

# ARM 嵌入式 Linux 系统开发从入门到精通

## 内容简介：

这是一本以实践为宗旨的嵌入式 ARM Linux 开发书籍，它不同于一般的教材重点讲述理论而缺乏实践的部分，也不同于许多类似书籍只针对特定开发板讲述，这对于没有开发板的读者来说很难掌握书中的内容。ARM 是当今最主流的嵌入式微处理器，本书以应用最广泛的新一代 ARM9 处理器为讲述对象。此外，Linux 是一个成熟而稳定的开放源代码操作系统，将 Linux 植入嵌入式设备具有众多的优点。本书分为三部分：第一部分讲述 ARM Linux 系统移植，其中包括嵌入式系统开发入门，交叉编译器的构建，BootLoader 的移植与实现以及 Linux 2.6 内核的编译与下载；第二部分讲述 ARM Linux 的驱动程序开发，其中包括最常见的字符设备驱动的分析，块设备驱动的分析以及网络设备驱动的分析。其中每一种类型的驱动都是利用典型的实例来讲述，使读者充分了解驱动程序的实现思想；第三部分讲述 Qt GUI 开发，其中包括 Qt 的具体安装，Qt 的核心技术，以及最新的 Qtopia Core 开发环境，最后利用实例来讲述 Qtopia Core 开发过程。总之，本书包括了嵌入式 Linux 系统移植，底层驱动实例的讲解以及上层应用的实例讲述，针对那些想从事嵌入式开发或已经从事嵌入式开发的读者来说无疑是一本难得的参考书籍。

## 前言：

嵌入式系统由于芯片、软件、网络和传感器等技术的不断发展，正在成为未来社会的“数字基因”。如今，人类已经进入了后 PC 时代，嵌入式技术已被广泛应用于科学研究、工程设计、军事技术以及文艺、商业等方方面面，成为后 PC 时代的主力军。与此同时，嵌入式 Linux 操作系统也在嵌入式领域蓬勃发展，它不仅继承了 Linux 源码开放，内核稳定性强，软件丰富等特点，而且还支持几乎所有的主流处理器和硬件平台。嵌入式硬件系统和 Linux 系统的有机结合，成为后 PC 时代计算机最普遍的应用形式。嵌入式 Linux 技术在中国有巨大的发展潜力和市场需求。有数据显示，未来两年里，在计算机、消费电子、通信、汽车电子、工业控制和军事国防这六大主要应用领域，嵌入式 Linux 产品将达到 80 亿美元的市场规模，可见这个行业的前景是非常乐观的。当然，Linux 嵌入式操作系统本身也有一定的局限性，就是开发难度过高，对于企业需要很高的技术实力。这就要求 Linux 系统厂商们不光要利用 Linux，更要掌握 Linux。此外，社会需要更多人加入到学习和使用 Linux 行业中来。

## 本书编写的目的：

嵌入式 Linux 属于一个交叉学科，并且也是一个高起点的学科，它涵盖了微电子技术、电子信息技术、计算机软件和硬件等多项技术领域的应用。另外学习嵌入式 Linux 最好具备相应的嵌入式开发板和软件，还需要有经验的人进行指导开发，目前国内大部分高校都很难达到这种要求，这也造成了目前国内嵌入式 Linux 开发人才极其缺乏的局面。

很多希望学习嵌入式 Linux 的人已经具备了一定的硬件知识，并且对操作系统原理，数据结构等都有相当的了解，但在 Linux 技术方面又是零起点。目前嵌入式 Linux 的书籍也是非常之多，但大部分都是要求读者有一定的 Linux 使用基础，对于初学者来说真的非常困难。写这本书的主要目的就是对那些没有 Linux 开发经验的初学者有个很好的指导参考作用，从而让他们少走弯路。

其次，笔者希望通过写书来总结这几年在工作中的项目经验，与更多的读者分享自己的技术，也是对自己的所做项目的一个巩固；通过写这本书，让笔者更加清楚了实践与理论之

间的联系，从而将自己的亲身经验和教训寄托在书中的每个章节。

本书的特点：

首先，本书涵盖了嵌入式 Linux 系统中最重要的三个部分：ARM Linux 系统移植，ARM Linux 驱动程序开发以及 Qt GUI 开发，这在同类书籍中比较少见。

其次，本书的讲述不依赖于具体某个厂家开发板，这样读者可以使用任意一款类似的开发板就可以进行实践学习，同时对于没有开发板的读者也可以学到更多的知识。

另外，本书提供了书中出现的所有实例的源代码，便于读者参考使用，更重要的是读者不用手动输入这些代码，从而节省时间。

本书的主要组成：

本书分为三个部分，共 12 章节，每一部分由 4 章内容组成。

第一部分讲述 ARM Linux 系统移植，首先第 1 章讲述嵌入式系统开发入门，主要针对初学者，讲述嵌入式系统的概要，ARM 处理器，ADS 工具，Linux 开发环境，以及 Linux 内核源码等。接着第 2 章讲述交叉编译工具链的构建，主要讲述交叉工具链的作用，使用分步法构建交叉工具链和使用 Crosstool 工具构建交叉工具链。第 3 章讲述嵌入式系统的 BootLoader，主要讲述嵌入式 BootLoader 的作用，基于 S3C2410 开发板的 U-Boot 分析与移植以及自己设计 BootLoader 的方法。最后第 4 章讲述嵌入式 Linux 内核移植，主要讲述移植的基本概念，内核配置、内核编译、内核下载以及构建根文件系统。

第二部分讲述 ARM Linux 驱动程序开发，首先第 5 章讲述 ARM Linux 驱动程序开发入门，主要讲述嵌入式 Linux 驱动介绍，简单的内核模块程序分析，以及 Linux 驱动开发的基本要点。接着第 6 章讲述字符设备驱动程序，主要讲述字符设备驱动相关的重要数据结构，字符设备驱动开发实例——触摸屏设备驱动开发。第 7 章讲述块设备驱动程序，主要讲述块设备相关的数据结构，块设备驱动开发实例——MMC/SD 设备驱动开发。最后第 8 章讲述网络设备驱动程序，主要讲述网络设备驱动相关的重要数据结构，网络设备驱动开发实例——CS8900A 网卡驱动开发。

第三部分讲述 Qt GUI 开发，首先第 9 章介绍了 Qt 的概要知识，包括 Linux 桌面 GUI 系统，Qt/X11，Qt/Opia Core 等，使读者对 Qt 及其在 Linux GUI 系统中的作用有个大概了解。紧接着第 10 章讲述了 Qt/X11 的安装以及非常详细的应用实例，使读者可以轻松的编写基本的 Qt 程序。第 11 章深入讨论了一些 Qt 的核心技术，重点是以 Qt 对象模型为基础的信号和槽等机制，我们通过剖析 Qt 的源代码来深入的学习 Qt 的这些核心技术，同时也为读者今后对 Qt 源代码的自行研习打下基础。最后第 12 章重点讲述 Qt/Opia Core 和 Qt/X11 的一些不同之处，包括轻量级的窗口系统，QCOP 进程间通信机制及调试工具 qvfb 等，使读者在熟悉了 Qt/X11 的基础上能够很快过渡到 Qt/Opia Core 开发。

本书的读者对象：

本书通俗易懂，可作为高等院校电子类、电气类、控制类、计算机类专业本科生、研究生学习嵌入式 Linux 的参考书目或自学教材，也可供广大希望转入嵌入式领域的科研和工程技术人员参考使用，还可作为广大嵌入式培训班的教材和教辅材料。

致谢：

首先要感谢这本书的另外一位作者欧文盛，书中 Qt GUI 部分（第 9 章到第 12 章）主要由他来完成，由于他这几年一直在国际知名的通信公司从事 Qt 方面的开发工作，所以这部分由他来完成，出版社和我都很放心。其次，我要感谢我的妻子，很特殊的是我写这本书的时间正是我妻子怀孕的期间，其实在写这本书之前已经得知妻子怀孕，所以本想放弃编写这本书，但是妻子却很坚定的支持我写这本书。所以，我认为这本书的完成离不开她对我的默默支持。其次，要感谢我的岳父、岳母，是他们对我妻子这段时间的精心照顾，才使得我有更多的时间投入到写书中。

最后，要感谢威盛电子的李松，易宏宇，周志勇，张磊等，他们为本书的完成也提供了很多的帮助。

鉴于作者水平有限，加之时间仓促，本书一定有不少错误与不清楚之处，希望得到广大读者批评与指正。有兴趣的读者可以发送 E-mail 到 [lyf99526@yahoo.com.cn](mailto:lyf99526@yahoo.com.cn) 或登录笔者的个人 Blog 来做技术上的交流：<http://mike2linus.blog.com.cn/>。

作者

2007 年 3 月 28 日

<b>第一部分</b>	<b>ARM LINUX 系统移植</b>	<b>12</b>
<b>第 1 章</b>	<b>嵌入式系统开发入门</b>	<b>13</b>
1.1	嵌入式系统介绍	13
1.1.1	嵌入式系统概述	13
1.1.2	嵌入式系统组成	15
1.2	ARM 介绍	16
1.2.1	ARM 处理器介绍	17
1.2.2	ARM 处理器的选型	18
1.2.3	S3C2410 微处理器介绍	18
1.3	ADS 集成开发环境介绍	20
1.3.1	ADS 软件组成	21
1.3.1.1	命令行开发工具	21
1.3.1.2	GUI 开发环境	23
1.3.1.3	实用程序	23
1.3.1.4	支持的软件	24
1.3.2	使用 Code Warrior IDE	24
1.3.2.1	创建项目工程	24
1.3.2.2	编译和链接项目工程	27
1.3.3	使用 AXD IDE	29
1.3.3.1	打开调试文件	29
1.3.3.2	设置断点	30
1.3.3.3	查看寄存器内容	30
1.3.3.4	查看变量值	31
1.4	嵌入式 LINUX 开发介绍	32
1.4.1	Linux 历史	32
1.4.2	Linux 开发环境	33
1.4.2.1	GCC 介绍	35
1.4.2.2	GNU Make 介绍	36
1.4.2.3	GDB 介绍	38
1.4.3	ARM Linux 系统开发流程	41
1.5	LINUX 内核介绍	43
1.5.1	Linux 内核目录结构	44
1.5.2	如何阅读 Linux 内核源代码	45
1.6	本章小节	47
1.7	常见问题	48
<b>第 2 章</b>	<b>交叉编译工具链的构建</b>	<b>49</b>
2.1	交叉编译工具链介绍	49
2.2	ARM LINUX 交叉编译工具链的构建	49
2.2.1	分步构建交叉编译链	50
2.2.1.1	建立工作目录	50
2.2.1.2	建立环境变量	51

2.2.1.3 编译、安装 Binutils.....	51
2.2.1.4 获得内核头文件.....	52
2.2.1.5 编译安装 boot-trap gcc .....	53
2.2.1.6 建立 glibc 库.....	54
2.2.1.7 编译安装完整的 gcc.....	55
2.2.1.8 测试交叉编译工具链 .....	55
2.2.2 用 Crosstool 工具构建交叉工具链.....	55
2.2.2.1 准备资源文件.....	56
2.2.2.2 建立脚本文件.....	56
2.2.2.3 建立配置文件.....	57
2.2.2.4 执行脚本 .....	57
2.2.2.5 添加环境变量.....	57
2.3 本章小节 .....	58
2.4 常见问题 .....	58
<b>第 3 章 嵌入式系统的 BOOTLOADER.....</b>	<b>60</b>
3.1 BOOTLOADER 概述.....	60
3.2 常用的嵌入式 LINUX BOOTLOADER.....	61
3.2.1 U-Boot.....	61
3.2.2 VIVI.....	61
3.2.3 Blob.....	62
3.2.4 RedBoot.....	62
3.2.5 ARMboot .....	63
3.2.6 DIY.....	63
3.3 基于 S3C2410 开发板的 BOOTLOADER 实现.....	63
3.3.1 S3C2410 开发板介绍.....	63
3.3.2 U-Boot 分析与移植.....	66
3.3.2.1 U-Boot Stage1 分析 .....	66
3.3.2.2 U-Boot Stage2 分析 .....	71
3.3.2.3 U-Boot 的移植过程.....	72
3.4 基于 S3C2410 开发板自己编写 BOOTLOADER .....	88
3.4.1 设计系统的启动流程.....	88
3.4.2 BootLoader 的具体实现.....	90
3.4.2.1 设置异常向量表.....	91
3.4.2.2 初始化看门狗和外围电路.....	92
3.4.2.3 初始化存储器.....	92
3.4.2.4 初始化堆栈 .....	93
3.4.2.5 初始化数据区.....	94
3.4.2.6 跳转到 C 程序 Main 函数.....	96
3.4.2.7 Main 函数的具体实现 .....	96
3.5 本章小节 .....	97
3.6 常见问题 .....	97
<b>第 4 章 嵌入式 LINUX 内核移植 .....</b>	<b>98</b>
4.1 移植的基本概念 .....	98

4.2 内核移植的准备 .....	99
4.3 内核移植 .....	100
4.3.1 内核配置.....	100
4.3.1.1 修改 Makefile .....	100
4.3.1.2 设置 NAND Flash 分区 .....	101
4.3.1.3 配置内核选项.....	104
4.3.2 内核编译.....	108
4.3.2.1 清除冗余文件.....	108
4.3.2.2 编译内核映像和模块 .....	108
4.3.2.3 安装模块 .....	109
4.3.3 内核下载.....	109
4.4 建立 LINUX 根文件系统.....	110
4.4.1 根文件系统的基本介绍.....	110
4.4.1.1 根文件系统的基本目录结构 .....	110
4.4.1.2 常见的根文件系统 .....	111
4.4.1.3 选择根文件系统.....	112
4.4.2 建立根文件系统.....	113
4.4.2.1 Cramfs 工具包的使用.....	113
4.4.2.2 构建 Cramfs 根文件系统.....	114
4.5 本章小节 .....	117
4.6 常见问题 .....	117
<b>第二部分 ARM LINUX 设备驱动程序开发.....</b>	<b>119</b>
<b>第 5 章 ARM LINUX 驱动程序开发入门 .....</b>	<b>120</b>
5.1 嵌入式 LINUX 驱动程序介绍.....	120
5.1.1 驱动程序的作用.....	120
5.1.2 Linux 设备驱动程序分类.....	121
5.2 最简单的内核模块举例.....	122
5.2.1 编写 Hello World 模块 .....	122
5.2.2 编写 Hello World 模块的 Makefile .....	124
5.2.3 加载和卸载 Hello World 模块.....	125
5.3 LINUX 驱动程序开发要点.....	125
5.3.1 内存与 I/O 端口.....	125
5.3.1.1 内存.....	126
5.3.1.2 I/O 端口.....	129
5.3.2 并发控制.....	130
5.3.2.1 自旋锁 (Spinlocks) .....	131
5.3.2.2 信号量 (Semaphores) .....	133
5.3.3 阻塞 (Blocking) 与非阻塞 (Nonblocking) .....	135
5.3.3.1 阻塞 (Blocking) 与非阻塞 (Nonblocking) 操作.....	135
5.3.3.2 异步通知 (Asynchronous Notification) .....	135
5.3.4 中断处理.....	136
5.3.4.1 Linux 中断及其相关函数 .....	136
5.3.4.2 ARM 中断处理.....	137

