动后就可访问磁盘了。

2. S3C2410/S3C2440 开发板上的 IDE 接口驱动程序移植

从前面的分析可以知道，IDE 接口驱动程序的移植只有一点：确定 IDE 的接口（地址、中断号、读写函数）。

开发板上的 IDE 接口硬件连线如图 23.2 所示。

在修改代码之前，先介绍一下 IDE 设备的寄存器。对硬盘、光盘、软盘等 IDE 设备的所有操作，都是通过读写它们的寄存器来完成的。这些寄存器分为两种：命令块寄存器、控制块寄存器。前者被用来给设备发送命令或是查询状态；后者被用来控制设备，它也可以用来查询状态。这两类寄存器通过 CS1、CS0、DA2～DA0 来分辨，它们的功能和地址信号如表

23.1 所示。

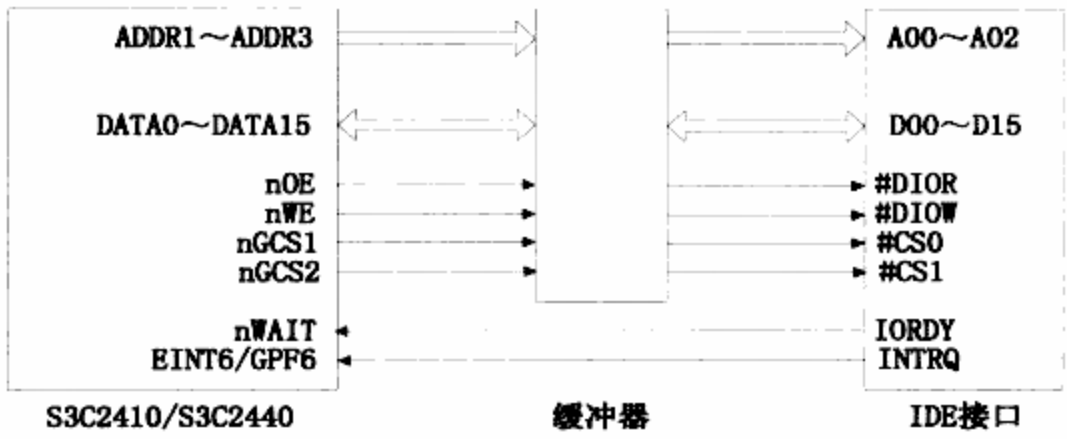


图 23.2 IDE 接口连线

表 23.1 IDE 设备寄存器及选择地址

地址					功能	
CS0	CS1	DA2	DA1	DA0	读操作	写操作
N	N	x	x	x	数据总线为高阻态	不使用
					控制块寄存器（Control block registers）	
N	A	0	x	x	数据总线为高阻态	不使用
N	A	1	0	x	数据总线为高阻态	
N	A	1	1	0	备份状态寄存器（Alternate Status）	设备控制寄存器
N	A	1	1	1	过时，不再使用	不使用
					命令块寄存器（Command block registers）	
A	N	0	0	0	数据寄存器	数据寄存器
A	N	0	0	1	错误寄存器	写前预补偿寄存器
A	N	0	1	0	扇区数寄存器	扇区数寄存器
A	N	0	1	1	扇区号寄存器	扇区号寄存器
A	N	1	0	0	柱面号寄存器（低字节）	柱面号寄存器（低字节）
A	N	1	0	1	柱面号寄存器（高字节）	柱面号寄存器（高字节）
A	N	1	1	0	驱动器/磁头寄存器	驱动器/磁头寄存器
A	N	1	1	1	主状态寄存器	命令寄存器
A	A	x	x	x	无效的地址	无效的地址

注：A 表示信号有效，即低电平；N 表示信号无效，即高电平；x 表示任意电平。

这些寄存器都是 8 位的。当设置好相关寄存器之后，就可以通过“数据端口”发送或读取数据了。“数据端口”的地址与数据寄存器一样，只不过传输的数据是 16 位的。

从图 23.2 和表 23.1 可知，命令块寄存器的基地址为 BANK1 的基地址，即 0x08000000，这些寄存器的地址分别为：0x08000000、0x08000002、0x08000004、……、0x0800000E；控

制块寄存器的基地址为BANK2的基地址，即0x10000000，只用到一个地址：0x1000000C。图23.2中，ADD3～ADD1连接到DA2～DA0，没有使用ADDR0，确定地址时需要注意到这点。另外，这些地址是物理地址，在内核中使用时需要映射为虚拟地址。

从图23.2还可以知道：中断引脚为EINT6，上升沿有效；使用nWAIT信号；数据位宽为16。

如前所述，只要“告诉”内核这些地址、中断号就可以了（还要进行相关设置使它们“可用”）。这些地址的读写函数使用前面设置的默认函数。

只需要修改两个文件：drivers/ide/arm/ide_arm.c、drivers/ide/Kconfig。

在drivers/ide/arm/ide_arm.c文件中增加ide_s3c24xx_init函数，修改后的文件如下（使用编译开关CONFIG_ARCH_S3C2410包含起来的代码是新加的）：

```
...
28 #elif defined(CONFIG_ARCH_S3C2410)
29 #include <linux/irq.h>
30 #include <asm/arch-s3c2410/regs-mem.h>
31 #include <asm/arch-s3c2410/regs-gpio.h>
32 # define IDE_ARM_IOs        {0x08000000, 0x10000000}    // IDE接口CS0、CS1
的物理基址
33 # define IDE_ARM_IRQPIN     S3C2410_GPF6                // 中断引脚
34 #else
...
39 #ifdef CONFIG_ARCH_S3C2410
40 /* Set hwif for S3C2410/S3C2440, by www.100ask.net */
41 static void __init ide_s3c24xx_init(void)
42 {
43     int i;
44     unsigned int oldval_bwscon;     /* 用来保存BWSCON寄存器的值 */
45     unsigned long mapaddr0;
46     unsigned long mapaddr1;
47     unsigned long baseaddr[] = IDE_ARM_IOs;
48     hw_regs_t hw;
49
50     /* 设置BANK1/2: 总线宽度为16 */
51     oldval_bwscon = readl(S3C2410_BWSCON);
52     writel((oldval_bwscon & ~((3<<4)|(3<<8))) \
53         | S3C2410_BWSCON_DW1_16 | S3C2410_BWSCON_WS1
54         | S3C2410_BWSCON_DW2_16 | S3C2410_BWSCON_WS2, S3C2410_BWSCON);
55
56     /* 设置BANK1/BANK2的时间参数 */
57       writel((S3C2410_BANKCON_Tacs4  |  S3C2410_BANKCON_Tcos4  |  S3C2410_
```

```
BANKCON_Tacc14
    58              | S3C2410_BANKCON_Tcoh4 | S3C2410_BANKCON_Tcah4 | S3C2410_
BANKCON_Tacp6
    59          | S3C2410_BANKCON_PMCnorm), S3C2410_BANKCON1);
    60           writel((S3C2410_BANKCON_Tacs4 | S3C2410_BANKCON_Tcos4 | S3C2410_
BANKCON_Tacc14
    61              | S3C2410_BANKCON_Tcoh4 | S3C2410_BANKCON_Tcah4 | S3C2410_
BANKCON_Tacp6
    62          | S3C2410_BANKCON_PMCnorm), S3C2410_BANKCON2);
    63
    64     /*
    65      * 设置 IDE 接口的地址, ADDR3~ADDR1 接到 IDE 接口的 A02~A00
    66      * 注意: 没有使用 ADDR0, 所以下面确定地址时, 都左移 1 位
    67      */
    68     mapaddr0 = (unsigned long)ioremap(baseaddr[0], 16);
    69     mapaddr1 = (unsigned long)ioremap(baseaddr[1], 16);
    70
    71     memset(&hw, 0, sizeof(hw));
    72
    73     for (i = IDE_DATA_OFFSET; i <= IDE_STATUS_OFFSET; i++)
    74         hw.io_ports[i] = mapaddr0 + (i<<1);                    // 命令块寄存器
    75
    76     hw.io_ports[IDE_CONTROL_OFFSET] = mapaddr1 + (6<<1);     // 控制块寄存器
    77
    78     /* 设置中断引脚 */
    79     hw.irq = s3c2410_gpio_getirq(IDE_ARM_IRQPIN);
    80     s3c2410_gpio_cfgpin(IDE_ARM_IRQPIN, S3C2410_GPIO_IRQ);
    81     set_irq_type(hw.irq, IRQF_TRIGGER_RISING);
    82
    83     /* 注册 IDE 接口 */
    84     ide_register_hw(&hw, 1, NULL);
    85 }
    86 #endif
    87
    88 void __init ide_arm_init(void)
    89 {
    90 #ifdef CONFIG_ARCH_S3C2410
    91     if (IDE_ARM_HOST) {
    92         ide_s3c24xx_init();
```

```
93      }
94 #else
95      if (IDE_ARM_HOST) {
96          hw_regs_t hw;
97
98          memset(&hw, 0, sizeof(hw));
99          ide_std_init_ports(&hw, IDE_ARM_IO, IDE_ARM_IO + 0x206);
100         hw.irq = IDE_ARM_IRQ;
101         ide_register_hw(&hw, 1, NULL);
102     }
103 #endif
104 }
```

第32、33行定义了IDE接口CS0、CS1的物理基址和中断引脚，它们在下面会用到。

第51～54行用来设置存储控制器，IDE接口使用BANK1、BANK2，数据总线位宽为16；还使用到了nWAIT信号。

第57～62行设置BANK1、BANK2的时序参数，现在设为比较宽松的值，基本都取最大值，读者可以根据硬盘特性进行调整。

第68～76行设置IDE接口的9个地址，第68、69两行首先将物理地址映射为虚拟地址，后面就是直接赋值了。需要注意的是：由于S3C2410/S3C2440的ADDR3～ADDR1接到IDE接口的DA0 2～DA00，没有使用ADDR0，所以是74、76行中地址的偏移都左移了1位。

第79～81行设置中断引脚，第79行的s3c2410_gpio_getirq函数的返回值就是中断号IRQ_ EINT6；第80行用来选择引脚功能为外部中断，第81行用来设置中断的触发方式为上升沿触发。第80～81行的功能完全可以在调用request_irq函数注册中断处理函数时，通过指定参数irqflags为IRQF_TRIGGER_RISING来完成，在这里之所以不使用这种方法是为了减少修改其他文件（在drivers/ide/ide-probe.c中注册中断处理函数）。

第84行调用ide_register_hw注册IDE接口，其实就是在将上面确定的地址、中断号填入某个不用的ide_hwifs[]表项中。

后面第88行开始的ide_arm_init函数直接调用ide_s3c24xx_init函数。

ide_arm_init函数在drivers/ide/ide.c文件中的init_ide_data函数中被调用，需要设置配置项CONFIG_IDE_ARM。ide_arm_init函数被调用时的代码如下：

```
static void __init init_ide_data (void)
{
...
#ifdef CONFIG_IDE_ARM
    ide_arm_init();
#endif
}
```

配置项CONFIG_IDE_ARM在drivers/ide/Kconfig中定义，代码如下：

```
config IDE_ARM
    def_bool ARM && (ARCH_A5K || ARCH_CLPS7500 || ARCH_RPC || ARCH_SHARK)
```

在配置菜单中看不到它，它取默认值。增加一个依赖条件 ARCH_S3C2410，新代码如下：

```
config IDE_ARM
    def_bool ARM && (ARCH_A5K || ARCH_CLPS7500 || ARCH_RPC || ARCH_SHARK ||
ARCH_S3C2410)
```

23.1.3 IDE 接口驱动程序测试

首先配置内核，需要增加不少配置项；然后还要移植一些分区、格式化的工具。

1. 配置、编译内核

为了支持硬盘、CDROM 等设备，需要在设备驱动、文件系统等方面设置相应的配置项。在内核根目录下执行“make menuconfig”后，按照下面的指示进行配置即可。

```
Device Drivers  --->
    <*> ATA/ATAPI/MFM/RLL support  --->
        <*>   Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support
        <*>     Include IDE/ATA-2 DISK support
        <*>     Include IDE/ATAPI CDROM support
        [*]     legacy /proc/ide/ support
        <*>     generic/default IDE chipset support

File systems  --->
    CD-ROM/DVD Filesystems  --->
        <*> ISO 9660 CDROM file system support
        [*]   Microsoft Joliet CDROM extensions
        [*]   Transparent decompression extension
        <*> UDF file system support
    DOS/FAT/NT Filesystems  --->
        <*> VFAT (Windows-95) fs support
        (936) Default codepage for FAT
        (cp936) Default iocharset for FAT
    Native Language Support  --->
        <*>   Simplified Chinese charset (CP936, GB2312)
```

值得一提的是上面的“(936) Default codepage for FAT”和“(cp936) Default iocharset for FAT”。

首先介绍一下字符集的概念：计算机中使用数值来表示字符，比如使用 0x41 来表示字符“A”。同一个数值在不同的字符集里可能表示不同的字符，比如数值 0xABB6 在 gb2312 字符集中是“东”字，在 bi5 字符集中却是“奎”字，在 UNICODE 字符集中没有对应的字符。

在FAT文件系统中存储一个短文件名时使用本地的字符集进行存储，这个字符集被称为“codepage”；存储长文件名时，使用UNICODE字符集。在Linux中，查看FAT文件系统的文件时，比如使用“ls”命令时，这些以“codepage”或UNICODE字符集保存的数值，还要转换为另一个字符集的数值，才发送到控制台上去显示。这个“显示用”的字符集就称为“iocharset”。

在Linux下挂接FAT文件系统时，经常碰到汉字的目录名、文件名显示为问号，就是因为在挂接文件系统时，没有正确设置“codepage”或“iocharset”所致。“codepage”和“iocharset”可以相同，也可以不同，比如我们可以这样挂接硬盘：

```
mount -o codepage=950,iocharset=cp936 /dev/hda2 /mnt
```

cp950表示字符集BIG5，cp936表示字符集GB2312，这个命令使得FAT文件系统中使用BIG5字符集保存的繁体文件名，可以通过GB2312字符集正确显示为简体字。

最后一项配置“Native Language Support”表示“本地语言支持”，就是将各种字符集编译进内核，或译编为模块。

2. 移植工具

在Busybox中已经有分区工具fdisk，还需要移植EXT2文件系统格式化工具mke2fs、FAT文件系统格式化工具mkdosfs。

（1）移植mke2fs。

mke2fs工具是开源项目e2fsprogs的一个工具，这个项目的源码可以从以下网址下载：http://sourceforge.net/projects/e2fsprogs/。

也可以使用/work/tools目录下的e2fsprogs-1.40.2.tar.gz。

解压缩后参照它的INSTALL文件即可编译。对于交叉编译，在执行“../configure”时需要指定交叉编译工具链和目标板。执行的命令如下：

```
$ cd /work/tools
$ tar xzf e2fsprogs-1.40.2.tar.gz
$ cd e2fsprogs-1.40.2/
$ mkdir build; cd build
$ ../configure --with-cc=arm-linux-gcc --with-linker=arm-linux-ld --enable-elf-shlibs --host=arm -prefix=/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux
$ make
$ make install-libs
```

在build/misc/目录下即生成mke2fs工具，把它放到开发板根目录/sbin下。

最后一条命令在/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/下的include、lib目录中安装一些头文件和库，比如uuid/uuid.h、libuuid.a、libuuid.so等，它们将在后面编译嵌入式GUI系统QTOPIA时用到。

（2）移植mkdosfs。

mkdosfs工具是开源项目dosfstools的一个工具，这个项目的源码可以从以下网址下载：http://ftp.debian.org/debian/pool/main/d/dosfstools/。

也可以使用/work/tools 目录下的 dosfstools_2.11.orig.tar.gz。

解压缩后直接修改它的 Makefile，修改其中的编译工具即可如下所示。

修改前：

```
CC = gcc
```

修改后：

```
CC = arm-linux-gcc
```

执行“make”命令后，在 mkdosfs/目录下即生成 mkdosfs 工具，把它放到开发板根目录/sbin 下。

分区、格式化、使用 IDE 接口设备

开发板上只有一个 IDE 接口，所以最多可以接两个设备：要么是主设备（hda）、要么是从设备（hdb），可以通过设置它们的跳线来确定谁是主设备、谁是从设备。

设备文件/dev/hda、/dev/hdb 表示整个磁盘，设备文件/dev/hda1、/dev/hda2、/dev/hdb1、/dev/hdb2 等表示磁盘的分区。初始化硬盘时，驱动程序会自动识别它的分区。

（1）创建设备文件：如果使用 mdev 机制，这个步骤可以省略。

进行下一步操作前，先在开发板根文件系统中建立几个设备文件，以下命令在开发板上执行。

```
# mknod /dev/hda  b 3 0
# mknod /dev/hda1 b 3 1
# mknod /dev/hda2 b 3 2
# mknod /dev/hda3 b 3 3
# mknod /dev/hda4 b 3 4

# mknod /dev/hdb  b 3 64
# mknod /dev/hdb1 b 3 65
# mknod /dev/hdb2 b 3 66
# mknod /dev/hdb3 b 3 67
# mknod /dev/hdb4 b 3 68
```

（2）分区。

分区工具 fdisk 操作的是整个设备，比如“fdisk /dev/hda”。fdisk 提供字符界面的菜单供用户进行各种操作：查看、增加、删除分区，查看主扇区的数据。需要注意的是：增加、删除分区等操作只是在内存中完成，还没有写入磁盘，这需要在主菜单中选择“w”才会将变化的数据写入磁盘。

如果不了解主分区、逻辑分区的概念，可以参考本章的附录。

（3）格式化。

分区之后就是格式化了，可以使用 mke2fs 工具将某个分区格式化为 EXT2 文件系统，或是使用 mkdosfs 工具格式化为 FAT 文件系统。mkdosfs 工具的默认格式为 FAT16，要格式化为 FAT32 需要增加参数“-F 32”。使用的命令示范如下：

```
# mke2fs /dev/hda1
# mkdosfs -F 32 /dev/hda2
```

（4）挂接磁盘。

对于已经格式化好的磁盘，直接使用 mount 命令即可挂接，之后就可以使用了。挂接命令示例如下：

```
// 对于 EXT2 文件系统，不需要指定字符集
# mount /dev/hda1 /mnt
// 对于 FAT 文件系统，指定 codepage 和 iocharset；可省略，因为内核已设置默认字符集为 cp936
# mount -o codepage=936,iocharset=cp936 /dev/hda2 /mnt
```

如果 IDE 接口上接了光驱，在启动内核时会看到类似以下的信息：

```
Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
hdb: BENQ DVD DD DW1650, ATAPI CD/DVD-ROM drive
ide0 at 0xc4872000-0xc4872007,0xc487400c on irq 50
hdb: ATAPI 48X DVD-ROM DVD-R CD-R/RW drive, 2048kB Cache
Uniform CD-ROM driver Revision: 3.20
```

这表示识别到了一个光驱，它是从设备（hdb）。光盘没有分区，直接使用“整个设备”，即/dev/hdb。在光驱中装入光盘后，挂接命令如下：

```
# mount -o iocharset=gb2312 /dev/hdb /mnt  # 要显示简体汉字，指定字符集 gb2312
```

23.2　SD 卡驱动程序移植

23.2.1　SD 卡相关概念介绍

MMC：MMC 就是 MultiMediaCard 的缩写，即多媒体卡。它是一种非易失性存储器件，体积小巧（24mm×32mm×1.4mm，类似一张邮票大小）、容量大、耗电量低、传输速度快，广泛应用于电子玩具、PDA、数码相机、手机、MP3 等设备中。以前的 MMC 规范的数据传输宽度只有 1 位，最新的 4.0 版 MMC 规范拓宽了 4 位、8 位带宽，时钟频率达到 52MHz 频率，从而支持 50MB/s 的传输速率。对于 SD 卡的“数据保全”特性，MMC 协会接纳了具有竞争性的安全卡标准——Secure MMC 1.1 版规范。

SD：SD 卡的英文名为 Secure Digital Memory Card，即安全数码卡。它在 MMC 的基础上发展而来，增加了两个主要特色：SD 卡强调数据的保全，可以设定所存储数据的使用权限，防止他人复制；另一个特色就是传输速度比 2.11 版的 MMC 卡快了 4 倍。在数据传输和物理规范上，SD 卡向前兼容 MMC 卡；在外观上，SD 卡尺寸为 24mm×32mm ×2.1mm，是比 MMC 卡更厚一些；这两个特点使得支持 SD 卡的设备也可以支持 MMC 卡。SD 卡和 2.11

版的 MMC 卡完全兼容。

SDIO：SDIO 是在 SD 标准上定义了一种外设接口，它和 SD 卡规范间的一个重要区别是增加了低速标准。SDIO 卡只需要 SPI 和 1 位 SD 传输模式。低速卡的目标应用是以最小的硬件开支支持低速 I/O 能力。低速卡支持类似调制解调器、条码扫描仪和 GPS 接受器等应用。对“组合”卡（存储器+ SDIO）而言，全速和 4 位操作对卡内存储器和 SDIO 部分都是强制要求的。

MMC/SD/SDIO 这 3 种存储卡都支持两种接口：对于 MMC 卡，称为 MMC 接口和 SPI 接口；对于 SD 卡、SDIO 卡，称为 SD 接口和 SPI 接口。SD 接口有 1 位和 4 位之分，上电时默认使用 1 位模式，设置 SD 主机后可以使用 4 位模式。

MCI：MCI 是 Multimedia Card Interface 的简称，即多媒体卡接口。上述的 MMC、SD、SDIO 卡定义的接口都属于 MCI 接口。MCI 这个术语在驱动程序中经常使用，很多文件、函数名字都包含“mci”。

23.2.2 SD 卡驱动程序移植

S3C2410/S3C2440 中集成了一个 MMC/SD/SDIO 主机控制器，用于访问外接的 MMC 卡、SD 卡或 SDIO 卡，它有如下特性。

- 支持 SD 存储卡规范 1.0、MMC 卡规范 2.11。
- 支持 SDIO 卡规范 1.0。
- 内部有 16 个字（64 字节）的 FIFO，用于发送、接收数据。
- 40 位的命令寄存器（SDICARG[31:0]+SDICCON[7:0]）。
- 136 位的回应寄存器（SDIRSPn[127:0]+ SDICSTA[7:0]）。
- 8 位的预分频逻辑电路：

  对于 S3C2410，Freq. = System Clock /（2（P + 1））；

  对于 S3C2440，Freq = System Clock/（P + 1）。
- CRC7 和 CRC16 校验码产生器。
- 支持查询、中断或者 DMA 传输模式。
- 数据总线的宽度可以是 1 位或 4 位，支持串流（Stream）或区块（Block）传输方式。
- 对于 SD 卡、SDIO 卡，传输数据时最高时钟频率为 25MHz。
- 对于 MMC 卡，传输数据时最高时钟频率为 20MHz。

Linux 2.6.22.6 尚未支持 S3C2410/S3C2440 的 MMC/SD/SDIO 控制器，需要移植驱动程序。在这之前，先介绍一下 MMC/SD 驱动的框架。这些驱动程序在内核 drivers/mmc 目录下。

1. 内核 MMC/SD 驱动程序框架

内核 drivers/mmc 目录下有 3 个子目录：card/、core/和 host/，这刚好表示了 MMC/SD 驱动程序的 3 个层次，层次结构如图 23.3 所示。

（1）区块层。

向文件系统层、用户空间提供文件操作的接口，主要文件是 card/目录下的 block.c，queue.c 向它提供了几个函数来操作队列。

区块层调用 core/目录下的 core.c、sysfs.c 提供的接口来识别存储卡的分区、读写存储卡

等功能。

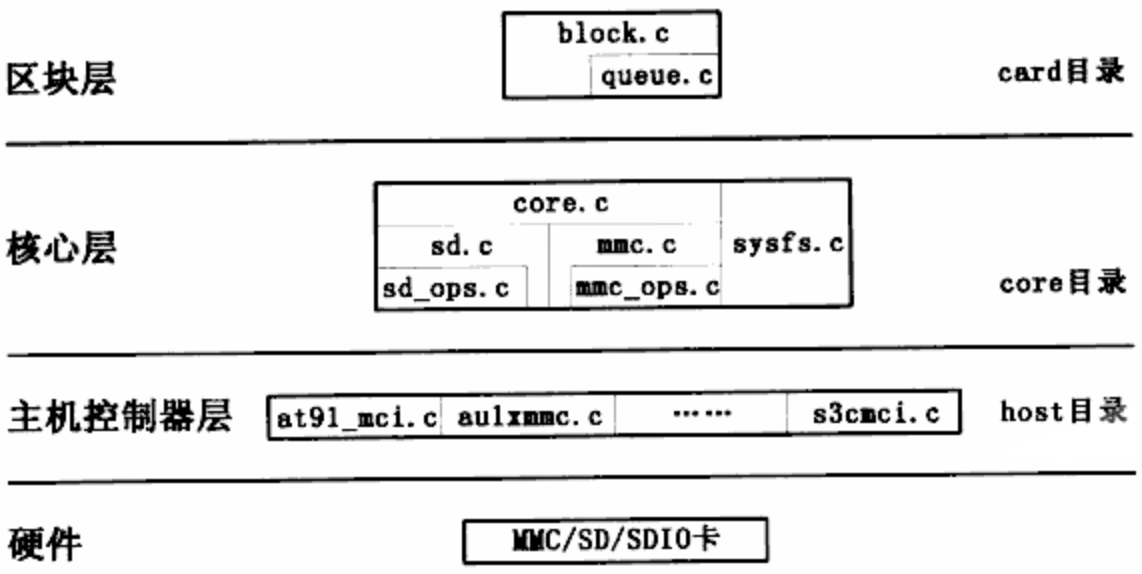


图 23.3　MMC/SD 驱动程序层次结构图

（2）核心层。

核心层代码在 core/目录下，它封装了 MMC/SD 命令，实现 MMC/SD 协议，它调用主机控制器层的接口完成存储卡的识别、设置、读写等。

如图 23.3 所示，core.c 文件由 sd.c、mmc.c 两个文件支撑，core.c 把 MMC 卡、SD 卡的共性抽象出来，它们的差别由 sd.c 和 sd_ops.c、mmc.c 和 mmc_ops.c 来完成。

sysfs.c 是 MMC/SD 驱动程序的 sysfs 文件系统的实现，它提供一些内核体系相关的函数来实现注册、调用驱动程序；在用户空间挂接 sysfs 文件系统后，可以从中看到 MMC/SD 的一些信息。

（3）主机控制器层。

核心层根据需要构造各种 MMC/SD 命令，这些命令怎么发送给 MMC/SD 卡呢？这通过主机控制器层来实现。这层是架构相关的，里面针对各款 CPU 提供了一个文件，目前支持的 CPU 还很少。

以本节即将移植的 s3cmci.c 为例，它首先进行一些低层设置，比如设置 MMC/SD/SDIO 控制器使用到的 GPIO 引脚、使能控制器、注册中断处理函数等，然后向上面的核心层增加一个主机（Host），这样核心层就可以调用 s3cmci.c 提供的函数来识别、使用具体存储卡了。

在向核心层增加主机之前，s3cmci.c 设置了一个 mmc_host_ops 结构，它实现了两个函数：发起访问请求的 request 函数，进行一些属性设置（时钟频率、数据线位宽等）的 set_ios 函数。以后上层对存储卡的操作都通过调用这两个函数来完成。

下面列出识别存储卡、区块层发起操作请求这两种情况下函数的主要调用关系，读者根据函数名称及所在的文件，就可以了解到上面讲述的层次结构。

仍以即将移植的 s3cmci.c 为例。

（1）识别存储卡。

```
s3c2410sdi_probe(host/s3c2410mci.c)
    mmc_alloc_host(core/core.c)
        mmc_rescan(core/core.c)
            mmc_attach_sd(core/sd.c)(SD卡的入口点)          // 尝试识别SD卡
                mmc_sd_init_card(core/sd.c)
```

```
                    mmc_all_send_cid(core/sd_ops.c)          // 读取存储卡的 CID
                        mmc_wait_for_cmd(core/core.c)        // 发起并等待请求完成
                            mmc_start_request(core/core.c)
                                host->ops->request(host,  mrq);//  host/s3cmci.c
的 s3cmci_request
    ……
                    mmc_switch_hs(sd.c)                      // 设置为高速模式
                        mmc_set_timing(core.c)               // 设置时钟
                            mmc_set_ios(core.c)
                                host->ops->set_ios(host,  ios);//  host/s3cmci.c
的 s3cmci_set_ios
    ……
            mmc_attach_mmc(core/mmc.c)(MMC 卡的入口点)          // 尝试识别 MMC 卡
            ……
```

（2）区块层发起操作请求。

```
mmc_blk_issue_rq(block/block.c)
    mmc_wait_for_req(core/core.c)              // 发起并等待请求完成
        mmc_start_request(core/core.c)
            host->ops->request(host, mrq);    // host/s3cmci.c 中的 s3cmci_request
```

2. S3C2410/S3C2440 的 MMC/SD/SDIO 控制器驱动程序移植

开发板上 MMC/SD 接口的连线如图 23.4 所示，nCD 接到外部中断引脚 EINT16，接上或拔下存储卡时会触发中断。

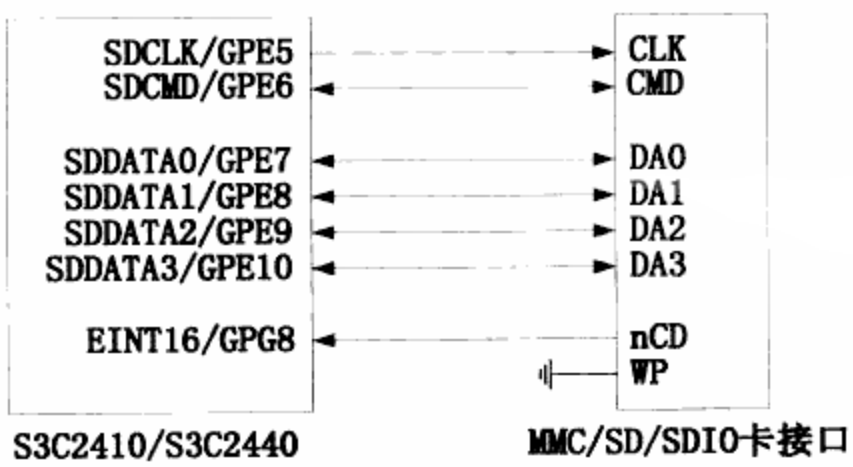


图 23.4 MMC/SD/SDIO 卡连线图

移植 MMC/SD/SDIO 控制器驱动程序分为 3 步骤：打补丁、增加 MMC/SD 平台设备、修改主机控制器驱动程序以指定上图中的 nCD 中断。

（1）给内核打补丁。

内核中要增加对 S3C2410/S3C2440 的 MMC/SD/SDIO 控制器的支持，需要打补丁。补丁文件可以从以下网址获得：

http://svn.openmoko.org/branches/src/target/kernel/2.6.21.x/patches/

下载其中含“s3cmci”、“s3c_mci”字样的6个文件，保存在s3c_mci_patch目录下（在/work/system/s3c_mci_patch.tar.bz2是已经下载好的），这6个文件为：

```
s3c_mci.patch
s3c_mci_platform.patch
s3cmci-dma-free.patch
s3cmci-stop-fix.patch
s3cmci-unfinished-write-fix.patch
s3cmci_dbg.patch
```

按照这些文件的次序，执行以下命令打上补丁（内核目录为linux-2.6.22.6，补丁文件所在目录为s3c_mci_patch，它们都在/work/system目录下）。

```
$ cd linux-2.6.22.6
$ patch -p1 < ../s3c_mci_patch/s3c_mci.patch
$ patch -p1 < ../s3c_mci_patch/s3c_mci_platform.patch
$ patch -p1 < ../s3c_mci_patch/s3cmci-dma-free.patch
$ patch -p1 < ../s3c_mci_patch/s3cmci-stop-fix.patch
$ patch -p1 < ../s3c_mci_patch/s3cmci-unfinished-write-fix.patch
$ patch -p1 < ../s3c_mci_patch/s3cmci_dbg.patch
```

这些补丁修改或添加的文件有：

```
drivers/mmc/host/Kconfig                          // 修改
drivers/mmc/host/Makefile                         // 修改
include/asm-arm/arch-s3c2410/regs-sdi.h           // 修改
include/asm-arm/arch-s3c2410/mci.h                // 新增
drivers/mmc/host/mmc_debug.c                      // 新增
drivers/mmc/host/mmc_debug.h                      // 新增
drivers/mmc/host/s3cmci.c                         // 新增
drivers/mmc/host/s3cmci.h                         // 新增
```

然后配置内核，增加MMC块设备驱动、s3c24xx的MMC/SD卡驱动，配置如下：

```
Device Drivers  --->
    <*> MMC/SD card support  --->
        [*]   MMC debugging
        <*>   MMC block device driver
        <*>   Samsung S3C SD/MMC Card Interface support
```

配置好后编译内核时，会发现MMC_ERR_DMA、MMC_ERR_BUSY、MMC_ERR_CANCELED这3个宏没有定义，在include/linux/mmc/core.h中增加以下3行即可：

```
#define MMC_ERR_DMA     6
#define MMC_ERR_BUSY    7
#define MMC_ERR_CANCELED 8
```

（2）增加MMC/SDI平台设备。

drivers/mmc/s3cmci.c文件的入口函数为s3cmci_init，代码如下：

```
1363 static int __init s3cmci_init(void)
1364 {
1365     platform_driver_register(&s3cmci_driver_2410);
1366     platform_driver_register(&s3cmci_driver_2412);
1367     platform_driver_register(&s3cmci_driver_2440);
1368     return 0;
1369 }
```

第1365～1367行向内核注册3个平台驱动，本书关心的两个驱动s3cmci_driver_2410、s3cmci_driver_2440的定义如下（在同一个文件中）：

```
1335 static struct platform_driver s3cmci_driver_2410 =
1336 {
1337     .driver.name    = "s3c2410-sdi",
1338     .probe      = s3cmci_probe_2410,
1339     .remove     = s3cmci_remove,
1340     .suspend    = s3cmci_suspend,
1341     .resume     = s3cmci_resume,
1342 };
……
1353 static struct platform_driver s3cmci_driver_2440 =
1354 {
1355     .driver.name    = "s3c2440-sdi",
1356     .probe      = s3cmci_probe_2440,
1357     .remove     = s3cmci_remove,
1358     .suspend    = s3cmci_suspend,
1359     .resume     = s3cmci_resume,
1360 };
```

当发现名称为“s3c2410-sdi”或“s3c2440-sdi”的平台设备时，会调用其中的s3cmci_probe _2410或s3cmci_probe_2440函数来枚举MMC/SD设备。

如上所述，要为MMC/SD驱动定义平台设备。在内核文件arch/arm/plat-s3c24xx/devs.c中，已经有一个平台设备的数据结构s3c_device_sdi（名称为“s3c2410-sdi”），只不过还没有使用它。

现在仿照它在以下两个文件中分别增加s3c2410_device_sdi、s3c2440_device_sdi结构，

并把它们加入设备列表中（就是smdk2410_devices[]、smdk2440_devices[]数组）。

> **注意** 下列代码前面的序号可能与内核目录/work/system/linux-2.6.22.6 下的文件不匹配，这是有可能的，如果/work/system/linux-2.6.22.6 曾经应用了补丁文件 linux-2.6.22.6-100ask24x0.patch，那么它是本书完结时修改的内核最终版本。而下面的代码，是读者按照前面章节逐步修改内核所得来的。

① 修改 arch/arm/mach-s3c2410/mach-smdk2410.c。

以下是修改的代码。

```
89 /* SDI */
90 static struct resource s3c2410_sdi_resource[] = {
91     [0] = {
92         .start = S3C2410_PA_SDI,
93         .end   = S3C2410_PA_SDI + S3C24XX_SZ_SDI - 1,
94         .flags = IORESOURCE_MEM,
95     },
96     [1] = {
97         .start = IRQ_SDI,
98         .end   = IRQ_SDI,
99         .flags = IORESOURCE_IRQ,
100     }
101
102 };
103
104 static struct platform_device s3c2410_device_sdi = {
105     .name         = "s3c2410-sdi",
106     .id       = -1,
107     .num_resources    = ARRAY_SIZE(s3c2410_sdi_resource),
108     .resource     = s3c2410_sdi_resource,
109 };
110
111
112 static struct platform_device *smdk2410_devices[] __initdata = {
…
118     &s3c2410_device_sdi,
119 };
120
```

② 修改 arch/arm/mach-s3c2440/mach-smdk2440.c。

```
169 /* SDI */
170 static struct resource s3c2440_sdi_resource[] = {
```

```
171     [0] = {
172         .start = S3C2410_PA_SDI,
173         .end   = S3C2410_PA_SDI + S3C24XX_SZ_SDI - 1,
174         .flags = IORESOURCE_MEM,
175     },
176     [1] = {
177         .start = IRQ_SDI,
178         .end   = IRQ_SDI,
179         .flags = IORESOURCE_IRQ,
180     }
181
182 };
183
184 static struct platform_device s3c2440_device_sdi = {
185     .name          = "s3c2440-sdi",
186     .id        = -1,
187     .num_resources    = ARRAY_SIZE(s3c2440_sdi_resource),
188     .resource      = s3c2440_sdi_resource,
189 };
190
191 static struct platform_device *smdk2440_devices[] __initdata = {
…
197     &s3c2440_device_sdi,
198 };
199
```