5.3.4.3 一个 Linux 中断相关的实例.....	139
5.3.5 内核调试.....	143
5.3.5.1 准备内核调试环境 .....	143
5.3.5.2 KDB 的基本用法.....	144
5.4 本章小结 .....	146
5.5 常见问题 .....	147
<b>第 6 章 字符设备驱动程序.....</b>	<b>148</b>
6.1 字符设备驱动介绍.....	148
6.1.1 字符设备驱动相关的重要结构.....	148
6.1.1.1 file_operations（文件操作）结构.....	148
6.1.1.2 file（文件）结构.....	151
6.1.1.3 inode（节点）结构 .....	152
6.1.2 主、次设备号.....	154
6.1.2.1 主、次设备号的内部表示.....	155
6.1.2.2 静态分配和释放主设备号.....	155
6.1.2.3 动态分配主设备号 .....	156
6.2 字符设备驱动开发实例.....	157
6.2.1 四线电阻式触摸屏原理.....	157
6.2.2 S3C2410 触摸屏工作原理 .....	158
6.2.3 S3C2410 的 ADC 和触摸屏接口特殊寄存器.....	159
6.2.3.1 ADC 控制（ADCCON）寄存器 .....	159
6.2.3.2 ADC 触摸屏控制（ADCTSC）寄存器 .....	160
6.2.3.3 ADC 开始延迟（ADCDLY）寄存器 .....	161
6.2.3.4 ADC 转化数据（ADCDAT0）寄存器 .....	161
6.2.3.5 ADC 转化数据(ADCDAT1)寄存器.....	162
6.2.4 触摸屏驱动概要设计.....	162
6.2.4.1 触摸屏硬件接口.....	162
6.2.4.2 触摸屏驱动程序流程设计 .....	163
6.2.5 触摸屏驱动程序分析.....	164
6.2.5.1 触摸屏设备初始化 .....	165
6.2.5.2 触摸屏设备文件操作 .....	168
6.2.5.3 open 和 release 方法 .....	168
6.2.5.4 read 和 poll 方法.....	169
6.2.5.5 触摸屏中断和 ADC 中断的实现 .....	170
6.2.6 配置和编译驱动程序.....	172
6.2.7 测试触摸屏驱动程序.....	173
6.2.8 触摸屏的校准.....	174
6.3 本章小节 .....	175
6.4 常见问题 .....	176
<b>第 7 章 块设备驱动程序 .....</b>	<b>177</b>
7.1 块设备驱动介绍 .....	177
7.1.1 块设备驱动相关的重要结构.....	177
7.1.1.1 block_device_operations（块设备操作）结构 .....	177

7.1.1.2 gendisk 结构 .....	178
7.1.1.3 request 结构 .....	180
7.1.2 请求处理 .....	182
7.1.2.1 request 函数 .....	182
7.1.2.2 request 函数实例 .....	182
7.2 块设备驱动开发实例 .....	183
7.2.1 MMC/SD 介绍 .....	184
7.2.2 S3C2410 提供的 SDI 接口 .....	186
7.2.3 SDI 相关的寄存器 .....	187
7.2.3.1 SDI 控制 (SDICON) 寄存器 .....	188
7.2.3.2 SDI 波特率预定标 (SDIPRE) 寄存器 .....	188
7.2.3.3 SDI 命令参数(SDICARG)寄存器 .....	188
7.2.3.4 SDI 命令控制(SDICCON)寄存器 .....	189
7.2.3.5 SDI 命令状态(SDICSTA)寄存器 .....	189
7.2.3.6 SDI 响应(SDIRSP)寄存器 .....	189
7.2.3.7 SDI 数据/占用定时器(SDIDTIMER)寄存器 .....	190
7.2.3.8 SDI 块大小(SDIBSIZE)寄存器 .....	190
7.2.4 MMC/SD 驱动概要设计 .....	191
7.2.4.1 MMC/SD 与主机的接口连接 .....	191
7.2.4.2 MMC/SD 驱动框架 .....	191
7.2.4.3 MMC 驱动的核心设计 .....	193
7.2.5 MMC 驱动程序分析 .....	193
7.2.5.1 MMC 初始化 .....	193
7.2.5.2 open 和 release 方法 .....	195
7.2.5.3 ioctl 方法 .....	196
7.2.5.4 MMC 驱动的 request 方法 .....	196
7.2.6 S3C2410 SDI 接口驱动分析 .....	198
7.2.6.1 SDI 初始化 .....	199
7.2.6.2 SDI 接口驱动方法 .....	199
7.2.7 配置和编译驱动程序 .....	200
7.3 本章小结 .....	200
7.4 常见问题 .....	200
<b>第 8 章 网络设备驱动程序 .....</b>	<b>202</b>
8.1 网络设备驱动介绍 .....	202
8.1.1 网络设备驱动相关的重要结构 .....	202
8.1.1.1 net_device 结构 .....	202
8.1.1.2 sk_buff 结构 .....	204
8.1.2 常见的网络术语 .....	205
8.1.2.1 常见的网络协议 .....	205
8.1.2.2 以太网介绍 .....	206
8.2 网络设备驱动开发实例 .....	207
8.2.1 CS8900A 介绍 .....	207
8.2.1.1 CS8900A 的组成部分介绍 .....	207
8.2.1.2 CS8900A 的系统应用 .....	208



8.2.2 CS8900A 网卡驱动概要设计 .....	209
8.2.2.1 CS8900A 网卡接口 .....	209
8.2.2.2 网络驱动程序的体系结构 .....	209
8.2.2.3 网络驱动程序的主要功能 .....	210
8.2.3 CS8900A 适配器驱动程序分析 .....	211
8.2.3.1 初始化 .....	211
8.2.3.2 open 和 stop 方法 .....	214
8.2.3.3 数据发送 .....	216
8.2.3.4 数据接收 .....	217
8.3 本章小结 .....	220
8.4 常见问题 .....	220
<b>第三部分 QT GUI 开发 .....</b>	<b>221</b>
<b>第 9 章 QT 概述 .....</b>	<b>222</b>
9.1 LINUX 下 GUI 介绍 .....	222
9.1.1 Linux 桌面 GUI 系统 .....	222
9.1.1.1 X Window 系统 .....	223
9.1.1.2 GNOME/Gtk+和 KDE/Qt .....	224
9.1.2 嵌入式 Linux 下的 GUI 系统 .....	226
9.2 QT/X11 介绍 .....	227
9.2.1 Qt 的历史和 Qt/X11 的由来 .....	227
9.2.2 Qt/X11 的版权问题 .....	228
9.2.3 Qt/X11 及 Qt/Windows 的系统架构图对比 .....	228
9.2.4 Qt 的特性简介 .....	228
9.3 QTOPIA CORE 介绍 .....	229
9.3.1 Qtopia Core 与 Qt/Embedded .....	229
9.3.2 Qtopia Core 的体系结构 .....	230
9.3.2.1 Frame Buffer(帧缓存)简介 .....	230
9.3.2.2 Qtopia Core 的窗口系统 .....	231
9.4 本章小结 .....	231
9.5 常见问题 .....	231
<b>第 10 章 QT/X11 初步 .....</b>	<b>233</b>
10.1 QT/X11 的安装 .....	233
10.1.1 Qt/X11 的下载及双重授权问题的说明 .....	233
10.1.2 Qt/X11 的安装详解 .....	234
10.2 QT 下的 HELLO WORLD .....	235
10.3 温度转换的小例子 .....	237
10.3.1 背景知识 .....	237
10.3.2 Quit 按钮 .....	237
10.3.3 摄氏温度的显示 .....	241
10.3.4 华氏温度的显示 .....	243
10.3.5 华氏温度和摄氏温度之间的转换 .....	247
10.3.6 保存当前的数值 .....	251

10.4 本章小结 .....	256
10.5 常见问题 .....	257
<b>第 11 章 QT 核心技术.....</b>	<b>258</b>
11.1 信号(SIGNALS)和槽(SLOTS).....	258
11.1.1 常见的 GUI 组件通信方式 .....	258
11.1.1.1 回调函数.....	258
11.1.1.2 面向对象的回调 .....	260
11.1.2 Qt 中的信号和槽(Signals and Slots).....	261
11.1.2.1 信号和槽历史和所带来的优点.....	261
11.1.2.2 信号 .....	261
11.1.2.3 槽.....	262
11.1.2.4 信号和槽的效率 .....	262
11.1.3 自定义信号和槽的小例子.....	263
11.2 QT 对象模型.....	266
11.2.1 元对象系统 (Meta-Object System) .....	266
11.2.2 信号和槽机制的实现.....	272
11.2.2.1 用 connection()建立连接.....	272
11.2.2.2 信号的发射和槽的执行 .....	278
11.2.3 元对象编译器 moc .....	282
11.2.3.1 在 Makefile 中使用 moc.....	282
11.2.3.2 moc 用法详解 .....	282
11.2.3.3 moc 及信号和槽机制的局限性.....	283
11.3 QT 的窗口系统.....	285
11.3.1 窗口部件之间的树型结构.....	285
11.3.2 窗口部件的布局管理 (Layout) .....	288
11.4 国际化.....	291
11.4.1 Qt 国际化的基本步骤.....	291
11.4.1.1 程序员的工作 .....	291
11.4.1.2 语言资源管理者和翻译工作者的工作.....	292
11.4.2 动态改变语言的小例子.....	292
11.4.3 一些注意事项.....	298
11.5 本章小结 .....	299
11.6 常见问题 .....	299
<b>第 12 章 QTOPIA CORE .....</b>	<b>301</b>
12.1 QTOPIA CORE 的安装.....	301
12.2 FRAME BUFFER 和 QVFB.....	302
12.2.1 Frame Buffer .....	302
12.2.2 编译 qvfb.....	304
12.2.3 在 qvfb 上运行 Qtopia Core 程序.....	305
<b># ./ DIGITALCLOCK -QWS -DISPLAY QVFB:0 .....</b>	<b>306</b>
12.3 移植 QT/X11 程序到 QTOPIA CORE 中 .....	307
12.4 轻量级的窗口系统 .....	309

12.5 进程间通信 .....	312
12.6 本章小结 .....	315
12.7 常见问题 .....	316
附录 A: 光盘内容 .....	317
附录 B: 参考文献 .....	317

## 第一部分 ARM Linux 系统移植

为了能让读者快速的了解嵌入式 ARM Linux 系统的开发过程,本书第一部分讲述 ARM Linux 的系统移植,这部分内容在实际工作中比较常见,是嵌入式 Linux 开发人员应该掌握的技能。该部分由四章组成:第一章介绍 ARM 嵌入式 Linux 系统概述,作为嵌入式开发入门的一章,非常适合初学者阅读,它包括嵌入式系统介绍,ARM 介绍,ADS 集成开发工具介绍,嵌入式 Linux 开发介绍以及 Linux 内核介绍,对那些刚接触嵌入式 Linux 开发的读者来说,通过本章学习将会对嵌入式 Linux 开发有个大概的了解和认识;第二章介绍交叉编译工具链的制作,对于非 X86 硬件平台的设备开发通常使用交叉编译工具链在 X86 机器上进行,该章内容是编译目标内核和程序的基础,它包括对交叉工具链的介绍,使用分步法构建交叉工具链和使用 Crosstool 构建交叉工具链,通过本章学习,读者将会对交叉工具链有深刻的认识以及可以构建自己的交叉工具链;第三章讲述 ARM Linux 的引导程序——BootLoader,这是内核移植的关键,没有一个良好的 BootLoader 来引导内核工作,再强大、稳定的内核也不能正常工作,它包括对 BootLoader 的介绍,U-boot 移植与分析以及讲述如何自己设计 BootLoader。通过本章学习,读者将会对 BootLoader 的作用有更清楚的认识,以及学会如何移植和设计 BootLoader;第四章讲述嵌入式 Linux 内核移植,也是实际工作中非常重要的内容,它包括移植的基本概念,内核配置,内核编译,以及根文件系统的构建。通过本章学习,读者将会对内核移植以及构建根文件系统有更深入的理解。

总之,读者通过对这部分内容的学习,将会了解构建嵌入式 Linux 系统所需要的一些工作,比如交叉工具链,BootLoader,内核配置,内核编译等。

# 第 1 章 嵌入式系统开发入门

本章学习目标：

- ！ 了解嵌入式系统基本概念
- ！ 了解嵌入式系统的基本组成
- ！ 了解 ARM 处理器的特点
- ！ 学会使用 ADS 集成开发工具（Code Warrior 和 AXD）
- ！ 熟悉 Linux 开发环境
- ！ 学会如何有效阅读 Linux 内核代码

## 1.1 嵌入式系统介绍

本章作为 ARM Linux 系统移植的第一章，也是本书的第一章。俗话说的好“良好的开始是成功的一半”，虽然这句话并不是真理，但是希望读者在学习任何东西之前都应该有坚定的学习态度和持之以恒的信念，同样学习本书也要有个良好的开端。首先介绍嵌入式系统的概述。

### 1.1.1 嵌入式系统概述

随着嵌入式系统在消费类电子、工业控制、航空航天、汽车电子、医疗保健、网络通信等各个领域的广泛应用，嵌入式系统这个名词已经被各行各业的人所熟悉，嵌入式系统已经走进了人们的生活。它正在以各种不同的形式悄悄地改变着人们的生产、生活方式。毋庸置疑，社会对嵌入式系统开发人员的需求也越来越大，所以现在越来越多的人已经加入到这个行业中来。嵌入式系统，英文为 **Embedded System**，从广义上讲，凡是带有微处理器的专用软、硬件系统都可称为嵌入式系统。如各类单片机和 **DSP** 系统，这些系统在完成较为单一的专业功能时具有简洁高效的特点。但是由于他们没有使用操作系统，所以管理系统硬件和软件的能力有限，在实现复杂的多任务功能时往往困难重重，甚至无法实现。从狭义上讲，那些使用嵌入式微处理器构成的独立系统，并且有自己的操作系统，具有特定功能，用于特定场合的系统。本书中所说的嵌入式系统是指狭义上的嵌入式系统。到目前为止，对于嵌入式系统还没有一个明确的定义。嵌入式系统的核心是嵌入式微处理器，该处理器都是 **RISC**（**Reduce Instruction Set Computing**，精简指令集计算机）\*（注 1）的处理器内核。