③ 修改 drivers/mmc/host/s3cmci.c，指定 nCD 中断。

只要在 s3cmci_def_pdata 结构中修改 gpio_detect 成员即可，将它从 0 改为 S3C2410_GPG8。修改后的代码如下：

```
1102 static struct s3c24xx_mci_pdata s3cmci_def_pdata = {
1103     .gpio_detect    = S3C2410_GPG8, /* by www.100ask.net */
1104     .set_power  = NULL,
1105     .ocr_avail  = MMC_VDD_32_33,
1106 };
```

s3cmci.c 中的函数会将 GPG8 引脚设备为外部中断（即 EINT16）、设置双边沿触发。

现在可以编译内核了，测试方法见 23.2.3 小节。

如果想尝试更新的代码，可以从以下网址下载正在开发的补丁：

http://svn.openmoko.org/developers/nbd/patches/

新代码里增加了很多功能，比如增加了对 SDIO 卡的支持、增加了对其他一些处理器的

支持。本章中，只使用前面下载的补丁，对于这些正在开发的补丁，读者感兴趣的话可以自行使用，方法是类似的。

23.2.3　SD卡驱动程序测试

使用新编译的内核启动系统，可以看到如下信息：

```
s3c2440-sdi s3c2440-sdi: powered down.
s3c2440-sdi s3c2440-sdi: initialisation done.
```

如果接上SD卡，还可以看到类似下面的信息：

```
s3c2440-sdi s3c2440-sdi: running at 0kHz (requested: 0kHz).
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi  s3c2440-sdi:  CMD[TIMEOUT]  #9  op:UNKNOWN(8)  arg:0x000001aa
flags:0x0875 retries:0 Status:nothing to complete
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi  s3c2440-sdi:  CMD[TIMEOUT]  #13  op:UNKNOWN(8)  arg:0x000001aa
flags:0x0875 retries:0 Status:nothing to complete
s3c2440-sdi s3c2440-sdi: running at 196kHz (requested: 195kHz).
s3c2440-sdi s3c2440-sdi: running at 25000kHz (requested: 25000kHz).
s3c2440-sdi s3c2440-sdi: running at 25000kHz (requested: 25000kHz).
mmcblk0: mmc0:b368 SD    1997312KiB
 mmcblk0:<7>mmc0: starting CMD18 arg 00000000 flags 00000035
 p1    // 如果已经分区，则会打印出类似这行的字样
```

这表明已经识别出了SD卡，然后就可以使用fdisk工具来分区，使用mke2fs或mkdosfs来格式化设备了。

如果根文件系统中没有使用mdev机制，在使用之前要先创建设备节点（主设备号可以通过“cat /proc/deivces”命令确定），mmcblk0表示整个SD卡，mmcblk0p1等表示上面的分区。

```
# mknod /dev/mmcblk0    b 179 0
# mknod /dev/mmcblk0p1  b 179 1
# mknod /dev/mmcblk0p2  b 179 2
# mknod /dev/mmcblk0p3  b 179 3
# mknod /dev/mmcblk0p4  b 179 4
```

在SD卡上进行分区、格式化、挂接的方法与硬盘一样，读者可以参考23.1.3节的内容，不再重复。

23.2.4 磁盘分区表

磁盘的分区表示形式有多种风格：BSD/SUN、IRIX/SGI、DOS。在 DOS 风格的分区表中，分区开始地址和大小是以两种不同的方式来存放的：以扇区数的绝对值来描述（占 32 位）和以柱面、磁头、扇区三个一组的形式（占 10+8+6 个位）来描述。前者编号从 0 开始；后者被称为 C/H/S 方式，已经过时，Linux 不使用。

磁盘一个扇区大小为 512 字节，第一个扇区被称为主引导记录（MBR，Master Boot Record）。MBR 中偏移地址 446～509 处存放了分区表，每个表项为 16 字节，可以存放 4 个表项。MBR 中偏移地址 510、511 处的数据为 0x55、0xAA。

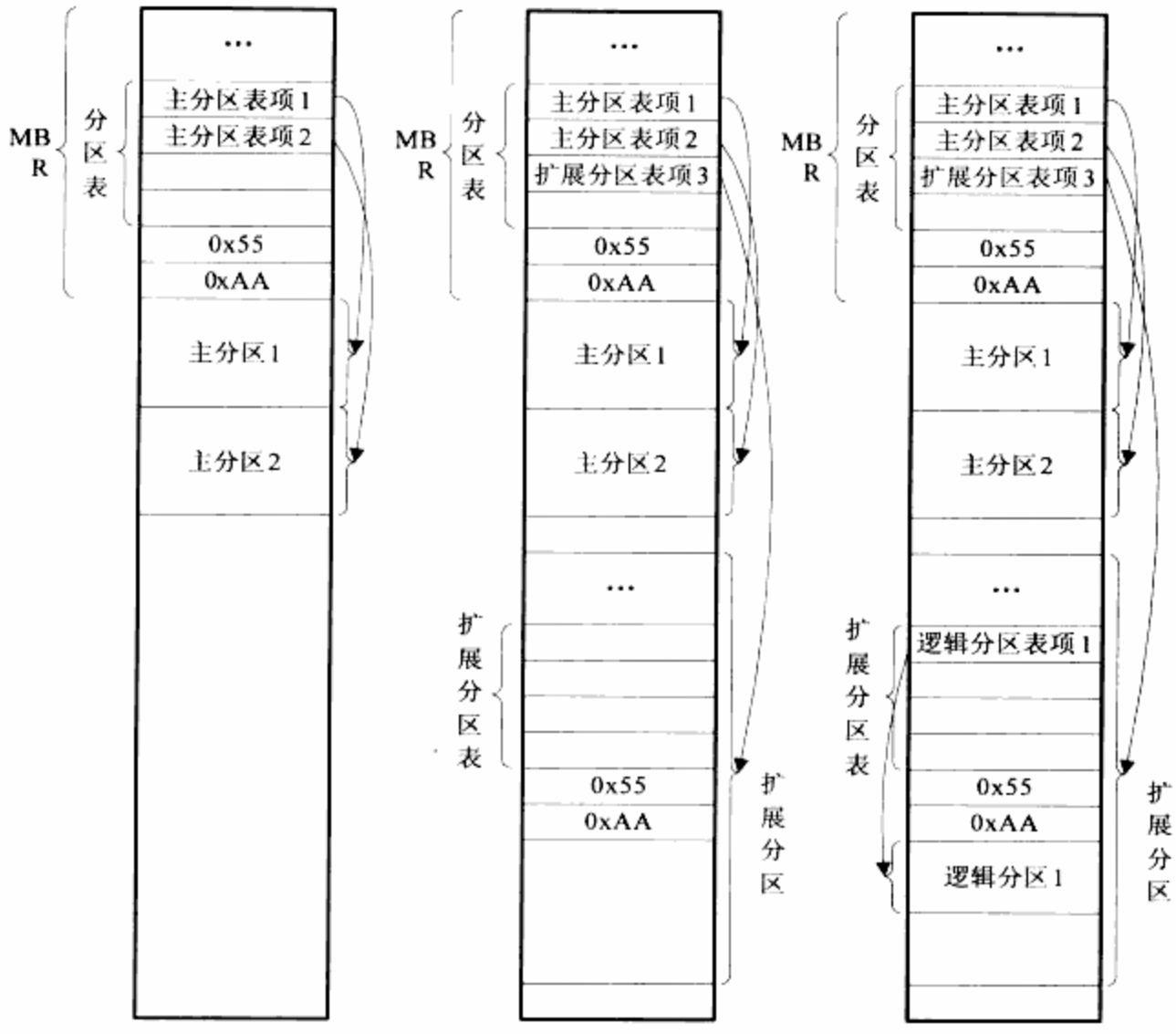
分区表项的数据结构如下（在 include/linux/genhd.h 中定义）：

```
struct partition {
    unsigned char boot_ind;     /* 引导标志。4个分区中同时只能有一个分区是可引导的
                                 * 0x00: 不从该分区引导操作系统
                                 * 0x80: 从该分区引导操作系统
                                 */
    unsigned char head;         /* 起始磁头 */
    unsigned char sector;       /* 起始扇区 */
    unsigned char cyl;          /* 起始柱面 */
    unsigned char sys_ind;      /* 分区类型，比如:
                                 * 0x05: 扩展分区
                                 * 0x06: FAT16
                                 * 0x83: Linux
                                 */
    unsigned char end_head;     /* 结束磁头 */
    unsigned char end_sector;   /* 结束扇区 */
    unsigned char end_cyl;      /* 结束柱面 */
    unsigned int start_sect;    /* 分区起始物理扇区号(从0计数) */
    unsigned int nr_sects;      /* 分区占用的扇区数 */
} __attribute__((packed));
```

其中的 head、sector、cyl、end_head、end_sector、end_cyl 在 Linux 中不再使用，而是使用 start_sect、nr_sects 来定义一个分区的开始扇区和大小。

由于 MBR 中只有 4 个分区表项，所以一个磁盘最多可以有 4 个主分区。如果要划分更多的分区，那么这 4 个表项中可以（只可以）用 1 个来作为扩展分区（表项中 sys_ind 等于 0x05）。当创建一个扩展分区时，扩展分区表也被创建。扩展分区就像一个独立的磁盘驱动器，它有自己的分区表，在它里面又可以进一步划分最多 4 个分区，也可以划分 1 个扩展分区。扩展分区里面进一步划分出来的分区被称为逻辑分区（logical partitions），与主分区（primary partitions）相对，扩展分区的分区表完全包含在扩展分区之内。

下面以图 23.5 来说明。

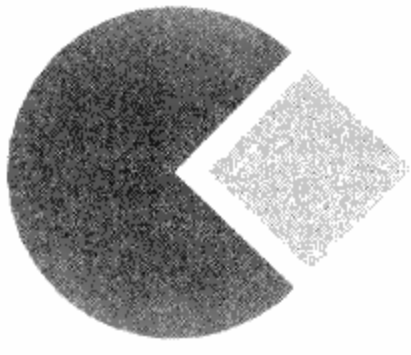


1. 有两个主分区

2. 划分了一个扩展分区后，会在扩展分区的第一个扇区创建分区表，不过此时分区表是空的

3. 划分一个逻辑分区
扩展分区就像一个独立的磁盘，在扩展分区中仍可以划分分区(它们被称为逻辑分区)，也可以继续划分扩展分区。分区表中“分区位置”是在此“新磁盘”中的偏移

图 23.5　分区表和分区示意图



第 24 章　LCD 和 USB 驱动程序移植

本章目标

- 了解 TTY 层下 LCD 和 USB 键盘驱动程序的框架
- 掌握移植 LCD 驱动程序的方法
- 使用 LCD 和 USB 设备

24.1　LCD 驱动程序移植

24.1.1　LCD 和 USB 键盘驱动程序框架

1．框架概述

在第 23 章介绍过控制台、终端的概念，具备人机交互功能的串口可以作为控制台和终端，同样，LCD 和键盘组合起来也可以。

对 LCD 的操作可以像串口一样，通过终端设备层的封装（/dev/tty*设备）来输出内容，也可以通过 frame buffer（/dev/fb*设备）直接在显存上“绘制”图像。

frame buffer 即帧缓冲，是一种独立于硬件的抽象图形设备，它使得应用程序可以通过一组定义良好的接口访问各类图形设备，不需要了解低层硬件细节。从用户的观点来看，frame buffer 设备与/dev 目录下其他设备没有区别，通过/dev/fb*设备文件来访问它（fb0 表示第一个 frame buffer 设备、fb1 表示第二个，…）。

frame buffer 设备提供了一些 ioctl 接口来查询、设置图形设备的属性，比如分辨率、像素位宽等，另外，它属于“普通的”内存设备，类似/dev/mem：可以读（read）、写（write）、移动访问位置（seek）以及将“这块内存”映射（mmap）给用户。不同的是 frame buffer 的内存不是所有的内存，而是显卡专用的内存。应用程序可以直接更改 frame buffer 内存中的数据，效果立刻就能在显示器上看到。

/dev/tty1 等终端设备文件通过显示驱动程序和键盘驱动程序（还有其他输入设备，比如触摸屏）为它们提供输出、输入功能。

TTY 和 frame buffer 驱动程序的框架如图 24.1 所示，输入设备以 USB 键盘为例。

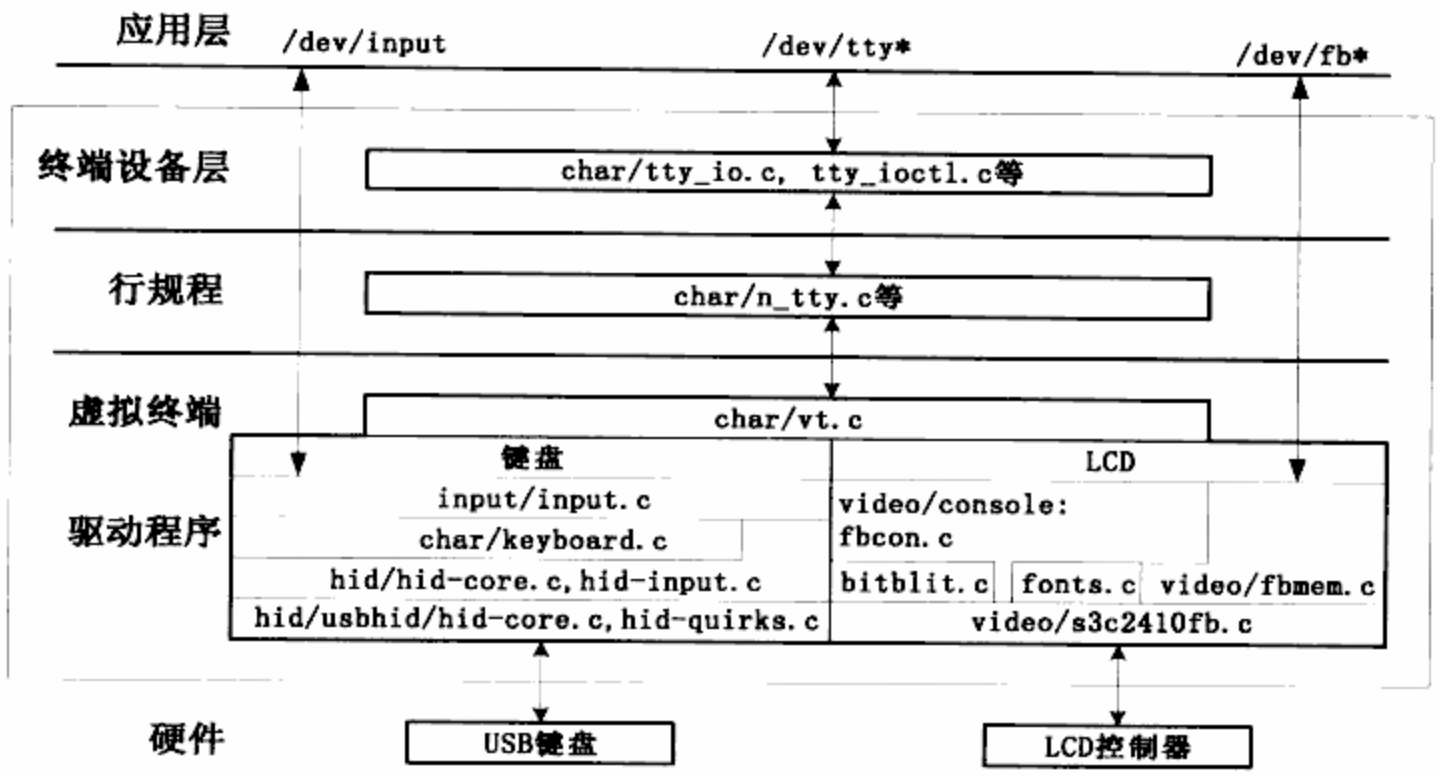


图 24.1　TTY 和 frame buffer 驱动程序的框架（这些文件都是在 drivers/目录下）

drivers/char/vt.c 用来支持显示器/键盘组成的终端设备，之所以被称为“虚拟终端”，是因为可以在一个物理终端设备上运行多个“虚拟终端”（也叫虚拟控制台），比如可以使用第 1 个虚拟终端来显示系统信息，使用第 2 个虚拟终端来运行文本模式程序，而在第 3 个虚拟终端运行图形程序。它们可以同时运行，使用一些组合键可以切换到某个虚拟终端上（通常是 Alt+Fn 键）。

虚拟终端层管理着这些“虚拟终端”，比如为它们分配缓冲区、切换虚拟终端时把它的内容输出到显示器、键盘有输入时把数据填入当前终端的缓冲区中。它向上提供了封装好的接口，向下通过调用显示器/键盘的接口完成输入、输出功能。

驱动程序层分为两类：键盘驱动程序和显示驱动程序。

（1）显示驱动程序。

drivers/console/fbcon.c 文件向上提供了一个很重要的数据结构 fb_con，所有的输出都是通过 fb_con 中的成员函数来实现的，bitblit.c、fonts.c 也都处于 drivers/console/目录下，它们和 drivers/video/fbmem.c 一起，实现 fb_con 结构中的函数。另外，fbmem.c 是 frame buffer 驱动程序，它向应用层提供/dev/fb*设备的访问接口，应用程序可以通过它绘制图形。

drivers/video/s3c2410fb.c 文件是架构相关的代码，它实现 LCD 控制器的初始化、向 fbmem.c 注册 frame buffer 设备，并提供一些与架构相关的函数，比如设置分辨率、像素位宽等需要操作寄存器的函数。

注意　图 24.1 是以 LCD 为例的，与 fbcon.c 相同地位的文件还有 vgacon.c 等文件，vgacon.c 用于一般 VGA 显卡。也有很多与 s3c2410fb.c 类似的文件，比如：sm501fb.c、vesafb.c 等。

（2）键盘驱动程序。

drivers/input/input.c 表示“输入设备”，有键盘、鼠标等。drivers/keyboard.c 是键盘驱动程序的封装，在它的下面，可以是一般的键盘，也可以是符合 HID 规范的键盘。HID 是英文“Human Interface Device”的缩写，它通常指 USB-HID 规范，但是也有其他类型的遵循 HID 规范的设备（比如蓝牙键盘、蓝牙鼠标）。所以 drivers/hid-core.c、hid-input.c 两个文件将 HID

规范的共性提炼出来，它们的下面是各类具体实现，比如 USB 的 drivers/hid/hidusb/hid-core.c、hid-quirks.c 等。

2. 操作实例

下面以几个操作的函数调用过程来理解 TTY 和 frame buffer 驱动程序的层次结构。注意：这只是为了在阅读内核源码时，给读者提供一些函数间调用的脉落关系。刚接触某类驱动时，了解各函数、结构间的调用关系是一件困难的事情。如果脱离源代码，这些内容几乎没什么用处。

每行代码后面的括号表示当前函数在哪个文件中实现，它们大多数是在 drivers/目录下，省略“drivers”字样。

（1）注册 frame buffer 设备时，显示 LOGO 的过程。

```
s3c2410fb_probe (video/s3c2410fb.c) ->
    register_framebuffer (fbmem.c) ->
        fb_info->node = i; // registered_fb[i]为空项，本例中 i=0
        registered_fb[i] = fb_info;
        fb_notifier_call_chain (fb_notify.c) // 它会调用 fbcon_event_notify
(fbcon.c 中) ->
            fbcon_fb_registered (video/console/fbcon.c)
                info_idx = idx//即 info->node，值为上面的“i”
                fbcon_takeover(1) (video/console/fbcon.c) ->
                    con2fb_map[i] = info_idx; // i=0, info_idx=0
                    take_over_console (char/vt.c) ->
                        register_con_driver (char/vt.c) ->
                            csw->con_startup(…) // 即 fbcon_startup(video/console/
fbcon.c) ->
                                info = registered_fb[info_idx];
                                info->fbops->fb_open(…) (video/s3c2410fb.c)

                        bind_con_driver (char/vt.c) ->
                            visual_init (char/vt.c) ->
                                vc->vc_sw->con_init // 即 fbcon_init ->
                                    fbcon_init (video/console/fbcon.c) ->
                                        // 以下“准备 LOGO”
                                        fbcon_prepare_logo (video/console/fbcon.c) ->
                                            fb_prepare_logo (video/fbmem.c) ->
                                                fb_logo.logo = fb_find_logo(depth);
// logo.c
                                打印: Console: switching to colour frame buffer device 30x40
                                update_screen(vc); (include/linux/vt_kern.h) ->
```

```
                                redraw_screen (char/vt.c) ->
                                    vc->vc_sw->con_switch(vc); // 即 fbcon_switch
(fbcon.c) ->
                                        fb_show_logo (video/fbmem.c) // 显示 LOGO
```

（2）对/dev/tty*调用 write 函数时的过程。

```
tty_write (char/tty_io.c) ->
    ld = tty_ldisc_ref_wait(tty) // 它就是 char/n_tty.c 中的 tty_ldisc_N_TTY
    do_tty_write(ld->write, tty, file, buf, count) (char/tty_io.c) ->
        write_chan (就是上面的 ld->write, char/n_tty.c 中 tty_ldisc_N_TTY 的成员函数) ->
            tty->driver->write (即 con_write, char/vt.c) ->
                do_con_write (char/vt.c) ->
                    vc->vc_sw->con_putcs (即 fbcon_putcs, video/console/fbcon.c) ->
                        ops->putcs (即 bit_putcs, video/console/bitblit.c) ->
                            dst = fb_get_buffer_offset (video/fbmem.c) // 获取要
写入的显存位置
                            bit_putcs_aligned/bit_putcs_unaligned
(video/console/bitblit.c)
                                src = vc->vc_font.data + (scr_readw(s++)&
                                    charmask)*cellsize;              // 获得字符的点阵
                                ..., __fb_pad_aligned_buffer (fb.h)   // 将点阵写入显存
```

在使用/dev/tty*作为控制台的 shell 中，运行某个程序时，如果里面有“printf("hello, world! ")”字样的语句，它会调用到内核的 tty_write 函数。

然后会调用行规程的 write_chan 函数，它又会调用“tty->driver->write”，对于串口，它是 drivers/serial/serial_core.c 中的 uart_write 函数，它直接输出 ASCII 字符；对于显示器，它是 drivers/char/vt.c 中的 con_write 函数，它更复杂。在 LCD 显示器上显示字符时，先要根据这些字符得到它们的点阵，然后再将它们“画出来”。

drivers/video/console/fbcon.c 中的 fbcon_putcs 函数通过 drivers/video/console/bitblit.c、drivers/video/fbmem.c 提供的一些函数来获得点阵、写到显存上去。其中的“vc->vc_font.data”指向某个字库，以字符为索引即可找到它的点阵。在 drivers/video/console/fonts.c 文件中定义了一个 fonts 数组，每个表项是一个字库，比如 font_vga_8x8、font_vga_8x16 等。在 deviers/video/fbcon.c 中初始化 frame buffer 控制台时，会把 vc->vc_font.data 指向某个字库。

（3）USB 键盘被按下时的函数调用过程。

与串口相似，键盘的读取以中断来驱动。以 USB 键盘为例，调用过程如下：

```
hid_irq_in (hid/usbhid/hid-core.c) ->
    hid_input_report (hid/hid-core.c) ->
        hid_input_field (hid/hid-core.c) ->
            hid_process_event (hid/hid-core.c) ->
                hidinput_hid_event (hid/hid-input.c) ->
```

```
                input_event (input/input.c) ->
                    dev->event(…)
                    handle->handler->event, 即 kbd_event(char/keyboard.c) ->
                        kbd_rawcode/kbd_keycode (char/keyboard.c) ->
                            put_queue(vc, data) (char/keyboard.c) -> //数据放
入终端缓冲区
                                tty_insert_flip_char  (include/linux/tty_flip.h)
// 放数据
                                con_schedule_flip (kbd_kern.h) // 唤醒等待数据的进程
```

hid_irq_in 是 USB 中断传输方式的中断处理函数，当键盘被按下时，它导致后续的一系列函数被调用。与图 24.1 对应，它从低层的 drviers/hid/usbhid/hid-core.c 一直向上调用到 drivers/input/input.c 中的 input_event 函数，接着 input_event 函数根据调用 drivers/char/keyboard.c 注册的处理函数将数据放入虚拟终端设备的缓冲区中，然后唤醒等待数据的进程。

24.1.2 S3C2410/S3C2440 LCD 控制器驱动程序移植

从图 24.1 可以知道，架构相关的代码为 drivers/video/s3c2410fb.c，移植的思想是一样的：先确定 LCD 控制器所用的资源，然后把它们加入平台设备结构，最后修改代码使这些资源可用。

硬件连线图如图 24.2 所示。

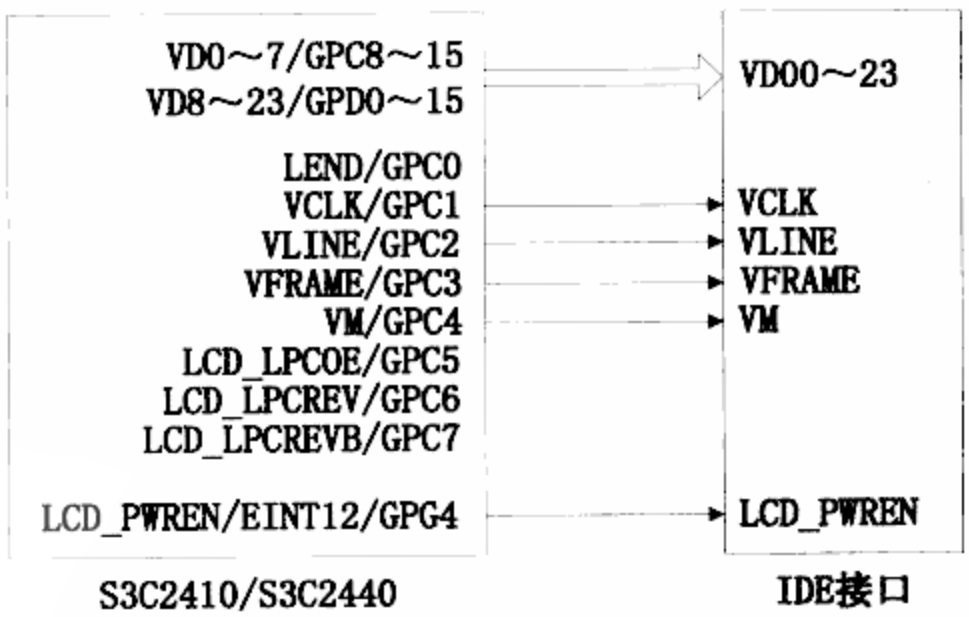


图 24.2 TFT LCD 连线图

1. 平台设备结构

LCD 控制器的平台设备在 arch/arm/plat-s3c24xx/devs.c 中定义，它所用的资源都是固定的，不需要任何改动。它的平台设备结构定义如下：

```
147 /* LCD Controller */
148
```

```
149 static struct resource s3c_lcd_resource[] = {
150     [0] = {
151         .start = S3C24XX_PA_LCD,
152         .end   = S3C24XX_PA_LCD + S3C24XX_SZ_LCD - 1,
153         .flags = IORESOURCE_MEM,
154     },
155     [1] = {
156         .start = IRQ_LCD,
157         .end   = IRQ_LCD,
158         .flags = IORESOURCE_IRQ,
159     }
160
161 };
162
163 static u64 s3c_device_lcd_dmamask = 0xffffffffUL;
164
165 struct platform_device s3c_device_lcd = {
166     .name          = "s3c2410-lcd",
167     .id        = -1,
168     .num_resources    = ARRAY_SIZE(s3c_lcd_resource),
169     .resource     = s3c_lcd_resource,
170     .dev              = {
171         .dma_mask       = &s3c_device_lcd_dmamask,
172         .coherent_dma_mask  = 0xffffffffUL
173     }
174 };
175
```

第165行定义的s3c_device_lcd结构，都已经加入了本书所用的S3C2410、S3C2440开发板的设备列表中了。

（1）arch/arm/mach-s3c2410/mach-smdk2410.c。

```
static struct platform_device *smdk2410_devices[] __initdata = {
...
    &s3c_device_lcd,
...
};
```

（2）arch/arm/mach-s3c2440/mach-smdk2440.c。

```
static struct platform_device *smdk2440_devices[] __initdata = {
...
```

```
    &s3c_device_lcd,
...
};
```

而 LCD 控制器驱动程序 drivers/video/s3c2410fb.c 的入口函数为：

```
1010 static struct platform_driver s3c2410fb_driver = {
1011     .probe      = s3c2410fb_probe,
1012     .remove     = s3c2410fb_remove,
1013     .suspend    = s3c2410fb_suspend,
1014     .resume     = s3c2410fb_resume,
1015     .driver     = {
1016         .name   = "s3c2410-lcd",
1017         .owner  = THIS_MODULE,
1018     },
1019 };
1020
1021 int __devinit s3c2410fb_init(void)
1022 {
1023     return platform_driver_register(&s3c2410fb_driver);
1024 }
1025
```

平台设备 s3c_device_lcd 和平台驱动 s3c2410fb_driver 的名字都是“s3c2410-lcd”，所以注册了 s3c2410fb_driver 之后，它的 s3c2410fb_probe 函数将被调用来设置 LCD 控制器。

读者可以发现，图 24.2 中各连线对应的 GPIO 引脚并没有在平台设备中体现，在什么地址设置它们呢？这需要阅读 s3c2410fb_probe 函数。

2. 底层驱动代码分析及修改

s3c2410fb_probe 函数完成初始化 LCD 控制器、注册中断处理函数、注册 frame buffer 设备等工作，它的流程图如图 24.3 所示。

这个函数中，与单板相关的就是其中的 mach_info 结构。它是平台设备 s3c_device_lcd 结构中的 dev.platform_data 成员，读者可以查看 s3c2410fb_init_registers 函数来了解它的功能。但是在前面看到的 s3c_device_lcd 结构中，并没有指定这个成员。它在其他函数中设置：对于 S3C2440，单板初始化函数 smdk2440_machine_init 调用 s3c24xx_fb_set_platdata 函数来设置；对于 S3C2410，没有设置。

smdk2440_machine_init 函数在 arch/arm/mach-s3c2440/mach-smdk2440.c 中，如下所示：

```
203 static void __init smdk2440_machine_init(void)
204 {
205     s3c24xx_fb_set_platdata(&smdk2440_lcd_cfg);
...
```

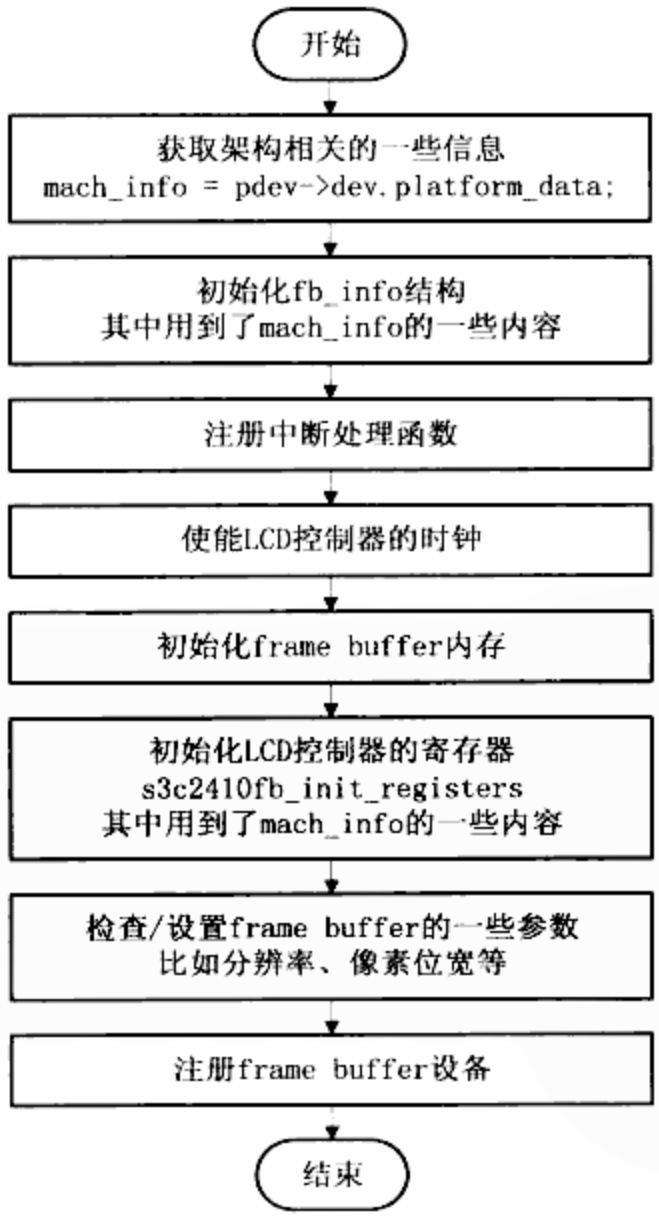


图 24.3　frame buffer 驱动程序初始化函数 s3c2410fb_probe 流程

smdk2440_lcd_cfg 结构表示 LCD 控制器的一些配置，比如分辨率、时间特性等，在后面会详细描述。

s3c24xx_fb_set_platdata 函数在 arch/arm/plat-s3c24xx/devs.c 中，它直接将参数 smdk2440_ lcd_cfg 赋给设置平台设备 s3c_device_lcd 结构中的 dev.platform_data 成员。代码如下：

```
178 void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)
179 {
...
184         memcpy(npd, pd, sizeof(*npd));
185         s3c_device_lcd.dev.platform_data = npd;
...
189 }
```

所以，对于 S3C2440，需要修改 smdk2440_lcd_cfg 结构；对于 S3C2410，仿照 S3C2410 增加一个 smdk2410_lcd_cfg 结构，并调用 s3c24xx_fb_set_platdata 函数来设置它。

smdk2440_lcd_cfg 是 s3c2410fb_mach_info 结构类型，这个类型在 include/asm-arm/arch-s3c2410/fb.h 文件中定义，下面分析它的各个成员的意义。

```
31 struct s3c2410fb_mach_info {
32     unsigned char   fixed_syncs;    /* do not update sync/border */
33
34     /* LCD types */
35     int     type;
36
37     /* Screen size */
38     int     width;
39     int     height;
40
41     /* Screen info */
42     struct s3c2410fb_val xres;
43     struct s3c2410fb_val yres;
44     struct s3c2410fb_val bpp;
45
46     /* lcd configuration registers */
47     struct s3c2410fb_hw  regs;
48
49     /* GPIOs */
50
51     unsigned long   gpcup;
52     unsigned long   gpcup_mask;
53     unsigned long   gpccon;
54     unsigned long   gpccon_mask;
55     unsigned long   gpdup;
56     unsigned long   gpdup_mask;
57     unsigned long   gpdcon;
58     unsigned long   gpdcon_mask;
59
60     /* lpc3600 control register */
61     unsigned long   lpcsel;
62 };
```

第 32 行的 fixed_syncs 被设为 1 时表示“固定的”时间参数和边框大小，这意味着用户程序无法调整分辨率等参数，因为底层驱动程序不修改时间参数和边框大小。从 s3c2410fb.c 中的相关代码来看，它就是不再重新设置 LCDCON2/3/4 寄存器中的相关位。

第 35 行的 type 表示 LCD 的类型，从 LCDCON1 寄存器位[6:5]可以知道它有 4 种取值，如下所示：

```
00 = 4-bit dual scan display mode (STN)
01 = 4-bit single scan display mode (STN)
10 = 8-bit single scan display mode (STN)
11 = TFT LCD panel
```

第38～39行的width、height用来设置图像的宽度和高度，它们取xres、yres的默认值。

第42～44行中s3c2410fb_val结构的定义如下，xres、yres和bpp分别表示图像宽度、高度和像素位宽的最小、最大、默认值。

```
struct s3c2410fb_val {
    unsigned int    defval;
    unsigned int    min;
    unsigned int    max;
};
```

第47行的“struct s3c2410fb_hw　regs”表示LCDCON1～LCDCON5共5个LCD控制器的控制寄存器。它们用来设置LCD类型、像素数据的格式、

第51～58行用来设置GPC、GPD两组GPIO引脚，比如gpcup和gpccon_mask两个成员被用来设置GPCUP寄存器：gpcup表示新值，gpccon_mask表示要设置的位。

第61行表示LPCSEL寄存器，它用来支持SEC公司（Samsung Electronics Company）生产的TFT LCD（称为SEC TFT LCDs）。对于一般LCD，不用设置这个寄存器。

本开发板使用240x320，16bpp的TFT LCD，内核自带的smdk2440_lcd_cfg结构并不适用于这个开发板，并且它的设置有一些错误：没有指定GPIO寄存器的值，“type”设置错了。原来的值如下：

```
static struct s3c2410fb_mach_info smdk2440_lcd_cfg __initdata = {
...
#if 0
        /* currently setup by downloader */
        .gpccon         = 0xaa940659,
        .gpccon_mask    = 0xffffffff,
        .gpcup          = 0x0000ffff,
        .gpcup_mask     = 0xffffffff,
        .gpdcon         = 0xaa84aaa0,
        .gpdcon_mask    = 0xffffffff,
        .gpdup          = 0x0000faff,
        .gpdup_mask     = 0xffffffff,
#endif
...
        .type       = S3C2410_LCDCON1_TFT16BPP,
```

```
...
};
```

它把 GPIO 的值屏蔽掉了，原因是“currently setup by downloader”，这也许是这个驱动的开发者在调试时，另外使用某种下载器来设置 GPIO。

上面的“type”被设为“S3C2410_LCDCON1_TFT16BPP”，这是错误的，“type”表示“类型”，而“S3C2410_LCDCON1_TFT16BPP”表示“TFT”类型下数据的格式。应该设为以下 4 个值之一：

```
#define S3C2410_LCDCON1_DSCAN4    (0<<5)
#define S3C2410_LCDCON1_STN4      (1<<5)
#define S3C2410_LCDCON1_STN8      (2<<5)
#define S3C2410_LCDCON1_TFT       (3<<5)
```

下面修改代码。

（1）对于 S3C2440 单板。

修改 smdk2440_lcd_cfg 结构，它在 arch/arm/mach-s3c2440/mach-smdk2440.c 文件中，修改后的代码如下：

```
104 /* LCD driver info */
105
106 static struct s3c2410fb_mach_info smdk2440_lcd_cfg __initdata = {
107     .regs   = {
108         .lcdcon1 =  S3C2410_LCDCON1_TFT16BPP | \
109                 S3C2410_LCDCON1_TFT | \
110                 S3C2410_LCDCON1_CLKVAL(0x04),
111
112         .lcdcon2 =  S3C2410_LCDCON2_VBPD(1) | \
113                 S3C2410_LCDCON2_LINEVAL(319) | \
114                 S3C2410_LCDCON2_VFPD(5) | \
115                 S3C2410_LCDCON2_VSPW(1),
116
117         .lcdcon3 =  S3C2410_LCDCON3_HBPD(36) | \
118                 S3C2410_LCDCON3_HOZVAL(239) | \
119                 S3C2410_LCDCON3_HFPD(19),
120
121         .lcdcon4 =  S3C2410_LCDCON4_MVAL(13) | \
122                 S3C2410_LCDCON4_HSPW(5),
123
124         .lcdcon5 =  S3C2410_LCDCON5_FRM565 |
125                 S3C2410_LCDCON5_INVVLINE |
126                 S3C2410_LCDCON5_INVVFRAME |
```

```
127                 S3C2410_LCDCON5_PWREN |
128                 S3C2410_LCDCON5_HWSWP,
129     },
130
131     .gpccon      = 0xaaaaaaaa,
132     .gpccon_mask =  0xffffffff,
133     .gpcup       = 0xffffffff,
134     .gpcup_mask  =  0xffffffff,
135
136     .gpdcon      = 0xaaaaaaaa,
137     .gpdcon_mask =  0xffffffff,
138     .gpdup       = 0xffffffff,
139     .gpdup_mask  =  0xffffffff,
140
141     .fixed_syncs =  1,
142     .type        =  S3C2410_LCDCON1_TFT,
143     .width       =  240,
144     .height      =  320,
145
146     .xres   = {
147         .min    =   240,
148         .max    =   240,
149         .defval =   240,
150     },
151
152     .yres   = {
153         .max    =   320,
154         .min    =   320,
155         .defval =   320,
156     },
157
158     .bpp    = {
159         .min    =   16,
160         .max    =   16,
161         .defval =   16,
162     },
163 };
164
```

在设置 GPIO 引脚时，我们把 GPC、GPD 的所有引脚都设置用于 LCD，虽然 16bpp 的

TFT LCD 没有完全用到这些引脚，但是在本书所用开发板中，这些引脚并没有另外用做其他用途。

（2）对于 S3C2410 单板。

仿照 arch/arm/mach-s3c2440/mach-smdk2440.c 来修改 arch/arm/mach-s3c2410/mach-smdk2410.c。

① 增加 smdk2410_lcd_cfg 结构。

直接把 smdk2440_lcd_cfg 的内容搬到 mach-smdk2410.c 中，改名为 smdk2410_lcd_cfg 即可。

② 使用 smdk2410_lcd_cfg 结构。

在 S3C2410 单板初始化函数 smdk2410_init 中，调用 s3c24xx_fb_set_platdata 函数。除第 92～149 行增加的 smdk2410_lcd_cfg 结构外，还要增加第 56 行、第 193 行，如下所示：

```
56 #include <asm/arch/fb.h>
...
90 /* LCD driver info, add by www.100ask.net */
91
92 static struct s3c2410fb_mach_info smdk2410_lcd_cfg __initdata = {
... /* 与 smdk2440_lcd_cfg 相同 */
149 };
...
191 static void __init smdk2410_init(void)
192 {
193     s3c24xx_fb_set_platdata(&smdk2410_lcd_cfg); // add by www.100ask.net
...
```

3. 配置内核以使用 LCD

对 LCD 的配置有两方面，一是 frame buffer 方面的配置，二是控制台方面的配置。

配置内容如下：

```
Device Drivers  --->
    Graphics support  --->
        <*> Support for frame buffer devices          // 支持 frame buffer
        <*> S3C2410 LCD framebuffer support           // 支持 S3C24xx
            Console display driver support  --->
            <*> Framebuffer Console support           // 支持 frame buffer 控制台
            [ ] Select compiled-in fonts              // 选择字库，默认为 VGA 8x8、
VGA 8x16 字库
        [*] Bootup logo  --->                         // 启动时显示 LOGO
            [*]   Standard 224-color Linux logo       // 选择 LOGO 图像，有单色、
16 色、244 色
```

对 frame buffer 控制台的使用，可以参考内核文档 Documentation/fb/fbcon.txt，它讲述了

如何通过命令行参数控制 frame buffer 控制台，比如选择字库、旋转图像等（这需要配置内核以增加相应功能）。

下面介绍一些常规用法。

（1）通过 LCD 显示内核信息。

以前使用串口作为控制台（指打印内核信息）时，命令行参数为“console=ttySAC0”，现在可以多加一项，比如：“console=ttySAC0 console=tty1”。

注意　tty1 表示第一个虚拟终端，而 tty2 表示第二个虚拟终端，而 tty0 表示当前的虚拟终端。

（2）操作/dev/tty1 设备输出字符：如果使用 mdev 机制，这个步骤可以省略。

首先如下创建设备文件，在单板步执行以下命令：

```
# mknod /dev/tty0 c 4 0
# mknod /dev/tty1 c 4 1
# mknod /dev/tty2 c 4 2
# mknod /dev/tty3 c 4 3
# mknod /dev/tty4 c 4 4
# mknod /dev/tty5 c 4 5
# mknod /dev/tty6 c 4 6
```

在串口控制台，使用“echo hello > /dev/tty0”命令可以在 LCD 上显示“hello”字符串。

（3）操作/dev/fb0 绘制图像。

首先如下创建设备文件（如果使用 mdev 机制，这个步骤可以省略）：

```
# mknod /dev/fb0 c 29 0
```

然后在 work/drivers_and_test/fb_test/目录下执行“make”命令编译 frame buffer 测试程序 fb_test，把它放到单板/usr/bin 目录下。执行“fb_test /dev/fb0”即可在 LCD 上看到很多同心圆，并且在控制台上可以看到如下字样，它打印出 frame buffer 的属性：

```
# fb /dev/fb0
fb_var_screeninfo values:
    xres:          240
    yres:          320
    xres_virtual:  240
    yres_virtual:  320
    xoffset:       0
    yoffset:       0
    bits_per_pixel: 16
    grayscale:     0
    red.offset:    11
    red.length:    5
    red.msb_right: 0
```

```
green.offset:   5
green.length:   6
green.msb_right:0
blue.offset:    0
blue.length:    5
blue.msb_right: 0
transp.offset:  0
transp.length:  0
transp.msb_right:0
nonstd:         0
activate:       0
height:         320
width:          240
accel_flags:    0
pixclock:       0
left_margin:    20
right_margin:   37
upper_margin:   2
lower_margin:   6
hsync_len:      6
vsync_len:      2
sync:           0
vmode:          0
240x320, 16bpp
```

24.2 USB 驱动程序移植

24.2.1 USB 驱动程序概述

USB（Universal Serial Bus）即“通用串行外部总线”，在各种场所已经大量使用。它接口简单（只有 5V 电源和地、两根数据线 D+和 D-），可以外接硬盘、键盘、鼠标、打印机等多种设备。要使用尽可能少的接口支持尽可能多的外设，USB 是一个好的选择，在嵌入式设备中尤其如此。

USB 总线规范有 1.1 版和 2.0 版。USB 1.1 支持两种传输速率：低速（Low Speed）1.5Mbit/s、全速（Full Speed）12Mbit/s，对于鼠标、键盘、CD-ROM 等设备，这样的速率是足够了。但是在访问硬盘、摄像机时，还是显得很慢。为此，USB 2.0 提供了一种更高的传输速率：高速（High Speed），它可以达到 480Mbit/s。USB 2.0 向下兼容 USB 1.1，可以将遵循 USB 1.1 规范的设备连接到 USB 2.0 控制器上，也可以把 USB 2.0 的设备连接到 USB 1.1 控制器上。

USB总线的硬件拓扑结构如图24.4所示。

USB主机控制器（USB Host Controller）通过根集线器（Root Hub）与其他USB设备相连。集线器也属于USB设备，通过它可以在一个USB接口上扩展出多个接口。除根集线器外，最多可以层叠（一个接着一个）5个集线器，每条USB电缆的最大长度是5m，所以USB总线的最大距离为30m（接上5个集线器）。一条USB总线上可以外接127个设备，包括根集线器和其他集线器。整个结构图是一个星状结构，一条USB总线上所有设备共享一条通往主机的数据通道，同一时刻只能有一个设备与主机通信。

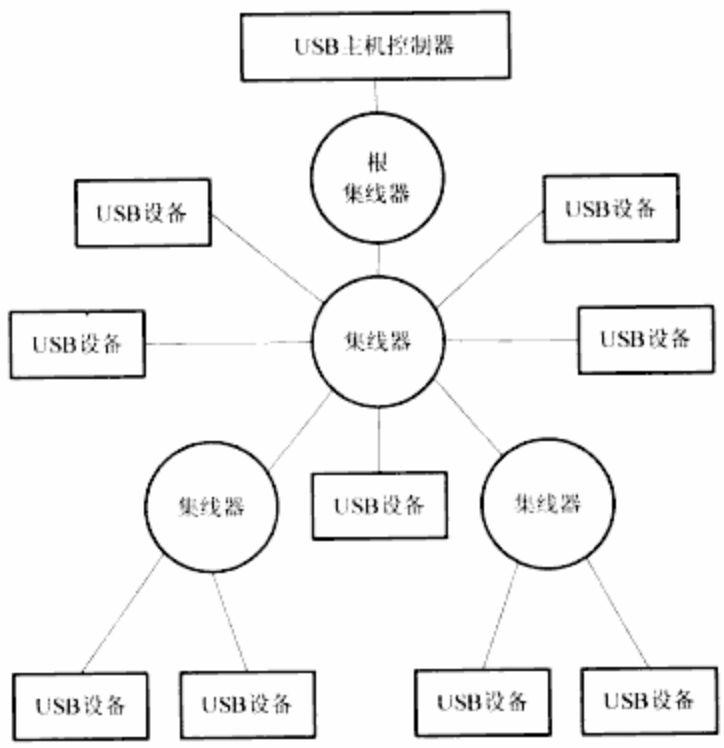


图24.4　USB总线硬件拓扑图

通过USB主机控制器来管理外接的USB设备，USB主机控制器共分3种：UHCI、OHCI和EHCI，其中的“HCI”表示“Host Controller Interface”。UCHI、OHCI属于USB 1.1的主机控制器规范，而EHCI是USB 2.0的主机控制器规范。UHCI（即Universal HCI），它是由Intel公司制订的标准，它的硬件所做的事情比较少，这使得软件比较复杂。与之相对的是OHCI（即Open HCI），它由Compaq、Microsoft和National Semiconductor联合制定，在硬件方面它具备更多的智能，使得软件相对简单。

> **注意**　这些差别只存在于底层的USB主机控制器的驱动程序，对它之上的软件没有影响。USB 2.0的主机控制器规范只有EHCI（即Enhanced HCI）一种。

在配置Linux内核的时候，经常可以看到“HCD”字样，它表示“Host Controller Drivers”，即主机控制器驱动程序。比如有uhci-hcd、ohci-hcd和ehci-hcd等驱动模块。

USB驱动程序分为两类：USB主机控制器驱动程序（Host Controller Drivers）、USB设备驱动程序（USB device drivers）。它们在内核中的层次如图24.5所示。

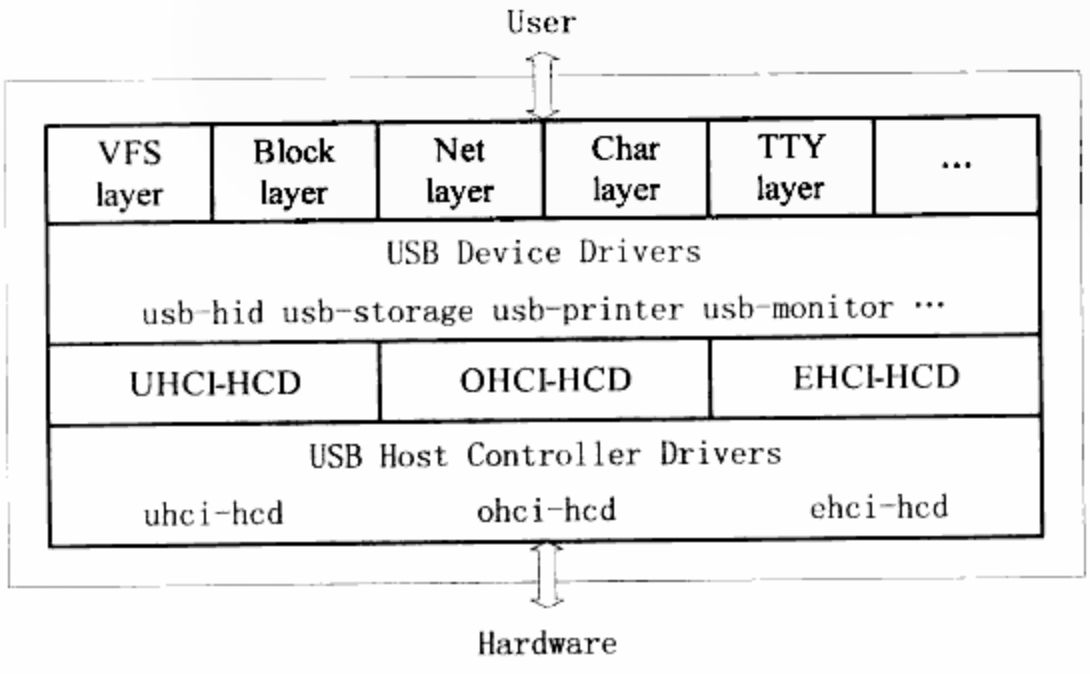


图24.5　USB驱动程序层次结构

USB主机控制器驱动程序提供访问USB设备的接口，它只是一个“数据通道”，至于这

些数据有什么作用，这要靠上层的 USB 设备驱动程序来解释。USB 设备驱动程序使用下层驱动提供的接口来访问 USB 设备，不需要关心传输的具体细节。

24.2.2 配置内核支持 USB 键盘、USB 鼠标和 USB 硬盘

S3C2410/S3C2440 的 USB 控制器有如下特性。

- 符合 OHCI 1.0 规范。
- 支持 USB 1.1 版本。
- 有两个插口。
- 支持低速设备和全速设备。

Linux 内核中对 OHCI 主机控制器支持完善，并有多种 USB 设备驱动程序。Linux 2.6.22.6 也已经支持 S3C2410/S3C2440 的 USB 控制器，只不过第二个插口上电后默认为 USB Device 插口，如果要将它改为 USB Host 插口（比如没有 USB 集线器，却需要同时接入 USB 键盘、USB 鼠标时），只要设置 MISCCR 寄存器的位 3 即可，所有的修改都在文件 drivers/usb/host/ohci-s3c2410.c 中完成，代码如下，其中的第 27、351、352 行是新加的。

```
27 #include <asm/arch/regs-gpio.h>
...
345 static int usb_hcd_s3c2410_probe (const struct hc_driver *driver,
346                   struct platform_device *dev)
347 {
348     struct usb_hcd *hcd = NULL;
349     int retval;
350
351     /* 2 host port */
352     writel(readl(S3C2410_MISCCR)|S3C2410_MISCCR_USBHOST, S3C2410_MISCCR);
353
354     s3c2410_usb_set_power(dev->dev.platform_data, 1, 1);
……
```

现在只需要配置内核启用它们：

```
Device Drivers  --->
    SCSI device support  --->
        <*> SCSI device support          // 要支持USB磁盘，这项要选上
        [*] legacy /proc/scsi/ support // 在/proc/scsi目录下提供一些信息
        <*> SCSI disk support            // SCSI硬盘，要支持U盘等，这项要先上

    USB support  --->
        <*> Support for Host-side USB    // USB主机控制器
        [*]   USB device filesystem      // 在/proc文件系统中提供一些信息，调试用
        <*>   OHCI HCD support           // OHCI主机控制器驱动程序
```

```
        <*> USB Mass Storage support      // USB存储设备

    HID Devices  --->
        <*> USB Human Interface Device (full HID) support    // USB 键盘、USB
鼠标等HID设备
        [*]   /dev/hiddev raw HID device support             // 以原始(raw)的
方式访问HID设备
```

USB控制器的时钟是在U-Boot中设置的(board/100ask24x0/100ask24x0.c中的board_init函数)，UCLK必须设为48MHz。如果读者使用其他bootloader，需要注意这点。

24.2.3 USB设备的使用

连接USB设备时需要注意：S3C2410/S3C2440既可以作为USB主机，也可以作为USB设备。作为USB主机时对外提供两个接口，对应板上叠起来的两个USB接口，下面的称为HOST1，上面的称为HOST2；作为USB设备时，对外也提供一个接口，对应板上的USB_DEVICE接口。

HOST2和USB_DEVICE在S3C2410/S3C2440上的引脚是复用的。要在开发板上使用两个USB设备时，除HOST1外，可以设置跳线使用HOST2；要使用更多的USB设备，必须通过USB集线器来连接。跳线方法请参考板上的标志。

1. 使用LCD和USB键盘作为终端

现有的内核已经支持LCD和USB键盘，可以使用它们来作为控制台、终端了。前面说过，在命令行参数中增加“console=tty1”就可以在LCD上显示内核信息，不过要想使用它们来登录系统，需要修改/etc/inittab文件，增加以下6行：

```
tty1::askfirst:-/bin/sh
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh
tty5::askfirst:-/bin/sh
tty6::askfirst:-/bin/sh
```

它们在6个虚拟终端上启动shell程序，接上USB键盘和LCD后，可以看到如下字样的提示信息：

```
Please press Enter to activate this console.
```

在键盘上按回车键，就可以像在串口终端上一样使用USB键盘、LCD来控制系统了。

按“Alt+F2”将会转换到第二个虚拟终端，将会再次看到上面的提示信息，按“Alt+Fn”可以切换到第*n*个虚拟终端。

执行“echo hello > /dev/tty0”将在当前终端上显示“hello”字符，即LCD上立刻就能看到。但是执行“echo hello > /dev/tty6”的话，需要按“Alt+F6”切换到第6个虚拟终端上才

能看到。

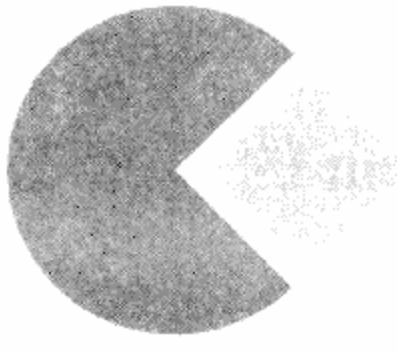
2. 使用 U 盘

首先在开发板上创建如下设备文件（如果使用 mdev 机制，这个步骤可以省略）。

```
# mknod dev/sda  b 8 0
# mknod dev/sda1 b 8 1
# mknod dev/sda2 b 8 2
# mknod dev/sda3 b 8 3
# mknod dev/sda4 b 8 4
```

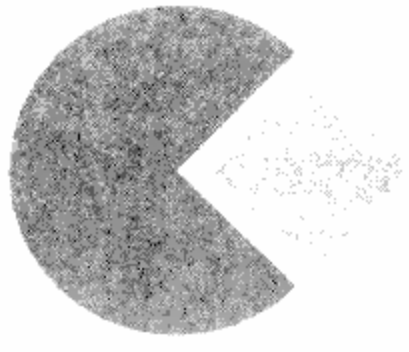
接 U 盘后，即可像前面使用硬盘、SD 卡一样来使用 U 盘了。比如：

```
# fdisk   /dev/sda                    // 进入菜单，对 U 盘进行分区，修改分了一个主分区
/dev/sda1
# mkdosfs -F 32 /dev/sda1             // 格式化为 FAT32 文件系统
# mount   /dev/sda1  /mnt             // 挂接
```



第5篇　嵌入式Linux系统应用开发篇

- 基于Qtopia的GUI开发
- 基于X的GUI开发
- Linux应用程序调试技术



第 25 章　基于 Qtopia 的 GUI 开发

本章目标

- 了解几种嵌入式 GUI 的特点
- 掌握 Qtopia 的移植
- 初步掌握集成开发工具 Kdevelop 的使用
- 学习简单 Qtopia 程序的编写方法
- 掌握在 x86 主机上模拟、调试嵌入式 GUI 程序

25.1　嵌入式 GUI 介绍

25.1.1　Linux 桌面 GUI 系统的发展

1. Linux 的 GUI 系统架构

首先了解 Linux 的 GUI 系统架构，无论是桌面 Linux 还是嵌入式 Linux，它们所涉及的一些概念是相似的。

本节中，下面内容来自陈汉仪先生所著的《The Embedded Linux GUI System》。

“UNIX 环境底下的图形窗口标准为 X Window System，Linux 算是 UNIX Like 的系统，上头跑的 GUI 系统是兼容于标准 X 的 XFree86 系统。下面以 X Window System 架构的思维来介绍各个系统，希望这样的介绍能够让您有清楚的观念。”

“依照 X 的逻辑，我大致划分了即 X Server（包括 Display、Input 等）、Graphic Library（底层绘图函数库）、Toolkitss（如 QT、GTK+等）、Window Manager、桌面环境、Internationalization（I18N）等几大类来剖析。”

- X Server

“X Window System 架构上有一项特点是别的 GUI 系统所没有的，这个特点就是 Client/Server 架构，请注意这跟一般我们熟知的某某服务器（Server 端）跟 PC 端（Client 端）相连接的情形不同。惟一类似的是 X Window System 本身也是采用网络架构设计。具体地说 X Client 就是我们在 X 上执行的软件（比如浏览器、各种办公软件等），X Server 则是负责

显示、传递使用者输入事件（包括键盘及鼠标等硬件装置的输入）。”

- Graphic Library

“我们可以把一幅图案想象成是由成千上万个细微小点所组成，这种小点的单位通常为 pixel（像素），在同一平方单位里头这些小点越多图案就越清晰、画质就越好，专业一点的解释便是分辨率高。我们要设计出一个窗口当然不可能一点一点地画上去，这样太过于旷日费时，基于这样的观念我们就会开始设计出高阶一点的函数来帮助我们完成这些繁琐的步骤，于是就出现了画点、画线、画矩形、画圆形、画不规则形、上色等高阶函数。通过这些高阶函数使得程序设计者不用去管画一条线要点几个点以及如何让显示器显示等零零总总低阶的工作，我们称绘图相关的一组函数库为 GUI 的基本：Graphic Library。”

- Toolkits

“有了点、线、面的函数之后，虽然已经去除大半的无聊工作，但是就开发窗口程序来说，还是显得非常没有效率，怎么办呢？只有继续将构成窗口的抽象组件（它们被称为构件 Widgets，在 Windows 下的对应术语为控件）例如：按钮、滚动条等抽离出来，重新定义一组更高阶的函数库，再配合上一些联系的语法函数就成了 Toolkits 这东西，目前以 QT、GTK+ 等较为流行。”

- Window Manger

“有了 Toolkits 我们可以很轻松地建立窗口软件（X Client），但是每个窗口软件只负责自己软件内的事务，那不同窗口间的沟通、协调（例如：窗口的切换、放大、缩小等），就没人管了，于是窗口管理器（Window Manager）就应运而生了。”

- 桌面环境

“在一个 Linux 系统中，光有窗口管理器是不够的——总得有东西给它管才行。桌面环境提供一整套图形界面下使用的程序，比如浏览器、邮件客户端、文件管理器、图形化桌面配置工具、桌面应用程序、办公软件等。在桌面 Linux 系统中，有两个主流的桌面环境：KDE、GNOME。”

- Internationalization

“国际化通常是我们东方语系国家的人比较关心的议题，但是很多软件一开始都由西方国家所主导开发，因此这点常常受到忽略，这个问题牵扯的层面非常广泛，上从语文的显示、输入，中至语文习惯，下到文字位元的处理，完整的解决是必须从头到脚彻底配合才能达成，只处理一半都只能说是一个跛脚的系统。”

“随着东方国家使用 GNU/Linux 的人口越来越多，I18N（之所以如此简称是因为 Internationalization 这个单词除去首尾两个字母后，还有 18 个字母）也就日益受到重视，目前底层 libc 部分已经有完整的支持，剩下来便是 GUI 系统的问题，由于处理一个字符时使用双字节所耗的资源较大、西方国家主导的系统多的情况下，有时候在一些取舍上，I18N 就被牺牲了。整体而言 Embedded Linux GUI 系统在 I18N 的程度通常都没 PC 端的好，只有在有需求时才会特别调校。”

“上述几点就是笔者借 X Window System 的分层架构，来指出一般的 GUI 系统所必须具备的功能，虽说 X 架构不错，但却不甚适用于嵌入式环境底下，因为相关程序过于庞大，因此有很多 Embedded Linux GUI 系统会把上述几点合并，甚至全部绑在一块，当然这样会失去

很多弹性与功能，但却也是一种权衡的作法。”

2. 桌面 Linux GUI 的发展

对于 Linux 的 GUI 系统，我们首先接触的是桌面的 GNOME、KDE 等。它们是什么概念呢？在了解嵌入式 Linux GUI 之前，先了解一下桌面 Linux GUI 系统的发展过程，这对于选择合适的嵌入式 Linux GUI 软件也会有所帮助。

《自由软件圣战——KDE/QT .VS. Gnome/Gtk》一文非常生动地了桌面 Linux GUI 的两大流派 KDE/QT 和 Gnome/Gtk 的发展和竞争过程。

Qt 是一个跨平台的 C++图形用户界面库，由挪威 TrollTech 公司出品，目前包括 Qt、基于 Framebuffer 的 Qt Embedded（现已改名为 Qtopia）、快速开发工具 Qt Designer、国际化工具 Qt Linguist 等部分。Qt 支持所有 UNIX 系统，当然也包括 Linux，还支持 WinNT/Win2k，Win95/98 平台。

Trolltech 公司在 1994 年成立，但是在 1992 年，成立 Trolltech 公司的那批程序员就已经开始设计 Qt 了，Qt 的第一个商业版本于 1995 年推出，然后 Qt 的发展就很快了，下面是 Qt 发展史上的一些里程碑。

- 1996 年 10 月 KDE 组织成立。
- 1998 年 4 月 5 日 Trolltech 的程序员在 5 天之内将 Netscape5.0 从 Motif 移植到 Qt 上。
- 1998 年 4 月 8 日 KDE Free Qt 基金会成立。
- 1998 年 7 月 9 日 Qt 1.40 发布。
- 1998 年 7 月 12 日 KDE 1.0 发布。
- 1999 年 3 月 4 日 QPL 1.0 发布。
- 1999 年 3 月 12 日 Qt 1.44 发布。
- 1999 年 6 月 25 日 Qt 2.0 发布。
- 1999 年 9 月 13 日 KDE 1.1.2 发布。
- 2000 年 3 月 20 日嵌入式 Qt 发布。
- 2000 年 9 月 6 日 Qt 2.2 发布。
- 2000 年 10 月 5 日 Qt 2.2.1 发布。
- 2000 年 10 月 30 日 Qt/Embedded 开始使用 GPL 宣言。
- 2000 年 9 月 4 日 Qt free edition 开始使用 GPL。

基本上，Qt 同 X Window 上的 Motif、Openwin、GTK 等图形界面库和 Windows 平台上的 MFC、OWL、VCL、ATL 是同类型的东西，但是 Qt 具有下列优点。

- 优良的跨平台特性。

Qt 支持下列操作系统：Windows 95/98、Windows NT、Linux、Solaris、SunOS、HP-UX、Digital UNIX （OSF/1，Tru64）、Irix、FreeBSD、B SD/OS、SCO、AIX、OS390，QNX 等。

- 面向对象。

Qt 的良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。Qt 提供了一种称为 signals/slots 的安全类型来替代 callback，这使得各个元件之间的协同工作变得十分简单。

- 丰富的 API。

Qt 包括多达 250 个以上的 C++ 类，还替供基于模板的 collections、serialization、file、I/O device、directory management、date/time 类，甚至还包括正则表达式的处理功能。

- 支持 2D/3D 图形渲染。
- 支持 OpenGL。
- 大量的开发文档。
- XML 支持。

但是真正使得 Qt 在自由软件界的众多 Widgets（如 Lesstif，Gtk，EZWGL，Xforms， fltk 等）中脱颖而出的还是基于 Qt 的重量级软件 KDE。

那么对于用户来说，如何在 Qt/GTK 中作出选择呢?一般来说，如果用户使用 C++，对库的稳定性、健壮性要求比较高，并且希望跨平台开发的话，那么使用 Qt 是较好的选择，但是值得注意的是，虽然 Qt 的 Free Edition 采用了 GPL 宣言，但是如果你开发 Windows 上的 Qt 软件或者是 UNIX 上的商业软件，还是需要向 Trolltech 公司支付版权费用的。

25.1.2 嵌入式 Linux 中的几种 GUI

KDE/QT 与 Gnome/Gtk 的竞争从桌面 GUI 系统扩展到嵌入式 GUI 系统，分别有 Qtopia 和 GtkFB。在嵌入式领域，GUI 种类繁多，比如 Microwindows、MiniGUI 等，下面介绍其中的几种。

1. QTE 和 QPE

前面说过 QT 是一个图形界面库，它需要配合底层的 X Server、X Libaray 等才能运行图形程序。TrollTech 公司也针对嵌入式环境推出了“Qt/Embedded”产品，简称为 QTE。与桌面版本不同，QTE 已经直接取代掉 X Server 及 X Library 等角色，将所有的功能全部整合在一起。它有如下特色：

- 与桌面 Qt 库使用相同的 API 接口。

开发者只需要学习与套 API 接口，基于 Qt 或 QTE 开发的程序只需要维护一套代码，就可以运行于多种桌面环境（比如 Windows, X11, Mac OS X）或者嵌入式 Linux 环境下。

- 针对嵌入式系统专门设计。

使用 QTE，可以设计出占用内存、Flash 更少的程序。

- QTE 包含自己的窗口系统。

QTE 不再需要其他底层库的支持，它已经包含了一切，可以直接在它上面开发、运行图形程序。

- 可配置的外观。

QTE 的 GUI 是可以高度配置的，这有利于客户开发自己独特的程序。

- 完全的模块化设计。

可以将不需要的功能模块去掉，这可以节省系统资源。QTE 号称最小可以缩到 800 KB 左右，最多为 3 MB（for Intel x86），这样的弹性让 QTE 更适合在嵌入式环境底下生存。

- 源代码完全开放。

这使得用户可以深入调节 QTE，或是了解它的具体实现。

- 与 Qtopia 完美整合。

Qtopia是Trolltech公司在QTE的基础上针对PDA和手机开发的应用平台和用户界面。基于QTE开发的应用程序可以轻易地迁移到基于Qtopia开发的设备上。

- 同Java的集成。

QTE可以同几种Java虚拟机集成。Java程序可以在基于QTE的工作平台上运行，提供同原程序相同的效果。

与PC桌面的KDE类似，Trolltech公司针对PDA软件推出了整体解决方案——QPE（Qt Plamtop Environment），从底层的GUI系统、Window Manager、Soft Keyboard到上层的PIM、浏览器、多媒体等全部一手包办。QPE目前已经改名为Qtopia，本章的目标是移植一个与PC的桌面类似的GUI系统（可以称为PDA）。

2. GtkFB

自从Qt推出了嵌入式版本之后，虽然GTK＋并非商业公司所发展，但也加紧脚步推出了GtkFB方案，其宗旨就是要为嵌入式系统推出一套基于GTK+的GUI解决方案。与QTE类似，GtkFB也跳过X层直接与FrameBuffer沟通，因此也具有QTE的几项优点。

（1）优点。

GtkFB的最大优点是可以使用强大的GTK+库，基于GTK+库的软件极大丰富，并且GtkFB跳过了“巨大的”X，适用于PDA等嵌入式设备。GtkFB所用的API与桌面系统所用API完全一样，这使得在桌面PC和嵌入式设备间移植软件、共享代码非常容易。

另一个优点是GtkFB是完全免费、完全公开的，它鼓励程序员进行修改以切合实际需要。它基于LGPL协议，用户无需公开他自己的代码，当然，被修改的库文件是要公开的。

另外，由于GtkFB不使用X协议，得以消除一些X协议所特有缺点。

（2）缺点。

GtkFB的最大缺点是它只能运行在单处理器系统上，这意味着无法使用其他处理器来分离、保护系统的不同部分，也很难使用GtkFB来设置大型的系统。

有些基于GTK+的程序直接调用X，如果不修改代码，它们无法在GtkFB上直接运行。GNOME库有一些对X的直接调用，所以基于GNOME库的程序需要一些修改。

X拥有大量成熟的驱动程序，有极好的硬件加速功能。GtkFB也支持硬件加速，但是目前这方面的工作进展很小。这意味着GtkFB在某些方面运行得比较慢，特别是大屏幕情况下。

Framebuffer不支持X的一些特性，比如网络透明、DGA等。

3. Microwindows

Microwindows Open Source Project成立的宗旨在于针对体积小的装置，建立一套先进的视窗环境，在Linux桌面上通过交叉编译可以很容易地制作出micro-windows的程序。MicroWindows能够在没有任何操作系统或其他图形系统的支持下运行，它能对裸显示设备进行直接操作。这样，MicroWindows就显得十分小巧，便于移植到各种硬件和软件系统上。

然而MicroWindows的免费版本进展一直很慢，几乎处于停顿状态，而且至今为止，国内没有任何一家公司专业对MicroWindows提供全面技术支持、服务和担保。

4. MiniGUI

MiniGUI 是我国做得比较好的自由软件之一，它是在 Linux 控制台上运行的多窗口图形操作系统，可以在以 Linux 为基础的应用平台上提供一个简单可行的 MiniGUI 支持系统。“小”是 MiniGUI 的特色，MiniGUI 可以应用在电视机顶盒、实时控制系统、掌上电脑等诸多场合。由于这是由我国自己开发的 GUI 系统，所以 MiniGUI 对于中文的支持最好。它支持 GB2312 与 BIG5 字符集，其他字符集也可以轻松加入。

5. 基于 Tiny X Server 的 X 架构

Tiny X Server 是 XFree86 Project 的一部分，由 Keith Packard 先生所发展，他本身就是 XFree86 项目的核心成员之一，一般的 X Server 都太过于庞大，因此 Keith Packard 就以 XFree86 为基础，精简而成 Tiny X Server，它的体积可以小到几百 KB，非常适合应用于嵌入式环境之中。

以纯 X Window System 搭配 Tiny X Server 架构来说，最大的优点就是弹性与开发速度，因为与桌面的 X 架构相同，因此以 Qt、GTK+等所开发的软件可以很容易的移植上来。

虽然移植速度快，但是却有体积大的缺点，由于很多软件本来是针对桌面环境所开发，因此无形之中功能都比较多样，有些并不适用于嵌入式环境，因此“调校”便成为采用此架构最大的课题，有时候重新改写都可能比调校所需的时间还短。

25.2 Qtopia 移植

25.2.1 主机开发环境的搭建

下面安装主机上的开发环境。

1. 安装 g++

编译 Qtopia 时，需要用到 Qt 自带的一些工具，有些是使用 C++编写的。Ubuntu 7.10 中没有 g++编译器，在 2.2.3 节中，已经安装了 g++。

2. 安装 X11 的相关库文件和开发包

在编译 Qtopia 时，会生成一些在主机上运行的工具，要用到 X11 的一些头文件、库，比如/usr/X11R6/include/X11/Xlib.h，所以需要安装 X11 开发包。执行如下命令即可：

```
    $ sudo apt-get -y install x-dev libx11-dev xlibs-static-dev x11proto-xext-dev
libxext-dev libqt3-mt-dev
    $ sudo mkdir -p /usr/X11R6/include
    $ sudo cp -rf /usr/include/X11 /usr/X11R6/include/          /* 把 X11 目录复
制过去 */
```

3. 安装集成开发环境

与Windows下的Visual Studio 6.0类似，Linux下也有多种集成开发环境。在开发Qt的GUI程序时，常使用Kdevelop。为了方便使用，将控制终端konsole也一起装上，这使得可以在Kdevelop的界面上使用命令行。

安装命令如下：

```
$ sudo apt-get install kdevelop3
$ sudo apt-get install konsole
```

安装完成后，在桌面的菜单Applications->Programming里可以看到很多Kdevelop工具，在Applications->System Tools里可以看到Kconfig工具。

25.2.2 交叉编译、安装Qtopia 2.2.0

在以前移植Qtopia是一件非常繁琐的事，需要先编译Qt/Embedded，它是底层基础；然后还要编译Qt/X11（它是桌面PC使用的Qt库），这步仅仅是因为要用到其中生成的一些工具（比如uic）；最后才编译Qtopia，它就是QPE（Qt Plamtop Environment），里面包含了众多的应用程序。从Qtopia 2.2.0开始，Qtopia的代码里就包含了Qtopia、Qt、Qt/Embedded和tmake（一个用来生成Makefile的工具）。Qtopia 2.2.0之后的版本是4.2.0，从它开始，开源版本不再支持PDA，所以我们将使用2.2.0的版本。可以使用/work/GUI/qtopia/qtopia-free-src-2.2.0.tar.gz，也可以下面的网址下载到源代码：

```
http://www.qtopia.org.cn/ftp/mirror/ftp.trolltech.com/qtopia/source/qtopia-free-src-2.2.0.tar.gz
```

编译Qtopia 2.2.0前，首先要编译、安装它所依赖的库，根据开发板的硬件特性修改配置文件，还要针对编译器的版本修改源代码，针对开发板的所用的C库（如果是uClibc的话）修改源代码。

即使对于一个成熟的产品，也不能指望它的代码不经修改就能编译成功，开发环境不同、代码本身的缺陷等都是修改的原因。下面的修改内容是笔者在编译Qtopia时，根据错误信息进行修改，再编译、再出错、再修改，如此反复总结出来的。如果读者在移植时，由于开发环境不一样而出错，那么请根据出错信息自行修正。

本书所用开发环境、工具如下。

① 主机系统：Ubuntu 7.10。

② 编译器版本：gcc/g++ 4.1.3；arm-linux-gcc/g++ 3.4.5。

③ 交叉编译器自带的库：glibc-2.3.6。

Qtopia的文档非常丰富，将源码包qtopia-free-src-2.2.0.tar.gz打开得到目录qtopia-free-2.2.0，顶层目录下的README.html就是所有文档的入口点，下面所指的文件都是相对于qtopia-free-2.2.0目录。