\*注 1：RISC 和 CISC(**Complex Instruction Set Computing**，复杂指令集计算机)是当前 CPU 的两种架构。它们的区别在于不同的 CPU 设计理念和方法。早期的 CPU 全部是 CISC 架构，它的设计目的是要用最少的机器语言指令来完成所需的计算任务。比如对于乘法运算，在 CISC 架构的 CPU 上，您可能需要这样一条指令：**MUL ADDRA, ADDRb** 就可以将 **ADDRA** 和 **ADDRb** 中的数相乘并将结果储存在 **ADDRA** 中。将 **ADDRA, ADDRb** 中的数据读入寄存器，相乘和将结果写回内存的操作全部依赖于 CPU 中设计的逻辑来实现。这种架构会增加 CPU 结构的复杂性和对 CPU 工艺的要求，但对于编译器的开发十分有利。比如 C 程序中的 **a\*=b** 就可以直接编译为一条乘法指令。今天只有 Intel 及其兼容 CPU 还在使用 CISC 架构。RISC 架构要求软件来指定各个操作步骤。上面的例子如果要在 RISC 架构上实现，将 **ADDRA, ADDRb** 中的数据读入寄存器，相乘和将结果写回内存的操作都必须由软件来实现，比如：**MOV A, ADDRA; MOV B, ADDRb;**

MUL A, B; STR ADDRA, A。这种架构可以降低 CPU 的复杂性以及允许在同样的工艺水平下生产出功能更强大的 CPU，但对于编译器的设计有更高的要求。

嵌入式微处理器一般具备以下 4 个主要特点：

1. 对实时多任务有很强的支持能力，能完成多任务并且有较短的中断响应时间，从而使内部的代码和实时内核的执行时间减少到最低限度。
2. 具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断。
3. 可扩展的处理器结构，以能最迅速地开展出满足应用的最高性能的嵌入式微处理器。
4. 嵌入式微处理器必须功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此。

嵌入式处理器内核按照体系结构分类：

#### 1. MIPS 处理器

MIPS 处理器是由美国 MIPS 公司研发出来的一套处理器体系，MIPS 公司是一家设计制造高性能、高档次及嵌入式 32 位和 64 位处理器的厂商，在 RISC 处理器方面占有重要地位。1984 年，MIPS 计算机公司成立。1992 年，SGI 收购了 MIPS 计算机公司。1998 年，MIPS 脱离 SGI，成为 MIPS 技术公司。MIPS 公司设计 RISC 处理器始于二十世纪八十年代初，1986 年推出 R2000 处理器，1988 年推 R3000 处理器，1991 年推出第一款 64 位商用微处理器 R4000。之后又陆续推出 R8000（于 1994 年）、R10000（于 1996 年）和 R12000（于 1997 年）等型号。随后，MIPS 公司的战略发生变化，把重点放在嵌入式系统。1999 年，MIPS 公司发布 MIPS32 和 MIPS64 架构标准，为未来 MIPS 处理器的开发奠定了基础。新的架构集成了所有原来 NIPS 指令集，并且增加了许多更强大的功能。MIPS 公司陆续开发了高性能、低功耗的 32 位处理器内核（core）MIPS324Kc 与高性能 64 位处理器内核 MIPS64 5Kc。2000 年，MIPS 公司发布了针对 MIPS32 4Kc 的版本以及 64 位 MIPS 64 20Kc 处理器内核。

#### 2. ARM 处理器

ARM（Advanced RISC Machines）处理器是由只设计内核的英国 ARM 公司研发出来的一套处理器体系，ARM 公司成立于 1990 年 11 月，其前身是 Acorn 计算机公司。ARM 是微处理器行业的一家知名企业，设计了大量高性能、廉价、耗能低的 RISC 处理器、相关技术及软件。技术具有性能高、成本低和能耗省的特点。适用于多种领域，比如嵌入控制、消费/教育类多媒体、DSP 和移动式应用等。ARM 将其技术授权给世界上许多著名的半导体、软件和 OEM 厂商，每个厂商得到的都是一套独一无二的 ARM 相关技术及服务。利用这种合伙关系，ARM 很快成为许多全球性 RISC 标准的缔造者。目前，总共有 30 家半导体公司与 ARM 签订了硬件技术使用许可协议，其中包括 Intel、IBM、三星电子、LG 半导体、NEC、SONY、飞利浦和国民半导体这样的大公司。至于软件系统的合伙人，则包括微软、升阳和 MRI 等一系列知名公司。ARM 架构是面向低预算市场设计的第一款 RISC 微处理器。

#### 3. PowerPC 处理器

二十世纪九十年代，IBM(国际商用机器公司)、Apple（苹果公司）和 Motorola（摩托罗拉）公司开发 PowerPC 芯片成功，并制造出基于 PowerPC 的多处理器计算机。PowerPC 架构的特点是伸缩性好、方便灵活。第一代 PowerPC 采用 0.6 微米的生产工艺，晶体管的集成度达到单芯片 300 万个。MPC860 和 MPC8260 是其最经典的两款 PowerPC 内核的嵌入式处理器。

#### 4. 68K/ColdFire 处理器

68K/ColdFire 处理器是 Motorola 公司独有的处理器体系。68K 内核是最早在嵌入式领域广泛应用的内核。其最著名的代表芯片是 68360。Coldfire 继承了 68K 的特点并继续兼容它。

由于嵌入式系统一般具有芯片集成度高, 软件代码小, 高度自动化, 响应速度快等特点, 特别适合于要求实时性和多任务的体系。RTOS (Real-Time Operating System, 实时操作系统) 是根据操作系统的工作特性而言的。实时是指物理进程的真实时间。实时操作系统是指具有实时性, 能支持实时控制系统工作的操作系统。首要任务是调度一切可利用的资源完成实时控制任务, 其次才着眼于提高计算机系统的使用效率, 重要特点是要满足对时间的限制和要求。一般 Windows、Unix、Linux 等桌面系统都属于分时操作系统, 在此有必要说明一下实时操作系统与分时操作系统的区别: 具体的说, 对于分时操作系统, 软件的执行在时间上的要求并不严格, 时间上的错误, 一般不会造成灾难性的后果。而对于实时操作系统, 主要任务是对事件进行实时的处理, 虽然事件可能在无法预知的时刻到达, 但是软件上必须在事件发生时能够在严格的时限内作出响应, 即使是在尖峰负荷下, 也应该如此, 系统时间响应的超时就意味着致命的失败。另外, 实时操作系统的重要特点是具有系统的可确定性, 即系统能对运行情况的最好和最坏等的情况能做出精确的估计。

到此为止, 读者应该对嵌入式系统有了大概的了解, 下面将介绍嵌入式系统的一般组成。

### 1.1.2 嵌入式系统组成

嵌入式系统一般由硬件平台和软件平台两部分组成, 如下图 1.1 所示。其中硬件平台由嵌入式微处理器和外围硬件设备组成, 而软件平台由嵌入式操作系统和应用软件组成。

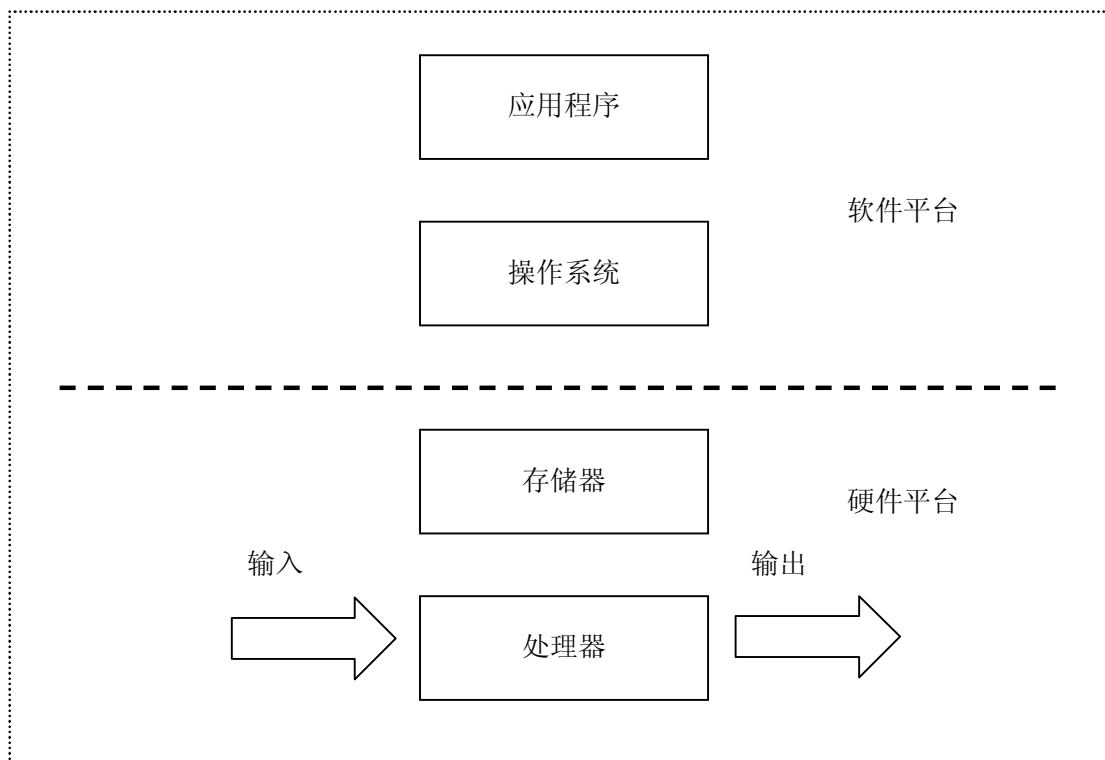


图 1.1 嵌入式系统的一般架构

随着芯片技术的不断发展，嵌入式处理器的主频也越来越高，通常主频都在 40M Hz 以上，有的甚至高达 500MHz。多处理器、多核处理器平台也逐渐应用在嵌入式领域，不过现在大量使用的还是 32 位单处理器组成的平台。一个典型的硬件平台如图 1.2 所示[2]。

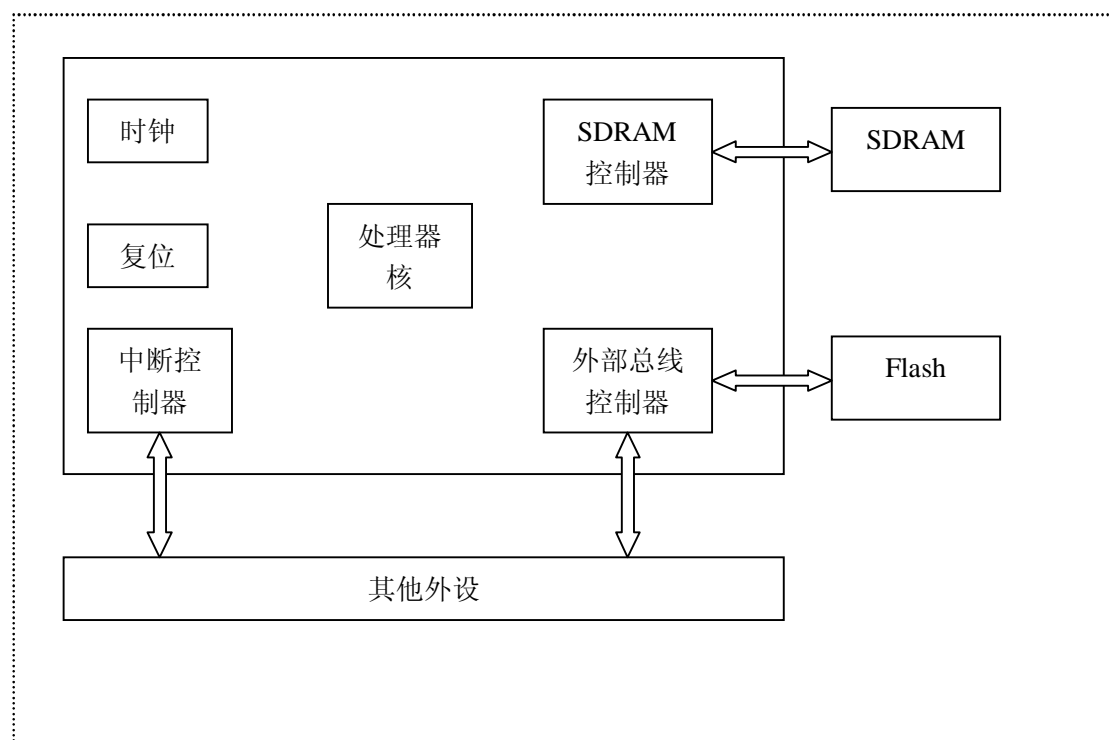


图 1.2 嵌入式硬件平台基本组成结构

嵌入式软件平台主要由嵌入式操作系统与应用软件组成。目前流行的嵌入式操作系统可以分为两类：一类是从运行在个人电脑上的操作系统向下移植到嵌入式系统中，形成的嵌入式操作系统，如微软公司的 Windows CE，SUN 公司的 Java 系统，朗讯科技公司的 Inferno，嵌入式 Linux 等。这类系统经过个人电脑或高性能计算机等产品的长期运行考验，技术日趋成熟，其相关的标准和软件开发方式已被用户普遍接受，同时积累了丰富的开发工具和应用软件资源。另一类是实时操作系统，如 WindRiver 公司的 VxWorks，ISI 的 pSOS，QNX 系统软件公司的 QNX，ATI 的 Nucleus，中国科学院凯思集团的 Hopen 嵌入式操作系统等，这类产品在操作系统的结构和实现上都针对所面向的应用领域，对实时性高可靠性等进行了精巧的设计，而且提供了独立而完备的系统开发和测试工具，较多地应用在军用产品和工业控制等领域中。目前常见的嵌入式系统有：Linux、uClinux、WinCE、PalmOS、Symbian、eCos、uCOS-II、VxWorks、pSOS、Nucleus、ThreadX、Rtems、QNX、INTEGRITY、OSE、C Executive 等。嵌入式操作系统的发展也必将带动新一轮科技竞争。

应用程序运行在嵌入式操作系统之上，一般情况下应用程序和操作系统是分开的。当处理器上带有 MMU(Memory Management Unit，存储器管理单元)，它可以从硬件上将应用程序和操作系统分开编译和管理，Linux、WinCE 就是这种分离机制。这样做的好处就是系统安全性更高，可维护性更强，更有利于各功能模块的划分。很多情况下在没有 MMU 的处理器，如 ARM7TDMI，经常应用程序和操作系统是编译在一起运行的，对于开发人员来说，操作系统更像一个函数库。

## 1.2 ARM 介绍

ARM 是 Advanced RISC Machines（高级精简指令系统处理器）的缩写，它既是一种微



处理器知识产权（IP）核，也是一个公司的名称。在上节中对 ARM 公司有了大概的介绍，这里就不再赘述，以下将介绍 ARM 处理器。

## 1.2.1 ARM 处理器介绍

ARM 处理器已经成功广泛地应用于无线通信、工业控制、消费类电子、网络产品等领域，并且保持持续增长的势头。目前，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75% 以上的市场份额。采用 RISC 架构的 ARM 微处理器一般具有如下特点：

1. 体积小、低功耗、低成本、高性能；
2. 支持 Thumb（16 位）/ARM（32 位）双指令集，能很好的兼容 8 位/16 位器件；
3. 大量使用寄存器，指令执行速度更快；
4. 大多数数据操作都在寄存器中完成；
5. 寻址方式灵活简单，执行效率高；
6. 指令长度固定。