先了解一下qtopia-free-2.2.0目录中的内容，它的子目录、文件如表25.1、25.2所示，请参考文档qtopia/doc/html/build-from-source.html。

表 25.1 qtopia-free-2.2.0 下的子目录

子 目 录	描 述
qtopia	所有 qtopia 的代码，即基于 qte 的上层 GUI 环境，或者称为 QPE
qt2	Qt 2.3.x 的源码，可以用于 X11 或嵌入式系统
dqt	Qt-x11-3.3.x 的源码，它提供一些工具用来编译 Qtopia 的桌面
tmake	tmake-1.14 的源码，tmake 是一个用来配置 Qt 2.3.x 的工具

表 25.2 qtopia-free-2.2.0 下的文件和脚本

文件/脚本	描 述
LICENSE	许可证
LODI	A List Of Deliverable Items，即本开发包中所有的文件列表
README.html	文档，用来做一些引导性介绍
configure	配置脚本
Makefile.in	一个临时的 Makefile，它被上面的配置脚本用来生成真正的 Makefile

1. 编译、安装 Qtopia 所依赖的库

文档 qtopia/doc/html/environment-prereq.html 描述了编译、执行 Qtopia 时需要满足的依赖条件，比如内核要支持 Frame Buffer、用到的库、系统命令、执行 Qtopia 时用到的设备、编译器的版本等。

本小节只关心所依赖的库，文档中列出了 jpeg、zlib、uuid 共 3 项，可以使用 qtopia-free-2.2.0 自带的 zlib 库，所以只需要编译、安装 jpeg 和 uuid。

（1）编译、安装 jpeg 库。

从 http://www.ijg.org/files/下载源码 jpegsrc.v6b.tar.gz，解压后得到目录 jpeg-6b。

先使用以下命令进行配置：

```
$ ./configure --enable-shared --enable-static          \
--prefix=/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux   \
--build=i386 --host=arm
```

然后修改生成的 Makefile，如下：

```
CC= gcc     改为: CC= arm-linux-gcc
AR= ar rc   改为: AR= arm-linux-ar rc
AR2= ranlib 改为: AR2= arm-linux-ranlib
```

最后是编译和安装，执行如下命令：

```
$ make
$ make install-lib
```

这将在/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux 中的 include 目录中生成一些头文件，在

lib目录中生成一些jpeg库文件。

（2）编译、安装uuid库。

本书23.1.3小节移植e2fsprogs-1.40.2开发包中的EXT2文件系统格式化工具mke2fs时，已经编译、安装了uuid库。

下面讲解对qtopia的修改，读者可以一个个地修改文件，也可以使用补丁文件：

```
$ cd qtopia-free-2.2.0/
$ patch -p1 < ../qtopia-free-2.2.0_100ask.patch
```

2. 修改配置文件

本节的目标是移植Qtopia后，能够使用USB键盘、USB鼠标来操作。这需要修改配置文件qtopia/src/qt/qconfig-qpe.h，将下面几行注释掉:QT_NO_QWS_CURSOR表示没有光标，QT_NO_QWS_MOUSE_AUTO表示没有鼠标，QT_NO_QWS_MOUSE_PC表示不支持与台式机类似的鼠标，如下所示：

```
48 #ifndef QT_NO_QWS_CURSOR
49 #define QT_NO_QWS_CURSOR
50 #endif
51 #ifndef QT_NO_QWS_MOUSE_AUTO
52 #define QT_NO_QWS_MOUSE_AUTO
53 #endif
54 #ifndef QT_NO_QWS_MOUSE_PC
55 #define QT_NO_QWS_MOUSE_PC
56 #endif
```

3. 针对交叉编译器的版本修改代码

根据文档qtopia/doc/html/environment-prereq.html，可以知道能用来编译qtopia-free-2.2.0的编译器版本有：2.95.2、3.2.4、3.3.0、3.3.3、3.3.4、3.4.1。

gcc-2.95是一个质量比较高的编译器，它的性能甚至优于3.0、3.2版本。很多工程（比如Linux内核）仍旧使用gcc-2.95.x，它产生的代码的质量和稳定性比很多高版本的编译器还要好。也有很多人使用gcc-2.95.x来编译qtopia-free-2.2.0（也许使用gcc-2.95.x时，本小节指定的文件不需要修改），本书使用的交叉编译器版本为3.4.5，需要修改qtopia-free-2.2.0中的几个文件。

```
vi qt2/tools/qembed/Makefile.in
```

（1）修改qt2/src/tools/qvaluestack.h。

第57行修改如下，否则编译时会出现remove函数无法识别的错误。

修改前：

```
57 remove( this->fromLast() );
```

修改后：

```
57 this->remove( this->fromLast() );
```

（2）修改 qt2/src/kernel/qwindowsystem_qws.h。

在文件的开头增加两个类的声明，加入下面两行：

```
class QWSInputMethod;
class QWSGestureMethod;
```

这两个类在这个文件的开始部分就被用到了，但是它们在文件的后部分定义。如果不在前面声明，新的编译器会导致“has not been declared”错误。

（3）修改 qtopia/src/libraries/qtopia/backend/event.cpp。

第 419 行修改如下。

修改前：

```
419    while ( !( i & day ) && i <= Event::SUN ) {
```

修改后：

```
419    while ( !( i & day ) && (int)i <= Event::SUN ) {
```

否则会出现“error: ISO C++ says that these are ambiguous”的错误，错误信息如下：

```
backend/event.cpp:419: error: ISO C++ says that these are ambiguous, even
though the worst conversion for the first is better than the worst conversion for
the second:
    backend/event.cpp:419: note: candidate 1: operator<=(int, int) <built-in>
    /work/GUI/qtopia-free-2.2.0/qt2/include/qstring.h:312: note: candidate 2:
int operator<=(char, QChar)
```

这表示进行“<=”运算时，编译器无法确定是使用自身的方法还是使用 qt2/include/qstring.h 中重载的“<=”运算符进行比较。

（4）修改导致“error: extra qualification”错误的文件。

在一个类的定义中，使用“类名::”来修饰它本身的成员函数显得很冗余，3.2 版本之后的 gcc 编译器将它视为错误，比如会看到“error: extra qualification”字样的错误信息。解决方法是去掉前面的修饰符“类名::”。

由于这个原因导致需要修改的文件有很多，根据出错信息来进行修改即可。

注意 当配置不同时，要编译的文件也不同，所以本节中修改的文件并不一定完全，只能保证这些修改适用于这个配置。

这些文件如下。

① qtopia/src/libraries/qtopia/qdawg.cpp。

第 294 行修改如下。

修改前：

```
294 QDawgPrivate::~QDawgPrivate()
```

修改后：

```
294 ~QDawgPrivate()
```

② qtopia/src/libraries/qtopia2/thumbnailview_p.h。

第81行修改如下。

修改前：

```
81    void ThumbnailItem::paintItem( QPainter*, const QColorGroup& );
```

修改后：

```
81    void paintItem( QPainter*, const QColorGroup& );
```

③ qtopia/src/libraries/qtopiapim/abtable_p.h。

第276行修改如下。

修改前：

```
276    QListViewItem* PhoneTypeSelector::addType(QListViewItem* prevItem,
```

修改后：

```
276    QListViewItem* addType(QListViewItem* prevItem,
```

④ qtopia/src/libraries/qtopiapim/numberentry_p.h。

第106行修改如下。

前改前：

```
106    bool NumberEntryDialog::eventFilter(QObject *o, QEvent *e);
```

修改后：

```
106    bool eventFilter(QObject *o, QEvent *e);
```

⑤ qtopia/src/libraries/mediaplayer/videoviewer.cpp。

第52行修改如下。

修改前：

```
52 SimpleVideoWidget::SimpleVideoWidget(QWidget *parent);
```

修改后：

```
52 SimpleVideoWidget(QWidget *parent);
```

⑥ qtopia/src/applications/addressbook/ablabel.h。

第78行修改如下。

修改前：

```
78  bool  AbLabel::decodeHref(const  QString&  href,  ServiceRequest*  req,
QString* pm) const;
```

修改后：

```
78 bool decodeHref(const QString& href, ServiceRequest* req, QString* pm)
const;
```

⑦ qtopia/src/games/minesweep/minefield.h。

第 105、106 行修改如下。

修改前：

```
105    void MineField::setState( State st );
106    void MineField::placeMines();
```

修改后：

```
105    void setState( State st );
106    void placeMines();
```

⑧ qtopia/src/settings/buttoneditor/buttoneditordialog.h。

第 56 行修改如下。

修改前：

```
56 ServiceRequest ButtonEditorDialog::actionFor(int cur) const;
```

修改后：

```
56 ServiceRequest actionFor(int cur) const;
```

⑨ qtopia/src/settings/qipkg/packagewizard.h。

第 106 行修改如下。

修改前：

```
106 PackageItem* PackageWizard::current() const;
```

修改后：

```
106 PackageItem* current() const;
```

⑩ qtopia/src/plugins/inputmethods/keyboard/keyboard.h。

第 60 行修改如下。

修改前：

```
60 KeyboardPicks::~KeyboardPicks();
```

修改后：

```
60 ~KeyboardPicks();
```

⑪ qtopia/src/plugins/decorations/polished/polished.h。

第 58 行修改如下。

修改前：

```
58 void PolishedDecoration::drawBlend( QPainter *, const QRect &r, const QColor
&c1, const QColor&c2 ) const;
```

修改后：

```
58 void drawBlend( QPainter *, const QRect &r, const QColor &c1, const QColor&c2 )
const;
```

⑫ qtopia/src/server/inputmethods.cpp。

第86行修改如下。

修改前：

```
86 IMToolButton::IMToolButton( QWidget *parent ) : QToolButton( parent )
```

修改后：

```
86 IMToolButton( QWidget *parent ) : QToolButton( parent )
```

4. 针对uClibc库修改代码：如果使用glibc，则这个步骤可以省略(本书使用glibc)

常用的C库有glibc和uClibc，后者常用于嵌入式系统。两者基本兼容，但是还是有一些区别。

（1）修改qtopia/src/3rdparty/libraries/rsync/config_linux.h。

将第47行如下注释掉，uClibc中没有变量program_invocation_short_name。

```
46 /* GNU extension of saving argv[0] to program_invocation_short_name */
47 //#define HAVE_PROGRAM_INVOCATION_NAME 1
```

否则在编译时会出现类似下面的错误：

```
3rdparty/libraries/rsync/trace.c:130:                                    error:
`program_invocation_short_name'
```

（2）修改qtopia/src/3rdparty/plugins/codecs/libffmpeg/mediapacketbuffer.h。

将第231、233行的pthread_yield函数修改为sched_yield，uClibc中没有定义pthread_yield函数，使用sched_yield来代替它。修改后的代码如下：

```
231         sched_yield(); //pthread_yield();
232         bufferMutex.signal();
233         sched_yield(); //pthread_yield();
```

5. 配置、编译、安装Qtopia

在配置之前，先执行如下命令复制两个文件，否则编译的时候会提示找不到这两个文件。

```
$ cd qtopia/src/libraries/qtopia/
$ cp custom-linux-cassiopeia-g++.h   custom-linux-arm-g++.h
```

```
$ cp custom-linux-cassiopeia-g++.cpp  custom-linux-arm-g++.cpp
$ cd -
```

（1）配置 qtopia-free-2.2.0。

配置命令如下，执行以下命令之后，就可以执行“make”命令编译了。

```
$ ./configure  -qte '-embedded -no-xft -xplatform linux-arm-g++ -qconfig qpe
-depths 16,32 -no-qvfb -system-jpeg -gif' -qpe '-xplatform linux-arm-g++ -edition
pda -displaysize 240x320' -qt2 '-no-xft' -dqt '-no-xft'
```

下面详细介绍相关的配置项。

在顶层目录下执行“./configure -help”命令就可以看到总体的配置信息，所有的配置信息由 3 个子目录的配置脚本 qtopia/configure、qt2/configure 和 dqt/configure 来提供。它们分别表示 3 个子产品（sub-product）的配置信息，要想详细了解某个子产品，可以使用如表 25.3 所示的命令。

表 25.3　　查看子产品（sub-product）配置信息的命令

子 产 品	命　令
Qtopia	qtopia/configure -help
Qt 2.3.x	qt2/configure -help
Qt 3.3.x	dqt/configure -help

注意　如果表中的命令执行时出错，这是因为一些环境变量（比如 QTDIR、QPEDIR）没有设置，可以简单地在顶层目录先执行“./configure”命令，它会生成 setQt2Env、setQpeEnv、setDqtEnv 等用来设置环境变量的脚本文件。当确定了配置参数后，先执行“make clean”，然后重新配置即可。

在顶层目录下执行“./configure -help”，可以看到它的用法为：

```
Usage:  configure [-qte 'cfg'] [-qpe 'cfg'] [-libpath path] [-prefix path]
                  [-debug] [-qtopiadesktop] [-dprefix path]
                  [-qt2'cfg'] [-dqt 'cfg']
```

① Qt/Embedded 的配置：

其中，-qte 'cfg'表示对 Qt/Embedded 的配置，将 cfg 置换成具体的参数即可。qtopia-free-2.2.0 预先定义了一些常用的配置，它们用 keypad、arm-keypad、no-keypad、arm-no-keypad 等缩写来代替。比如可以这样指定配置项：

```
$ ./configure -qte arm-no-keypad
```

它的意义等同于（请参考“./configure -help”帮助信息）：

```
$ ./configure -qte  '-embedded -no-xft -xplatform linux-sharp-g++ -qconfig qpe
-depths 16,32 -no-qvfb -system-jpeg -gif'
```

这些配置项基本符合本书所用开发板，只需要将 linux-sharp-g++改为 linux-arm-g++。

这些参数的意义，可以使用“qt2/configure -help”命令查看帮助。这些参数如表 25.4、25.5 所示。

表 25.4 qt2（包括 Qt/Embedded）的配置选项

配置项（带有*号的表示默认选项）	描　述
* -release	编译、连接 Qt 时，没有调试信息
-debug	编译、连接 Qt 时，加入调试信息
* -shared	创建 Qt 的动态链接库 libqt.so
-static	创建 Qt 的静态链接库 libqt.a
* -no-gif	不支持 GIF 格式的图形
-gif	支持 GIF 格式的图形
-no-sm	不支持“X Session Management”
* -sm	支持“X Session Management”
* -no-thread	不支持线程（Do not compile with Threading Support）
-thread	支持线程（Compile with Threading Support）
* -qt-zlib	使用 Qt 自带的 zlib 库
-system-zlib	使用操作系统的 zlib 库
* -qt-libpng	使用 Qt 自带的 pnp 库
-system-libpng	使用操作系统的 pnp 库
* -no-mng	不支持 mng（mng 是流行的 png 图片格式的动画扩展）
-system-libmng	使用操作系统的 mng 库
* -no-jpeg	不支持 jpeg
-system-jpeg	使用操作系统的 jpeg 库
* -no-nas-sound	不支持“网络音响”（Network Audio System）
-system-nas-sound	使用操作系统的 NAS libaudio 库
-no-<module>	禁止某个模块，“<module>”可以是这 4 个之一： opengl、table、network、canvas
-kde	编译工具 designer 时，让它支持 KDE 2。这样，KDE 2 的控件（widget）就可以在 designer 上直接使用
-tslib	使能触摸屏的处理函数
-no-g++-exceptions	一个编译选项： Disable exceptions on platforms using the GNU C++ compiler by using the -fno-exceptions flag
-no-xft	不支持 Anti-Aliased 字体
-xft	支持 Anti-Aliased 字体，这要用到 xft 库
-platform target	指定主机平台，请查看 qt2/PLATFORMS 获知支持的平台
-xplatform target	指定变叉编译的平台，请查看 qt2/PLATFORMS
-Istring	指定额外的头文件目录

续表

配置项（带有*号的表示默认选项）	描　述
-Lstring	指定额外的链接库目录
-Rstring	指定额外的动态键接库目录（dynamic library runtime search path）
-lstring	指定额外的库

Qt/Embedded only（以下选项只适用于 Qt/Embedded）。

表 25.5　　Qt2 的配置选项（只适用于 Qt/Embedded）

-embedded	使能 Qt/Embedded
-qconfig local	使用配置文件 qt2/src/tools/qconfig-<local>.h，不指定的话将使用 qt2/src/tools/qconfig.h
-depths list	支持的像素位宽，使用逗号分开。可取的值有： v、4、8、16、18、24、32 （“v”表示 VGA16）
-accel-snap	显卡方面的加速功能
-accel-voodoo3	
-accel-mach64	
-accel-matrox	
-qvfb	使能虚拟的 Frame Buffer（X11-based Qt Virtual Frame Buffer），它通常用于在 x86 主机上调试程序
-vnc	支持 VNC 服务器（Virtual Network Computing）
-keypad-mode	这两个选项用于 Qtopia Phone 版，在 PDA 版中不需要
-keypad-input	

② Qtopia 的配置。

-qpe 'cfg'表示对 Qtopia 的配置，将 cfg 置换成具体的参数即可。qtopia-free-2.2.0 预先定义了一些常用的配置，它们用 phone、arm-phone、pda、arm-pda、core、arm-core 等缩写来代替。比如可以这样指定配置项：

```
-qpe arm-pda
```

它的意义等同于（请参考“./configure -help”帮助信息）：

```
-qpe '-xplatform linux-sharp-g++ -edition pda -displaysize 240x320'
```

这些配置项基本符合我们的开发板，只需要将 linux-sharp-g++改为 linux-arm-g++。

这些参数的意义，可以使用下面命令查看帮助，第一条指令用来设置环境变量，否则第二条指令无法运行。

```
$ . ./setQpeEnv
$ qtopia/configure -help
```

这些帮助信息没有给出具体解释，不过可以从它们的名字上知道各配置项的含义。

> **注意**　第一条命令“. ./setQpeEnv”中的第一个“.”表示脚本内容用于设置“当前的”环境变量。如果不在前面加上“.”，则这些环境变量只在这个脚本的执行过程中有效。

③ Qt2、Qt3的配置。

只要按如下指定即可，它们表示不使用Anti-Aliased字体，否则会编译出错（没有安装xft库）。

```
-qt2 '-no-xft' -dqt '-no-xft'
```

其他的配置项都是一些路径的设置，我们使用默认即可。这3类配置项组合起来，就得到本小节开头的配置命令。

（2）编译、安装qtopia-free-2.2.0。

执行配置命令时，会询问是否接受Qtopia免费版本的许可协议，回答yes，如下：

```
Do you accept the terms of the Qtopia Free Edition License? yes
```

配置完成后，会得到一些操作指示，如下：

```
Qtopia is now configured.

Type "make"                to build the qtopia bundle (and the tools, if required).
Type "make install"        to install Qtopia.
Type "make cleaninstall" to install Qtopia after removing the image first (avoid
stale files in the image).
    Type "make clean"      to clean the qtopia bundle.

    Type "make tools"      to build the tools bundle.
    Type "make cleantools"  to clean the tools bundle.

    To manually build a particular component (eg. because it failed to build)
    source the set...Env script. eg. . ./setQpeEnv; cd $QPEDIR; make
```

执行“make”进行编译，它将在以下目录中生成可执行文件和库文件。

```
qtopia/bin
qtopia/lib
qtopia/plugins
```

另外，字库文件在qt2/lib/fonts/目录下。

然后执行“make install”进行安装，它将把所有必需的目录、文件复制到qtopia/image/opt/Qtopia目录下。

6. 在开发板上安装、运行Qtopia

假设开发板的根文件系统保存在主机上的/work/nfs_root/fs_qtopia目录中，它从fs_mini_mdev复制得到。

```
$ cd /work/nfs_root
$ sudo cp -rf fs_mini_mdev fs_qtopia
$ sudo chown book:book fs_qtopia -R
```

然后，按照下述 7 个步骤在 fs_qtopia 目录中加入对 Qtopia 的支持。

（1）复制 Qtopia 所依赖的 jpeg 库、uuid 库。

```
$ cd /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/
$ cp libjpeg.so* /work/nfs_root/fs_qtopia/lib/ -d
$ cp libuuid.so* /work/nfs_root/fs_qtopia/lib/ -d
```

（2）复制字库。

```
$ cd /work/GUI/qtopia/qtopia-free-2.2.0/
$ cp -rf qt2/lib/fonts qtopia/image/opt/Qtopia/lib/
```

（3）将 qtopia/image/opt/整个目录复制到开发板根目录上。

```
$ cd /work/GUI/qtopia/qtopia-free-2.2.0/
$ cp -rf qtopia/image/opt /work/nfs_root/fs_qtopia
```

（4）创建时区文件。

直接使用主机中的时区文件。

```
$ cd /work/nfs_root/fs_qtopia
$ mkdir -p usr/share/zoneinfo/
$ cp -rf /usr/share/zoneinfo/America  usr/share/zoneinfo/
$ cp /usr/share/zoneinfo/zone.tab  usr/share/zoneinfo/
```

（5）伪造触摸屏校验文件。

Qtopia 第一次启动时，会自动运行触摸屏校验程序。由于本书没有移植触摸屏的驱动程序，这将导致校验失败，使得无法进入系统。可以通过在开发板根文件系统中下建立一个触摸屏的校验文件 etc/pointercal，内容为:1 0 1 0 1 1 65536，它可以让系统不执行校验程序。

（6）建立一个脚本文件，用来运行 qtopia。

在开发板根目录/bin 下建立 qpe.sh 文件，它用来设置环境变量、启动 Qtopia，内容如下：

```
#!/bin/sh
export HOME=/root
export QTDIR=/opt/Qtopia
export QPEDIR=/opt/Qtopia
export QWS_DISPLAY=LinuxFb:/dev/fb0
export QWS_KEYBOARD="TTY:/dev/tty1"
export QWS_MOUSE_PROTO="USB:/dev/mouse0"
export PATH=$QPEDIR/bin:$PATH
export LD_LIBRARY_PATH=$QPEDIR/lib:$LD_LIBRARY_PATH
$QPEDIR/bin/qpe &
```

前面几行用来设置环境变量，比如指定一些目录、指定动态库的路径、指定输入输出设备等，可以参考文档 dqt/doc/html/emb-envvars.html。

需要指出的是，Qtopia 启动后，它在$HOME 目录下存放运行过程中产生的配置文件；其中文本编辑器、媒体播放器等程序使用的文件也存放在$HOME/Documents 目录下，这意味着可以在这个目录下存放音乐、影视文件。

最后一行启动 Qtopia。

启动界面可以参考图 25.5，中间的是第一次启动时的界面，然后按照提示设置语言、时区、时间后就可以得到右边的界面，这时可以在里面启动各个程序。

注意　由于上面设置了“HOME=/root”，Qtopia 运行时产生的信息将保存在/root 目录下。所以还要建立/root 目录，命令为：“$ mkdir –p /work/nfs_root/fs_qtopia/root”。

（7）修改根文件系统的启动脚本。

运行 Qtopia 时，需要用到临时目录/tmp，为减少对 Flash 的擦写，在/tmp 目录上挂接 tmpfs 文件系统。

首先建立/tmp 目录。

```
$ mkdir -p /work/nfs_root/fs_qtopia/tmp
```

然后修改/work/nfs_root/fs_qtopia/etc/fstab 文件，加入一行。

```
tmpfs          /tmp          tmpfs  defaults        0    0
```

最后，修改启动脚本/work/nfs_root/fs_qtopia/etc/init.d/rcS，在最后加入以下一行。

```
/bin/qpe.sh &
```

还要修改它的属性。

```
$ chmod +x /work/nfs_root/fs_qtopia/bin/qpe.sh
```

需要注意，在/work/nfs_root/fs_qtopia/etc/inittab 中，不能再用/dev/tty1 来启动控制台，否则 Qtopia 启动时无法使用键盘。如下将这行禁止掉：

```
#tty1::askfirst:-/bin/sh
```

现在，可以使用 mkyaffsimage 工具将/work/nfs_root/fs_qtopia/制作成 yaffs 映象文件，烧到开发板上；或者通过 NFS 从/work/nfs_root/fs_qtopia/上启动系统。系统启动后，就可以在 LCD 上看到 Qtopia 的桌面了（第一次启动时，需要进行一些设置）。注意：启动之前，先接入 USB 键盘、USB 鼠标，还不能做到即插即用。

25.2.3　开发自己的 Qt GUI 程序

开发 Qt GUI 程序时，需要用到集成开发工具 Kdevelop、Qt 中自带的 designer、qmake、uic 和 moc 工具，当然，交叉编译工具必不可少。

Kdevelop 给用户提供了方便的操作界面，用来编译、运行、调试程序。它还包含有 Qt GUI 程序的模板，可以用来产生一个框架，然后再在上面进行修改。在生成模板时，会产生一个工程文件（.pro）、一个图形界面文件（.ui），还有一些源文件（.h、.cpp）。

本节中调用 qmake 工具根据.pro 文件来生成 Makefile 文件，这使得程序员可以摆脱编写

枯燥并且容易出错的 Makefile。

designer 工具被用来打开.ui 文件，得到一个可视化的界面，程序员可以在 designer 中修改这个界面，达到“所见即所得”的效果。

当使用 designer 工具编辑好界面之后，通过 uic 工具将.ui 文件转换为 C++源代码，这将生成一些.h、.cpp 文件。uic 就是“User Interface Compiler”的缩写，即用户界面编译器。

moc 是“Meta Object Compiler”的缩写，即元对象编译器。Qt 著名的 signal/slot 机制必须借助 moc 工具才能实现：moc 检查一个 C++源文件，如果发现它包含有 Q_OBJECT 宏，则生成另一个 C++源文件，里面包含了“元对象代码”(meta object cod)。用 moc 产生的 C++源文件必须与类实现一起进行编译和连接，或者用#include 语句将其包含到类的源文件中。

使用 qmake 生成的 Makefile 中有使用 uic、moc 工具生成源文件的规则，这使得在编译程序时，会自动调用 uic、moc 来生成源文件。

综上所述，开发 Qt GUI 程序时，先使用 Kdevelop 工具生成程序框架，然后使用 designer 修改图形界面，接着使用 qmake 生成 Makefile 文件；编译程序时，Makefile 会调用 uic 将界面文件（.ui）转换为 C++源文件，调用 moc 生成 C++源文件以实现“元对象”。

下面以一个最简单的 GUI 程序“helloworld”为例，分步骤介绍开发过程。

1. 使用 Kdevelop 工具生成程序框架

从 Ubuntu 7.10 的桌面菜单中启动 Kdevelop，位置如下：

```
Applications -> Programming -> KDevelop: C/C++
```

然后创建一个新工程。

（1）在“Project”菜单中选择“New Project”。

（2）打开左边的“C++”目录，在“Embedded”子目录下可以看见多种工程的模板，如图 25.1 所示。

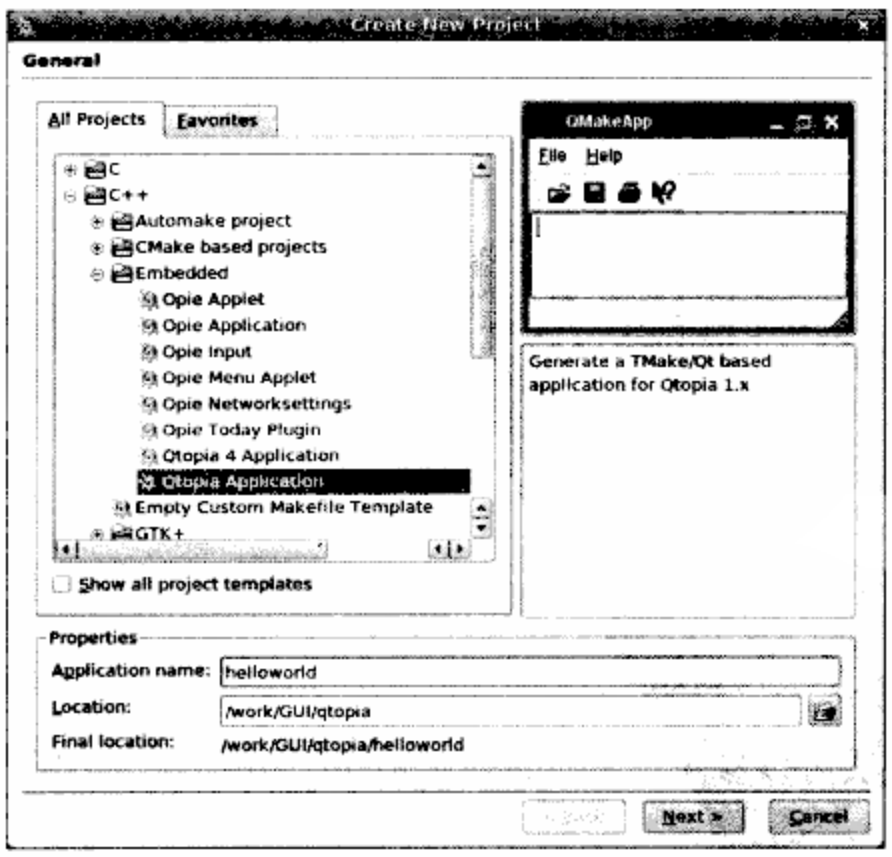


图 25.1　新建一个 Qtopia 工程

（3）选择“Qtopia Application”。

（4）指定工程存放的位置，输入工程的名字“helloworld”。

然后在后面出现的界面中，选择默认值，最后生成了一个框架，在 helloworld 目录下有如下文件：

```
$ ls
COPYING    helloworld.control  helloworld.h        helloworld.png  templates
Doxyfile   helloworld.cpp      helloworld.html     helloworld.pro
helloworldbase.ui  helloworld.desktop  helloworld.kdevelop  main.cpp
```

2. 使用 designer 修改图形界面

helloworld 目录下的 helloworldbase.ui 可以使用 designer 工具来打开和编辑。

> **注意**　designer 目前有 3 个版本，Qt2、Qt3、Qt4。使用 Qt3、Qt4 版本的 designer 打开.ui 文件后，它将不能用于 Qt2。

在本书移植的 qtopia-free-2.2.0 中，Qt 的版本为 2.3.x，所以只能使用 Qt2 版本的 designer 来处理.ui 文件。这个工具在编译 qtopia 时，在 qt2/bin/目录下已经生成了，为方便使用并区别于其他版本，把它复制到/usr/local/bin 目录下，并改名为 designer-qt2。

```
$ sudo cp /work/GUI/qtopia/qtopia-free-2.2.0/qt2/bin/designer /usr/local/bin/
designer-qt2
```

在桌面控制终端下执行“designer-qt2 &”启动它，然后使用它打开 helloworldbase.ui 文件，可以看到如图 25.2 所示的界面。

双击其中的文字，修改为“Hello, world!”，然后拖动边框改变它的样式，结果如图 25.3 所示。

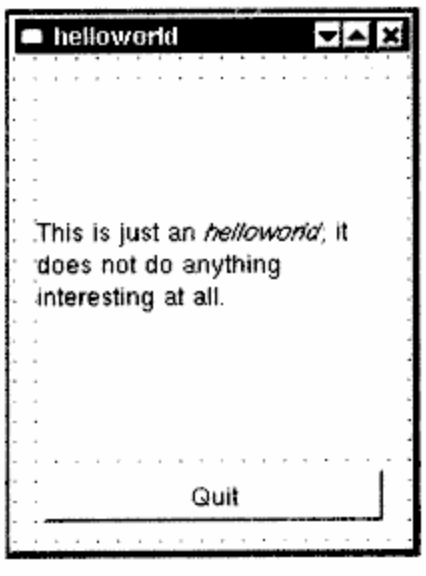


图 25.2　KDevelop 生成的 helloworld 工程的界面

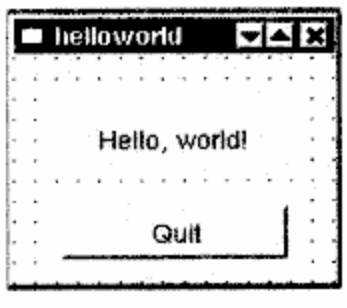


图 25.3　使用 designer-qt2 修改过的 helloworld 工程的界面

3. 使用 qmake 生成 Makefile 文件

qmake 工具的用法可以参考文档 dqt/doc/html/qmake-manual.html，它读取当前目录下的.pro 文件来产生 Makefile。

在执行它之前，需要指定 3 个环境。

（1）QTDIR：表示 Qt 的目录，即 qtopia-free-2.2.0 下的 qt2 目录。

（2）QPEDIR：表示 qpe（即 qtopia）的目录，即 qtopia-free-2.2.0 下的 qtopia 目录。

（3）QMAKESPEC：在 QMAKESPEC 指定的目录下有个 qmake.conf 文件，它定义了一些平台相关、编译器相关的信息，比如操作系统的类型、编译器的名称、编译选项、头文件目录、库目录等。qmake.conf 文件中，使用到了 QTDIR 和 QPEDIR 这两个环境变量。

本小节的目的是编译可以在 ARM 开发板上运行的 GUI 程序，在控制终端上，进入 helloworld 目录，执行以下命令：

```
$ export QTDIR=/work/GUI/qtopia/qtopia-free-2.2.0/qt2
$ export QPEDIR=/work/GUI/qtopia/qtopia-free-2.2.0/qtopia
$ export QMAKESPEC=/work/GUI/qtopia/qtopia-free-2.2.0/qtopia/mkspecs/qws/
linux-arm-g++
$ $QPEDIR/bin/qmake
```

qmake 将生成一个 Makefile 文件，在这个目录下直接执行“Make”命令就可以编译生成可执行程序 helloworld；也可以使用 Kdevelop 来编译程序。

可以在 Makefile 中看到，它会使用$（QTDIR）/bin/uic、$（QTDIR）/bin/moc 这两个工具来生成一些 C++源文件。要想进一步了解这两个工具，可以参考以下文档：

```
dqt/doc/html/uic.html
dqt/doc/html/moc.html
```

4．使用 Kdevelop 工具编译程序

从这时起，Kdevelop 只是一个“集成”工具，即它只是将其他工具集成起来，通过一个图形化的操作界面方便程序员使用而已。比如编译程序时，它仍依赖于工程中的 Makefile。

使用 Kdevelop 之前，也需要设置一些环境变量：编译程序时用到的环境变量属于被称为“Make Options”，与之对应的还有“Run Options”，它用来设置运行程序时的一些参数（由于 helloworld 工程为交叉编译，不能在主机上运行，“Run Options”的设置可以省略）。

“Make Options”位置如下，需要设置 Makefile 中用到的环境变量：

```
Project --> Project Options --> Make Options
```

本程序只需要设置 QTDIR、QPEDIR 两个环境变量，如图 25.4 所示。

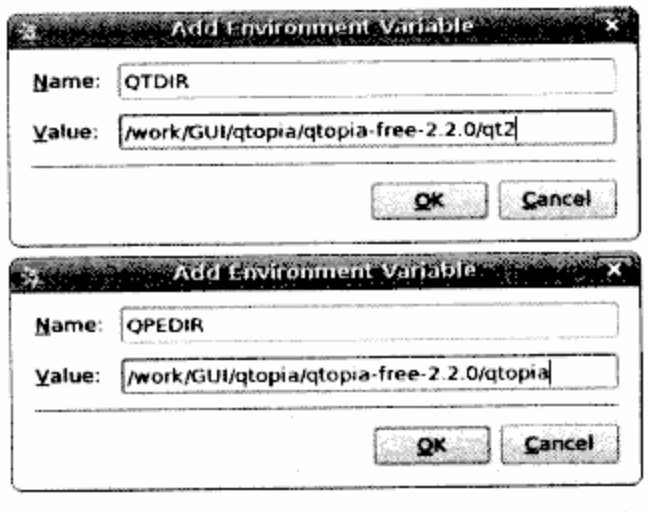


图 25.4　设置“Make Options”中的环境变量

现在编译程序很简单了，只需要按下“F8”键，或者在菜单“Build”中选择“Build Project”。在 Kdevelop 的消息窗口可以看到如下编译信息：

```
cd '/work/embedded_book_source/GUI/helloworld' && QPEDIR="/work/GUI/qtopia
/qtopia-free-2.2.0/qtopia" QTDIR="/work/GUI/qtopia/qtopia-free-2.2.0/qt2" make
generating .ui/release-shared/helloworldbase.h (uic)
compiling main.cpp (g++)
compiling helloworld.cpp (g++)
generating .ui/release-shared/helloworldbase.cpp (uic)
compiling helloworldbase.cpp (g++)
generating .moc/release-shared/moc_helloworld.cpp (moc)
compiling moc_helloworld.cpp (g++)
generating .moc/release-shared/moc_helloworldbase.cpp (moc)
compiling moc_helloworldbase.cpp (g++)
linking helloworld (g++)
*** Success ***
```

可见，它只是设置好了环境变量之后，再执行“make”命令而已。

5. 将可执行程序放到根文件系统中

本小节的目标是在 Qtopia 的桌面生成一个小图标，单击它时就运行 helloworld。这需要 3 个文件：图标文件、可执行程序、把它们两者联系起来的文件（称为桌面文件）。这 3 个文件在 helloworld 目录下都生成了，分别为：helloworld.png、helloworld 和 helloworld.desktop，只要将它们放入开发板根文件系统中相应的目录中即可。

（1）图标文件 helloworld.png 放到/opt/Qtopia/pics/中。

（2）可执行程序 helloworld 放到/opt/Qtopia/bin/中。

（3）桌面文件 helloworld.desktop 放到/opt/Qtopia/apps/Applications/中。

这 3 种文件之间的关系、桌面文件的格式，可以参考文档 qtopia/doc/html/files.html。

重启系统就可以看到桌面上多了一个名为“helloworld”的图标，单击它将得到与图 25.2 类似的界面。

25.2.4 在主机上使用模拟软件开发、调试嵌入式 Qt GUI 程序

Qtopia 中提供了一个工具 qvfb 来模拟实际的 Frame Buffer 设备，这使得可以在主机上运行为嵌入式设备开发的 GUI 程序，极大地便利了开发、调试工作。

与前两节的内容相似，要在主机上开发、运行“嵌入式 GUI 程序”也要完成以下两件事。

（1）编译、安装 Qtopia 2.2.0。

（2）开发自己的 Qt GUI 程序。

其中详细的过程与前两者完全对应，只是在对各工具的配置不同。换个角度来说，x86 也是一种“嵌入式处理器”，编译、安装、运行程序时与 ARM 处理器并无特别之处。

下面简要讲述这些过程。

1. 编译、安装 Qtopia 2.2.0

用于主机所的 qtopia-free-2.2.0 代码与用于 ARM 开发板上的代码完全一样，前面对它做的修改完全适用于主机，所以下面不再讲述对代码的修改。

（1）编译、安装 Qtopia 所依赖的库。

有两个库：uuid 和 jped，下面只给出执行的命令，不再解释过程。

① uuid 库。

```
$ cd /work/tools/e2fsprogs-1.40.2/
$ mkdir build_x86; cd build_x86
$ ../configure --enable-elf-shlibs
$ make
$ sudo make install-libs
```

② jpeg 库。

```
$ tar xzf jpegsrc.v6b.tar.gz        // 重新解压
$ cd jpeg-6b
$ ./configure --enable-shared --enable-static --prefix=/usr
$ make
$ sudo make install-lib
```

这些命令最终会在/usr/lib、/usr/include 目录下安装库文件、头文件。

（2）配置、编译、安装 qtopia。

将源文件/work/GUI/qtopia-free-src-2.2.0.tar.gz 解压所得目录命名为 qtopia-free-2.2.0_x86，然后按照进 25.2.2 小节的方法修改它。或者，也可以使用补丁文件 qtopia-free-2.2.0_100ask.patch，执行以下命令：

```
$ cd qtopia-free-2.2.0_x86
$ patch -p1 < ../qtopia-free-2.2.0_100ask.patch
```

然后，执行以下配置、编译、安装命令：

```
$ ./configure -qte '-embedded -no-xft -qconfig qpe -depths 16,32 -system-jpeg
-gif'-qpe'-edition pda -displaysize 240x320'-qt2'-no-xft'-dqt'-no-xft'
$ make
$ make install
```

配置项与前面在移植 ARM 平台时的相比，少了“-xplatform linux-arm-g++”、“-no-qvfb”选项，后者表示支持 qvfb。

这些命令将在 qtopia-free-2.2.0_x86/qtopia/image/目录下生成一个 opt 目录，里面存放了所有运行 qtopia 所必需的文件，比如库文件、可执行程序、图标等（除了字库）。字库在 qtopia-free-2.2.0_x86/qt2/lib/fonts/目录下，使用以下命令复制它：

```
$ cp -rf qt2/lib/fonts qtopia/image/opt/Qtopia/lib/
```

以后运行、调试程序时，都是基于 qtopia-free-2.2.0_x86/qtopia/image/opt/Qtopia 目录。

2. 在qvfb上启动qtopia

在编译qtopia的过程中，qvfb工具已经在qtopia-free-2.2.0_x86/qt2/bin/目录下生成了。为方便使用，可以将它复制到/usr/local/bin目录下。

在桌面控制终端下使用以下命令启动它，可以看见如图25.5左边的界面。

```
$ sudo cp qt2/bin/qvfb  /usr/local/bin/
$ qvfb &
```

图25.3右侧界面所示的虚拟frame buffer中什么都没有，因为还没有启动Qtopia。与在ARM开发板上启动Qtopia相似，也需要设置一些环境变量。为方便使用，可以建立一个脚本，称为qpe_x86.sh，内容如下：

```
#!/bin/sh
export HOME=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/image/root
export QTDIR=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/image/opt/Qtopia
export QPEDIR=$QTDIR
export PATH=$QPEDIR/bin:$PATH
export LD_LIBRARY_PATH=$QPEDIR/lib:$LD_LIBRARY_PATH
$QPEDIR/bin/qpe &
```

① qvfb程序必须先运行；qpe_x86.sh里设置的HOME环境变量可以设为其他目录，这个目录必须存在。第一次运行时，使用以下命令建立这个目录：

```
$ mkdir /work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/image/root
```

注意

② 其中QTDIR、QPEDIR、LD_LIBRARY_PATH表示的目录都在“image”目录中，这个目录是在编译qtopia的最后执行“make install”命令时生成的，它就是最终的结果。

然后，修改脚本的属性并执行（在桌面控制终端下执行）。

```
$ chmod +x  qpe_x86.sh
$ ./qpe_x86.sh
```

就可以看见如图25.5中间的界面了，然后使用鼠标点击桌面，按照提示设置语言、时区、时间后就可以得到图25.5右边的界面，这时可以在里面启动各个程序。

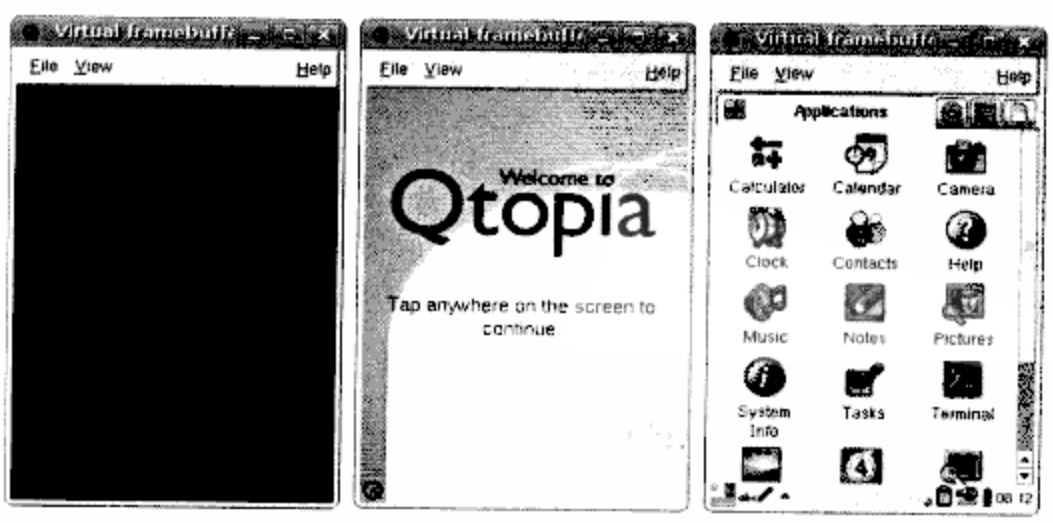


图25.5　在qvfb上运行Qtopia时的启动界面

3. 使用 Kdevelop 开发、编译、运行、调试 Qt GUI 程序

依照 25.2.3 小节的方法建立“helloworld_x86”工程，它的建立、编译方法与“helloworld”工程基本一样，只是编译、运行时要设置的环境变量不一样，下面简单讲述这些过程，本节将重点放在 Kdevelop 的使用上。

（1）建立、编译“helloworld_x86”工程。

建立 helloworld_x86 工程后，使用 qmake 生成 Makefile。进入 helloworld_x86 目录，执行以下命令：

```
$ export QTDIR=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qt2
$ export QPEDIR=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia
$ export QMAKESPEC=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/mkspecs/qws/
linux-x86-g++
$ $QPEDIR/bin/qmake
```

然后修改 Kdevelop 中的“Project --> Project Options --> Make Options”，将 QTDIR、QPEDIR 设为上面代码中的值。

现在可以在 Kdevelop 中，按下“F8”键编译程序了。

（2）运行 helloworld_x86。

启动方法有两种：使用命令行手工启动、通过 Kdevelop 启动。

① 手工启动 helloworld_x86。

其步骤与上面在 qvfb 上启动 Qtopia 是一样的，先运行 qvfb，然后设置环境变量，最后运行 helloworld_x86。为方便，后两个步骤写入一个脚本文件中，名字为 helloworld_x86.sh，内容如下（它与 qpe_x86.sh 内容相似）：

```
#!/bin/sh
export HOME=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/image/root
export
QTDIR=/work/GUI/qtopia/qtopia-free-2.2.0_x86/qtopia/image/opt/Qtopia
export QPEDIR=$QTDIR
export PATH=$QPEDIR/bin
export LD_LIBRARY_PATH=$QPEDIR/lib
./helloworld_x86 -qws &
```

要单独运行某个 Qt 程序，需要使用“-qws”把它作为一个服务来启动，还要修改脚本，如下所示：

```
$ chmod +x ./helloworld_x86.sh
```

在 qvfb 上启动 helloworld 的命令如下：

```
$ qvfb &
$ ./helloworld_x86.sh
```

这时可以看见如图25.6所示的界面。

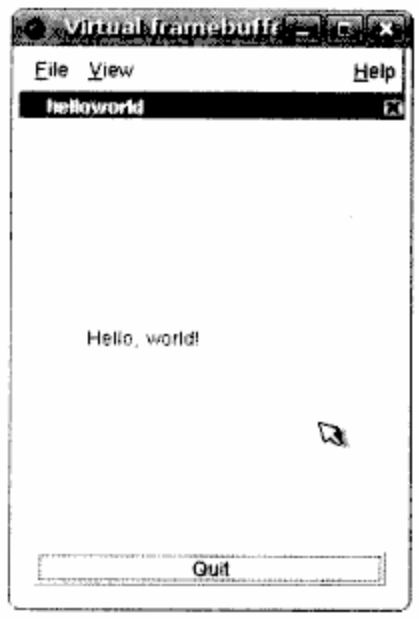


图25.6　在qvfb上运行helloworld时的界面

② 通过Kdevelop启动helloworld。

这需要设置“Run Options”，即运行程序时的参数。“Run Options”位置如下：

```
Project --> Project Options --> Run Options
```

需要在“Main Program”中指定“Executable”（运行的程序，设为/work/GUI/qtopia/helloworld_x86/helloworld_x86）、“Run Arguments”（运行参数，设为-qws），在“Environment Variables”中按照helloworld_x86.sh文件设置环境变量（注意：所有环境变量都必须展开，即它们的值不能用其他变量来定义），如图25.7所示。

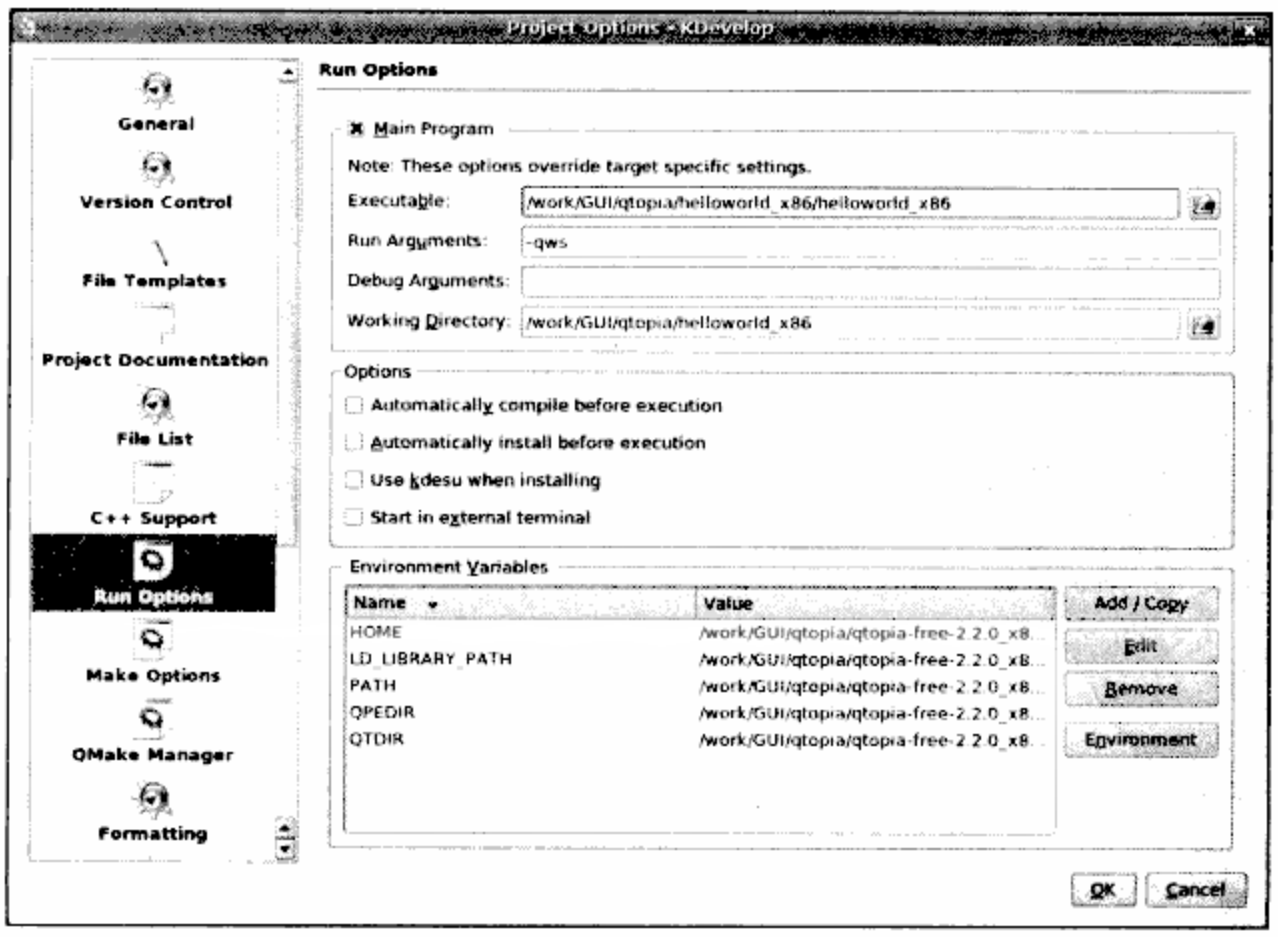


图25.7　设置“Run Options”

与手动启动程序类似，也要先在控制终端中启动qvfb。然后就可以在Kdevelop中运行

helloworld_x86，启动方法：使用快捷键“Shift + F9”，或者使用菜单“Build --> Execute Main Program”。

（3）调试“helloworld_x86”工程。

使用 Kdevelop 来调试程序并不方便，可以使用 DDD 和 GDB 来调试。

首先修改工程文件 helloworld_x86.pro，将“warn_on release”改为“warn_on debug”，如下：

```
#CONFIG          = qt warn_on release
CONFIG          = qt warn_on debug
```

然后重新生成 Makefile，方法与前面一样。不同的 helloworld_x86.pro 文件，产生不同的 Makefile。生成 Makefile 之后，执行“make distclean”清除以前的编译结果，再执行“make”生成带调试信息的可执行程序 helloworld_x86。

最后，使用 DDD 调用 GDB，启动 helloworld_x86，方法如下。

① 修改 helloworld_x86.sh 最后一行。

```
./helloworld_x86 -qws &
改为:
ddd gdb --args ./helloworld_x86 -qws &
```

② 启动 qvfb 后，在 helloworld_x86 目录下运行 helloworld_x86.sh。

```
$ qvfb &
$ ./helloworld_x86.sh
```

程序启动后，就可以看到类似图 25.8 的调试界面，可以在里面设置断点、单步执行、查看结果等。

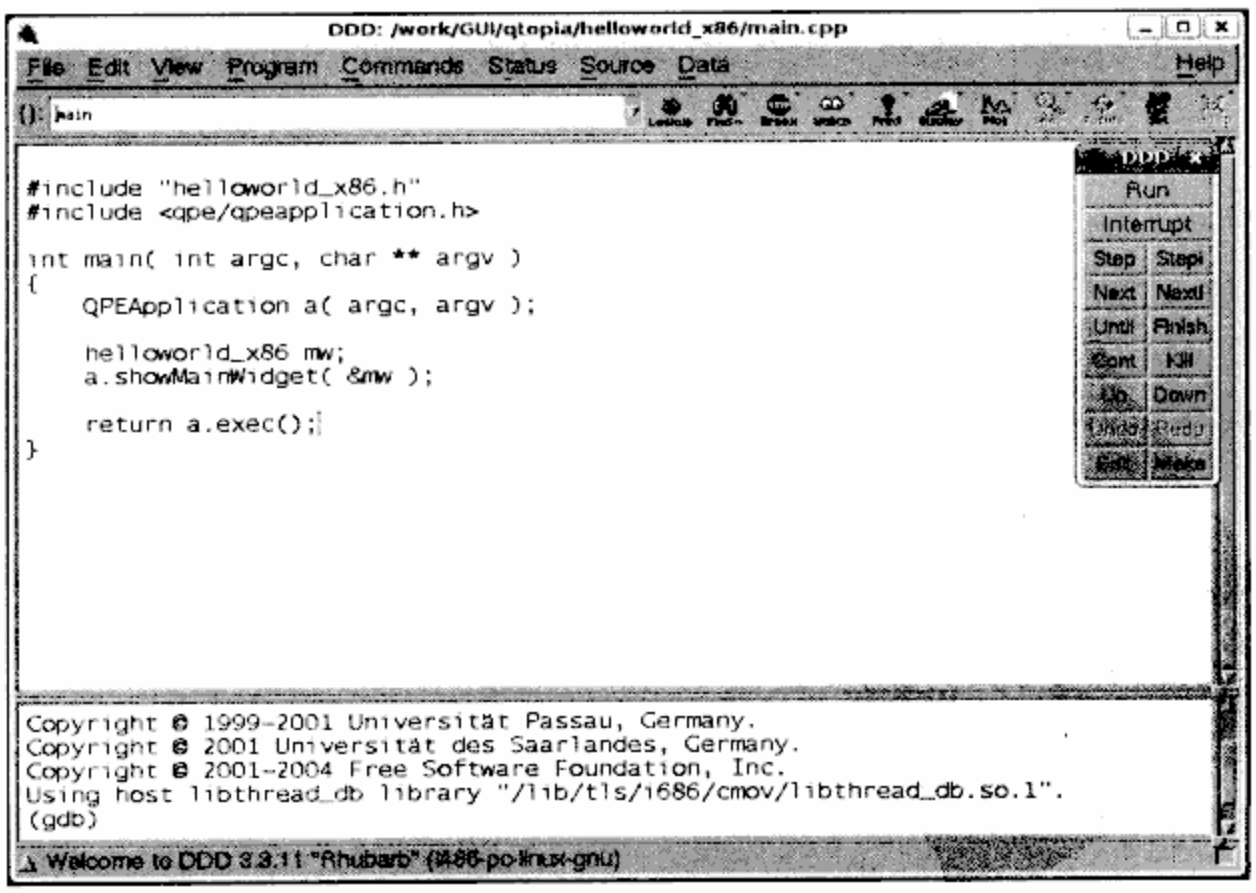
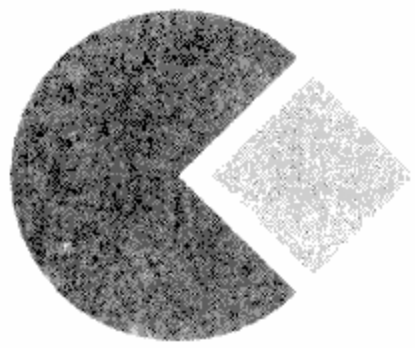


图 25.8 使用 DDD 调试 helloworld_x86



第 26 章　基于 X 的 GUI 开发

本章目标

- 理解 X Window 中各部件的作用
- 掌握搭建基于 X 的 GUI 系统的方法，移植一个桌面环境
- 掌握移植 GUI 程序的方法
- 掌握裁剪文件系统的方法

26.1　X Window 概述

在嵌入式 GUI 领域，以前人们都倾向于跨过 X，比如上节移植的 Qtopia，还有基于 GTK 的 GPE 等，它们都是直接操作硬件，这主要是基于两个方面的考虑：X 本身就很大，运行时消耗资源多。但是自从 KDrive（也叫 Tiny X、TinyX）出现，并且随着嵌入式设备的存储容量越来越大、CPU 运算速度的进一步提高，现在很多公司开始基于 X 设计它们的 GUI 系统，因为基于 X 的资源“极大”非富，很多软件都是基于 X 的。

KDrive 也被称为 TinyX，是由 XFree86 项目的核心开发人员 Keith Packard 针对内存很小的应用环境开发的 X server，它不是一个单独的项目，而是对标准 X server 的裁剪、配置。在 PC Linux 中，支持渲染（RENDER）、不支持字体缩放的 KDrive 最小可以达到 700KB，这非常适用于嵌入式领域。

本节将介绍基于 X 的 GUI 系统移植，目标是得到一个在单板上运行的桌面系统。

26.1.1　X 协议介绍

与 Windows 不同，Linux 的图形界面完全在用户态实现：窗口的管理（叠加、最大化、最小化）、桌面、文件浏览器等，都对应某一个应用程序。大多数的 PC Linux 都是基于 X 协议来实现图形系统。

X 指的是一种协议，在讲述它之前先回顾前面测试 LCD 所用的程序 fb_test，它的用法如下：

```
$ fb_test /dev/fb0
```

它调用 mmap 函数获得 LCD 的 Frame Buffer 地址后，直接在里面填充数据以显示图像。

但是当有多个程序需要操作 LCD 时，每个程序都直接操作硬件的方法并不可取。必须提供一种机制来串行化各个程序对 LCD 的访问，否则屏幕会将被画得一团糟。对键盘、鼠标等设备也有同样的问题。

X 协议就是用来处理这类问题的，它规定了这样一种操作方式：屏幕、键盘、鼠标等输入/输出设备由一个名为 X server 的程序来统一管理，其他的应用程序被称为 X client。比如当应用程序需要画一个圆时，它向 X server 提出请求：请在（x, y）坐标处绘制一个半径为 r 的红色的圆；剩下的工作就由 X server 操作具体硬件来完成了。X server 还负责捕捉键盘、鼠标的输入事件。比如当鼠标的左键被按下时，X server 会告诉“对这类事件感兴趣的” X client 程序：鼠标左键被按下了，请处理。

X server 和 X client 之间通过一定的协议进行通信，这也意味着它们可以位于不同的机器、不同的操作系统。实际上，在远程桌面等技术出现之前，在 UNIX、Linux 之类的系统中就可以通过 X 进行远程操作了。需要注意的是，X 中 server 和 client 的位置与传统的服务器、客户端刚好相反。X server 和 X client 指的是程序，传统的服务器、客户端指的是机器。以一个例子来说明：我们的个人电脑为 Windows 机器，通过 Xmanage 等远程控制工具登录到 Linux 服务器上去后，可以在 Xmanage 中指示 Linux 服务器执行一个程序，比如文本编辑器 Gedit。这时，登录工具 Xmanage 被称为 X server，而在 Linux 服务器上运行的 Gedit 程序反倒被称为 X client。

X 既然只是一种协议，那么它的代码实现就可以有多种方式，现在流行的有 XFree86、Xorg 两种。它们的关系如下。

（1）XFree86。

- XFree86 是由 X11R6 发展出来的最初专门给 Intel X86 结构 PC 机使用的 X Window 的系统。
- 而后 XFree86 发展成为几乎适用于所有类 UNIX 操作系统的 X Window 系统。
- XFree86 是一个开放源代码的基于 X11 的桌面基础构架。
- Red Hat 9 中使用的 X Window 系统就是 XFree86 4.3。
- XFree86 从 2004 年发布的版本 4.4 起不再遵从 GPL 许可证发行，而是遵循新的 XFree86 1.1 许可证。
- 由于 XFree86 不再遵从 GPL 许可证发行，导致许多发行套件不再使用 XFree86，转而使用 Xorg。

（2）Xorg。

- Xorg 是由 X.Org 基金会发行的开放源代码 X Window 系统实现的 X 服务。
- Xorg 遵从 GPL 许可证发行。
- Xorg 基于 XFree86 4.4RC2 和 X11R6.6 的代码。
- X.Org 基金会在 2004 年 4 月发布了 X11R6.7。
- 在 2005 年 2 月发布了 X11R6.8.2
- 在 2007 年 9 月发布了 X11R7.3

基于 X 实现的 GUI 系统就称为 X Window System，简称为 X Window 或 X。也常把 X server（这里指一个程序）及它提供的服务简称为 X。通过 X，使得图形应用程序不需要关心硬件的细节，它们只需要告诉 X 怎样显示即可，X 监听应用程序的请求，并将这些指示转换为实际的硬件操作。但是，X 并不控制这些应用程序在屏幕上的显示位置和显示内容。

26.1.2 窗口管理器（Window manager）

图形程序常常有一个矩形状的外观，称之为窗口（Window）。由于X只给图形程序提供了显示的硬件实现，所以需要额外的程序来管理窗口，这个程序被称为窗口管理器（Window manager）。它也只是一个普通的应用程序，特殊之处在于它是用来管理其他窗口的。

窗口管理器控制着屏幕上的窗口：它们的样式及操作。它决定窗口的边框样式，比如最大、最小、关闭这3个常见的按钮就是窗口管理器提供的。通过窗口管理器，可以对窗口进行各种操作，比如移动、隐藏、改变大小、关闭等。它控制着当前哪个窗口可以接收键盘、鼠标的输入，哪个窗口处于显示器的最上层。它还控制着进行上述操作的方式：使用鼠标左键还是右键、可以使用哪些快捷键等。

除上述功能外，有些窗口管理器还提供了额外的功能。比如提供一个或多个菜单，使得用户可以通过它们来启动程序；提供虚拟桌面（virtual desktops），有多个桌面，可以在它们之间进行切换，这就像切换应用程序一样，只不过它是切换“整个桌面”；有些窗口管理器还提供了图形界面的配置工具。

26.1.3 桌面环境（Desktop environment）

通过窗口管理器，已经可以在一个桌面进行各类日常工作了，对太多数用户来说，这已经足够了。但是如果想要更多的功能，比如想在桌面上放置一样图标，单击它就可以启动程序等，这不是窗口管理器的职责。这时候，需要一个桌面管理器（Desktop manager），或称之为桌面环境（Desktop environment）。

就如它的名字所示，桌面管理器管理整个桌面，但是它不直接处理窗口。比如桌面管理器在桌面上提供一个或多个任务栏、额外的菜单、图标、其他各种工具（文件管理器、查找工具、文件编辑器）的快捷方式（借用Windows的概念）等。

在PC Linux中，主流的两个桌面管理器是KDE（K Desktop Environment）和GNOME（GNU Network Object Model Environment）。它们有很多不同，但是有一点是相同的：它们都必须使用一个窗口管理器。桌面管理器管理整个桌面，但是对窗口的控制仍由窗口管理器来完成。KDE里已经集成了一个窗口管理器，而GNOME使用其他窗口管理器（这使得我们可以轻松更换其他窗口管理器）。

X、窗口管理器、桌面管理器，这3个概念在后面的移植过程中，读者会得到深入而形象理解。

26.2 交叉编译工具包Scratchbox

在UNIX系统上，知名的“make”工具可以协助开发程序，但是随着程序开发复杂度的提升，已经很难用有限的“make rules”来满足多变的需求，所以Cygnus/RedHat的Tom Tromey就设计了autoconf与automake等工具，期望大幅降低异质性平台开发的困难。这也包含所谓的cross-compile，为了克服不同平台的函数库编译落差，libtool也被提出，立意甚好但往往让我们遇到不少问题，比方说知名的hacker-Casey Marshall就在与这些上万行的工具程序奋战一段时间后，写了篇短文“Avoiding libtool minefields when cross-compiling”。autoconf与

automake 等工具最核心的想法就是希望编译过程可以简化成如下：

```
./configure --host=arm-linux \
--prefix=/usr \
--enable-shared \
--enable-Feature1 \
--disable-Feature2
```

在前面编译各种程序时，本书也都是使用这种方式。但是在更大型的软件、更复杂的编译脚本中，这种方法显得不足：如果脚本没有为交叉编译考虑周全，就需要进行大量修改。

人们提出了各种解决方法，比如知名的 OpenEmbedded 项目、Scratchbox 项目等。本书使用后者。

26.2.1 Scratchbox 介绍

现在的很多程序包里并没有 Makefile，而是先通过配置命令自动产生，然后才能执行“make”命令进行编译。大多数程序的默认平台为 x86，在主机上编译程序时通常执行以下 3 个命令即可。

```
$ ./configure
$ make
$ make install
```

但是进行交叉编译时，需要更改配置参数，比如指定目标机器使得生成的 Makefile 中使用交叉编译工具，指定最终结果存放的地址以免覆盖主机的文件导致系统崩溃。常用的配置命令格式如下：

```
$ ./configure -host=arm-linux --prefix=/other/install/dir
```

如果配置文件本身有缺陷，导致生成的 Makefile 文件没有使用交叉编译工具，还需要手工修改，这在前面编译 zlib 库时碰到过。另外，在编译某些程序时，它会用到自身生成的工具，但是这些经过交叉编译得来的工具并不能在主机上运行，这需要给配置文件打补丁，在编译这个工具时使用本地编译器。

对于小型软件，或是对于从一开始就为交叉编译考虑周全的软件，这类修改很少或几乎没有。但是对于大型软件，比如 X，它最开始是为 x86 的机器设计的，要对它进行交叉编译就非常困难。X 的最近版本已经考虑了交叉编译，但是仍需要少量修改。即使如此，由于最开始时并不知道需要修改什么地方，只好先编译、等待出错、修改、再编译，如此反复才能最终编译成功，这使得效率极其低下。

问题的根源在于存在两套编译工具链，想象一下，如果开发板资源足够丰富、运算速度足够快，就可以在上面编译程序，这将不存在“交叉编译”。嵌入式硬件的性能还没有达到这个地步，但是可以在主机进行模拟，这就是 Scratchbox 的由来。

Scratchbox 是一个为编译嵌入式 Linux 软件提供便利的系统，它有以下有两个主要的功能。

（1）包装交叉编译工具链，以“本地工具链”的形式运行。

比如在 Scratchbox 中执行以下编译命令：

```
> gcc -o test hello.c
```

它与在主机上执行以下命令的结果是一样的：

```
$ arm-linux-gcc -o test hello.c
```

（2）模拟目标机的指令，使得它们可以在主机上运行。

上面两个命令产生的 ARM 程序，可以在 Scratchbox 中运行（但是不能在主机中直接运行），就好像在目标机上运行一样。

26.2.2 安装 Scratchbox 及编译工具

再次约定：在主机上执行的命令，提示符为“$”；在主机中启动 Scratchbox，然后在 Scratchbox 里执行的命令，提示符为“>”；在单板上执行的命令，提示符为“#”。比如同是执行“ls”命令，用下面 3 行表示：

```
$ ls
> ls
# ls
```

从本节开始，在主机上要用到两个终端，一个用来直接操作主机，一个用来启动 Scratchbox，然后就在 Scratchbox 环境里进行操作。

1. 安装 Scratchbox

这包含两部分内容：安装 Scratchbox 本身，安装编译工具链。第一部分很简单，将/work/scratchbox 目录下的所有*.tar.gz 文件解压到/scratchbox 目录下即可。为了保持所有代码都存放在同一个分区里，将/scratchbox 设为到/work/scratchbox 的连接。执行以下命令进行安装：

```
$ cd /
$ sudo ln -s /work/scratchbox scratchbox
$ cd /scratchbox
$ for f in $(ls *.tar.gz); do tar xzf $f -C /; done
```

这些压缩文件分为以下两部分。

① Scratchbox 的核心文件，它们构建了 Scratchbox 的基本运行系统，核心文件。

```
scratchbox-core-1.0.8-i386.tar.gz
scratchbox-devkit-cputransp-1.0.3-i386.tar.gz
scratchbox-devkit-debian-1.0.9-i386.tar.gz
scratchbox-devkit-doctools-1.0.7-i386.tar.gz
scratchbox-devkit-perl-1.0.4-i386.tar.gz
scratchbox-libs-1.0.8-i386.tar.gz
```

② 编译工具。

```
scratchbox-toolchain-host-gcc-1.0.8-i386.tar.gz
```

这个文件被解压缩后，将在/scratchbox/compilers/目录下生成一个子目录 host-gcc，它表示“主机编译工具链”。在 Scratchbox 里面可以选择各种编译工具链，比如“host-gcc”或其

他交叉编译工具链，使用 gcc 等命令时，实际执行的是这些被选择的工具链中的相应命令。

Scratchbox、Ubuntu 7.10 使用的 glibc 库版本不同，这导致第 2 章制作的工具链 arm-linux-gcc-3.4.5-glibc-2.3.6.tar.bz2 在 Scratchbox 里不能使用，需要在 Scratchbox 里使用“host-gcc”重新制作。

2. 运行 Scratchbox

第一次运行前先进行一些设置，执行以下命令，使用默认设置即可：

```
$ sudo /scratchbox/run_me_first.sh
$ sudo /scratchbox/sbin/sbox_adduser book
```

这时候运行 groups 命令，如果看到了 sbox 组名，就可以使用 Scratchbox 了。否则先退出系统，重新登录即可。

注意 这时候 Scratchbox 已经启动，执行“/scratchbox/login”即可登录 Scratchbox。

以后启动 PC 时，要执行以下命令才能启动、登录 scratchbox：

```
$ sudo /scratchbox/sbin/sbox_ctl start
$ /scratchbox/login
```

登录 Scratchbox 后，可以看到如下信息：

```
book@100ask:/$ /scratchbox/login

You dont have active target in scratchbox chroot.
Please create one by running "sb-menu" before continuing
Welcome to Scratchbox, the cross-compilation toolkit!

Use 'sb-menu' to change your compilation target.
See /scratchbox/doc/ for documentation.

sb-conf: No current target
[sbox-: ~] >
```

Scratchbox 是一个改变根文件系统（chroot）的 Linux 系统，进入 Scratchbox 后，它的根文件系统位于原来主机的/scratchbox/users/<username>目录下。比如对于用户名 book，它在 Scratchbox 里的根文件系统就位于原来主机的/scratchbox/users/boot 目录下。

现在可以执行 sb-menu 命令设置目标（target），这将在后面实际使用时介绍。

26.2.3 在 Scratchbox 里安装交叉编译工具链

1. 使用制作好的工具链

可以使用/work/tools/目录下已经编译好的工具链：scratchbox-arm- linux-gcc-3.4.5-

glibc-2.3.6.tar.gz。使用以下命令解压即可。

```
$ sudo chown book:sbox /scratchbox/compilers -R
$ tar xzf scratchbox-arm-linux-gcc-3.4.5-glibc-2.3.6.tar.gz -C /
```

2. 自己制作工具链

也可以自己制作交叉编译工具链：启动 Scratchbox 后，选择 host-gcc 作为编译器，然后使用与第二章相似的方法通过 crosstool 来制作交叉编译工具链，然后稍做修改即可。下面分步介绍。

（1）使用 sb-menu 命令创建一个目标（target），选择选择 host-gcc 作为编译器。

依照以下步骤进行设置。

① 执行“/scratchbox/login”命令进入 Scratchbox 后，再执行 sb-menu 命令。

```
[sbox-: ~] > sb-menu
```