ARM 微处理器目前包括下面几个系列，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

- 2 ARM7 系列：一般包括 ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ 几种内核。ARM7TDMI 是目前使用最广泛的 32 位嵌入式 RISC 处理器之一，主要应用工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。TDMI 的基本含义为：
  - T：支持 16 为压缩指令集 Thumb；
  - D：支持片上 Debug；
  - M：内嵌硬件乘法器（Multiplier）
  - I：嵌入式 ICE，支持片上断点和调试点
- 2 ARM9 系列：包含 ARM920T、ARM922T 和 ARM940T 三种类型，主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数字照相机和数字摄像机等。其中本书中介绍的 S3C2410 就是 ARM9 系列的 ARM920T 类型。ARM9 具有以下特点：
  - l 5 级流水线，指令执行效率更高。
  - l 提供 1.1MIPS/MHz 的哈佛结构。
  - l 支持 32 位元 ARM 指令集和 16 位元 Thumb 指令集。
  - l 支持 32 位元的高速 AMBA 汇流排界面。
  - l 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
  - l 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。
- 2 ARM9E 系列：包含 ARM926EJ-S、ARM946E-S 和 ARM966E-S 三种类型。主要应用于下一代无线设备、数字消费品、成像设备、工业控制、存储设备和网络设备等领域。
- 2 ARM10E 系列：包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 三种类型。主要应用于下一代无线设备、数字消费品、成像设备、工业控制、通信和信息系统等领域。
- 2 SecurCore 系列：包含 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC210 四种类型，主要应用于一些对安全性要求较高的应用产品及应用系统，如电子商务、电子政务、电子银行业务、网络和认证系统等领域。

- 2 Intel 的 Xscale: Xscale 处理器是基于 ARMv5TE 架构的解决方案, 是一款全性能、高成本效益比、低功耗的处理器。它支持 16 位的 Thumb 指令和 DSP 指令集, 已使用在许多移动电话、个人数字助理和网络产品等场合。
- 2 Intel 的 StrongARM: StrongARM SA-1100 处理器是采用 ARM 架构高度整合的 32 位元 RISC 微处理器。它融合了 Intel 公司的设计和处理技术以及 ARM 架构的电源效率, 采用在软件上相容 ARMv4 架构、同时采用具有 Intel 技术优点的架构。Intel StrongARM 处理器是便携型通讯产品和消费类电子产品的理想选择, 已成功应用于多家公司的掌上电脑系列[3]。

其中, ARM7、ARM9、ARM9E 和 ARM10 为 4 个通用处理器系列, 每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。Intel 的 Xscale 和 StrongARM 也是应用非常广泛的嵌入式处理器系列。

### 1.2.2 ARM 处理器的选型

基于 ARM 为内核的处理器已经越来越多, 并且种类繁多, 在选择开发基于 ARM 的嵌入式系统时, 首要任务就是选择 ARM 微处理器。下面讲述在选择 ARM 微处理器的一般准则[3]。

1. ARM 微处理器内核的选择
  - Ø 如果使用 Windows CE 或标准 Linux 等操作系统, 就需要选择 ARM720T 以上带有 MMU 功能的 ARM 晶片。
  - Ø ARM720T、ARM920T、ARM922T、ARM946T、Strong-ARM 都带有 MMU 功能。
  - Ø 而 ARM7TDMI 则没有 MMU, 不支持 Windows CE 和标准 Linux, 但目前有 uCLinux 等不需要 MMU 支持的操作系统可执行 ARM7TDMI 硬件平台。
2. 系统的工作频率
  - Ø 系统的工作频率在很大程度上决定了 ARM 微处理器的处理能力。
  - Ø ARM7 系列微处理器的典型处理速度为 0.9MIPS/MHz, 常见的 ARM7 晶片系统主时钟为 20MHz-133MHz。
  - Ø ARM9 系列微处理器的典型处理速度为 1.1MIPS/MHz, 常见的 ARM9 的系统主时钟为 100MHz-233MHz, ARM10 最高可以达到 700MHz。
3. 晶片内部存储体的容量
  - Ø 大多数的 ARM 微处理器晶片内部存储体的容量都不太大。
  - Ø 如 ATMEL 的 AT91F40162 就具有最高 2MB 的晶片内部存储空间。
4. 晶片内部周围电路选择
  - Ø 如 USB 接口、IIS 接口、IIC 接口、LCD 控制器、键盘接口、RTC、ADC 和 DAC、DSP 等, 设计者应该分析系统的需求, 尽可能采用晶片内部周围电路完成所需的功能, 这样既可以简化系统的设计, 同时提高系统的可靠性。

除了上面介绍的四个方面准则之外, 还有许多其它的因素考虑, 比如价格、兼容性等等。总之, 根据设计的需求选择一款适合自己系统的 ARM 处理器是非常重要的。

### 1.2.3 S3C2410 微处理器介绍

S3C2410 是三星电子开发的一种 32 位 RISC 微处理器, 它是基于 ARM920T 内核开发

的。S3C2410 是面向低价格、低功耗和高性能的手持设备和小型设备而设计。S3C2410 的具体特点有以下[4]:

#### I 系统管理

- Ø 支持小端/大端方式
- Ø 地址空间: 128M 字节每一个 Bank (总共 1G 字节)
- Ø 每个 BANK 可编程为 8/16/32 位数据总线
- Ø BANK0 到 BANK6 采用固定起始地址和大小
- Ø BANK7 具有可编程的 BANK 起始地址和大小
- Ø 共 8 个存储器 BANK
- Ø 前 6 个存储器 BANK 用于 ROM、SRAM 和其他
- Ø 另外两个存储器 BANK 用于 ROM、SRAM 和同步 DRAM
- Ø 支持等待信号用以延长总线周期
- Ø 支持掉电时的 SDRAM 自刷新模式
- Ø 支持不同类型的 ROM 引导 (NOR/NAND Flash、EEPROM 和其他)。

#### I S3C2410 的 SoC 芯片集成单元

- Ø 内部 1.8V, 存储器 3.3V, 外部 I/O 3.3V, 16KB 数据 CACHE, 16KB 指令 CACHE, MMU
- Ø 内置外部存储器控制器 (SDRAM 控制和芯片选择逻辑)
- Ø LCD 控制器, 一个 LCD 专用 DMA
- Ø 4 个带外部请求线的 DMA
- Ø 3 个通用异步串行端口 (IrDA1.0, 16-Byte Tx FIFO, and 16-Byte Rx FIFO), 2 通道 SPI
- Ø 一个多主 I<sup>2</sup>C 总线, 一个 I<sup>2</sup>S 总线控制器
- Ø SD 主接口版本 1.0 和多媒体卡协议版本 2.11 兼容
- Ø 两个 USB HOST, 一个 USB DEVICE (VER1.1)
- Ø 4 个 PWM 定时器和一个内部定时器
- Ø 看门狗定时器
- Ø 117 个通用 I/O
- Ø 24 个外部中断
- Ø 电源控制模式: 标准、慢速、休眠、掉电
- Ø 8 通道 10 位 ADC 和触摸屏接口
- Ø 带日历功能的实时时钟
- Ø 芯片内置 PLL
- Ø 设计用于手持设备和通用嵌入式系统
- Ø 16/32 位 RISC 体系结构, 使用 ARM920T CPU 核的强大指令集
- Ø 带 MMU 的先进的体系结构支持 WinCE、EPOC32、Linux
- Ø 指令缓存 (CACHE)、数据缓存、写缓冲和物理地址 TAG RAM, 减小了对主存储器带宽和性能的影响
- Ø ARM920T CPU 核支持 ARM 调试的体系结构
- Ø 内部先进的位控制器总线 (AMBA) (AMBA2.0, AHB/APB)

其中, S3C2410 的芯片结构图 1.3 所示:

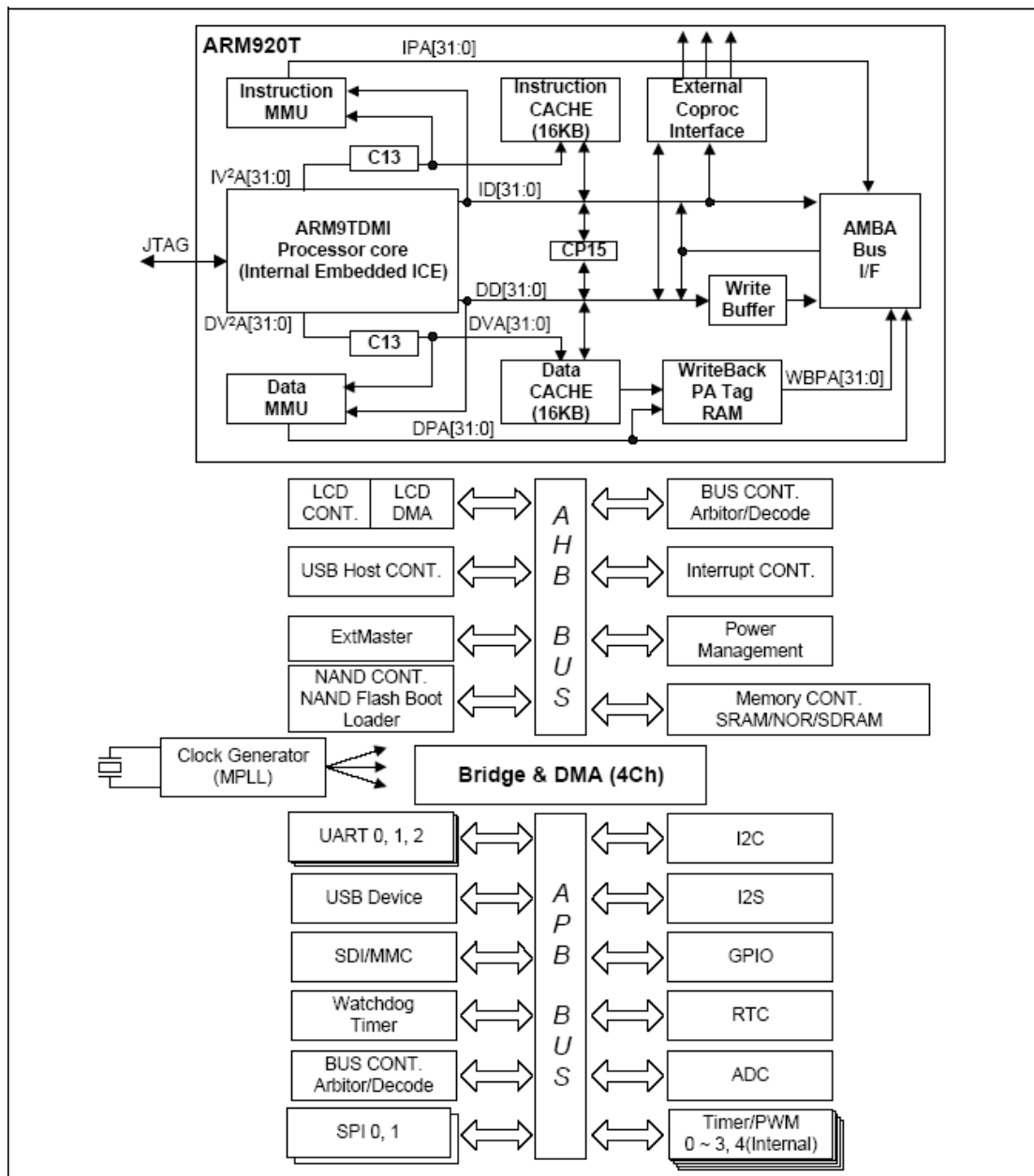


图 1.3 S3C2410 芯片结构

## 1.3 ADS 集成开发环境介绍

ADS 全称为 ARM Developer Suite，是 ARM 公司推出的新一代 ARM 集成开发工具。现在 ADS 的最新版本是 1.2，它取代了早期的 ADS1.1 和 ADS1.0。在 ADS 工具诞生之前，一直使用的是 ARM SDT 工具，目前 ARM SDT 工具已经慢慢被淘汰。ADS 除了可以安装在 Windows NT4，Windows 2000，Windows 98 和 Windows 95 操作系统下，还支持 Windows XP 和 Windows Me 操作系统。

### 1.3.1 ADS 软件组成

ADS 由命令行开发工具，GUI（Graphics User Interface，图形用户界面）开发环境(Code Warrior 和 AXD)，实用程序和支持软件组成。有了这些部件，用户就可以为 ARM 系列的 RISC 处理器编写和调试自己的开发应用程序了。下面将分别介绍这四个组成部分。

#### 1.3.1.1 命令行开发工具

命令行开发工具在实际应用中相对比较广泛，用其最大的好处就是可以将许多编译命令写在一个脚本文件中，然后只执行该脚本文件就可以让工具自动完成所有编译的工作。命令行中常用的命令如下：

##### Ø armcc

armcc 是 ARM C 编译器。这个编译器通过了 Plum Hall C Validation Suite 为 ANSI C 的一致性测试。armcc 用于将用 ANSI C 编写的程序编译成 32 位 ARM 指令代码。

在命令控制台环境下，输入以下命令：

```
> armcc -help
```

将可以查看 armcc 的语法格式以及最常用的一些操作选项

armcc 最基本的用法为：

```
> armcc [options] file1 file2 ... fileN
```

这里的 option 是编译器所需要的选项，file1,file2...fileN 是相关的文件名。

以下简单介绍一些最常用的操作选项：

-c: 表示只进行编译不链接文件；

-C: (注意：这是大写的 C)禁止预编译器将注释行移走；

-D<symbol>: 定义预处理宏，相当于在源程序开头使用了宏定义语句

-E: 仅仅是对 C 源代码进行预处理就停止；

-g<options>: 指定是否在生成的目标文件中包含调试信息表；

-I<directory>: 将 directory 所指的路径添加到#include 的搜索路径列表中去；

-J<directory>: 用 directory 所指的路径代替默认的对#include 的搜索路径；

-o<file>: 指定编译器最终生成的输出文件名。

-O0: 不优化；

-O1: 这是控制代码优化的编译选项，大写字母 O 后面跟的数字不同，表示的优化级别就不同，-O1 关闭了影响调试结果的优化功能；

-O2: 该优化级别提供了最大的优化功能；

-S: 对源程序进行预处理和编译，自动生成汇编文件而不是目标文件；

-U<symbol>: 取消预处理宏名，相当于在源文件开头，使用语句#undef symbol;

-W<options>: 关闭所有的或被选择的警告信息；

有关更详细的选项说明，读者可查看 ADS 软件的在线帮助文件。

##### Ø armcpp

armcpp 是 ARM C++编译器。它将 ISO C++ 或 EC++ 编译成 32 位 ARM 指令代码。该编译器的命令选项和 armcc 的选项基本一样，这里就不再重复。