将会出现如图 26.1 所示的主菜单。

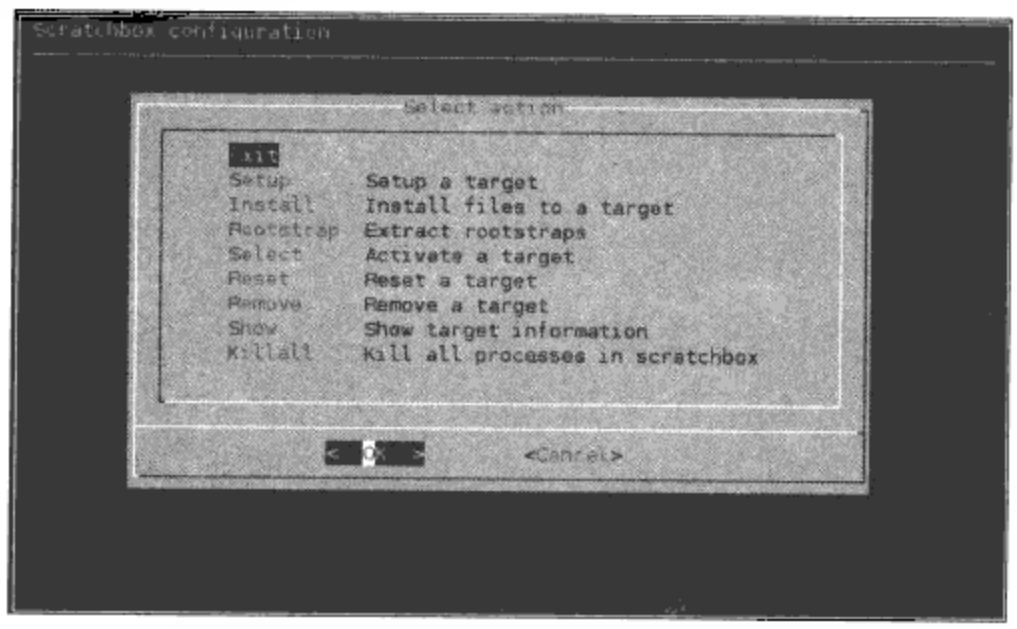


图 26.1　Scratchbox 的主菜单

② 选择其中的“Setup”项，进入下一级菜单，如图 26.2 所示。

图 26.2　创建新目标（target）

③ 第一次使用时，选择“Create a new target”，出现一个提示框，输入这个新目标的名

字，取为 x86，如图 26.3 所示。

④ 选择“host-gcc”作为编译器，目前也只有一个编译器可供选择，如图 26.4 所示。

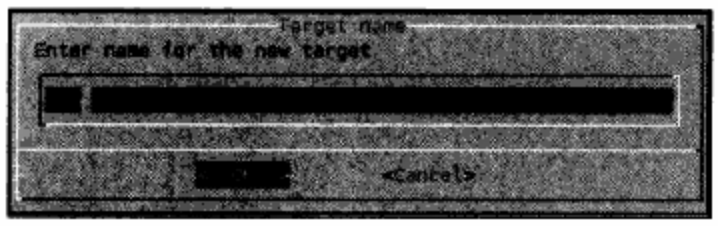


图 26.3 目标名称

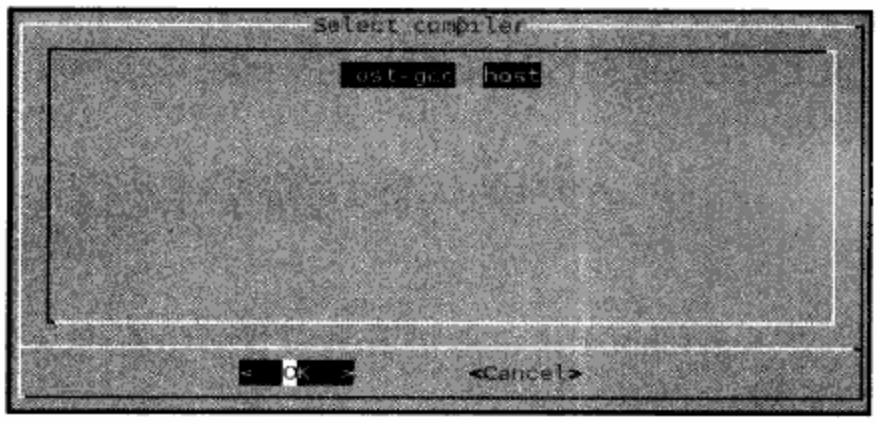


图 26.4 选择编译器

⑤ 选择开发工具，除“cputransp”（host-gcc 是主机编译器，不需要模拟）外，其他的都选上，如图 26.5 所示。

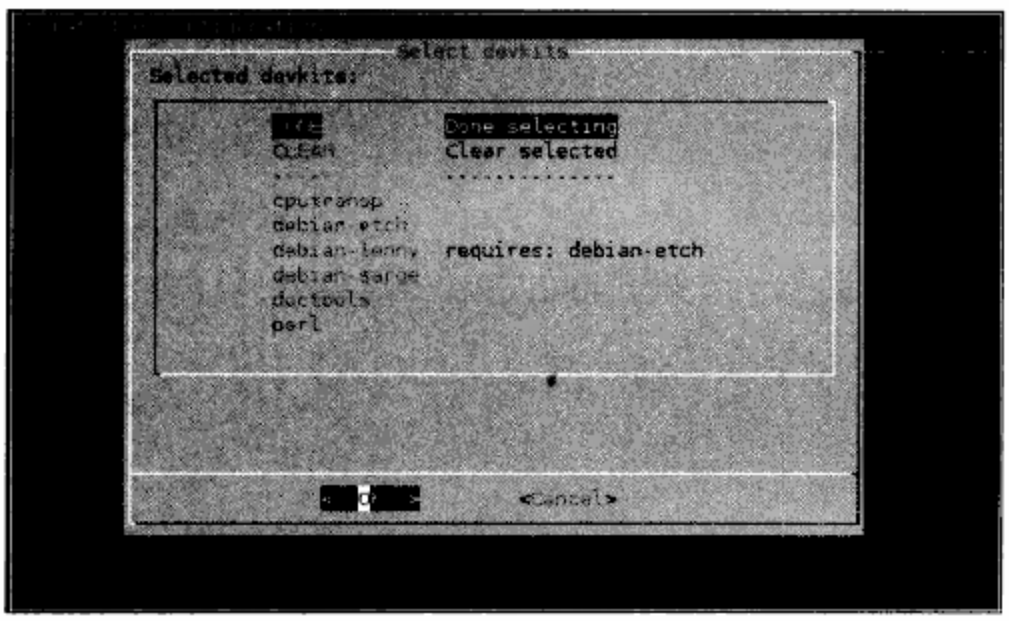
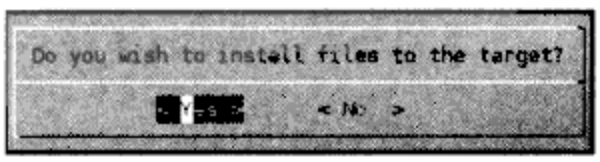


图 26.5 选择开发工具

选择完毕后，选择“Done selecting”进入下一级菜单“Select CPU-transparency method”。由于刚才没有选择“cputransp”，所以这个菜单中没有东西可选，跳过进入后一个菜单。

⑥ 选择“Yes”，表示安装刚才选择的开发工具，在随后出现的菜单中，使用空格键把“/etc”、“Devkits”都选上，然后按回车键，如图 26.6 所示。



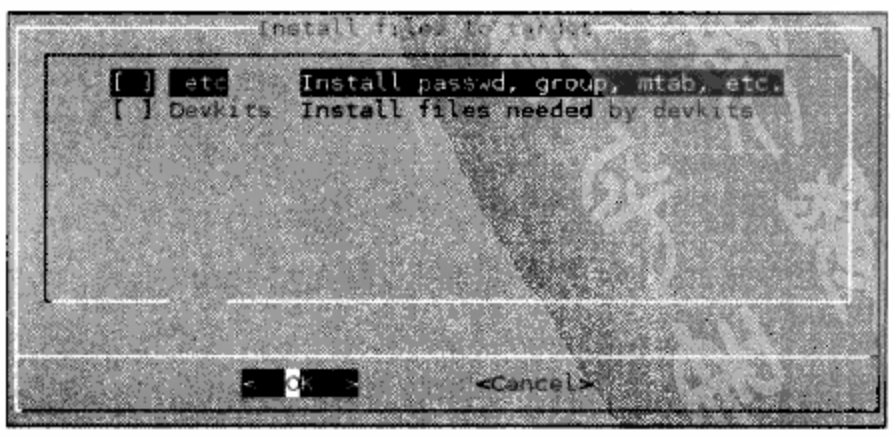


图 26.6 安装文件

⑦ 所有的准备工作已经完成，出现最后一个菜单，选择“Yes”使用这个新目标，如图 26.7 所示。

（2）修改 crosstool 脚本，编译交叉工具链。

为与前面的章节目录保持一致，先在/scratchbox/users/book 目录下建一个 work 目录，然后将工作分区挂接到这个目录上（可以先执行“cat /proc/mounts”确定挂接哪个设备）。执行以下命令：

图 26.7 选择使用新目标

```
$ mkdir -p /scratchbox/users/book/work
$ sudo mount /dev/sdb1 /scratchbox/users/book/work
```

在 Scratchbox 里，即可看到根目录下有个 work 子目录：

```
[sbox-x86: ~] > ls /work
```

在进行后续操作前，先保证 book 用户有权限读写/scratchbox/compilers 目录：

```
$ sudo chown book:sbox /scratchbox/compilers -R
```

编译工具链的步骤如下所示。

① 解压。

```
[sbox-x86: ~] > cd /work/tools/create_crosstools/
[sbox-x86: /work/tools/create_crosstools] > tar xzf crosstool-0.43.tar.gz
```

② 复制补丁，比第 2 章多一个补丁 ld-2.15-scratchbox_NATIVE.patch。

```
[sbox-x86: /work/tools/create_crosstools] > cp glibc-2.3.6-version-info.
h_err.patch crosstool-0.43/patches/glibc-2.3.6/
[sbox-x86: /work/tools/create_crosstools] > cp ld-2.15-scratchbox_NATIVE.
patch crosstool-0.43/patches/binutils-2.15
```

③ 修改 crosstool 脚本。

进入 crosstool-0.43 目录，需要修改 3 个文件：demo-arm-softfloat.sh、arm-softfloat.dat、all.sh。

- 修改 demo-arm-softfloat.sh，修改第 7、8 两行。

```
07 TARBALLS_DIR=/work/tools/create_crosstools/src_gcc_glibc
08 RESULT_TOP=/scratchbox/compilers
```

- 修改 arm-softfloat.dat。

```
02 TARGET=arm-softfloat-linux-gnu
改为:
02 TARGET=arm-linux
```

它表示编译出来的工具样式为 arm-linux-gcc、arm-linux-ld 等，这是常用的名字。

- 修改 all.sh。

修改 PREFIX，将结果存放在/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6 目录下。

```
70 PREFIX=${PREFIX-$RESULT_TOP/$TOOLCOMBO/$TARGET}
改为:
70 PREFIX=${PREFIX-$RESULT_TOP/$TOOLCOMBO}
```

④ 最后，在 crosstool-0.43 目录下执行以下命令进行编译。

```
> unset LD_LIBRARY_PATH                    /* 使用 crosstool 制作工具链时，不能设置
                                            LD_LIBRARY_PATH */
> ./demo-arm-softfloat.sh
```

（3）设置新建的交叉工具链。

还需要进行一些设置才能在 Scratchbox 中以 gcc、ld 等命令直接调用刚才生成的工具链。

① 建立编译工具的连接。

```
[sbox-x86: /] > cd /scratchbox/compilers/gcc-3.4.5-glibc-2.3.6/bin/
[sbox-x86: /scratchbox/compilers/gcc-3.4.5-glibc-2.3.6/bin] > for f in 'ls';
do ln -s $f sbox-$f; done
```

② 在/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6 目录下建立一个名为 compiler-name 的文件，内容为：

```
gcc-3.4.5-glibc-2.3.6:/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6:arm:lin
ux:glibc:arm
```

这些内容以“:”号分隔，它们的意义为。

- 编译器的名字，必须与 compiler-name 文件所在的目录名相同。
- 编译器的绝对路径。
- 目标板的处理器架构。
- sub-arch。
- 使用的 C 库是 glibc 还是 uclibc 。
- 目标板的处理器的模拟器。

注意 编译器的绝对路径指定了转换的路径，目本主板的处理器架构和 sub-arch 合起来再加上前缀“sbox-”就是：sbox-arm-linux。

所以，/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6/bin/sbox-arm-linux-gcc 就是在 Scratchbox 中执行 gcc 后，真正执行的程序，它是 arm-linux-gcc 的连接。

③ 在/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6 目录下建立一个名为 target_setup.sh 的文件，内容为：

```
#!/bin/sh

target=$1
source=/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6/arm-linux

mkdir -p $target/lib
mkdir -p $target/usr/lib
mkdir -p $target/usr/include
mkdir -p $target/usr/bin
```

```
cp -af $source/lib/*     $target/lib/
cp -af $source/include/*  $target/usr/include/

ln -sf /bin/cpp $target/lib/
ln -sf /bin/cc  $target/usr/bin/

chmod +x $target/lib/ld-*.so
```

在创建使用 gcc-3.4.5-glibc-2.3.6 编译器的目标时，这个脚本被用来复制库文件、头文件等。

④ 在/scratchbox/compilers/gcc-3.4.5-glibc-2.3.6 目录下建立 gcc.specs 文件：

```
*cross_compile:
0
%rename cpp old_cpp
*cpp:
-isystem /usr/local/include -isystem /usr/include %(old_cpp)
```

（4）使用新建的交叉工具链建立一个新的目标（target）。

在 Scratchbox 里执行 sb-menu 命令，参考前面的方法建立一个目标。

① 命名为 arm，也可以取其他名字。

② 选择编译器时，选择“gcc-3.4.5-glibc-2.3.6”。

③ 选择开发工具时，全部选上（包括“cputransp”）。

在后面的菜单中会给出一系列的 CPU 模拟器，选中用于 ARM 的最高版本“qemu-arm-0.8.2-sb2”，如图 26.8 所示。

④ 当出现如图 26.9 所示的界面时，选择“No”。

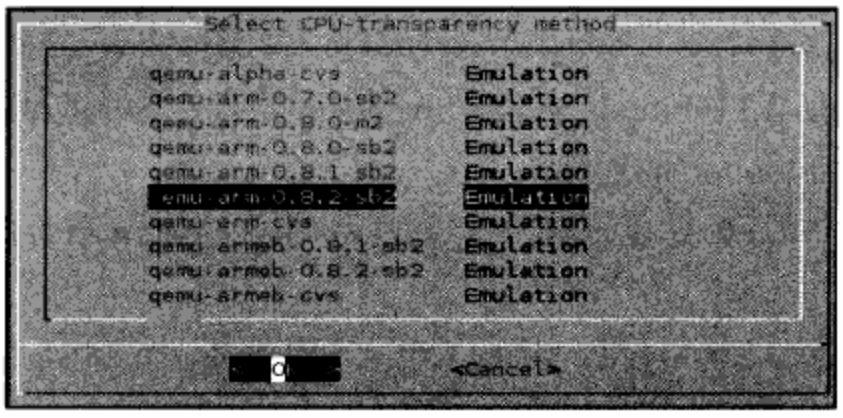


图 26.8　选择 CPU 模拟器

图 26.9　选择是否安装 rootstrap（一个制作文件系统映象的工具）

其他的操作与前面完全一样，当创建好目标后，先测试一下，执行以下命令：

```
[sbox-arm: ~] > tar xfz /scratchbox/packages/hello-world.tar.gz      /* 解压 */
[sbox-arm: ~] > cd hello-world/
[sbox-arm: ~/hello-world] > ./autogen.sh       /* 生成 configure 脚本，配置程序，
生成 Makefile */
```

```
[sbox-arm: ~/hello-world] > make                    /* 编译 */
[sbox-arm: ~/hello-world] > file hello              /* 查看文件信息，可以发现它是 ARM
                                                    程序 */
hello: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.4.3,
dynamically linked (uses shared libs), not stripped
[sbox-arm: ~/hello-world] > ./hello                 /* 执行 */
Hello World!
```

可见，虽然可执行程序 hello 是为 ARM 处理器编译的，但它在 scratchbox 中也可以运行。以后就可以在 Scratchbox 里进行开发了。

26.2.4 安装其他开发工具

Scratchbox 可以认为是一个全新的 Linux 安装版，里面缺乏一些开发包。比如编译 busybox 时先要执行“make menuconfig”进行配置，这个命令需要 ncurses 开发包；编译很多程序的时候，要用到 libtool 工具，这些工具需要自己安装。

下面的安装指令都是在 Scratchbox 的 arm 目标中执行的，这些工具的源代码在/work/tools 目录下。

1. 安装 ncurses

```
> tar xzf ncurses.tar.gz
> cd ncurses-5.6
> ./configure --with-shared --prefix=/usr
> make
> make install
```

2. 安装 libtool

```
> tar xzf libtool_1.5.6.orig.tar.gz
> cd libtool-1.5.6/
> ./configure --prefix=/usr
> make
> make install
```

现在 Scratchbox 的开发环境基本完成。

注意

每次登录 Scratchbox 后，先使用以下命令设置环境变量 GCC_EXEC_PREFIX 才能正确编译程序，否则连接器不会查找/usr/lib 目录（后面两个反斜线是必须的）：

```
> export GCC_EXEC_PREFIX=/usr/lib//
```

为了避免每次都要设置环境变量 GCC_EXEC_PREFIX，在/scratchbox/users/book/ targets 目录下建立一个文件 arm.environment，内容如下：

```
export GCC_EXEC_PREFIX=/usr/lib//
```

总结一下使用 Scratchbox 的命令。

```
$ sudo mount /dev/sdb1 /scratchbox/users/book/work      /* 挂接工作分区 */
$ sudo /scratchbox/sbin/sbox_ctl start                  /* 启动 */
$ /scratchbox/login                                     /* 登录 */
[sbox-arm: ~] > export GCC_EXEC_PREFIX=/usr/lib//    /* 如果建立了arm.environment,则不
                                                        需要 */
```

26.3 移植 X

本节先编译 X 的依赖软件，再介绍为交叉编译 X 所进行的修改，最后进行实验。

26.3.1 编译软件的基本知识

对于以压缩包发布的软件，在它的目录下通常都有一个配置脚本 configure，它的作用是确定编译参数（比如头文件位置、连接库位置等），然后生成 Makefile 以编译程序。可以进入该软件的目录，执行“./configure -help”命令查看使用帮助。

一个程序能正确编译、连接、运行需要满足 3 个条件：预处理时能找到头文件，连接时能找到库，运行时能找到库。下面分别介绍。

1. 指定头文件位置

在程序中常用两种方法来包含头文件：

```
#incldue <headerfile.h>
#incldue "headerfile.h"
```

它们的区别是，对于第二种方法，首先在源文件当前目录下查找头文件，如果找不到，再像第一种方法一样去编译命令指定、系统预设的目录去查找。这些“指定的”、“预设的”目录在什么地方呢？“指定的”头文件目录是编译程序时使用“-I”指定的目录，“预设的”的头文件目录是由编译器自己决定的。通过一个例子可以看到这点，执行以下命令：

```
$ mkdir -p /work/AAA/include          /* 临时目录，测试用 */
$ mkdir -p /work/BBB/include          /* 临时目录，测试用 */
$ export C_INCLUDE_PATH=/work/AAA/include
$ echo 'main(){}' | arm-linux-gcc -I/work/BBB/include -E -v -
```

得到以下输出内容，从中可以看到查找头文件时的路径及优先顺序：

```
...
#include "..." search starts here:
#include <...> search starts here:
```

```
    /work/BBB/include
    /work/AAA/include
    /work/tools/gcc-3.4.5-glibc-2.3.6/lib/gcc/arm-linux/3.4.5/include
    /work/tools/gcc-3.4.5-glibc-2.3.6/lib/gcc/arm-linux/3.4.5/../../../../arm-
linux/sys-include
    /work/tools/gcc-3.4.5-glibc-2.3.6/lib/gcc/arm-linux/3.4.5/../../../../
arm-linux/include
  End of search list.
  ...
```

可以总结出头文件的查找路径及优先顺序。

① 如果源文件中使用双引号来包含头文件，则首先在源文件当前目录查找头文件。

② 如果编译时使用“-I/some/dir”，则在/some/dir 中查找。

③ 如果设置了环境变量 C_INCLUDE_PATH，则在它指定的目录中查找。

④ 最后在编译器预设的路径中查找，这是不需要指定的。

所以，编译程序时如果出现了找不到头文件的错误，可以通过设置 C_INCLUDE_PATH 或给编译器设置“-I”选项来指定头文件目录，这可以在执行配置命令 configure 之前设置 C_INCLUDE_PATH 或 CFLAGS，如果不设置 CFLAGS，它的默认值为“-g -O2”，比如：

```
$ export C_INCLUDE_PATH="/some/dir/1:/some/dir/2"
$ export CFLAGS="-g -O2 -I/some/dir"          # 如果设置了 C_INCLUDE_PATH，就可以
                                              # 不设置 CFLAGS
$ ./configure
```

还有更好的方法，当明确知道要使用哪个动态库时，可以通过 pkg-config 命令获知要使用这个库时编译时的参数、连接时的参数。先执行以下命令体验一下：

```
$ export PKG_CONFIG_PATH=/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/
pkgconfig
$ pkg-config --cflags uuid
-I/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/include
```

最后一行是输出结果，表示使用这个库时头文件的目录。

pkg-config 程序在环境变量 PKG_CONFIG_PATH 指定的目录中找到 uuid.pc 文件（库名加上.pc），根据这个文件就可以知道使用 uuid 库时的编译参数、连接参数了。所以，使用 pkg-config 要保证两点：设置正确的环境变量 PKG_CONFIG_PATH，有相应的.pc 文件。在生成大多数的库文件时，.pc 文件也会自动生成。

配置的时候怎样使用 pkg-config 来确定这些参数？在 configure 文件中，常常可以看到类似如下字样的语句：

```
pkg_cv_XLIB_CFLAGS='$PKG_CONFIG --cflags "x11" 2>/dev/null'
```

其中的$PKG_CONFIG 就是 pkg-config 程序，这个语句将得到使用 libx11 库时的编译参

数，这些参数在配置结束后会被写入 Makefile 中。所以，通过 pkg-config 来确定编译参数的最终结果，也是给编译器（比如 arm-linux-gcc）传递“-I”选项。如果配置文件中对某些库没有使用 pkg-config 来确定它的编译选项，或是没有相应的.pc 文件，那就只能通过设置环境变量 CFLAGS 来指定了。

总结一下，在配置前，可以设置以下 3 项（前两项重合，可选其一）来找到头文件（第 3 项不一定能起作用）。

① 设置环境变量 CFLASG，比如：

```
$ export CFLAGS="-g -O2 -I/work/crossbuild/include -I/work/crossbuild/GTK/include"
```

② 设置环境变量 C_INCLUDE_PATH，比如：

```
$ export C_INCLUDE_PATH="/work/crossbuild/X/include:/work/crossbuild/GTK/include"
```

③ 设置环境变量 PKG_CONFIG_PATH，比如：

```
$ export PKG_CONFIG_PATH=/work/crossbuild/X/lib/pkgconfig:/work/crossbuild/GTK/lib/pkgconfig
```

在后面的移植中，本书使用后两种方法。

2. 指定连接时库的位置、名称

连接程序时，通常使用类似以下的命令：

```
$ arm-linux-gcc -o appfile 1.o 2.o -L/some/lib/dir -labc
```

它表示将生成可执行程序 appfile，这个文件由 1.o、2.o 和库 abc 组成，库 abc 位于 /some/lib/dir 目录下，它的名称为 libabc.so。

通过“-L”选项指定库的路径能够解决大部分问题，但是当库之间存在依赖关系时，比如库 abc 依赖于库 xyz，上述命令会给出“找不到库 xyz”的错误。这时可以通过“-rpath-link”或“-rpath”来指定依赖库的路径。这 3 个选项之间的作用及关系如下。

① “-L”指定连接时库的搜索路径，这些库使用“-l”来显示指定，比如“-labc”表示的库文件为 libabc.so。

② “-rpath-link”比“-L”多一项功能，它指定的目录还可以用于搜索依赖库。

③ “-rpath”比“-rpath-link”多一项功能，它指定的目录会被编译进程序中，当程序运行时，首先从这些目录中寻找库。

对于交叉编译（本地编译的连接库查找路径方法比它多几种），连接时库搜索路径及优先顺序如下。

① 使用“-rpath-link”指定的目录。

② 使用“-rpath”指定的目录。

③ 使用“-L”指定的目录。

④ 连接器的默认连接目录，这通常在交叉编译器的目录下。

由于编译、运行时库的路径并不相同，所以在后面的移植中，本书使用方法①，而不使

用方法②。在编译一些程序时，配置文件也自动指定了“-rpath”目录，它在运行时也起作用。

怎样指定“-rpath-link”呢？连接器 arm-linux-ld 通常是由 arm-linux-gcc 间接启动的，而 arm-linux-gcc 并不认识“-rpath-link”选项，所以需要在前面加上关键字“-Wl,”表示这个选项用于连接器。在执行配置命令 configure 之前设置 LDFLAGS 即可，比如：

```
$ export LDFLAGS="-Wl,-rpath-link -Wl,/work/crossbuild/X/lib -Wl,-rpath-link -Wl,
/work/crossbuild/GTK/lib"
$ ./configure
```

configure 文件通常会自动指定“-L”选项，它也可能使用 pkg-config 来确定“-L”选项。比如在 configure 文件中，常常可以看到如下字样的语句：

```
pkg_cv_XLIB_LIBS=`$PKG_CONFIG --libs "x11" 2>/dev/null`
```

3. 指定运行时库的位置

运行时库的查找路径及优先顺序如下。

① 编译程序时使用“-rpath”指定的目录。

② 环境变量 LD_LIBRARY_PATH 指定的目录（它可以指定多个目录，以冒号分隔）。

③ 文件/etc/ld.so.cache 中指定的目录，这个文件是 ldconfig 程序读取/etc/ld.so.conf 文件生成的。

④ 默认路径：/lib、/usr/lib。

26.3.2 编译 X 的依赖软件

要编译具备完全功能的 X，需要安装表 26.1 中的软件，表中的备注表示对于本书编译的 KDrive 是否需要这个软件。

表 26.1　　X 所依赖的软件和工具

软件名（最低版本）	功　能	备　注
expat 1.95.8	用于解析 XML（可扩展置标语言，EXtensible Markup Language）的 C 库	需要
gperf	用于编译 xcb-ut	不需要
xcb	xcb 表示“X protocol C-language Binding”，它是用来取代 xlib 的，能提供更好的性能	需要，GIT 库自带源码
freetype 2.1.8 或 freetype 2.1.9	跨平台的字体绘制引擎，为各种应用程序提供通用的字体文件访问接口	需要
fontconfig 2.2	用于字体的配置和访问，即负责字体的安装确认和匹配	需要
libpng 1.2.8	png 图形编码解码程序库	需要
zlib 1.1.4 或 zlib1.2.3	通用的压缩、解压缩函数库	需要
Mesa 6.5.2	一个三维计算机图形函数库，如果 X 要支持 GLX，则需要这个库	不需要

续表

软件名（最低版本）	功　能	备　注
libdrm	给 X 窗口系统提供直接访问显示设备的接口。如果编译 Mesa 的话，这个库也需要	不需要
xmlto	XML 前端，如果编译 libXcomposite 和 libXtst 的 man 手册时，需要这个工具	不需要
gettext	用于系统的国际化（I18N）和本地化（L10N），可以在编译程序的时候使用本国语言支持（Native Language Support（NLS）），可以使程序的输出使用用户设置的语言而不是英文	不需要

这些软件代码所在目录为/work/GUI/xwindow/X/deps，解压后，配置、编译、安装即可。

本章编译的所有软件都安装在 Scratchbox 里的/usr 目录下（它对应主机的/scratchbox/users/book/targets/arm/usr 目录），配置时需要指定“—prefix=/usr”。它们之间也存在一些依赖关系，可以按照以下顺序编译。进入/work/GUI/xwindow/X/deps 目录后，分别执行以下命令即可编译。

1. zlib

```
> tar xzf zlib-1.2.3.tar.gz
> cd zlib-1.2.3
> ./configure --shared --prefix=/usr
> make
> make install
```

2. libpng

```
> tar xjf libpng-1.2.23.tar.bz2
> cd libpng-1.2.23
> ./configure --prefix=/usr
> make
> make install
```

3. expat

```
> tar xzf expat-2.0.1.tar.gz
> cd expat-2.0.1
> ./configure --prefix=/usr
> make
> make install
```

4. freetype

```
> tar xjf freetype-2.3.5.tar.bz2
> cd freetype-2.3.5
```

```
> ./configure --prefix=/usr
> make
> make install
```

5. libxml-2.0

libxml-2.0 并不包含在 Xorg 的依赖里边，但是在编译下面的 fontconfig 之前，必须先编译 libxml-2.0。

```
> tar xzf libxml2-2.6.30.tar.gz
> cd libxml2-2.6.30
> ./configure --prefix=/usr
> make
> make install
```

6. fontconfig

```
> tar xzf fontconfig-2.5.0.tar.gz
> cd fontconfig-2.5.0
> ./configure --prefix=/usr
> make
> make install
```

7. libdrm

```
> tar xjf libdrm-2.3.0.tar.bz2
> cd libdrm-2.3.0
> ./configure --prefix=/usr
> make
> make install
```

8. openssl

编译 X server 时用到 openssl。

```
> tar xzf openssl-0.9.8g.tar.gz
> cd openssl-0.9.8g
> ./Configure --prefix=/usr --openssldir=/usr/openssl linux-generic32 -DL_
ENDIAN
> make
> make install
```

26.3.3　编译 Xorg

这节的目标如下。

- 编译库文件（统称为 Xlib），后面的窗口管理器、GUI 程序等会用到它们。
- 编译 Xfbdev，这是个 X server 可执行程序。
- 编译一些基于 X 的应用程序，比如窗口管理器、计算器等，以供实验与测试。

1. 下载源码

git_xorg.sh 里没有下载字库，可以使用主机里的字库；如果想自己编译字库，在 git_xorg.sh 中增加下面一行：

```
do_dir xorg font "${font}"                    # 下载字库
```

将 git_xorg.sh 文件放到/work/GUI/xwindow/X/Xorg 目录下，执行以下命令就可以下载到最新的代码。

```
$ chmod +x ./git_xorg.sh
$ ./git_xorg.sh
```

> **注意**　读者自行下载的代码相对于光盘中的 Xorg_git_20071119.tar.bz2 可能已经有了更新，可以先使用 Xorg_git_20071119.tar.bz2（在/work/GUI/xwindow/X 目录下，解压后即得 Xorg 目录），当掌握了移植方法后再下载最新版本。

当下载完成后，可以看到如下目录：

```
$ ls
app  data  doc  driver  drm  font  git_xorg.sh  lib  pixman  proto  util  xcb
xserver
```

这些目录的内容、作用正如它们的名字一样，如表 26.2 所示。

表 26.2　　Xorg 源码包的结构

目 录 名	作　用	目 录 名	作　用
app	基于 Xlib 的应用程序	lib	X 的函数库（*）
data	数据，目前只有两个目录：bitmaps（表示位图的数组）、cursors（各类光标形状）	pixman	像素处理库，X 和 cairo（一个二维图形库）会用到它
doc	文档	proto	众多协议的头文件
driver	驱动，目前有两类驱动：input、video。这些驱动是应用层的模块，X 可以动态加载它们来操作不同的硬件（当然，内核中要有相应的驱动程序支撑）	util	系统工具
drm	Libdrm 库及一些内核驱动模块	xcb	用来取代 xlib 的库（*）
font	字库	xserver	X server

> **注意** xlib 和 Xorg/lib 目录下的库不是一个概念。xlib 是一个 C 语言库，X client 程序用它来与 X 进行通信，可以使用 xcb 来代替它。

2. 分析编译脚本、修改文件

本书对 Xorg 的修改都制成了补丁文件 Xorg_git_20071119_100ask.patch，执行以下命令，打上补丁（建议读者先编译，等出错时再参考补丁文件进行修改）。

```
$ cd /work/GUI/xwindow/X/Xorg
$ patch -p1 < ../Xorg_git_20071119_100ask.patch
```

这些需要修改的文件可以分为 3 类。

① util/modules/build.sh，通过它来编译所有的程序，可以修改它以便不编译某些程序，或是加入一些配置参数。

② 代码本身的错误。

③ 增加功能。

下面首先分析修改后的 build.sh 文件，了解编译过程，然后讲述其他文件的修改原因。

① build.sh 分析及修改。

与 Xorg 中各个子目录对应，在 build.sh 里分别使用一个函数来编译它们，比如 build_proto、build_app、build_xserver、build_font 等函数。以 build_lib 函数为例，它负责编译 lib 目录。build_lib 函数的内容如下：

```
build_lib() {
    build lib libxtrans
    build lib libXau
    build lib libXdmcp
...
}
```

它调用 build 子函数编译 lib 目录下的各个子目录。

build 函数是整个 build.sh 文件的核心，它的调用方法如下：

```
build module component
```

module 是指源文件顶层目录下的各个子目录名，比如 app、lib、xserver 等，component 是指 module 所表示的目录下的某个子目录。调用 build 函数后，它将进入<module>/<component>目录中编译程序。

build 函数的本质为：使用 build.sh 中指定的参数，调用具体目录下的 autogen.sh 脚本产生 Makefile 文件，然后执行“make”、“make install”进行编译、安装。本书对 build.sh 所做的修改主要是设置配置参数。修改后的代码如下，以补丁文件的格式给出以便对照。

- 修改编译 X server 时的配置参数，生成 Xfbdev。

```
if test "$1" = "xserver" && test -n "$MESAPATH"; then
MOD_SPECIFIC="--with-mesa-source=${MESAPATH}"
```

```
    fi
+   if test "$1" = "xserver"; then
+   MOD_SPECIFIC="${MOD_SPECIFIC} --enable-kdrive --enable-xfbdev
+                 --disable-ipv6 \
+                 --disable-xorg \
+                 --disable-xnest \
+                 --disable-xvfb \
+                 --disable-xevie \
+                 --disable-xwin \
+                 --disable-xsdl \
+                 --disable-xephyr \
+                 --disable-xfake \
+                 --disable-kdrive-vesa \
+          --disable-dri"
+   fi
```

- 去除不需要的软件。

```
-build_doc
+#build_doc
...
-build_mesa
+#build_mesa
...
-build_driver
-build_data
+#build_driver
+#build_data
...
-build_util
+#build_util
```

- 去除编译时出错的软件（在本书中没有使用它们）。

```
-    build app xdriinfo
+#    build app xdriinfo
...
-    build app xprop
+#    build app xprop
...
-    build app xsm
```

```
+#    build app xsm
...
```

② 本身错误的代码。

这类文件的错误，多与头文件有关，共有 3 个文件。

- xserver/hw/kdrive/src/kaa.c。

```
-    pPixmap = fbCreatePixmapBpp (pScreen, w, h, depth, bpp);
+    pPixmap = fbCreatePixmapBpp (pScreen, w, h, depth, bpp, usage_hint);
```

- xserver/hw/xfree86/modes/xf86Crtc.c。

```
+#include "xf86Priv.h"
 #include "xf86.h"
```

- xserver/hw/xfree86/os-support/xf86_OSlib.h。

```
-#  include <sys/kd.h>
+//#  include <sys/kd.h>
+#  include <linux/kd.h>
```

使用<sys/kd.h>会导致<linux/types.h>没有被包含进去，这使得在 types.h 中定义的数据类型无法使用。

③ 修改 xserver/hw/kdrive/src/kmode.c 以支持 240×320 的分辨率。

```
     /*  H    V    Hz   KHz */
         /*  FP       BP       BLANK    POLARITY */

+    {   240,    320,    60,  16256,
+                   17,     12,     32,     KdSyncNegative,
+                   1,      11,     14,     KdSyncNegative,
+    },
+
```

参考 320×240，增加对 240×320 的支持。实际上对于 Xfbdev，只用到了前面的 3 个参数：240 和 320 表示分辨率，60 表示刷新频率（设为一个比较低的值，只是用于比较，没有实际使用）。

> **注意** 由于在 util/modules/build.sh 中屏蔽了对很多程序的编译，如果读者要编译它们，有可能出错，需要自行分析、解决。

3. 编译、测试

文件修改完毕后，执行以下命令即可编译 Xorg。

```
> cd /work/GUI/xwindow/X/Xorg
> ./util/modular/build.sh -b /usr
```

① “/usr”必须放在最后面。

注意　② 可以加入“-n”使得编译的时候即使出错也不退出，而是继续编译下一个软件。

③ 可以使用“-r module/component”，它将从module/component目录中的软件开始重新往下编译。

所有的结果都放在Scratchbox里/usr目录下（它就是主机上的/scratchbox/users/book/targets/arm/usr目录），里面包含了X的所有部件：库、应用程序、Xfbdev（就是X server）、字库、头文件、文档等。它们加起来体积庞大，在放到单板上之前需要去除执行程序时不需要的部件，这在后面会说明。

先使用nfs进行试验，假设开发板的根文件系统保存在主机上的/work/nfs_root/fs_xwindow目录中，把编译Xorg所得的结果全部复制到fs_xwindow中，即把Scratchbox中/usr目录下的文件复制到单板根文件系统的/usr目录下。

```
$ mkdir -p /work/nfs_root/fs_xwindow
$ cp -rf /scratchbox/users/book/targets/arm/usr /work/nfs_root/fs_xwindow/
```

再将fs_mini_mdev目录下的所有文件复制到fs_xwindow下。

```
$ cd /work/nfs_root
$ sudo cp -rf fs_mini_mdev/* fs_xwindow
$ sudo chown book:book fs_xwindow -R
```

通过nfs将/work/nfs_root/fs_x作为根文件系统启动单板后，即可进行测试。

① 启动X server。

在单板上使用以下命令启动X server。

```
# Xfbdev -mouse mouse -keybd keyboard &
```

在LCD上可以看到如图26.10所示的界面，光标可以移动。这时候，输入、输出设备已经准备好，可以运行基于X的程序了。

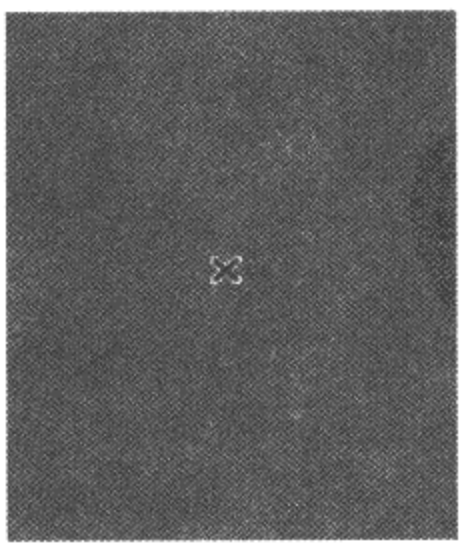


图26.10　Xfbdev启动界面

② 启动基于X的应用程序。

首先设置环境变量DISPLAY，它表示之后运行的程序在什么地方输出，从什么地方获得输入，它有以下3种格式。

```
hostname:D.S
host/unix:D.S
:D.S
```

hostname 是与显示器直接连接的机器的名称，也可以是 IP。第一种格式使用 TCP 端口“D+6000”进行通信；第二种格式使用“Unix domain sockets”进行通信；省略 hostname 时即为第三种格式，它表示 X client 与 X sever 位于同一台机器上，系统会为 X client 选择最有效率的方式与 X server 通信。D 表示“Display number”，它是一组显示器、键盘、鼠标的组合，从 0 开始编号，设置 DISPLAY 环境变量时 D 不可省略。S 表示“Screen number”，表示连接到显示器上的多个屏幕（这属于软件的范畴），也从 0 开始编号，通常设为 0（这时可以省略）。

使用以下命令设置环境变量 DISPLAY。

```
# export DISPLAY=:0
```

然后，启动 xcalc 和 xeyes。

```
# xcalc &
# xeyes &
```

LCD 上出现如图 26.11 所示的界面，可以使用鼠标、键盘操作计算器，图中的眼睛会随着光标移动。但是，这两个程序的图像重叠在一起，无法进行窗口移动、缩小等操作，这是窗口管理器的功能。

③ 启动窗口管理器。

在上面两个实验的基础上，执行以下命令启动窗口管理器。

```
# twm &
```

可以看到 xcalc 和 xeyes 这两个程序的窗口周围被加上了边框，可以按住它进行移动，单击左上角的圆点可以最小化窗口，如图 26.12 所示。

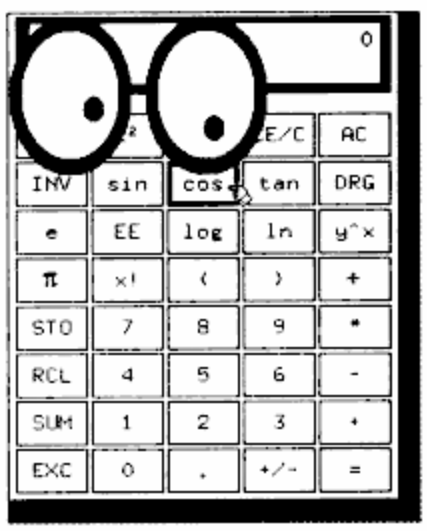


图 26.11 运行基于 X 的应用程序

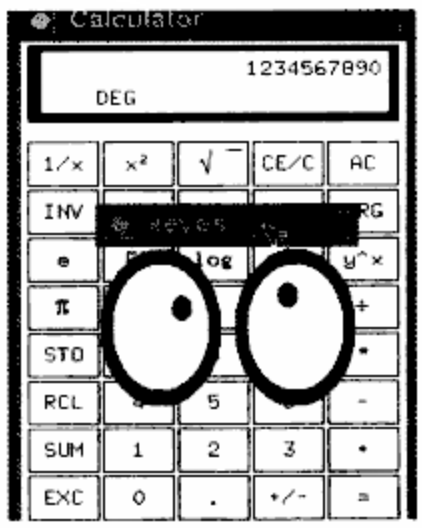


图 26.12 运行窗口管理器 twm 后的界面

26.4 移植 Matchbox

本节移植 Matchbox，它为嵌入式设备提供了一个开放源代码的基于 X Window 的 GUI

环境。相比于 KDE、GNOME 等桌面环境，它体积小巧、易于裁剪。Matchbox 的核心是一个小型的窗口管理器，还有其他一些程序：一个面板、一个桌面、一个共享功能程序库、一些小的面板应用程序。Matchbox 的风格是基于 PDA 的，这与 PC 机不同。

26.4.1 下载源代码

/work/GUI/xwindow 目录下的 matchbox_release.tar.bz2 就是本书即将使用的代码，解压即可。

```
$ cd /work/GUI/xwindow
$ tar xjf matchbox_release.tar.bz2
```

读者也可以通过以下网址自己下载所需要的软件包。

```
http://matchbox-project.org/download.html
```

假设源码目录为 matchbox_release，它下面的内容如下所示：

```
$ ls
libmatchbox-1.9.tar.gz matchbox-common-0.9.1.tar.gz matchbox-panel-0.9.3.tar.gz
matchbox-autobuild.sh      matchbox-desktop-0.9.tar.gz      matchbox-window-
manager-1.2.tar.gz
```

其中：

① 脚本文件 matchbox-autobuild.sh 被用来编译 Matchbox。

② libmatchbox-1.9.tar.gz 是 Matchbox 的基本库，其他程序都依赖于它。

③ matchbox-common-0.9.1.tar.gz 中含有图标及一些配置数据。

如果要安装 matchbox-panel 或 matchbox-desktop，则必须先安装 matchbox-common。

④ matchbox-window-manager-1.2.tar.gz 是窗口管理器。

⑤ matchbox-panel-0.9.3.tar.gz 是控制面板。

⑥ matchbox-desktop-0.9.tar.gz 是桌面管理器。

⑦ 其他的文件在本书中没有用到，它们大多是 Matchbox 应用程序。

26.4.2 编译 Matchbox

Matchbox 也要在 Scratchbox 中编译。

Matchbox 的各个程序的编译方法与前面编译 Xorg 的依赖很相似，对于非“release”版本(比如本书直接通过 svn 下载的代码)运行 autogen.sh 进行配置、生成 Makefile；对于 release”版本运行 configure 进行配置、生成 Makefile；最后执行 make、make install 命令进行编译、安装。

脚本文件 matchbox-autobuild.sh 已经将这些步骤包装好，并设置了一些配置参数，稍做修改即可。它编译的程序是 26.4.1 小节中的②～⑥共 5 个程序包。

1. 安装所依赖的 jpeg 库

源码为/work/GUI/xwindow/jpegsrc.v6b.tar.gz，在 Scratchbox 里编译之前先修改它的 configure 脚本的第 1562 行，否则无法自动检测出机器类型。

```
-  $srcdir/ltconfig $disable_shared $disable_static $srcdir/ltmain.sh
+  $srcdir/ltconfig $disable_shared $disable_static $srcdir/ltmain.sh arm
```

然后执行以下命令编译。

```
> ./configure --enable-shared --enable-static --prefix=/usr
> make
> make install
```

2. 编译、安装 Matchbox

首先修改 matchbox-autobuild.sh。

① 将 DEST 改为“/usr”，它将在配置程序时作为“—prefix”的值。

```
-DEST="/tmp/mb"
+DEST="/usr"
```

② 指定“不是编译 CVS 版本的代码”，而是编译“release 版本”，这两者在编译过程中有点差别：前要使用 autogen.sh 生成 configure 文件，再进行配置；后者直接调用 configure 进行配置。

```
-BUILD_CVS="y"
+BUILD_CVS="n"
```

③ 指定软件包的版本。

```
MBLIB="libmatchbox"
-MBLIB_V="1.2"
+MBLIB_V="1.9"
 MBCMN="matchbox-common"
-MBCMN_V="0.8"
-MBCMN_MV=
+MBCMN_V="0.9"
+MBCMN_MV=".1"
 MBWM="matchbox-window-manager"
-MBWM_V="0.8"
-MBWM_MV=".2"
+MBWM_V="1.2"
+MBWM_MV=
 MBPANEL="matchbox-panel"
-MBPANEL_V="0.8"
-MBPANEL_MV=".2"
+MBPANEL_V="0.9"
+MBPANEL_MV=".3"
```

```
 MBDSKTP="matchbox-desktop"
-MBDSKTP_V="0.8"
-MBDSKTP_MV=".1"
+MBDSKTP_V="0.9"
+MBDSKTP_MV=
```

④ 将所有 wget 开头的下载命令屏蔽掉，因为在当前目录中这些源码已经存在，如下所示：

```
-  wget "${SRC_URL}${MBLIB}/${MBLIB_V}/${MBLIB}-${MBLIB_V}.tar.gz" || exit
+  #wget "${SRC_URL}${MBLIB}/${MBLIB_V}/${MBLIB}-${MBLIB_V}.tar.gz" || exit
```

然后，在 Scratchbox 中运行 matchbox-autobuild.sh 即可编译 Matchbox。

最后将编译结果复制到 nfs 中。

```
$ mkdir -p /work/nfs_root/fs_xwindow
$ cp -rf /scratchbox/users/book/targets/arm/usr /work/nfs_root/fs_xwindow/
```

还要将 fs_mini_mdev 目录下的所有文件重新复制到 fs_xwindow 下。上述复制命令会修改原来 fs_xwindow 目录中其他子目录的内容，比如 fs_xwindow/bin/busybox，这可能是个系统 bug。

```
$ cd /work/nfs_root
$ sudo cp -rf fs_mini_mdev/* fs_xwindow
$ sudo chown book:book fs_xwindow -R
```

另外，NFS 目录中配置文件/usr/etc/fonts/fonts.conf 里指定了字符文件所在目录为/usr/share/fonts 和~/.fonts。

```
<!-- Font directory list -->
        <dir>/usr/share/fonts</dir>
        <dir>~/.fonts</dir>
<!--
```

但是/usr/share/font 还没有构建，可以使用第 26.3 节移植 X 时所生成的字符目录/usr/lib/X11/fonts，执行以下命令即可。

```
$ cd /work/nfs_root/fs_xwindow
$ ln -s /usr/lib/X11/fonts usr/share/
```

26.4.3 运行、试验 Matchbox

下面以两种方法运行 Matchbox，读者可以从中理解各个部件的作用。

1. 使用脚本 matchbox-session 启动所有程序

```
# Xfbdev -mouse mouse -keybd keyboard &
# export DISPLAY=:0
```

```
# export HOME=/root
# matchbox-session &
```

matchbox-session 启动的程序有：matchbox-window-manager、matchbox-desktop、matchbox-panel；而 matchbox-panel 自己又启动了 mb-applet-menu-launcher 和 mb-applet-clock，就是下图左下角的按钮、右下角的时钟。

启动界面如图 26.13 所示。

可以继续启动其他程序，比如 xeyes、xcalc，请读者自行实验。

2. 逐个启动 Matchbox 程序以观察其作用

（1）窗口管理器 matchbox-window-manager。

启动 Xfbdev 后，先启动 xeyes 和 xcalc，然后启动 matchbox-window-manager。

```
# Xfbdev -mouse mouse -keybd keyboard &
# export DISPLAY=:0
# export HOME=/root
# xcalc &
# xeyes &
# matchbox-window-manager &
```

从图 26.14 可知，Matchbox 的窗口是占满屏幕的（不能移动、改变大小，这适用于 PDA 等掌上设备），窗口管理器在程序的上面加了个边框，可以看到程序名称，可以关闭程序，可以单击左上角的小箭头切换程序。

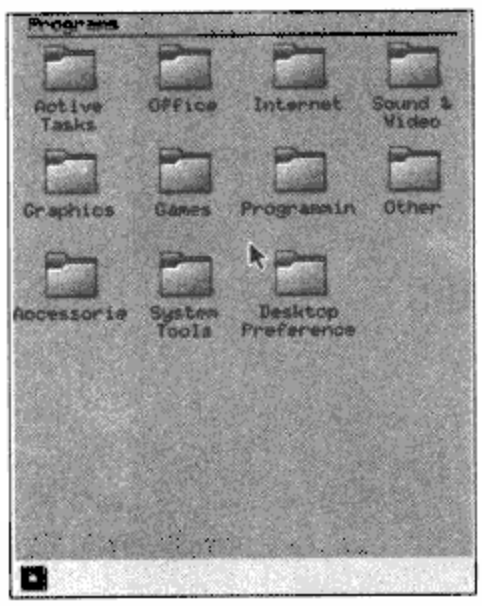


图 26.13 Mathbox 启动界面

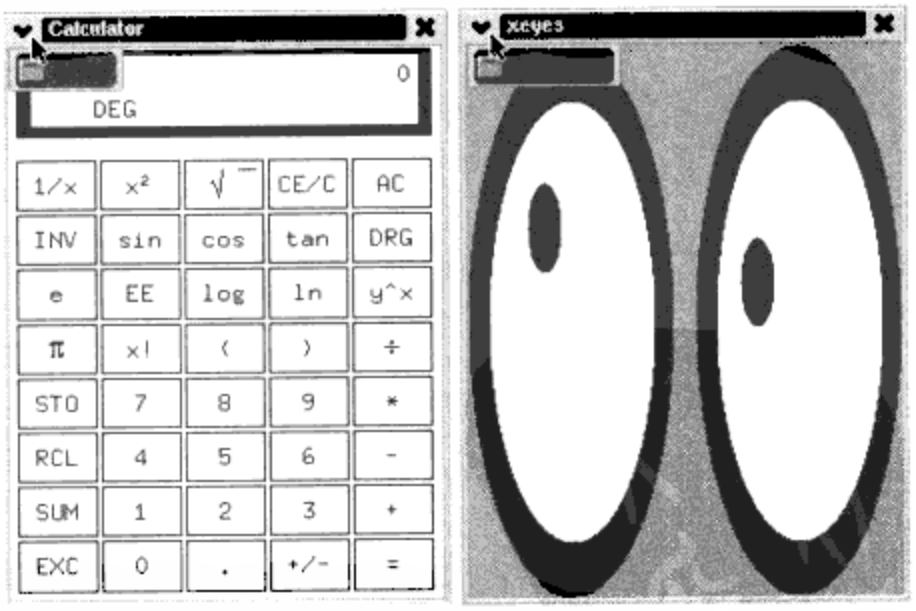


图 26.14 matchbox-window-manager 的效果

（2）桌面 matchbox-desktop。

重启系统后执行以下命令，界面如图 26.15 左边的图所示。

```
# Xfbdev -mouse mouse -keybd keyboard &
# export DISPLAY=:0
# matchbox-desktop &
```

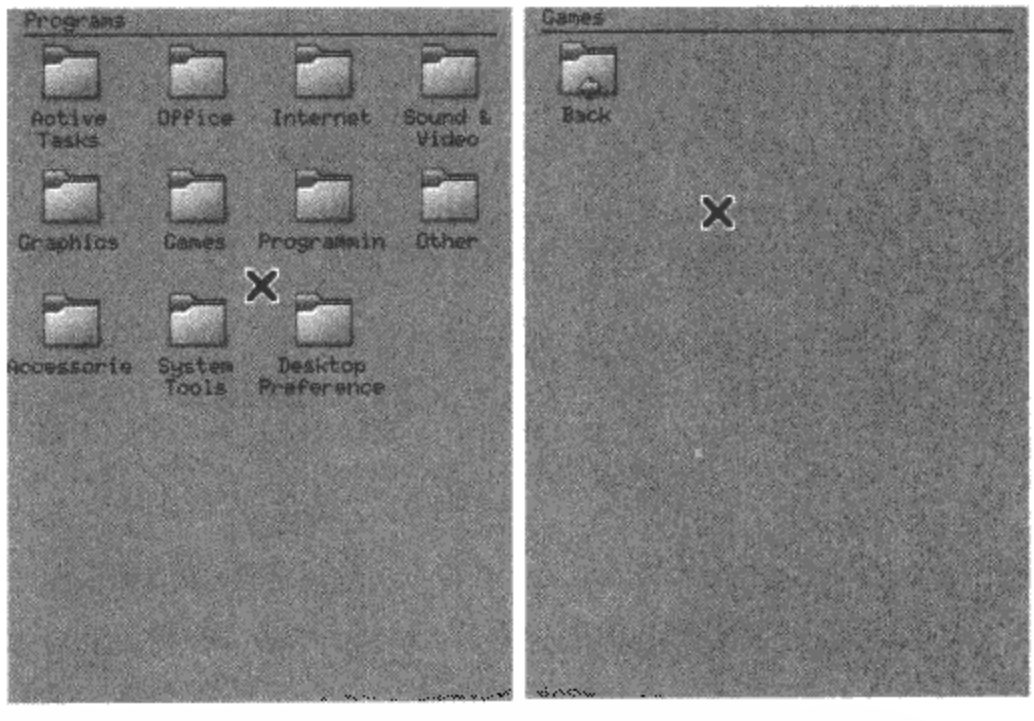


图 26.15　matchbox-desktop 的效果

单击图中各个文件夹将进入下一级界面，比如单击“Games”进入右边的图，再单击“Back”返回。现在各个文件夹是空的。

（3）面板 matchbox-panel。

执行以下命令，界面如图 26.16 左边的图所示，其中“--orientation south”表示面板将坐靠南面（即上下左右四个方向中，坐靠下面的边沿）。

```
# Xfbdev -mouse mouse -keybd keyboard &
# export DISPLAY=:0
# matchbox-panel --orientation south &
```

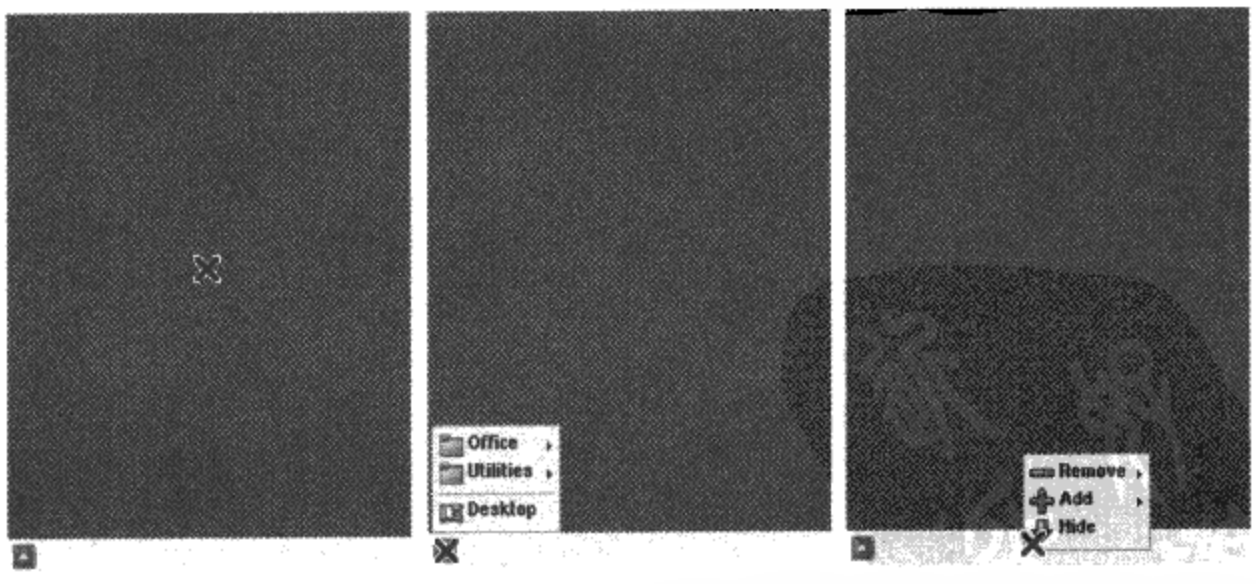


图 26.16　matchbox-panel 的效果

默认情况下 matchbox-panel 会启动小程序 mb-applet-menu-launcher(对应左下角的按钮)。这类小程序被称为“applet”。

通过“菜单启动器”(就是左下角的按钮）可以启动其他程序，请参考图 26.16 中间的图。

单击面板的空白位置，并按住一会儿，会弹出图 26.16 右边的图，可以在面板上增加、删除各个小程序（applet)，或者隐藏面板。

26.5 移植 GTK+

26.5.1 GTK+介绍

1. GTK+概述

GTK 原意为 GIMP 工具箱（GIMP ToolKit），最初是为 GIMP（GNU 图形处理程序）开发的控件集，是一个用于创建图形用户界面的多平台工具。它包含有基本的控件和一些很复杂的控件，例如文件选择控件和颜色选择控件，可以用来创建按钮、菜单及其他图形对象。GTK 已经发展为 Linux 下开发基于 X Window 图形界面应用程序的主流开发工具之一，它遵循 LGPL 协议，所以可以用来开发任何软件：开放源码软件、自由软件、商业软件、非自由的软件。

GTK 有时候被用来泛指 GTK，但是如果跟 GTK+比较的时候，一般指的是老的 GTK 库："+"表示与老的 GKT 库相比，做了巨大的改动。

GTK+有两个主要版本分支，其区别也是非常大的，那就是 GTK+1.2 和 GTK+2.x，目前一般都是用的 GTK+2.x。为了软件包的名字不发生冲突，有些 Linux 发行版将 GTK+2.x 命名为 GTK2。

2. GTK+的结构

基于 X、使用 GTK+开发的 GUI 程序的结构如图 26.17 所示。

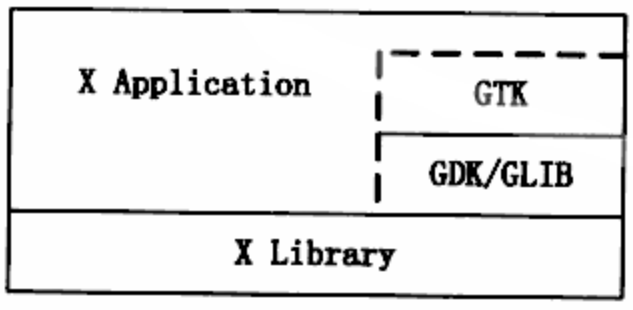


图 26.17 基于 GTK、X 的 GUI 程序结构

XLIB 被用来与 X server 通信。

Glib 是 GTK+和 GNOME 工程的基础底层核心程序库，是一个综合用途的实用的轻量级的 C 程序库。它定义很多数据结构，比如数组（长度可变）、单（双）向链表、hash 表、队列、还有关系；实现了一些函数，比如字符串的处理等；还针对可移植性封装了一些函数。它的功能与 glibc（glibc 是标准 C 库的实现）有些重合，但是注意 GLIB 与 glibc 并不相同。

GDK 是底层的图形函数库，它包含 GTK 所使用的基本图形操作函数，比如基本图元、颜色、事件处理、图像和位图、窗口、拖放函数等。GTK+不会直接与 X Window 打交道，而是通过中间层 GDK。GDK 屏蔽了不同窗口系统的差异，这可以简化编程并提高可移植性。

26.5.2 GTK+移植

1. GTK+的依赖

要编译 GTK+，必须保证一些工具、一些库已经被安装。

（1）编译工具 pkg-config 和 make。

在 Scratchbox，这两个工具已经存在。

（2）所依赖的库。

这些库如表 26.3 所示，其中的 Glib、Pango 和 ATK 也是由开发 GTK+的人员进行开发的。

表 26.3　　GTK+所依赖的库

软 件 名	功　能	备　注
GLib	提供 C 语言类型、操作函数	需要
Pango	国际化文本处理库	需要
ATK	可访问性工具箱（Accessibility Toolkit）	需要
libiconv	如果系统里没有 iconv 函数，则需要这个库	不需要
libintl	如果系统里没有 gettext 函数，则需要这个库	不需要
JPEG、PNG 和 TIFF 库	支持这 3 种图形格式的库，它们是 3 个库	png 在编译 Xorg 时已经安装，jpeg 库在编译 matchbox 时也已经安装，只要编译 tiff 库
X 库	提供 X 协议的封装等	已经安装
fontconfig	用于字体的配置和访问，简言之就是负责字体的安装确认和匹配	在编译 Xorg 时已经安装
Cairo	二维图形库	需要

2. 编译依赖库

GTK+的代码在/work/GUI/xwindow/GTK 目录下，依赖库在/work/GUI/xwindow/GTK/deps 目录下。

> **注意**　最先编译 Glib，然后编译 Cairo，再编译 Pango。编译 Pango 之前必须编译 Cairo。

（1）Glib。

```
> tar xzf glib-2.12.9.tar.gz
> cd glib-2.12.9/
> ./configure --prefix=/usr
> make
> make install
```

（2）Cairo。

先解压缩。

```
> tar xzf cairo-1.2.6.tar.gz
> cd cairo-1.2.6/
```

然后修改 src/cairo-type1-subset.c，加上下面一行，否则编译时会出现“isspace 未定义”的错误。

```
#include <ctype.h>
```

最后编译，配置时需要加上“--enable-pdf”，否则在编译 GTK+时出现找不到 cairo-pdf.h 的错误。

```
> ./configure  --enable-pdf --prefix=/usr
> make
> make install
```

（3）Pango。

```
> tar xzf pango-1.16.4.tar.gz
> cd pango-1.16.4/
> ./configure --prefix=/usr
> make
> make install
```

（4）ATK。

```
> tar xjf atk-1.9.1.tar.bz2
> cd atk-1.9.1/
> ./configure --prefix=/usr
> make
> make install
```

（5）TIFF 库。

```
> tar xzf tiff-3.7.4.tar.gz
> cd tiff-3.7.4/
> ./configure --prefix=/usr
> make
> make install
```

3. 编译 GTK+

GTK+的代码在/work/GUI/xwindow/GTK 目录下，执行以下命令编译、安装。

```
> tar xjf gtk+-2.10.9.tar.bz2
> cd gtk+-2.10.9/
> ./configure --prefix=/usr
> make
> make install
```

最后，编译结果都保存在 Scratchbox 里的/usr 目录中。

26.6 移植基于 GTK+/X 的 GUI 程序

本节的目的是移植两个 GUI 程序：xterm、gtkboard。xterm 完全基于 Xlib，它是一个终

端模拟器——使X应用程式视窗看起来像普通终端机一样的程序，可以在上面输入各种命令操作 Linux，就像串口终端一样。gtkboard 基于 GTK+，它是 30 多个游戏的集合。通过这两个例子，读者将学到如何将应用程序集成进 Matchbox 的 GUI 环境中。

26.6.1 xterm 移植

本节的目标是：在桌面文件夹“System Tools”下面建立 xterm 的图标，单击它时将启动 xterm。

源码为/work/GUI/xwindow/apps/xterm.tar.gz。

```
> tar xzf xterm.tar.gz
> cd xterm-229/
> ./configure --x-includes=/usr/include --x-libraries=/usr/lib --prefix=/usr
> make
> make install
```

然后将可执行文件 xterm 复制到 NFS 文件系统的/usr/bin 目录下，将图标复制到 NFS 文件系统的/usr/share/pixmaps 目录下。

```
$ cd /work/GUI/xwindow/apps/xterm-229/
$ cp xterm  /work/nfs_root/fs_xwindow/usr/bin
$ cp icons/xterm-color_32x32.xpm  /work/nfs_root/fs_xwindow/usr/share/pixmaps
```

可以想象，要通过单击一个图标启动一个程序，需要 3 个文件：图标文件、可执行程序、把它们两者联系起来的文件（称为桌面文件）。前两个文件已经复制到好了，下面建立桌面文件 xterm.desktop，将它放在 NFS 文件系统的/usr/share/applications 目录下，内容为：

```
01 [Desktop Entry]
02 Name=Xterm
03 Comment=Terminal for X Window System
04 Exec=xterm
05 Type=Application
06 Icon=xterm-color_32x32.xpm
07 Categories=System
```

第 1 行表示这是一个“桌面条目”（Desktop Entry）。

第 2 行表示显示在图标下方的名字。

第 3 行是注释，可以省略。

第 4 行表示对应的程序，单击图标时将运行它。

第 5 行表示桌面条目的类型，共有 3 种类型：Application（应用程序）、Link（连接）、Directory（目录）。这是可以扩展的，比如 Matchbox 中扩展了“PanelApp”类型。

第 6 行表示图标文件，不要使用绝对路径，系统会在（install prefix）/share/pixmaps 目录下查找。在本书中，“(install prefix)”就是/usr。

第 7 行表示“类别”，它决定了这个图标显示在哪里，下面说明。

NFS 文件系统的/usr/share/applications 目录下有很多.desktop 文件，它们表示各个应用程序及其图标。这些图标出现在 Matchbox 桌面的哪个文件夹里呢？显然，需要有相关的“文件夹”文件来描述，它们在 NFS 文件系统的/usr/share/matchbox/vfolders 目录下。分为以下 3 类。

① Root.order。

这个文件用来确定在桌面显示的文件夹及它们的顺序。

② Root.directory。

它表示整个桌面。

③ 其他的.directory 文件。

每个.directory 文件对应在桌面上显示的一个文件夹，它的结构与.desktop 文件相同，差别在于它用来描述文件夹，而.desktop 用来描述应用程序。.directory 文件里定义了一个名为“Match”的键值，它被用来与.desktop 文件的“Categories”键值比较：这决定了.desktop 文件表示的图标出现在哪个文件夹里。比如在 xterm.desktop 文件里，“Categories=System”；而在 System.directory 里：“Name=System Tools”、“Match=System”，这表示 xterm 程序对应的图标出现在桌面的“System Tools”文件夹里。

建立 xterm.desktop 文件后，重新启动 Matchbox，单击“System Tools”文件夹，可以看到如图 26.18 左边所示的界面，单击 xterm 的图标得到右边界面，可以在里面执行各种命令。

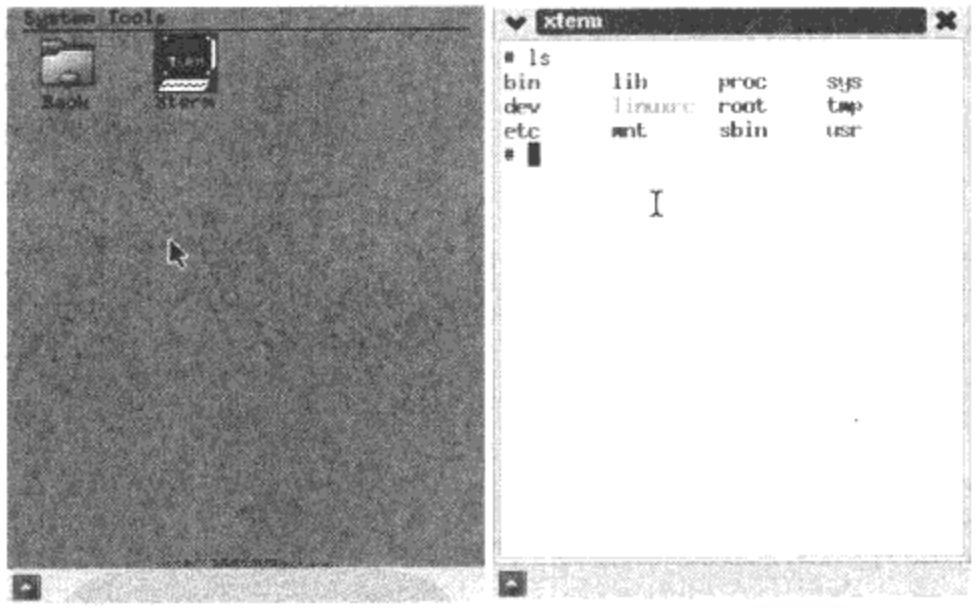


图 26.18 将 xterm 集成进桌面“System Tools”文件夹中

26.6.2 gtkboard 移植

本节的目标是：在桌面文件夹“Games”下面建立 gtkboard 的图标，单击它时将启动 gtkboard。

源码为/work/GUI/xwindow/apps/gtkboard-0.11pre0.tar.gz，先解压缩。

```
> tar xzf gtkboard-0.11pre0.tar.gz
> cd gtkboard-0.11pre0/
```

本书没有移植 SDL 库（SDL：Simple DirectMedia Layer），需要修改 src/sound.c 文件，并且在配置时增加“--disable-sdl”选项，修改如下：

```
+#ifdef HAVE_SDL
                if (opt_verbose) fprintf (stderr, "Using sound directory %s\n",
sound_dir);
+#endif
```

然后配置、编译。

```
> ./configure --disable-sdl --prefix=/usr
> make
> make install
```

执行以下命令看看最终要安装什么文件。

```
> mkdir tmp
> make install prefix=$PWD/tmp
> ls tmp/
bin  share
> ls tmp/share/
pixmaps  sounds
```

可以看到在 bin 目录下有可执行文件 gtkboard，share/pixmaps 目录有图标文件 gtkboard.png，share/sounds/目录下是各种声音文件。

然后将 Scratchbox 中/usr 目录下的所有文件复制到 NFS 文件系统/usr 目录下（前面移植 GTK+时，GTK+库也在 Scratchbox 中/usr 目录下），在 Scratchbox 外部执行以下命令：

```
$ cp -rf /scratchbox/users/book/targets/arm/usr /work/nfs_root/fs_xwindow/
$ cd /work/nfs_root
$ sudo cp -rf fs_mini_mdev/* fs_xwindow
$ sudo chown book:book fs_xwindow -R
$ cd /work/nfs_root/fs_xwindow
$ ln -s /usr/lib/X11/fonts usr/share/
```

最后在 NFS 文件系统/work/nfs_root/fs_xwindow/usr/share/applications 目录下建立 gtkboard.desktop 文件，内容如下：

```
01 [Desktop Entry]
02 Name=gtkboard
03 Comment=A set of many games
04 Exec=gtkboard
05 Type=Application
06 Icon=gtkboard.png
07 Categories=Game
```

重新启动 Matchbox，单击“Games”文件夹，可以看到如图 26.19 左边所示的界面，单

击 gtkboard 的图标，在“Game”菜单里选择各种游戏（如中图所示），右边的图是一个拼图游戏。

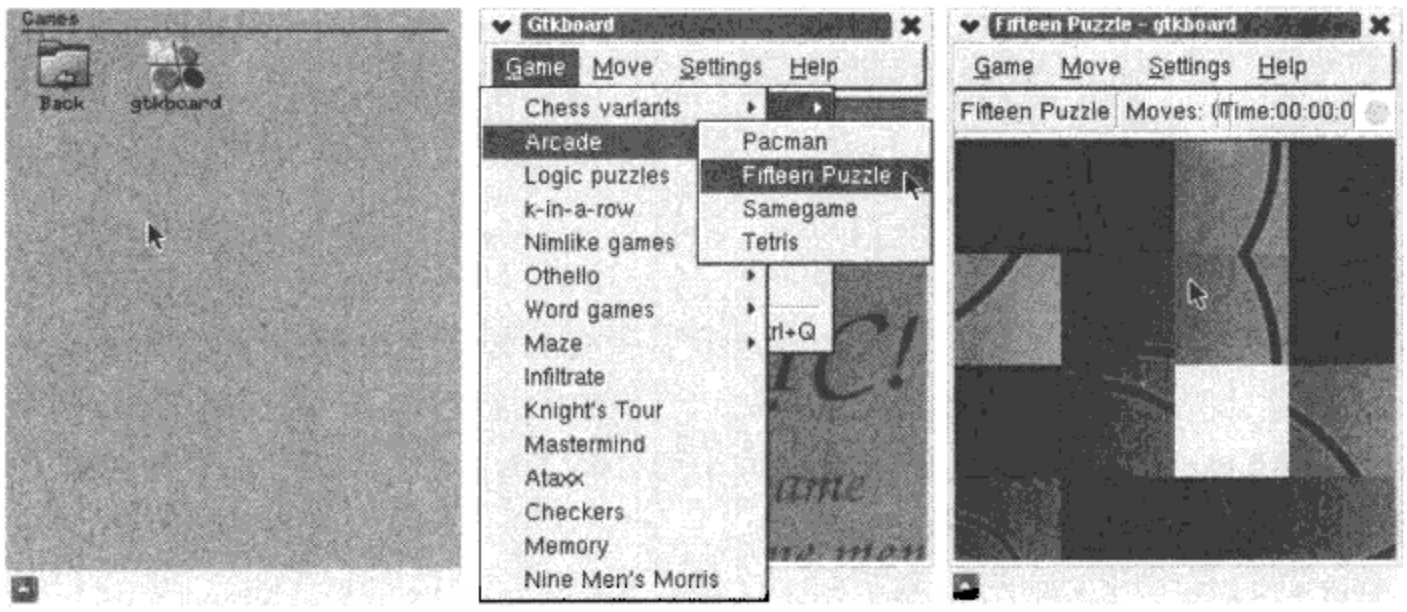


图 26.19 gtkboard 界面

启动 gtkboard 后，在串口控制台中会看到如下警告信息：

```
(gtkboard:855): Gdk-WARNING **: Error converting from UTF-8 to STRING:
Conversion from character set 'UTF-8' to 'ISO-8859-1' is not supported
```

解决方法为：把编译器 lib 目录下的 gconv 文件夹复制到 NFS 文件系统的/usr/lib 目录中。可以只选取其中的两个文件（gconv-modules 和 ISO8859-1.so）。

```
$ cd /scratchbox/compilers/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/gconv/
$ mkdir /work/nfs_root/fs_xwindow/usr/lib/gconv
$ cp gconv-modules /work/nfs_root/fs_xwindow/usr/lib/gconv
$ cp ISO8859-1.so  /work/nfs_root/fs_xwindow/usr/lib/gconv
```

最后，运行 gtkboard 时，会有如下警告信息：

```
(gtkboard:798): GLib-WARNING **: getpwuid_r(): failed due to unknown user id (0)
```

执行“whoami”命令，也会有类似的信息：

```
# whoami
whoami: unknown uid 0
```

原因是“不知道用户 ID 0 对应的用户名是什么”，解决方法为：在根文件系统中构建一个/etc/passwd 文件，并且将库文件 libnss_files*复制到根文件系统的 lib 目录。

/etc/passwd 文件中包含了所有用户登录名清单，为所有用户指定了主目录，在登录时使用的 shell 程序名称等，还保存了用户口令，给每个用户提供系统识别号。它是一个纯文本文件，每行采用了相同的格式：

```
name:password:uid:gid:comment:home:shell
```

本书不使用密码，上述“password”可以为空。/etc/passwd 文件内容如下：

```
root::0:0:root:/root:/bin/sh
```

使用以下命令复制库文件 libnss_files*：

```
$ cd /work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/
$ cp libnss_files* /work/nfs_root/fs_xwindow/lib/ -d
```

26.6.3 裁剪文件系统

本章配置程序时，都是使用“—prefix=/usr”，所有的编译结果都保存在 Scratchbox 的/usr 目录下。为了方便，在上面进行实验时，直接将这个目录复制到 NFS 文件系统/work/nfs_root/fs_xwindow 下。fs_xwindow 大小为 314MB，而开发板的 root 分区只有 54MB，需要裁剪。裁剪方法可以分为以下 3 类。

① 删除开发程序时用到的静态库、头文件、文档。

② 去除不需要的应用程序、库。

③ 各个程序、库含有一些符号信息（symbols，供调试用），使用 arm-linux-strip 工具去除。

下面的操作都是在基于/work/nfs_root/fs_xwindow 目录中进行的。

1. 删除开发资料

① 静态库。

```
$ cd usr/lib
$ for i in $(find -name *.a); do rm -rf $i; done
```

② 头文件。

```
$ rm -rf usr/include
```

③ 文档。

```
$ rm -rf usr/share/gtk-doc usr/share/doc usr/man usr/share/man usr/info
usr/openssl/man
```

④ 开发工具。

```
$ rm -rf usr/share/glib-2.0 usr/share/build-essential usr/share/libtool
usr/lib/pkgconfig
```

2. 删除不需要的应用程序和库

在编译 Xorg 时，生成了很多应用程序和库，它们没有全部用到。

本书用到的应用程序有下面 4 类，其他的程序都删除。

① busybox。

② Xorg 的程序：Xfbdev、xcalc、xeyes 和 twm。

③ Matchbox 的应用程序，它们的名字都以 mb 或 matchbox 开头。

④ 另外移植的 xterm 和 gtkboard。

执行以下命令删除不需要的应用程序。

```
$ cd usr/bin
$ for i in $(ls); \
do [ ! -L $i ] && \
    case $i in \
        busybox|Xfbdev|xcalc|xeyes|twm|mb*|matchbox*|xterm|gtkboard) ;;  \
        *)rm -f $i;; \
    esac; \
done
```

对于不需要的库，可以通过 ldd 命令来反向确定。使用 ldd 命令可以查看一个可执行程序用到的动态库，将上面几个程序使用到的库记录下来后，就可以据此删除 lib/、usr/lib 目录下其他没被用到的动态库了。