##### Ø tcc

tcc 是 Thumb C 编译器。该编译器通过了 Plum Hall C Validation Suite 为 ANSI 一致性的测试。tcc 将 ANSI C 源代码编译成 16 位的 Thumb 指令代码。同时它的编译选项和用法

类似 armcc，具体使用请参考 ADS 软件的在线帮助文件。

#### Ø tcpp

tcpp 是 Thumb C++ 编译器。它将 ISO C++ 和 EC++ 源码编译成 16 位 Thumb 指令代码。同时它的编译选项和用法类似 armcc，具体使用请参考 ADS 软件的在线帮助文件。

#### Ø armasm

armasm 是 ARM 和 Thumb 的汇编器。它对用 ARM 汇编语言和 Thumb 汇编语言写的源代码进行汇编。在命令行输入：armasm -help 将会看到 armasm 汇编器的用法以及它的编译选项。

```
> armasm [options] sourcefile objectfile
> armasm [options] -o objectfile sourcefile
```

上述是关于 armasm 两种基本用法，其中 options 为它的选项，常用的选项如下：

- LIST: 写一个列表文件在指定的文件
- Depend: 保存编译后的依赖源文件
- Errors: 将标准出错的诊断信息放到指定的文件中
- I: 添加目录到源文件的搜索路径
- PreDefine: 预执行一个 SET{L,A,S}指令
- NOCache: 源缓冲关（默认是开）
- MaxCache: 定义最大缓冲的大小（默认是 8M）
- NOWarn: 关闭打印告警信息
- G: 输出调试表
- APCS: 使预定义匹配已选择 proc-call 标准
- Help: 打印帮助信息
- Littleend: Little-endian ARM
- Bigend: Big-endian ARM
- MEMACCESS: 说明目标内存系统的属性
- M: 写源文件依赖性列表到标准输出
- MD: 写源文件依赖性列表到标准输入
- CPU: 设置目标 ARM 内核类型
- FPU: 设置目标 FP 体系版本，SOFTVFP, SOFTFPA, VFP, FPA, NONE 之一
- 16: 汇编 16 位 Thumb 指令
- 32: 汇编 32 位 ARM 指令

#### Ø armlink

armlink 是 ARM 链接器。该命令既可以将编译得到的一个或多个目标文件和相关的一个或多个库文件进行链接，生成一个可执行文件，也可以将多个目标文件部分链接成一个目标文件，以供进一步的链接。ARM 链接器生成的是 ELF 格式的可执行映像文件。armlink 的一般用法如下：

```
> armlink option-list input-file-list
```

其中，option-list: 是一个区分大小写的选项表；input-file-list: 是一系列库和对象文件。关于 armlink 的具体使用请参考 ADS 软件的在线帮助文件。

#### Ø armsd

armsd 是 ARM 和 Thumb 的符号调试器。它能够进行源码级的程序调试。用户可以在用 C 或汇编语言写的代码中进行单步调试，设置断点，查看变量值和内存单元的内容。armsd 的一般用法如下：

```
> armsd [options] [<imagefile> [<arguments>]]
```

其中, options: 是一系列调试选项; imagefile: 定义一个 AIF 或 ELF 文件的名称; arguments: 是被 imagefile 接受的命令行参数。关于 armsd 的具体使用请参考 ADS 软件的在线帮助文件。

讲到这里我们可以举一个简单的应用实例, 来看看关于常用的 ARM 命令行是如何使用的。以附件光盘中的 SWI (Software Interrupter) 参考项目为例, 它的编译命令如下:

```
armasm -g a_swi.s
armcc -c -g -O1 main.c
armcc -c -g -O1 c_swi.c
armlink a_swi.o main.o c_swi.o -o swi.axf
```

其中, armasm 命令用来编译 ARM 汇编代码, armcc 用来编译 C 代码, armlink 用来最终链接目标文件为 ELF 格式的可执行映像文件。

### 1.3.1.2 GUI 开发环境

ADS GUI 开发环境包含 Code Warrior 和 AXD 两种, 其中 Code Warrior 是集成开发工具, 而 AXD 是调试工具。下面将分别介绍这两个工具。

CodeWarrior for ARM 是一套完整的集成开发工具, 充分发挥了 ARM RISC 的优势, 使产品开发人员能够很好的应用尖端的片上系统技术。该工具是专为基于 ARM RISC 的处理器而设计的, 它可加速并简化嵌入式开发过程中的每一个环节, 使得开发人员只需通过一个集成软件开发环境就能研制出 ARM 产品, 在整个开发周期中, 开发人员无需离开 CodeWarrior 开发环境, 因此节省了在操做工具上花的时间, 使得开发人员有更多的精力投入到代码编写上来, CodeWarrior 集成开发环境(IDE)为管理和开发项目提供了简单多样化的图形用户界面。用户可以使用 ADS 的 CodeWarrior IDE 为 ARM 和 Thumb 处理器开发用 C, C++, 或 ARM 汇编语言的程序代码。CodeWarrior IDE 缩短了用户开发项目代码的周期, 主要是由于: 一是全面的项目管理功能, 二是子函数的代码导航功能, 使得用户迅速找到程序中的子函数。关于 CodeWarrior 的具体使用将在下一节中具体介绍。

AXD(ARM eXtended Debugger), 即 ARM 扩展调试器。调试器本身是一个软件, 用户通过这个软件使用调试代理可以对包含有调试信息的, 正在运行的可执行代码进行比如变量的查看, 断点的控制等调试操作。调试代理既不是被调试的程序, 也不是调试器。在 ARM 体系中, 它有这几种方式: Multi-ICE(Multi-processor in-circuit emulator), ARMulator 和 Angel。其中 Multi-ICE 是一个独立的产品, 是 ARM 公司自己的 JTAG 在线仿真器, 不是由 ADS 提供的。AXD 可以在 Windows 和 UNIX 下, 进行程序的调试。它为用 C, C++, 和汇编语言编写的源代码提供了一个全面的 Windows 和 UNIX 环境。后面的章节会具体介绍 AXD 工具的使用方法。

### 1.3.1.3 实用程序

ADS 除了提供上述工具外, 它还提供以下的实用工具来配合前面介绍的命令行开发工具的使用。

#### Ø Flash downloader

用于把二进制映像文件下载到 ARM 开发板上的 Flash 存储器的工具

#### Ø fromELF

这是 ARM 映像文件转换工具。该命令将 ELF 格式的文件作为输入文件, 将该格式转换为各种输出格式的文件, 包括 plain binary(BIN 格式映像文件), Motorola 32-bit S-record

format(Motorola 32 位 S 格式映像文件), Intel Hex 32 format(Intel 32 位格式映像文件), 和 Verilog-like hex format(Verilog 16 进制文件)。fromELF 命令也能够为输入映像文件产生文本信息, 例如代码和数据长度。

#### Ø armar

ARM 库函数生成器将一系列 ELF 格式的目标文件以库函数的形式集合在一起, 用户可以把一个库传递给一个链接器以代替几个 ELF 文件。

### 1.3.1.4 支持的软件

ADS 为用户提供 ARMulator 软件, 使用户可以在软件仿真的环境下或者在基于 ARM 的硬件环境调试用户应用程序。ARMulator 是一个 ARM 指令集仿真器, 集成在 ARM 的调试器 AXD 中, 它提供对 ARM 处理器的指令集的仿真, 为 ARM 和 Thumb 提供精确的模拟。用户可以在硬件尚未做好的情况下, 开发程序代码。

关于 ADS 软件主要由上述四个部分组成, 下面将介绍在实际工作中经常用到的 Code Warrior 和 AXD 工具的基本使用。

## 1.3.2 使用 Code Warrior IDE

Code Warrior IDE 提供一个简单通用的图形化用户界面用于管理软件开发项目。可以利用 Code Warrior IDE 开发 C, C++ 和 ARM 汇编代码以 ARM 和 Thumb 处理器为对象。下面将通过一个实例来讲述 Code Warrior IDE 的具体使用, 为了使读者容易理解, 这里还是以附件光盘中提供的 SWI 项目为例, 讲述 Code Warrior IDE 工具的使用。

### 1.3.2.1 创建项目工程

建立项目工程是嵌入式实际开发中必不可少的一部分, 因为工程将所有的源码文件组织在一起, 并能够决定最终生成文件存放的路径, 输出的格式等。在 CodeWarrior 中新建一个工程的方法有两种, 可以在工具栏中单击“New”按钮, 也可以在“File”菜单中选择“New...”菜单。这样就会打开一个如图 1.4 所示的对话框。



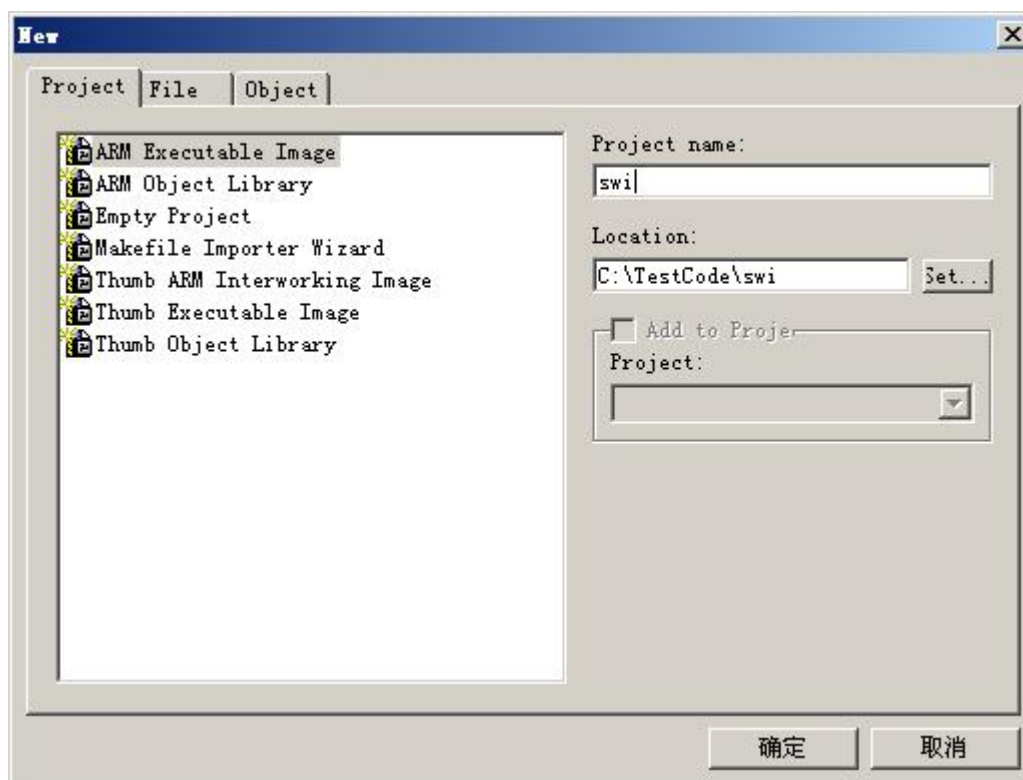


图 1.4 新建工程对话框

在 Project 可选框中为用户提供了 7 种可选择的工程类型，分别是：

- Ø ARM Executable Image: 用于由 ARM 指令的代码生成一个 ELF 格式的可执行映像文件；
- Ø ARM Object Library: 用于由 ARM 指令的代码生成一个 armar 格式的目标文件库；
- Ø Empty Project: 用于创建一个不包含任何库或源文件的工程；
- Ø Makefile Importer Wizard: 用于将 Visual C 的 nmake 或 GNU make 文件转入到 CodeWarrior IDE 工程文件；
- Ø Thumb ARM Interworking Image: 用于由 ARM 指令和 Thumb 指令的混和代码生成一个可执行的 ELF 格式的映像文件；
- Ø Thumb Executable image: 用于由 Thumb 指令创建一个可执行的 ELF 格式的映像文件；
- Ø Thumb Object Library: 用于由 Thumb 指令的代码生成一个 armar 格式的目标文件库。

在这里选择 ARM Executable Image，然后在“Project name:”里输入名为 swi 的工程文件名。接着在“Location:”项中点击“Set...”按钮选择项目工程存放的位置，这里存放的位置为 C:\TestCode。最后点击“确定”，即可建立一个新的名为 swi 的工程。

这个时候会出现 swi.mcp 的窗口，如图 1.5 所示，有三个标签页，分别为 files, link order, target 默认的是显示第一个标签页 files。通过在该标签页点击鼠标右键，选中“Add Files...”可以把要用到的源程序添加到工程中。

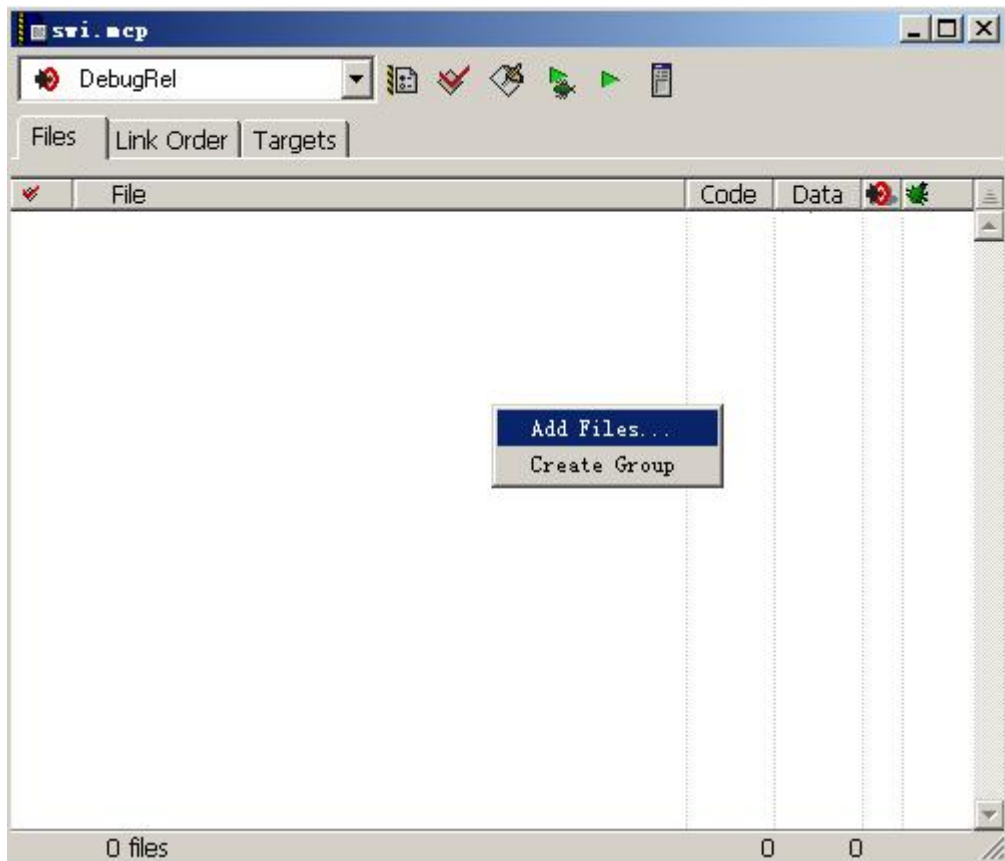


图 1.5 添加源文件到工程中

为工程添加源码常用的方法有两种，既可以使用入图 1.5 所示方法，也可以在“Project”菜单项中，选择“Add Files...”，这两种方法都会打开文件浏览框，用户可以把已经存在的文件添加到工程中来。当选中要添加的文件时，会出现一个对话框，如图 1.6 所示，询问用户把文件添加到何类目标中，在这里，我们选择 DebugRel 目标。这里我们添加了 swi.h, a\_swi.s 和 main.c 文件。