主机中的 ldd 命令无法处理 ARM 程序，可以在 Scratchbox 里查看，比如对于 Xfbdev：

```
> ldd /usr/bin/Xfbdev
        libXfont.so.1 => /usr/lib/libXfont.so.1 (0x00000000)
        libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x00000000)
        libfontenc.so.1 => /usr/lib/libfontenc.so.1 (0x00000000)
        libz.so.1 => /usr/lib/libz.so.1 (0x00000000)
        libpixman-1.so.0 => /usr/lib/libpixman-1.so.0 (0x00000000)
        libXv.so.1 => /usr/lib/libXv.so.1 (0x00000000)
        libXext.so.6 => /usr/lib/libXext.so.6 (0x00000000)
        libX11.so.6 => /usr/lib/libX11.so.6 (0x00000000)
        libxcb-xlib.so.0 => /usr/lib/libxcb-xlib.so.0 (0x00000000)
        libxcb.so.1 => /usr/lib/libxcb.so.1 (0x00000000)
        libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0x00000000)
        libdl.so.2 => /lib/libdl.so.2 (0x00000000)
        libXau.so.6 => /usr/lib/libXau.so.6 (0x00000000)
        libm.so.6 => /lib/libm.so.6 (0x00000000)
        librt.so.1 => /lib/librt.so.1 (0x00000000)
        libc.so.6 => /lib/libc.so.6 (0x00000000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x00000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00000000)
```

它用到了 libXfont.so.1 等 17 个库，最后的/lib/ld-linux.so.2 是连接器/加载器。手工选出这些程序使用的库是一件繁琐的事，可以使用脚本/work/tools/getlib.sh 来自动完成，在 Scratchbox 里执行。

```
$ /work/tools/getlib.sh  /  /tmplib  cp.log
```

getlib.sh 文件中定义了要使用的应用程序，上面的指令将生成一个名为 tmpdir 的目录，里面存放了这些应用程序所依赖的动态库。将原来 lib、usr/lib 目录下的动态库删除后，就可以使用 tmpdir 去覆盖它们了，指令如下：

```
$ cd /work/nfs_root/fs_xwindow
$ rm -f lib/*.so*
$ rm -f usr/lib/*.so*
$ cp -rfd /scratchbox/users/book/tmplib/*  /work/nfs_root/fs_xwindow/
```

不包括符号连接文件，原来 usr/lib 目录下共有 106 个库，lib 目录下共有 28 个库；挑出不使用的库后，usr/lib 目录下只有 43 个库，lib 目录下只有 11 个库。

3. 删除辅助调试的信息

使用 file 命令查看某个库文件或应用程序时，会发现类似以下的内容：

```
$ file usr/lib/libX11.so.6.2.0
usr/lib/libX11.so.6.2.0: ELF 32-bit LSB shared object, ARM, version 1, not
stripped
```

“not stripped”表示还没有裁剪符号表等供调试用的信息，这使得文件非常庞大。比如对于上面的 libX11.so.6.2.0 文件，裁剪前后的大小为 10961842 字节和 973240 字节。

裁剪符号信息，可以大幅减小程序，这通过 arm-linux-strip 命令来进行，分别进入 bin、sbin、lib、usr/bin、usr/sbin、usr/lib 目录，然后执行以下命令。

```
$ cd /work/nfs_root/fs_xwindow
$ for dir in bin sbin lib usr/bin usr/sbin usr/lib; \
do \
    cd $dir; \
    for file in $(ls); do [ ! -L $file ] && arm-linux-strip $file; done ; \
    cd -; \
done
```

4. 删除、替换其他文件

usr/share/locale 目录下是对各种语言的支持文件，只保留英文和中文。使用以下命令删除非“en”、“zh”开头的目录，这个目录的大小将从 18MB 变为 1.1MB。

```
$ cd /work/nfs_root/fs_xwindow/usr/share/locale
$ for i in $(ls);\
do \
    case $i in \
        en*|zh*)echo $i;;  \
        *)rm -rf $i;; \
```

```
    esac; \
done
```

usr/share/gtk-2.0 目录下是一些源代码，可以删除。

```
$ rm -rf usr/share/gtk-2.0
```

usr/share/目录下的 dpatch、cdbs 子目录，被用来维护 debian 包，可以删除。

```
$ rm -rf usr/share/dpatch  usr/share/cdbs
```

usr/lib/terminfo 目录的内容被用来描述打印机和终端的能力，大小为 6.2MB，可以删除（主机上相同的目录只有 56KB，有需要时可以考虑使用它）。

```
$ /work/nfs_root/fs_xwindow
$ rm -rf usr/lib/terminfo
$ rm -rf usr/share/terminfo
```

NFS 文件系统中的字库/usr/lib/X11/fonts 是从 Xorg 中编译出来的，它有 49MB；主机上/usr/share/fonts/X11 目录下的字库只有 19MB。在对字库的设置、使用不清楚时，可以直接使用主机中的字库。把它们复制到 NFS 文件系统中。

```
$ rm -rf /work/nfs_root/fs_xwindow/usr/lib/X11/fonts/*
$ cp -rf /usr/share/fonts/X11/* /work/nfs_root/fs_xwindow/usr/lib/X11/fonts
```

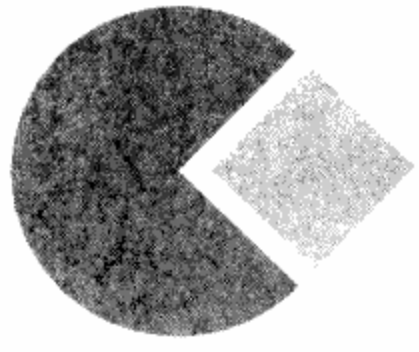
经过上面的裁剪之后，NFS 文件系统/work/nfs_root/fs_xwindow 的大小从 314MB 变成了 55MB。如果对系统很熟悉，还有很大的裁剪余地，比如字库 usr/lib/X11/fonts 目录有 19MB，可以去掉不使用的字库（支持中文的字库可以做到 3MB 以下）。

为了让系统启动后自动运行基于 X 的 GUI 系统，需要修改 etc/init.d/rcS 文件。设置 HOME 环境变量将使得 Matchbox 运行时建立的文件夹“.matchbox”位于/root 目录下，里面是一些配置信息；“sleep 1”使得在运行 Matchbox 的程序之前，确保 Xfbdev 已经初始化完毕。增加的内容如下：

```
export HOME=/root
export DISPLAY=:0
Xfbdev -mouse mouse -keybd keyboard >/dev/null 2>&1 &
sleep 1
matchbox-session >/dev/null 2>&1 &
```

现在可以使用 mkyaffsimage 工具制作 yaffs 文件系统映象，执行以下命令：

```
$ cd /work/nfs_root/
$ mkyaffsimage fs_xwindow fs_xwindow.yaffs
```



第 27 章　Linux 应用程序调试技术

本章目标

- 掌握使用 strace 工具跟踪系统调用和信号的方法
- 掌握各类内存测试工具，比如 memwatch
- 掌握使用库函数 backtrace 和 backtrace_symbols 来定位段错误

程序的调试在开发过程中占据了大部分的时间，在 Linux 中经常使用 GDB 工具来调试程序，GDB 的相关资料很丰富。本书介绍其他的调试方法，它们小巧、方便。

27.1　使用 strace 工具跟踪系统调用和信号

27.1.1　strace 介绍及移植

1．strace 介绍

strace 是一个很有用的诊断、学习、调试工具。使用时无需重新编译程序，这也使得可以用来跟踪没有源代码的程序。系统调用和信号是发生在用户空间和内核空间边界处的事件，检查这些边界事件有助于隔离错误、检查完整性、跟踪程序。

使用 strace 工具来执行程序时，它会记录程序执行过程中调用的系统调用、接收到的信号。通过查看记录结果，可以知道程序打开了哪些文件（open）、打开是否成功、对文件进行了哪些读写操作（read、wirte、ioctl 等）、映射了哪些内存（mmap）、向系统申请了多少内存等。

2．strace 移植

strace 的源码为/work/debug/strace-4.5.15.tar.bz2，在 ARM 平台上使用时要打上补丁 strace-fix-arm-bad-syscall.patch（它也在/work/debug 目录下）。

首先执行以下命令给 strace 打上补丁。

```
$ tar xjf strace-4.5.15.tar.bz2
$ cd strace-4.5.15/
$ patch -p1 < ../strace-fix-arm-bad-syscall.patch
```

然后在 strace-4.5.15 目录下执行以下命令编译 strace。

```
$ ./configure --host=arm-linux CC=arm-linux-gcc
$ make
```

上述命令将在 strace-4.5.15 目录下生成一个名为 strace 的可执行程序，将它复制到开发板根文件系统中即可使用。

27.1.2 使用 strace 来调试程序

1. strace 的用法

直接运行 strace 可以看到它的用法，如下所示：

```
# strace
usage: strace [-dffhiqrtttTvVxx] [-a column] [-e expr] ... [-o file]
              [-p pid] ... [-s strsize] [-u username] [-E var=val] ...
              [command [arg ...]]
   or: strace -c [-e expr] ... [-O overhead] [-S sortby] [-E var=val] ...
              [command [arg ...]]
```

上面的“[command [arg ...]]”表示要执行的程序及其参数；前面是各种选项。下面是几个常用的选项。

-f：除了跟踪当前进程外，还跟踪其子进程。

-o file：将输出信息写到文件 file 中，而不是显示到标准错误输出（stderr）。

-p pid：绑定到一个由 pid 对应的正在运行的进程。此参数常用来调试后台进程。

-t：打印各个系统调用被调用时的绝对时间，相观察程序各部分的执行时间时可以使用这个选项。

-tt：与-t 选项相似，打印的时间精度为μs。

-r：与-t 选项相似，打印的时间为相对时间。

2. strace 输出结果分析

使用 strace 的最简单的例子为：

```
# strace cat /dev/null
```

它的输出结果如下，其中的省略号表示还有其他系统调用，本书没有将它们打印出来。

```
01 execve("/bin/cat", ["cat", "/dev/null"], [/* 6 vars */]) = 0
02 ……
```

```
03 open("/lib/libcrypt.so.1", O_RDONLY)      = 3
04 ……
05 open("/lib/libm.so.6", O_RDONLY)          = 3
06 ……
07 open("/lib/libc.so.6", O_RDONLY)          = 3
08 ……
09 open("/dev/null", O_RDONLY|O_LARGEFILE)  = 3
10 read(3, "", 8192)                         = 0
11 close(3)                                  = 0
12 …
```

第1行表示通过系统调用execve来建立一个进程，它就是“cat /dev/null”对应的进程。在控制台中执行各种命令，比如“ls”、“cd”时，都是通过系统调用execve来建立它们的进程的。通过strace，可以看到程序运行的细节。

第2～7行打开动态连接库，如果cat程序是静态连接的，这几个步骤将不需要。

第9～10行才是“cat /dev/null”命令的真正处理过程，首先打开“/dev/null”文件，然后读取它的内容。

在strace的输出结果中，每一行对应一个系统调用：左起是系统调用的名字，紧接着是被包含在括号中的参数，最后是它的返回值，比如上面输出结果的第7行。

系统调用出错时（返回值通常是-1），在返回值的后面会打印错误记号及其注释，比如：

```
open("/foo/bar", O_RDONLY) = -1 ENOENT (No such file or directory)
```

接收到信号时会将信号记号及其注释打印出来，比如执行以下命令使用strace在后台启动一个sleep进程，然后向这个进程发送SIGINT信号。

```
# strace -o sleep.log sleep 100 &
# kill -INT 868          // 假设sleep的进程号为868，可以使用ps命令查看
```

在sleep.log文件中可以发现如下字样，表示接收到SIGINT信号，它是“Interrupt”信号。

```
--- SIGINT (Interrupt) @ 0 (0) ---
+++ killed by SIGINT +++
```

对于系统调用的参数，有多种打印格式，往往让人一目了然。下面是一些常见的格式。

```
01 open("xyzzy", O_WRONLY|O_APPEND|O_CREAT, 0666) = 3
02 lstat("/dev/null", {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
03 lstat("/foo/bar", 0xb004) = -1 ENOENT (No such file or directory)
04 read(3, "root::0:0:System Administrator:/"..., 1024) = 422
```

第1行的系统调用open有3个参数，第一个为文件名，使用字符串格式表示；第二个为flag标志，它由3个按位相或的宏组成；第三个为mode参数，它使用八进制表示。

第2行系统调用lstat的第二个参数的数据类型为“struct stat”，它被展开了。lstat是第一个参数是输入参数，第二个参数是输出参数。如果系统调用失败，相应的输出参数不会被展

开，比如第 3 行的第二个参数。

当参数是字符串指针时，这些字符串将被打印出来。默认只打印字符串的前 32 字节，其余字节使用省略号表示，这个省略号紧跟在双引号包含起来的被打印字符之后，比如第 4 行的第二个参数。

对于比较简单的指针或者数组，它们的内容被中括号包含起来，其中的各个元素使用逗号来分离，比如：

```
getgroups(32, [100, 0]) = 2
```

最后，位的集合（bit-sets）也是使用中括号包含起来的，其中的元素以空格分离，比如：

```
sigprocmask(SIG_BLOCK, [CHLD TTOU], [ ]) = 0
```

上面代码的第二个参数是两个信号 SIGCHLD 和 SIGTTOU 的集合。有时候集合的元素很多，只打印出不使用的元素比较直观，这时可以加上前缀“~”，比如下面语句中第二个参数表示信号的全集：

```
sigprocmask(SIG_UNBLOCK, ~[], NULL) = 0
```

3. 调试程序

在前面移植基于 X 的 GUI 程序时，就多次使用 strace 工具来跟踪程序，根据其中的出错信息建立了一些必需的目录，复制字库到特定的目录等。当不了解一个程序依赖于哪些目录和文件时，可以使用 strace 工具来跟踪它。

下面举几个例子来说明如何使用 strace 来调试程序。

（1）使用 strace 来定位 gtkboard 的警告信息。

在第 26 章移植 gtkboard 时，它启动之后在串口控制台中可以看到如下警告信息：

```
(gtkboard:855): Gdk-WARNING **: Error converting from UTF-8 to STRING:
Conversion from character set 'UTF-8' to 'ISO-8859-1' is not supported
```

解决方法为：把编译器 lib 目录下的 gconv 文件夹复制到开发板根文件系统的/usr/lib 目录中。

是怎么知道这个解决方法的呢？如果深入分析 gtkboard 的代码，自然可以知道解决方法，但是效率太低，可以借助 strace 来分析。

通过以下命令启动 gtkboard。

```
# strace -o gtkboard.log gtkboard &
```

然后查看 gtkboard.log 文件，发现如下字样：

```
open("/usr/lib/gconv/gconv-modules.cache", O_RDONLY) = -1 ENOENT (No such file
or directory)
open("/usr/lib/gconv/gconv-modules", O_RDONLY) = -1 ENOENT (No such file or
directory)
write(2, "\n(gtkboard:855): GLib-WARNING **"..., 82) = 82
```

而交叉编译工具链中 lib 目录下刚好有 gconv 目录，把它复制到开发板根文件系统的/usr/lib 目录后重新启动 gtkboard，这些警告信息消失。

（2）使用 strace 来测量程序的执行时间。

如果发现某个程序突然执行得很慢，通常需要找出其中哪部分代码执行的时间过长。使用 strace 工具可以轻易达到这个目的。

执行以下命令，可以发现第 2 行和第 3 行的时间相差 2s 左右，符合“sleep 2”意图。

```
# strace -r sleep 2
……
01     0.002608 rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
02     0.002392 nanosleep({2, 0}, {2, 0}) = 0
03     2.005517 exit_group(0)                  = ?
```

27.2 内存调试工具

27.2.1 使用 memwatch 进行内存调试

1. memwatch 介绍

段错误与内存错误是 C 语言编程中经常碰到的问题，段错误的调试与解决在下节讲述。所以内存错误是指使用动态分配的内存时出现的各种错误，比如内存泄漏（即调用 malloc 分配的内存没有使用 free 释放掉）和缓冲区溢出（例如对越界存储动态分配的数组）是一些常见的问题。

memwatch 由 Johan Lindh 编写，是一个开放源代码 C 语言内存错误检测工具。它可以跟踪程序中的内存泄漏和错误，支持 ANSI C，提供结果日志纪录，能检测双重释放（double-free）、错误释放（erroneous free）、没有释放的内存（unfreed memory）、溢出和下溢等。

memwatch 并不是一个可以单独运行的程序，它提供一套实现动态内存管理、检测的代码，用它们来代替标准 C 库中的相关函数。它有两个文件：memwatch.h 和 memwatch.c。前者将 malloc、free 等库函数重新定义为在 memwatch.c 文件中实现的相应库函数，部分代码如下：

```
#define malloc(n)       mwMalloc(n,__FILE__,__LINE__)
#define strdup(p)       mwStrdup(p,__FILE__,__LINE__)
#define realloc(p,n)    mwRealloc(p,n,__FILE__,__LINE__)
#define calloc(n,m)     mwCalloc(n,m,__FILE__,__LINE__)
#define free(p)         mwFree(p,__FILE__,__LINE__)
```

memwatch.h 和 memwatch.c 在/work/debug/memwatch-2.71.tar.gz 压缩包中，下面会用到它们。

2. memwatch 调试实例

要使用 memwatch，需要完成以下 3 点。

① 在代码中加入头文件 memwatch.h。

② 程序的代码与 memwatch.c 一起编译、连接。

③ 使用 gcc 编译器进行编译时要定义宏 MEMWATCH、MEMWATCH_STDIO，即在编译程序时增加“-DMEMWATCH -DMEMWATCH_STDIO”标志。

实例程序在/work/debug/examples/memtest 目录下，名为 memtest.c，代码如下：

```
01 #include <stdlib.h>
02 #include <stdio.h>
03 #include "memwatch.h"
04
05 int main(void)
06 {
07     char *ptr1;
08     char *ptr2;
09
10     ptr1 = malloc(512);
11     ptr2 = malloc(512);
12
13     ptr1[512] = 'A'
14
15     ptr2 = ptr1;
16
17     free(ptr2);
18     free(ptr1);
19
20     return 0;
21 }
```

其中，第 13 行的错误为缓冲区溢出；第 15 行修改 ptr2 变量的值后，将导致第 17、18 行释放 ptr1 对应的内存两次（double-free），而原来 ptr2 对应的内存没有被释放。

下面使用 memwatch 查看是否能检查出这些错误。

① 源文件 memtest.c 是第 3 行中，已经包含了头文件 memwatch.h。

② /work/debug/examples/memtest 目录下的 Makefile 内容如下，编译程序时增加了“-DMEMWATCH -DMEMWATCH_STDIO”标志，并且 memwatch.c 也一同编译、连接了。

```
CC=arm-linux-gcc
CFLAGS=-DMEMWATCH -DMEMWATCH_STDIO

memtest: memtest.o memwatch.o
```

```
	$(CC) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $^

clean:
	rm -f memtest *.o
```

在/work/debug/examples/memtest目录下执行make命令，即可生成可执行程序memtest，将它复制到根文件系统中然后运行，会生成一个记录文件memwatch.log，其内容如下：

```
01
02 ============ MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh ============
03
04 Started at Thu Jan  1 13:50:30 1970
05
06 Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
07 mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32
08
09 overflow: <3> memtest.c(17), 512 bytes alloc'd at <1> memtest.c(10)
10 double-free: <4> memtest.c(18), 0x1a1fc was freed from memtest.c(17)
11
12 Stopped at Thu Jan  1 13:50:30 1970
13
14 unfreed: <2> memtest.c(11), 512 bytes at 0x1a42c    {FE FE FE FE FE FE FE
FE FE FE FE FE FE FE FE FE ................}
15
16 Memory usage statistics (global):
17  N)umber of allocations made: 2
18  L)argest memory usage      : 1024
19  T)otal of all alloc() calls: 1024
20  U)nfreed bytes totals      : 512
```

第9行表示发生了缓冲区上溢，“memtest.c（17）”并不是表示在memtest.c是第17行发生了上溢，它表示这个错误是当程序执行到memtest.c的第17行“free（ptr2）”时才检测到的；“512 bytes alloc’d at <1> memtest.c（10）”表示这个出错的缓冲区大小为512字节，是在memtest.c的第10行分配的。根据这些信息查看代码，可以容易地发现是memtest.c的第13行代码导致这个错误。

第10行表示发生了双重释放（double-free）的错误，其中的“memtest.c（18）”表示这个错误是当程序执行到memtest.c的第18行“free（ptr1）”时才检测到的；“0x1a1fc was freed from memtest.c（17）”表示首地址为0x1a1fc的内存在memtest.c是第17行已经被释放过了。

第 14 行表示有一块内存没被释放（unfreed memory），“memtest.c（11），512 bytes at 0x1a42c”表示这块内存是在 memtest.c 的第 11 行分配，大小为 512 字节，首地址为 0x1a42c。

第 16～20 行是一些统计信息：第 17 行表示分配了两次内存，第 18 行表示程序结束时能够使用的最大动态内存，第 19 行表示总共分配的动态内存，第 20 行表示未释放的内存。

27.2.2 其他内存工具介绍：mtrace、dmalloc、yamd

memwatch 是使用得最为广泛的内存错误检测工具，但是还有其他相同类型的工具，它们要么功能不像 memwatch 一样强大（比如 mtrace），要么使用相对复杂（比如 dmalloc），要么只能用于 x86（比如 yamd）。

1. mtrace

mtrace 是 GNU C 库中的一个函数，它是最简单的内存泄漏检测工具，可以探测出由于不成对使用的分配、释放内存函数时引起的内存泄漏。它在 mcheck.h 文件中声明，原型如下：

```
void mtrace(void);
void muntrace(void);
```

它给 malloc、realloc、free 这 3 个库函数安装钩子函数。当程序执行这 3 个函数时，相应的钩子函数会记录分配内存、释放内存的信息，这些信息被存放在环境变量 MALLOC_TRACE 指定的文件中。如果没有设置环境变量 MALLOC_TRACE，则 mtrace 函数不起作用。这些输出信息不便于阅读，可以使用一个名字也叫 mtrace 的 perl 脚本来显示这些信息（用法为“mtrace logfile”）。

可以调用 muntrace 函数来去掉这些钩子函数。

要使用 mtrace 函数，需要确保以下 3 点。

① 程序中要包含头文件 mcheck.h。

② 在程序开头调用 mtrace 函数。

③ 执行程序之前设置环境变量 MALLOC_TRACE，它是一个文件名，用来保存输出信息。

mtrace 函数的功能很弱小，当出现更严重的内存错误（比如多次释放）时，它导致程序无法执行。另外，uClibc 库中没有 mtrace 函数。由于这些缺点，mtrace 函数很少使用。

2. dmalloc

dmalloc（Debug Malloc Library）是一个函数库，它会重新定义大部分与内存有关的函数，包括 malloc、free、memset、memcpy、strcat、strcpy 等，还会在程序的结束点（exit point）将剩余未释放的动态内存释放掉，然后将结果写进记录文件。

从 http://dmalloc.com/网站下载 dmalloc 后编译、安装，可以得到以下文件。

① 头文件 dmalloc.h。

② 动态库文件 libdmalloc.so。

③ 可执行程序 dmalloc，它被用来设置 dmalloc 的环境变量，也可以自己设置环境

变量。

④ 其他文件。

使用 dmalloc 的步骤如下。

① 首先在程序代码中包含头文件 dmalloc.h，并将动态库文件 libdmalloc.so 连接进程序中（连接时加上“-ldmalloc”选项）。

② 然后设置环境变量 DMALLOC_OPTIONS，有以下两种方法。

- 手工设置：比如执行以下命令。

```
# export DMALLOC_OPTIONS=log=logfile,debug=0x3
```

“log=logfile”表示把检查结果写入名为 logfile 的文件中；“debug=0x3”是一个十六进制代码，被称为“debug bitmask”，它的每一个值为 1 的位用来决定 dmalloc 记录哪些数据，这些位在 debug_tok.h 中定义，部分含义如下：

none（0x0）：取消 dmalloc 所有功能（不能与其他选项共享）。

log-stats（0x1）：记录基本内存运作的总结数据。

log-non-free（0x2）：记录未被释放的内存。

log-known（0x4）：只记录有程序指针参考的未释放内存（不能与 0x2 共享）。

log-trans（0x8）：记录内存分配及释放的事件。

log-admin（0x20）：记录分配及释放内存请求的数据。

log-blocks（0x40）：记录内存区块的运作。

根据这些位的含义，可知“debug=0x3”表示记录基本内存运作的总结数据、记录未被释放的内存。

- 使用可执行程序 dmalloc 进行设置。

比如执行以下命令，它的功能与上面手动设置是一样的。

```
# dmalloc -p log-stats -p log-non-free -l logfile
```

③ 最后运行程序：直接运行程序，就可以在 logfile 文件中看到检查结果。

dmalloc 的更详细说明可以查看文档 docs/dmalloc.html。

3. yamd

yamd 名为“Yet Another Malloc Debugger”，可以用来查找 C 和 C++中动态的、与内存分配有关的问题。相比于上述介绍的 memwatch、mtrace、dmalloc、yamd 等工具，使用 yamd 时不用修改源代码，只需要在编译程序时加上“-g”选项。

注意　目前 yamd 只能用于 x86 平台。

yamd 的源码可从 http://www.cs.hmc.edu/～nate/yamd/网站下载，编译后生成一个可执行文件 run-yamd，使用它来执行要调试的程序。比如有个名为 test.c 的程序，可以使用以下命令编译、运行，然后可以看到 yamd 的输出信息。

```
$ gcc -g -o test test.c
$ run-yamd ./test
```

27.3 段错误的调试方法

27.3.1 使用库函数 backtrace 和 backtrace_symbols 定位段错误

访问没有权限或是根本不存在的内存时，会产生段错误（Segmentation fault），它很常见，比如访问空字符串等都会引起这类错误。

使用 GDB 可以找到段错误发生的地方，鉴于 GDB 的相关资料非常丰富，本书使用其他方法来调试。

使用库函数 backtrace 和 backtrace_symbols 来进行栈回溯，可以知道发生错误时函数的调用关系，这不依赖于其他工具。这两个函数的原型如下：

```
int backtrace(void **buffer, int size);
char **backtrace_symbols(void *const *buffer, int size);
```

C 语言中，A 函数调用 B 函数时，会在栈中保存一个地址，当 B 函数执行完毕之后，程序返回这个地址继续执行；而这个地址处于 A 函数中，可以根据 B 函数的返回地址反向找到它的调用者为 A。根据栈中的返回地址向上回溯栈，一级一级地找到各个调用函数，就可以得到完整的调用关系。

backtrace 函数就是利用这个原理得到这些调用关系的，它分析栈的内容，找到各级调用的返回地址，将它们保存在字符串数组 buffer 中，最大数目由参数 size 指定，它的返回值表示所确定的返回地址的数目。如果返回值小于或等于参数 size，表示栈中所有的内容都被分析了；如果栈很大，要想回溯所有内容，就需要增大字符串数组 buffer、增大参数 size。

backtrace_symbols 函数将这些返回地址转换为描述性的字符串，返回地址保存在字符串数组 buffer 中，参数 size 表示它们的数目。这些描述性的字符串的格式为：

```
程序名（函数名+偏移）［返回地址］
```

其中的“函数名”是根据返回地址找到的函数名称，偏移是这个返回地址与这个函数的首地址之间的偏移。如果找不到函数名称，则小括号中的“（函数名+偏移）”不打印。

backtrace_symbols 函数将这些描述性的字符串保存在一个字符串数组中，作为它的返回值。需要注意，使用完毕后这个字符串数组需要释放掉，但是它的元素（即各个字符串）不需要也不能释放。

综上所述，库函数 backtrace 和 backtrace_symbols 通常一起使用，示例代码如下：

```
void DebugBacktrace(void)
{
#define SIZE 100
    void *array[SIZE];
    int size, i;
    char **strings;

    size = backtrace (array, SIZE);
```

```
    fprintf (stderr, "\nBacktrace (%d deep):\n", size);
    strings = backtrace_symbols (array, size);
    for (i = 0; i < size; i++)
       fprintf (stderr, "%d: %s\n", i, strings[i]);
    free (strings);
}
```

这两个函数能显示正确的结果是基于以下假设的。

① 编译程序时，gcc 的优化选项是 0。

② 内联（inline）函数没有栈。

③ 尾调用的优化使得一个“stack frame”替换另一个“stack frame”（Tail-call optimization causes one stack frame to replace another）。

注意　连接程序时，使用“-rdynamic”选项，这使得程序包含更多的符号（symbol）。静态（static）函数的符号没有导出来，所以使用这些函数进行回溯时无法找到静态函数的符号。

27.3.2　段错误调试实例

当程序发生段错误时，内核会向程序发送 SIGSEGV 信号，这个信号的默认处理行为是使程序退出。可以修改信号处理函数，在程序退出之前使用库函数 backtrace 和 backtrace_symbols 打印出函数的调用关系，这有助于找到出错的代码及出错因素。

这需要修改代码，只要将 SIGSEGV 信号的处理函数设为 DebugBacktrace 函数即可。示例代码在/work/debug/examples/segfault 目录中，源程序为 segfault.c，代码如下：

```
01 #include <stdio.h>
02 #include <signal.h>
03 #include <execinfo.h>
04
05 void A(int a);
06 void B(int b);
07 void C(int c);
08 void DebugBacktrace(void);
09
10 void A(int a)
11 {
12     printf("%d: A call B\n", a);
13     B(2);
14 }
15
16 void B(int b)
17 {
18     printf("%d: B call C\n", b);
```

```
19      C(3);       /* 这个函数调用将导致段错误 */
20  }
21
22  void C(int c)
23  {
24      char *p = (char *)c;
25      *p = 'A';   /* 如果参数 c 不是一个可用的地址值，则这条语句导致段错误 */
26      printf("%d: function C\n", c);
27  }
28
29  /* SIGSEGV信号的处理函数，回溯栈，打印函数的调用关系 */
30  void DebugBacktrace(void)
31  {
32  #define SIZE 100
33      void *array[SIZE];
34      int size, i;
35      char **strings;
36
37      fprintf (stderr, "\nSegmentation fault\n");
38      size = backtrace (array, SIZE);
39      fprintf (stderr, "Backtrace (%d deep):\n", size);
40      strings = backtrace_symbols (array, size);
41      for (i = 0; i < size; i++)
42          fprintf (stderr, "%d: %s\n", i, strings[i]);
43      free (strings);
44      exit(-1);
45  }
46
47  int main(int argc, char **argv)
48  {
49      char a;
50
51      /* 设置SIGSEGV信号的处理函数 */
52      signal(SIGSEGV, DebugBacktrace);
53
54      A(1);
55      C(&a);
56
57      return 0;
```

```
58 }
59
```

第1～27行的代码定义了A、B、C共3个函数，A调用B，B调用C。在函数C中，第24、25行的代码有漏洞，如果参数c不是一个可用的地址值，则第25行的赋值语句导致段错误。在函数B中，故意使第19行调用函数C时传入一个非法地址。

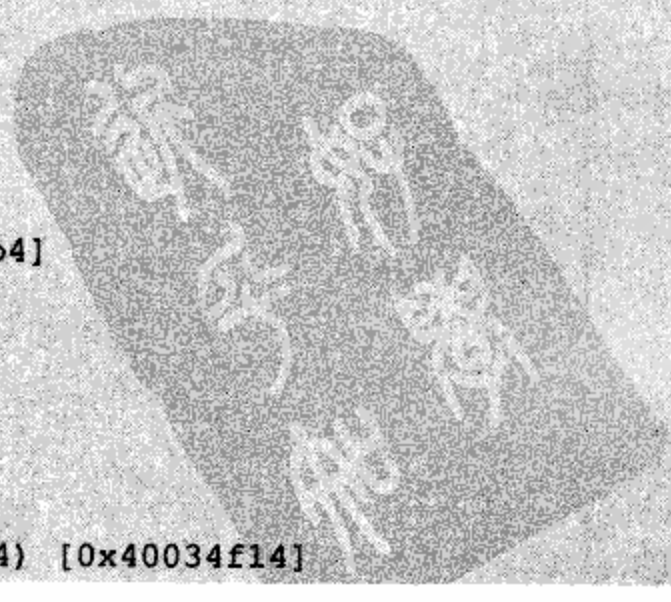
第30行是信号处理函数，它通过库函数backtrace和backtrace_symbols获得并打印函数的调用关系后，退出程序（第44行）。这个函数是本实例的重点，读者可以在自己的应用程序代码中加入这个函数，然后使用第52行的函数将它设为SIGSEGV信号的处理函数。

segfault.c的Makefile如下，可以看到，编译时选项为“-g -O0”，连接时选项为“-rdynamic”。

```
CC=arm-linux-gcc
CFLAGS=-g -O0
LDFLAGS=-rdynamic

segfault: segfault.o
        $(CC) $(LDFLAGS) -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $^

clean:
        rm -f segfault *.o
```

在/work/debug/examples/segfault目录下执行make命令生成可执行程序segfault，把它复制到开发板根文件系统/usr/bin目录下。运行后可以看到如下信息：

```
# segfault
1: A call B
2: B call C

Segmentation fault
Backtrace (6 deep):
0: segfault(DebugBacktrace+0x30) [0x89b4]
1: /lib/libc.so.6 [0x4004bd10]
2: segfault(B+0x28) [0x893c]
3: segfault(A+0x28) [0x890c]
4: segfault(main+0x2c) [0x8a84]
5: /lib/libc.so.6(__libc_start_main+0xe4) [0x40034f14]
```



标号0～5行表示从下到上的函数调用关系：标号5的__libc_start_main函数调用第4行的main函数，main函数调用标号3的A函数，A函数调用标号2的B函数。B调用C时出

错，这导致内核发出 SIGSEGV 信号，这时正常的程序流程被打断，信号处理函数 DebugBacktrace 被强行调用，标号 1、0 的行表示处理信号时的函数调用关系。

从这几个标号可以看出，当 main 函数调用 A、A 调用 B 时，出现段错误。这些调用关系可以帮助开发人员缩小定位错误的范围，在很复杂的程序中尤其如此。

从上面的输出信息“2: segfault（B+0x28）[0x893c]”中可以知道，B 函数调用的某个函数的返回地址为 0x893c，这个地址前面的语句就是调用下一级函数。

单从这些调用关系还是不能直接看出是在函数 C 中出错，这时要用到 segfault 程序的反汇编代码。使用下面指令进行反汇编：

```
$ arm-linux-objdump -D segfault > segfault.dis
```

反汇编文件 segfault.dis 的部分内容如下：

```
00008914 <B>:
……
    8938:       eb000001        bl      8944 <C>
    893c:       e89da808        ldmia   sp, {r3, fp, sp, pc}
```

可以看到，返回地址 0x893c 前面的代码调用函数 C，所以可以确定是在函数 C 中出错。

参 考 文 献

[1] 杜春雷. ARM 体系结构与编程[M]. 北京：清华大学出版社，2003

[2] R.Rajsuman. SoC 设计与测试[M]. 北京：北京航空航天大学出版社，2003

[3] Jan Axelson. USB 大全[M]. 北京：中国电力出版社，2005

[4] 毛德操，胡希明. LINUX 内核源代码情景分析[M].浙江：浙江大学出版社，2001

[5] 徐明. GCC 中文手册[EB/OL].http://cmpp.linuxforum.net/

[6] 詹荣开. 嵌入式系统 Boot Loader 技术内幕[EB/OL].
http://www.ibm.com/developerworks/cn/linux/l-btloader/index.html

[7] 何立民. 嵌入式系统的定义与发展历史[EB/OL].
http://www.mesnet.com.cn/htm/article_view.asp?id=1079&keyword=%C7%B6%C8%EB%CA%BD%CF%B5%CD%B3%B5%C4%B6%A8%D2%E5%D3%EB%B7%A2%D5%B9%C0%FA%CA%B7&kind=%CB%F9%D3%D0

[8] 何立民. I^2C 总线的串行扩充技术[EB/OL].
http://www.symcukf.com/DataSheet/016/001/3.pdf

[9] Felix. Linux Framebuffer Driver writing HOWTO[EB/OL].
http://www.felixwoo.com/article.asp?id=78

[10] 深圳远峰公司. s3c2410 的触摸屏及模数转换[EB/OL].
http://www.embedon.com/aticle-show.asp?id=3

[11] 金步国. Linux 2.6.19.x 内核编译配置选项简介[EB/OL].
http://lamp.linux.gov.cn/Linux/kernel_options.html

[12] 杨沙洲. Linux 启动过程综述[EB/OL].
http://www-128.ibm.com/developerworks/cn/linux/kernel/startup/

[13] Charles Manning. YAFFS: the NAND-specific flash file system - Introductory Article [EB/OL].http://www.yaffs.net/yaffs-nand-specific-flash-file-system-introductory-article

[14] Rusty Russell，Daniel Quinlan，Christopher Yeoh. Filesystem Hierarchy Standard[EB/OL].
http://www.pathname.com/fhs/

[15] M. Tim Jones. Linux 网络栈剖析[EB/OL].
http://www.ibm.com/developerworks/cn/linux/l-linux-networking-stack/

[16] 司马余. SD 和 MMC 记忆卡介面技术[EB/OL].
http://www.52rd.com/S_TXT/2006_8/TXT4761.htm

[17] Jollen. Linux 2.6 的 MMC Core[EB/OL].
http://www.jollen.org/blog/2007/01/linux_26_mmc_core.html

[18] gowdy. Linux USB FAQ[EB/OL]. http://www.linux-usb.org/FAQ.html

[19] Jonathan Corbet，Alessandro Rubini，Greg Kroah-Hartman. Linux Device Drivers 3rd[EB/OL].http://lwn.net/Kernel/LDD3/

[20] 陈汉仪. Embedded Linux GUI System[EB/OL].
http://www.study-area.org/linux/embedded/articles/Embedded_Linux_GUI/Embedded_Linux_GUI.html
[21] 于明俭. 自由软件圣战 – KDE/QT .VS. Gnome/Gtk[EB/OL].
http://blog.csdn.net/xusually/archive/2007/10/17/1830101.aspx