在建立好一个工程时，默认的 target 是 DebugRel，还有另外两个可用的 target，分别为 Realse 和 Debug，这三个 target 的含义分别为：

**DebugRel:** 使用该目标，在生成目标的时候，会为每一个源文件生成调试信息；

**Debug:** 使用该目标为每一个源文件生成最完整的调试信息；

**Release:** 使用该目标不会生成任何调试信息。

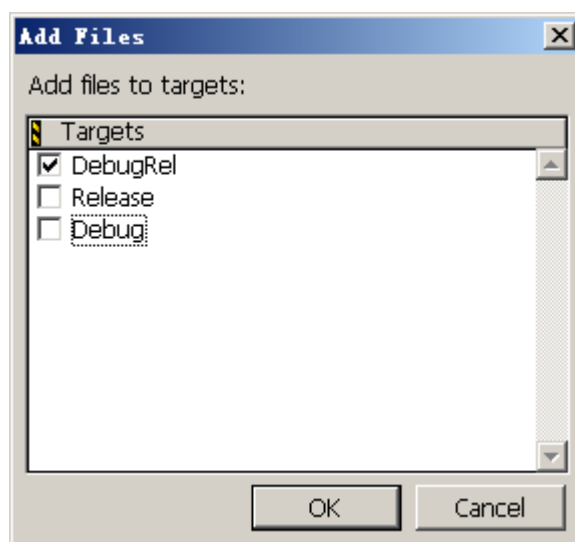


图 1.6 选择生成目标类型

到目前为止，一个完整的名为 swi 的项目工程已经建立，下面该对工程进行编译和链接工作。

### 1.3.2.2 编译和链接项目工程

在编译 swi 项目之前，先进行设置，点击 Edit 菜单，选择 “DebugRel Settings...”，或者按 Alt + F7 快捷键，显示如图 1.7 对话框。

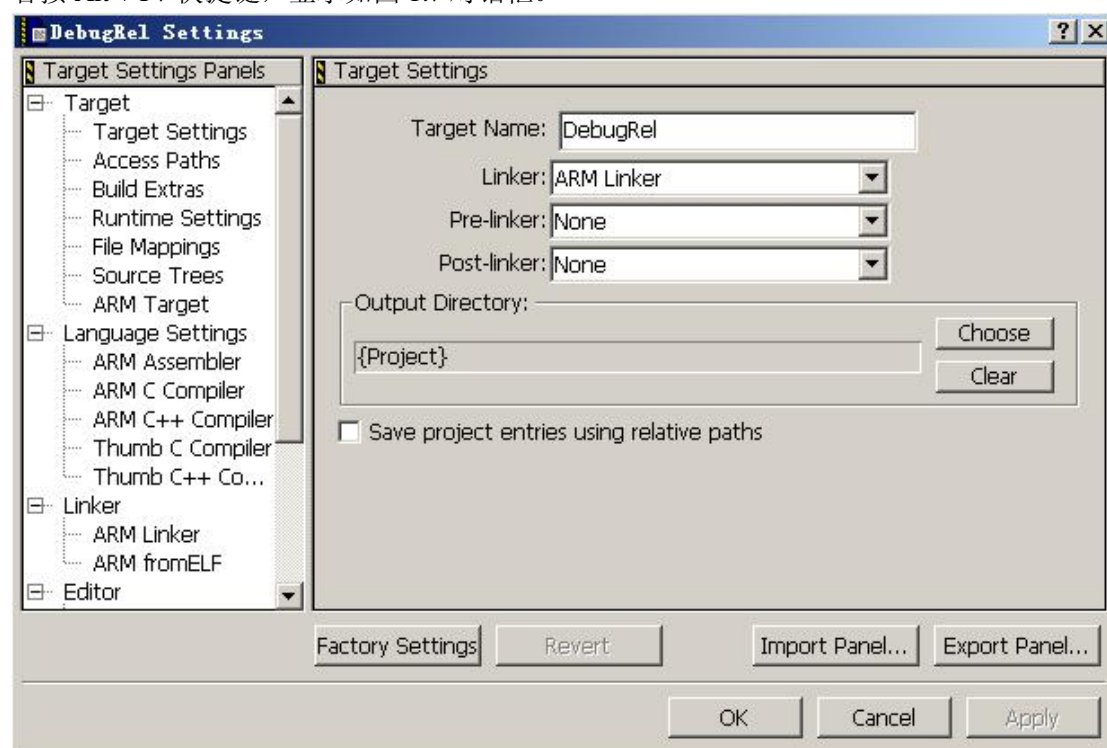


图 1.7 DebugRel 设置对话框

图 1.7 的最左边部分是目标设置面板，它包括如下几个大的设置对象：

#### Ø Target 设置选项

Target Settings: 包括 Target Name, Linker, Pre-linker 和 Post-linker 等设置。

Access Paths: 主要是用于项目的路径设置。  
 Build Extras: 主要用于 Build 附加的选项设置。  
 Runtime Settings: 包括一般设置、环境设置等。  
 File Mappings: 包含映射信息, 文件类型, 编辑语言等。  
 Source Trees: 包含源代码树结构信息, 以及路径选择等。  
 ARM Target: 定义输出 image 文件名和类型等。

Ø Language Settings 设置选项

ARM Assembler: 对 ARM 汇编语言的支持选项设置。  
 ARM C Compiler: 对 C 语言的支持选项设置。  
 ARM C++ Compiler: 对 C++语言的支持选项设置。  
 Thumb C Compiler: 对 Thumb C 语言的支持选项设置。  
 Thumb C++ Compiler: 对 Thumb C++语言的支持选项设置。

Ø Linker 设置选项

ARM Linker: 对输出的链接类型、RO、RW Base 地址设置等选项。  
 ARM fromELF: 定义输出文件格式以及路径等。

Ø Edit 设置选项

Custom Keywords: 对客户化关键字高亮颜色的设置。

Ø Debugger 设置选项

Other Executables: 制定其他的可执行文件来调试当调试该目标板时。  
 Debugger Settings: 对调试器的一些基本设置。  
 ARM Debugger: 选择调试时调试器 (AXD, Armsd 和其他等) 的选择。  
 ARM Runner: 选择运行时的调试器 (AXD, Armsd 和其他等) 的选择。

Ø Miscellaneous 设置选项

ARM Features: 设置一些受限制的特性。

接下来点击 CodeWarrior IDE 的菜单 Project 下的 make 菜单, 就可以对 swi 工程进行编译和链接了。整个编译链接之后生成如图 1.8 所示:

Code	R0 Data	RW Data	ZI Data	Debug
772	60	4	0	4744
9408	312	0	300	4320
10180	372	4	300	9064
<b>Object Totals</b>				
<b>Library Totals</b>				
<b>Grand Totals</b>				
Total R0 Size(Code + R0 Data)				10552 ( 10.30kB)
Total RW Size(RW Data + ZI Data)				304 ( 0.30kB)
Total ROM Size(Code + R0 Data + RW Data)				10556 ( 10.31kB)

图 1.8 编译和链接的之后

在工程 swi 所在的目录下，会生成一个名为：工程名\_data 的目录，即 swi\_data 的目录，在这个目录下不同类别的目标对应不同的目录。在本例中由于我们使用的是 DebugRe 目标，所以生成的最终文件都应该在该目录下。进入到 DebugRel 目录中去，读者会看到 make 后生成的映像文件和二进制文件，映像文件用于调试，二进制文件可以烧写到目标板的 Flash 中运行。关于 Code Warrior IDE 的具体使用请参考 ADS 软件的在线帮助文件。

### 1.3.3 使用 AXD IDE

AXD 是 ADS 软件中独立于 CodeWarrior IDE 的图形软件，打开 AXD 软件，默认是打开的目标是 ARMulator。ARMulator 也是调试的时候最常用的一种调试工具，本节主要是结合 ARMulator 介绍在 AXD 中进行代码调试的方法和过程，使读者对 AXD 的调试有初步的了解。要使用 AXD 必须首先要生成包含有调试信息的程序，在上节中，已经生成的 swi.axf 文件就是含有调试信息的可执行 ELF 格式的映像文件。这一节还是以 swi 工程为例讲述 AXD 调试工具的基本用法。

#### 1.3.3.1 打开调试文件

在菜单 File 中选择“Load image...”选项，打开 Load Image 对话框，找到要装载的.axf 映像文件，点击“打开”按钮，就把映像文件装载到目标内存中了。在所打开的映像文件中会有一个蓝色的箭头指示当前执行的位置。如图 1.9 所示：

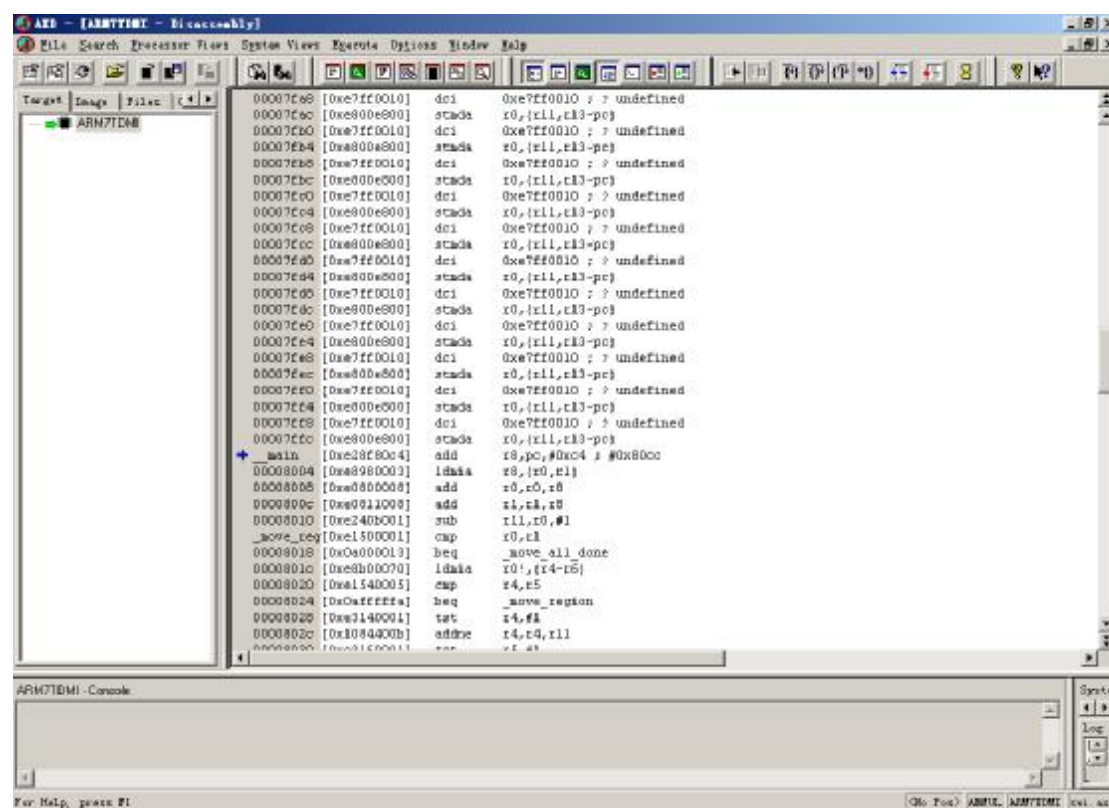


图 1.9 打开 swi 调试文件

此外，在菜单 File 中还有一个“Load Debug Symbols....”选项，该选项是用来调式那些调试器不能访问调试符号的情况，比如调试装载在 ROM 中的 image。通常“Load image...”



选项用来调试装载在 RAM 中的代码。

在菜单 Execute 中选择“Go”，将运行代码。要想进行单步的代码调试，在 Execute 菜单中选择“Step”选项，或用 F10 即可以单步执行代码，窗口中蓝色箭头会发生相应的移动。

### 1.3.3.2 设置断点

有时候，用户可能希望程序在执行到某处时，查看一些所关心的变量值，此时可以通过设置断点达到此要求。将光标移动到要进行断点设置的代码处，在 Execute 菜单中，选择“Toggle Breakpoint”或按 F9，就会在光标所在行的起始位置出现一个红色实心圆点，表明该处为已设为断点。假设本例中给 62 行代码设置断点，首先将光标移至 62 行，然后按 F9 或点击“Toggle Breakpoint”按钮，此时如图 1.10 所示：

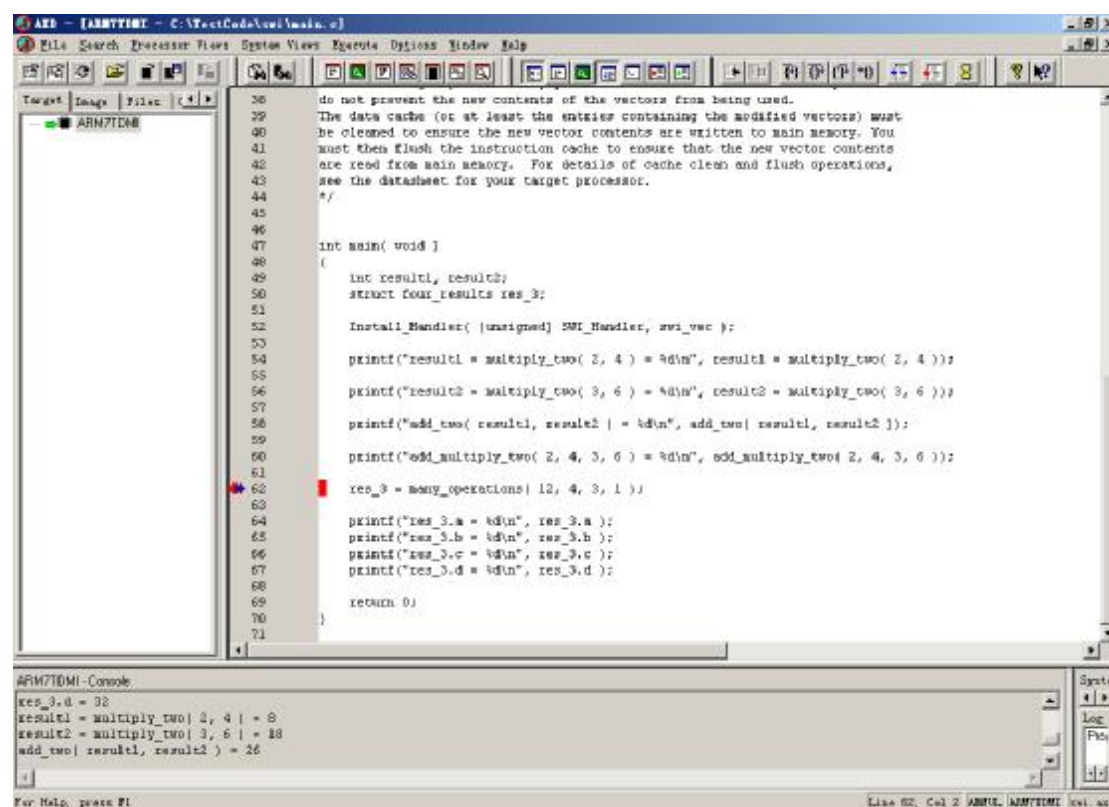


图 1.10 设置断点

### 1.3.3.3 查看寄存器内容

查看寄存器的值在实际嵌入式开发调试中经常使用，使用方法：从 Processor Views 菜单中选择“Memory”选项，如图 1.11 所示。在 Memory Start address 选择框中，用户可以根据要查看的存储器的地址输入起始地址，在下面的表格中会列出连续的 64 个地址。从图 1.11 中可以看出地址为 0x0 的存储器中的初始值为 0x E7FF0010，注意因为用的是 little-endian\*（注 2），所以读数据的时候注意高地址中存放的是高字节，低地址存放的是低字节。

\*注 2：Big-endian 和 Little-endian 是用来表述一组有序的字节数存放在计算机内存中时的顺序的术语。Big-endian 是将高位字节（序列中最重要的值）先存放在低地址处的顺序，而 Little-endian 是将低位字

节（序列中最不重要的值）先存放在低地址处的顺序。举例来说，在使用 Big-endian 顺序的计算机中，要存储一个十六进制数 5F48 所需要的字节将会以 5F48 的形式存储（比如 5F 存放在内存的 1000 位置，而 48 将会被存储在 1001 位置）。而在使用 Little-endian 顺序的系统中，存储的形式将会是 485F（48 在地址 1000 处，5F 在地址 1001 处）。如果将 0x5F48 写到以 0x0000 开始的地址中，则存放的顺序如下：

地址	Big-endian	Little-endian
0x0000	5F	48
0x0001	48	5F

IBM 的 370 种大型机、大多数基于 RISC 的计算机以及 Motorola 的微处理器使用的是 Big-endian 顺序，TCP/IP 协议也是。而 Intel 的处理器和 DEC 公司的一些程序则使用的 Little-endian 方式。

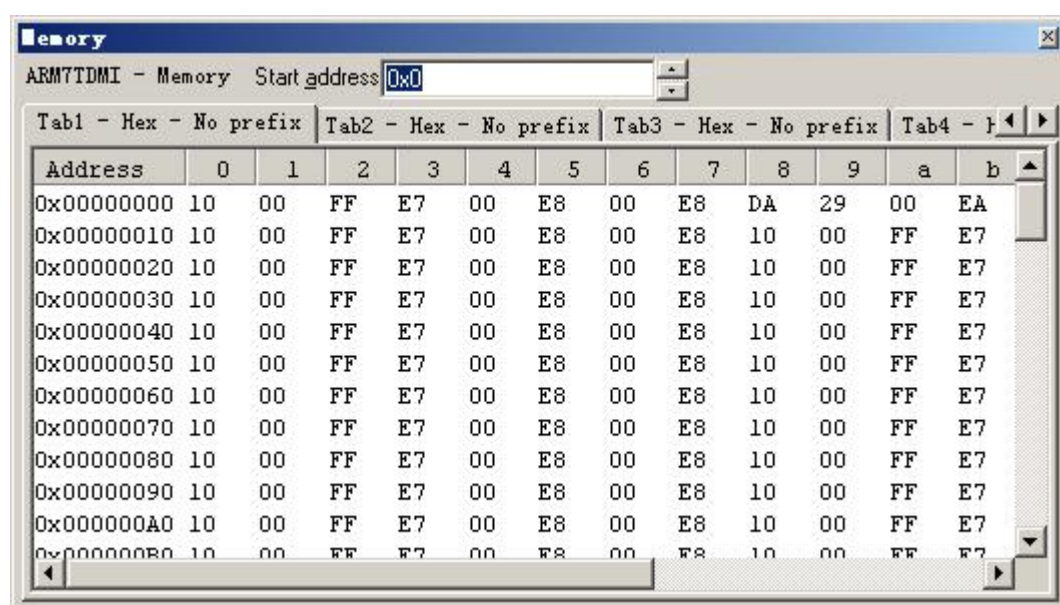


图 1.11 查看寄存器值

### 1.3.3.4 查看变量值

在调试过程中，经常需要查看某个变量的值，在 AXD 工具中，查看变量值的方法是：先用鼠标选中要查看的变量，然后鼠标右击，在弹出的对话框中选择“Watch..”，将会显示指定变量的详细信息。此处以 62 行的 res\_3 为要查看的变量，先选中 res\_3 变量，然后鼠标右击，选择“Watch..”项，将弹出如图 1.12 的对话框，该对话框显示了 res\_3 变量的地址和值等详细信息。

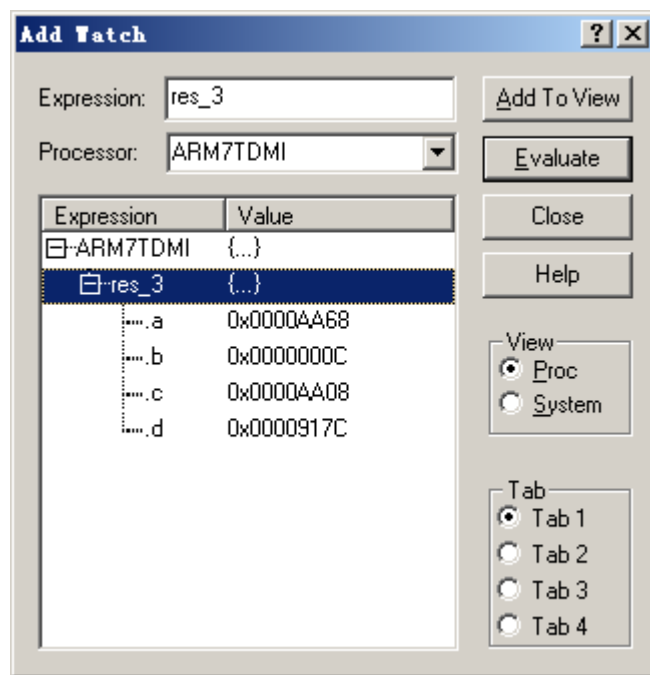


图 1.12 查看变量对话框

总之，AXD 工具的使用方法还有很多，关于 AXD IDE 的具体使用请参考 ADS 软件的在线帮助文件，这里不再赘述。

## 1.4 嵌入式 Linux 开发介绍

在这一节中主要讲述 Linux 开发的基础知识，其中包括 Linux 的发展历史，Linux 的开发环境和 ARM Linux 系统的开发流程，首先让我们来看一下 Linux 的发展历史。

### 1.4.1 Linux 历史

Linux 是 Unix 操作系统的一个克隆，由名叫 Linus Torvalds 的大学生在 1991 年开发诞生的。Linus Torvalds 将他写的操作系统源代码放在了 Internet 上，受到很多计算机爱好者的热烈欢迎，并且这些计算机爱好者不断地添加新的功能和特性，并不断的提高它的稳定性。在 1994 年，Linux 1.0 正式发布。现在，Linux 已经成为一个功能超强的 32 位操作系统。Linux 为嵌入操作系统提供了一个极有吸引力的选择，它是个和 Unix 相似、以核心为基础的、完全内存保护、多任务多进程的操作系统。支持广泛的计算机硬件，包括 X86, Alpha, Sparc, MIPS, PPC, ARM, NEC, MOTOROLA 等现有的大部分芯片。源代码全部公开，任何人都可以修改并在 GNU 通用公共许可证(GNU General Public License)下发行，这样开发人员可以对操作系统进行定制。同时由于有 GPL 的控制，大家开发的东西大都相互兼容，不会走向分裂之路。Linux 用户遇到问题时可以通过 Internet 向网上成千上万的 Linux 开发者请教，这使得最困难的问题也有办法解决。Linux 带有 Unix 用户熟悉的完善的开发工具，几乎所有的 Unix 系统的应用软件都已移植到了 Linux 上。Linux 还提供了强大的网络功能，有多种可选择窗口管理器(X windows)。其强大的语言编译器 gcc、g++等也可以很容易得到。不但成熟完善、而且使用方便。

关于嵌入式 Linux 的发展也如同 Linux 发展一样非常迅速，在 1999 年，Linux 开始根植



于嵌入式系统开发，同年9月在嵌入式系统会议（Embedded System Conference, ESC）上许多公司宣布支持嵌入式Linux，这些公司包括FSM Labs, MontaVista, Zentropix 和 Lineo 等。在2000年，Samsung 公司推出一款名为Yopy的PDA，其应用嵌入式Linux系统。同年Ericsson 公司推出一款名为HS210的基于Linux的无绳带屏电话，它可以通过无线连接上网，打电话，收发E-mail等功能。同年许多公司采用嵌入式Linux在他们的产品线上。在2001年，最重要的宣布就是发布了Linux内核2.4，该版本被后期采用到许多嵌入式Linux分支中。在2002年，可以看到许多上市的基于Linux的产品，并且Linux已经在向数字娱乐领域发展。在2003年，Motorola 宣布A760手机采用Linux作为它的嵌入式操作系统，这一年Linux也在小型办公市场上发展很快。在2004年，LynuxWorks 发布基于Linux2.6内核的BlueCat。它作为第一个基于Linux2.6内核的商业嵌入式Linux。在2005年，基于Linux2.6内核的嵌入式产品已经非常广泛，尤其是基于ARM内核的芯片已经广泛使用Linux为其操作系统。现在许多公司已经采用嵌入式Linux作为他们新的设计方案。目前，AMD，ARM，TI，Motorola，Intel 和 IBM 等知名企业把Linux作为一个首选的操作系统。相信嵌入式Linux的发展会越来越好，用户也会越来越多。

## 1.4.2 Linux 开发环境

习惯在Windows下编成的开发人员经常会感觉在Linux系统下编程很复杂，比如环境变量、编译器的选择以及繁琐的命令等等都会让他们头疼，因为往往是Windows下这些东西都已经做好，并且基本上都是图形界面的设置，不像Linux下大都用命令行形式执行。其实Linux环境下编程并不像许多人想象的那么难，一旦你熟悉了Linux操作系统的基本原理和编译原理，相对来说你会觉得Linux开发更容易一些，因为它能让你清楚地知道程序之间的编译关系，以及内部的逻辑结构，总之会让你清楚地知道你所编译的项目而不是只了解最上层的一些应用。常见的Linux开发环境有以下三种组合方式：

### 1. Windows 操作系统 + Cygwin 工具

Cygwin 于1995年开始开发，是cygnus solutions 公司（已经被Red Hat 公司收购）的产品。Cygwin是一个windows平台下的Linux模拟环境。它包括一个DLL(cygwin1.dll)，这个dll为POSIX系统提供接口调用的模拟层，还有一系列模拟linux平台的工具。Cygwin的dll可以用于windonws95之后的x86系列windows上面。其API竭尽模拟单个Unix和linux的规范。另外Cygwin和linux之间的重要区别是：一是C函数库的不同，前者用newlib而后者用的是glibc。二是shell不同，前者用ash而在大多数linux发行版上用的是bash。Windows + Cygwin 组合的开发方式非常适合初学者使用，笔者学习Linux环境下的开发也是从使用Cygwin开始的，但是这种组合不能开发QT等GUI，因为它没有提供X服务器。Cygwin的下载站点是<http://www.cygwin.com/>。Cygwin在Windows系统上的打开界面如图1.13所示。

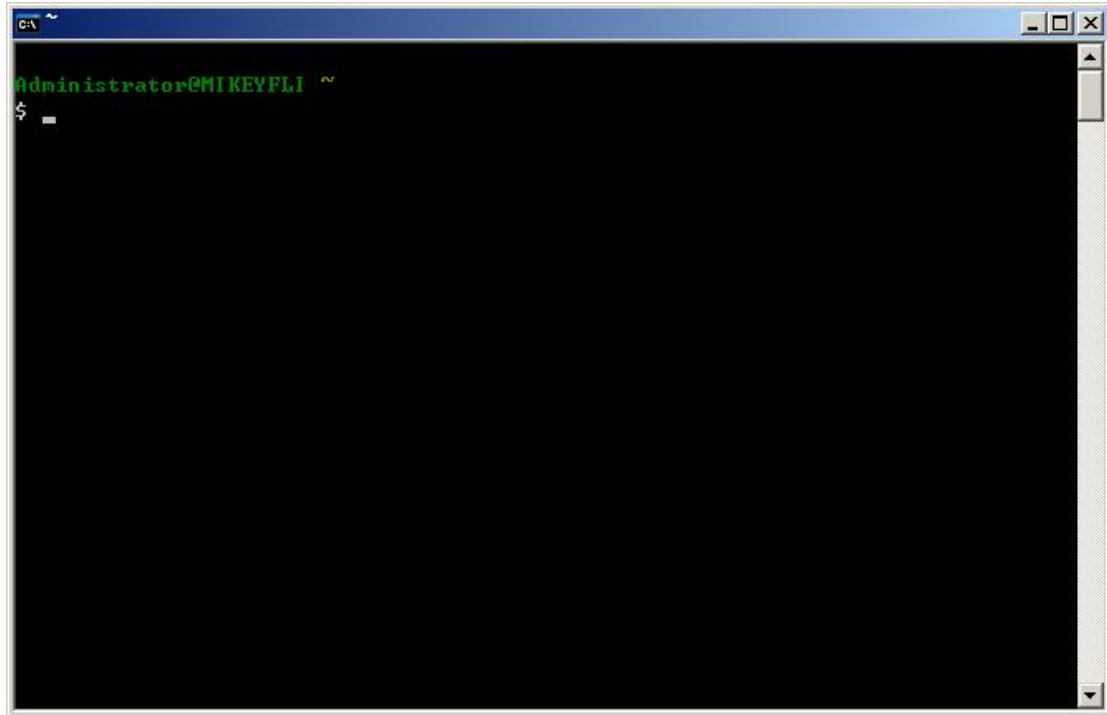


图 1.13 Cygwin 工具

## 2. Windows 操作系统 + VMware 工具 + Linux 操作系统

VMware 是一个“虚拟机”软件。它使你可以在一台机器上同时运行二个或更多的操作系统,比如 WIN2000 / WINNT / WIN9X / DOS / LINUX 系统。与“多启动”系统相比,VMware 采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统,在系统切换时需要重新启动机器。VMware 是真正“同时”运行多个操作系统在主系统的平台上,就象 Word / Excel 那种标准 Windows 应用程序那样切换。Windows + VMware 这种组合对于实际开发应用来说比较广泛,因为在 VMware 工具中可以安装 Linux 系统,可以完全实现 Linux 系统的开发。几乎和在真正的 Linux 系统下开发没有什么区别,并且其最大的好处是在 Linux 系统和 Windows 系统的之间的切换是非常的方便。所以笔者推荐读者使用这种组合方式学习 Linux 系统开发,因为它可以开发 Qt 等图形用户界面程序,与 Cygwin 工具相比,它更接近 Linux 真是环境。关于 VMware 的具体了解可参考 <http://www.vmware.com> 站点。图 1.14 所示的是 VMware 工具 + RedHat 9.0 系统在 Windows 系统下的一个登陆界面。



图 1.14 VMware 工具 + Reahat 9.0 操作系统

### 3. Linux 操作系统 + 自带的开发工具

这种组合是最完整和最权威的 Linux 系统开发方式，不过对于那些习惯 Windows 系统的 Linux 初学者来说比较困难，因为 Linux 下的许多操作都是基于命令行的，所以需要记住常用的命令，并且与 Windows 系统下的文件共享比较困难。一般常用的 Linux 系统有：Red Hat，红旗 Linux 等。

总之，以上三种 Linux 环境的开发组合读者可以根据自己的兴趣进行选择，Linux 环境下开发经常用到的工具有 GCC 或 gcc，Make 和 GDB 或 gdb，以下将逐一介绍。

#### 1.4.2.1 GCC 介绍

在 Linux 下编译程序一般都用 GCC (GNU C Compile) 开发工具，无论你是编译内核代码还是应用程序，一般都用 GCC 工具来完成。GCC 是一个全功能的 ANSI C 兼容编译器。使用 GCC 通常后跟一些选项和文件名来使用。GCC 命令的基本用法如下，本书中所有命令操作都是基于 bash (Bourne Again shell)，常见的 shell\* (注 3) 有 Bourne shell (/bin/sh)，C shell (/bin/csh)，Korn shell (/bin/ksh)，Bourne again shell (/bin/bash) 等。

*注 3：什么是 Shell？Shell 是一种具备特殊功能的程序，它是介于使用者和 UNIX/Linux 操作系统之核心程序 (kernel) 间的一个接口。众所周知，对计算机下命令需要透过命令 (command) 或程序 (program) 来执行；程序由编译器 (compiler) 将程序转为二进制代码，可是命令呢？其实 shell 也是一个程序，它由输入设备读取命令，再将其转为计算机可以了解的机器码，然后执行它。	
--	--

关于 gcc 命令的使用语法如下:

```
# gcc [options] [filenames]
```

命令行[options] (选项) 指定的操作将在命令行上每个给出的文件上执行, GCC 有超过 100 个的编译选项可用, 这些选项中的许多你可能永远都不会用到, 但一些主要的选项将会频繁用到。很多的 GCC 选项包括一个以上的字符。因此你必须为每个选项指定各自的连字符, 并且就象大多数 Linux 命令一样你不能在一个单独的连字符后跟一组选项, 例如, 下面的两个命令是不同的:

```
# gcc -p -g test.c
# gcc -pg test.c
```

第一条命令告诉 GCC 编译 test.c 时为 prof 命令建立剖析(profile)信息并且把调试信息加入到可执行的文件里。而第二条命令只告诉 GCC 为 gprof 命令建立剖析信息。所以在使用多个选项时一定要注意。

当你不用任何选项编译一个程序时, GCC 将会建立(假定编译成功)一个名为 a.out 的可执行文件。例如, 下面的命令将在当前目录下产生一个叫 a.out 的文件:

```
# gcc test.c
```

你可以用 -o 编译选项来为将产生的可执行文件指定一个文件名来代替 a.out。 例如, 将一个叫 test.c 的 C 程序编译为名叫 test 的可执行文件, 你将输入下面的命令:

```
# gcc -o test test.c
```

注意, 当你使用 -o 选项时, -o 后面必须跟一个文件名。

GCC 同样有指定编译器处理多少的编译选项, -c 选项告诉 GCC 仅把源代码编译为目标代码而跳过汇编和连接的步骤。这个选项使用的非常频繁因为它使得编译多个 C 程序时速度更快并且更易于管理。缺省时 GCC 建立的目标代码文件有一个 .o 的扩展名。-S 编译选项告诉 GCC 在为 C 代码产生了汇编语言文件后停止编译。GCC 产生的汇编语言文件的缺省扩展名是 .s。 -E 选项指示编译器仅对输入文件进行预处理。当这个选项被使用时, 预处理器的输出被送到标准输出而不是储存在文件里。

当你用 GCC 编译 C 代码时, 它会试着用最少的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行次序, 编译后的代码没有经过优化。有很多选项可用于告诉 GCC 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是 -O 和 -O2 选项。-O 选项告诉 GCC 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。-O2 选项告诉 GCC 产生尽可能小和尽可能快的代码。-O2 选项将使编译的速度比使用 -O 时慢。但通常产生的代码执行速度会更快。如果了解 GCC 的详细描述, 请参考 GCC 的指南页, 在命令行上键入 man gcc 就可以看到所有 GCC 的选项说明。

### 1.4.2.2 GNU Make 介绍

GNU Make 工具是 Linux 下非常重要的一个开发工具, 当你编译只有几个源文件的程序时也许觉得 Make 工具并没多大意义, 但是当你开发一个庞大的软件系统(比如成千上万个源文件)时, Make 工具就变得必不可少。作为一个 Linux 开发人员, 熟悉 make 工具的使用以及编写自己的 Makefile 是必需的。在 Linux 环境下使用 GNU 的 make 工具能够比较容易的构建一个属于你自己的工程, 整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。在 make 命令后不仅可以出现宏定义, 还可以跟其他命令行参数, 这些参数指定了需要编译的目标文件。其标准形式为:

```
target1 [target2 ...]:[:][dependent1 ...][;commands][#...]
```

`[(tab) commands][#...]`

方括号中间的部分表示可选项。Targets 和 dependents 当中可以包含字符、数字、句点和"/"符号。除了引用，commands 中不能含有"#"，因为"#在这里代表注释行的开始，也不允许换行。

在通常的情况下命令行参数中只含有一个":", 此时 command 序列通常和 makefile 文件中某些定义文件间依赖关系的描述行有关。如果与目标相关连的那些描述行指定了相关的 command 序列，那么就执行这些相关的 command 命令，即使在分号和(tab)后面的 command 字段甚至有可能是 NULL。如果那些与目标相关连的行没有指定 command，那么将调用系统默认的目标文件生成规则。

如果命令行参数中含有两个冒号"::", 则此时的 command 序列也许会 and makefile 中所有描述文件依赖关系的行有关。此时将执行那些与目标相关连的描述行所指向的相关命令。同时还将执行 build-in 规则。

如果在执行 command 命令时返回了一个非"0"的出错信号，例如 makefile 文件中出现了错误的目标文件名或者出现了以连字符打头的命令字符串，make 操作一般会就此终止，但如果 make 后带有"-i"参数，则 make 将忽略此类出错信号。

Make 命令本身可带有四种参数：标志、宏定义、描述文件名和目标文件名。其标准形式为：

`make [flags] [macro definitions] [targets]`

Unix/Linux 系统下标志位 flags 选项及其含义为：

-f file 指定 file 文件为描述文件，如果 file 参数为 "-" 符，那么描述文件指向标准输入。如果没有 "-f" 参数，则系统将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。在 Linux 中，GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文件。

- i 忽略命令执行返回的出错信息。
- s 沉默模式，在执行之前不输出相应的命令行信息。
- r 禁止使用 build-in 规则。
- n 非执行模式，输出所有执行命令，但并不执行。
- t 更新目标文件。
- q make 操作将根据目标文件是否已经更新返回"0"或非"0"的状态信息。
- p 输出所有宏定义和目标文件描述。
- d Debug 模式，输出有关文件的调试信息。

Linux 下 make 标志位的常用选项与 Unix 系统中稍有不同，下面我们只列出了不同部分：

- C dir 在读取 makefile 之前改变到指定的目录 dir。
- I dir 当包含其他 makefile 文件时，利用该选项指定搜索目录。
- h help 文档，显示所有的 make 选项。
- w 在处理 makefile 之前和之后，都显示工作目录。

通过命令行参数中的 target，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中第一个目标文件。为了能快速的对 make 和 Makefile 有个大体了解，这里给出一个最简单的 Makefile 实例。假设源文件有四个：main.c, file1.c, file2.c, file1.h 和 file2.h。Makefile 文件编写如下：

```
1 # 最简单的 Makefile
2 CC=gcc
3 exec=test.exe
```

```

4  obj=main.o file1.o file2.o
5  $(exec) : $(obj)
6      $(CC) -o $(exec) $(obj)
7  $(objects) : %.o : %.c
8      $(CC) -c $<
9
10 .PHONY: clean
11 clean:
12     -rm $(exec) $(obj)

```

Makefile 文件有几个非常有用的变量：分别是 \$@、\$\*、\$?、\$^、\$<，其代表的意义分别是：

```

$@ -- 完整的目标文件，包括扩展名
$* -- 目标文件去掉后缀的部分
$^ -- 所有的依赖文件
$< -- 比目标文件更新的依赖文件
$? -- 表示被修改的文件

```

在此解释一下上述的 Makefile，这是一个非常简单的 makefile，make 从最上面开始。其中#号用来注释行用得，所以第 1 行是注释行。第 2-4 行，用来定义变量，其中定义了编译器为 gcc；可执行文件为 test.exe；目标文件有 main.o，file1.o 和 file2.o。第 5 行，表示了可执行文件依赖于目标文件，注意在引用变量前一定要加\$符号，否则系统不能正确引用该变量。第 6 行，该行是命令行，值得注意的是命令行前一定是以[Tab]键开始，否则系统不能执行命令。该行等价于“gcc -o test.exe main.o file1.o file2.o”。第 7 行，表示目标文件依赖于具体的源文件。第 8 行，意思是当有文件更新时执行编译。第 10-12 行，建立一个执行 make 的清除选项，实现的功能是删除可执行文件和目标文件。

在 Makefile 建立完成后，在有 Makefile 文件的目录下通过命令行输入以下命令：

```
# make
```

上述命令的作用就是执行源代码的编译，编译的顺序和逻辑由所编写的 Makefile 文件决定，以上述的 Makefile 为例，执行后会生成一个文件名为 test.exe 的可执行性文件，然后输入以下命令执行该文件：

```
# ./test
```

此外，可以在命令行输入以下命令来清除可执行文件 test.exe，目标文件 main.o，file1.o 和 file2.o。

```
# make clean
```

通过上面的例子可以对 Make 和 Makefile 有了感性的认识，在实际工作中 makefile 文件会比较庞大，相对比较复杂，不过万变不离其宗，它的实现方式和目的都是一样，通过具体实践应用读者一定会觉得 Makefile 的编写并不是什么难事。

### 1.4.2.3 GDB 介绍

Linux 包含了一个叫 GDB (GNU DeBugger) 的 GNU 调试程序。GDB 是一个用来调试 C 和 C++ 程序的强大调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。以下是 GDB 所提供的一些基本功能：

- Ø 监视程序中变量的值
- Ø 设置断点以使程序在指定的代码行上停止执行

## Ø 单步执行代码

在命令行上键入 `gdb` 并按回车键就可以运行 GDB 了，如果已经正常安装的话，GDB 将被启动并且将在屏幕上看到以下类似的内容：

```
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

GDB 支持很多的命令使你能实现不同的功能。这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令，如表 1.1 列出了 `gdb` 调试时会用到的一些命令。想了解 GDB 的详细使用请参考 GDB 的指南页。

表 1.1 GDB 常用命令描述

命令	命令描述
<code>break</code>	在代码里设置断点，这将使程序执行到这里时被挂起
<code>file</code>	装入想要调试的可执行文件
<code>kill</code>	终止正在调试的程序
<code>list</code>	列出产生执行文件的源代码的一部分
<code>make</code>	使你在不退出 <code>gdb</code> 时就可以重新产生可执行文件
<code>next</code>	执行一行源代码但不进入函数内部
<code>print</code>	显示表达式的值
<code>quit</code>	终止 <code>gdb</code>
<code>run</code>	执行当前被调试的程序
<code>shell</code>	使你能不离开 <code>gdb</code> 就执行 <code>shell</code> 命令
<code>step</code>	执行一行源代码而且进入函数内部
<code>watch</code>	监视一个变量的值而不管它何时被改变

接下来举一个简单的 GDB 使用实例，使读者能够初步了解在 Linux 系统下应用程序的调试过程。

首先建立一个被调试的程序名叫 `test.c`，文件内容如下：

```
#include <stdio.h>
int main()
{
    char str1[]="Hello,world";
    char str2[11]="";
    int i=0;

    while(str1[i]!='\0')
    {
        str2[i]=str1[i];
        i++;
    }
}
```

```
printf("The string1 is:%s.\n",str1);
printf("The string2 is:%s.\n",str2);
return 0;
}
```

然后对这个源程序进行编译，编译和执行命令如下：

```
# gcc -g -o test test.c
# ./test
The string1 is:Hello,world.
The string2 is:Hello,world?E?Hello,world.
```

注意上面的命令中多了一个-g 选项，该选项的含义是为接下来调试作准备的，如果在编译时没有加这个选项，那么就不能直接运行 gdb 命令进行调试该程序，不过可以通过另外一种方式来代替它，就是进入 gdb 以后，在 gdb 命令输入行输入以下命令来调试该程序：

```
(gdb) gdb test
```

通过上面的执行结果可以看出，test 程序的目的是打印结果应该为：

```
The string1 is:Hello,world.
```

```
The string2 is:Hello,world.
```

但实际打印结果却是：

```
The string1 is:Hello,world.
```

```
The string2 is:Hello,world?E?Hello,world.
```

通过以上输出的结果可以得知出错的大概位置就在 while 循环执行过程中，接下来用 gdb 进行调试，进入 gdb 调试界面后会显示如下：

```
(gdb) list
1      #include <stdio.h>
2      int main()
3      {
4          char str1[]="Hello,world";
5          char str2[11]="";
6          int i=0;
7
8          while(str1[i]!='\0')
9          {
10             str2[i]=str1[i];
(gdb) list
11             i++;
12         }
13
14         printf("The string1 is:%s.\n",str1);
15         printf("The string2 is:%s.\n",str2);
16         return 0;
17     }
(gdb)
Line number 18 out of range; test.c has 17 lines.
```

其中用 list 命令用来显示要调适的程序，一般需要好几页才能显示完程序，所以显示下



一页程序继续用 `list` 命令或输入回车即可。下来就要对程序中可能会出错的地方设置断点，设置断点的方法如下，设置断点在第 8 行。

```
(gdb) break 8
Breakpoint 1 at 0x8048373: file test.c, line 8.
```

然后用 `run` 命令运行程序，显示信息如下：

```
(gdb) run
Starting program: /home/mike/test

Breakpoint 1, main () at test.c:8
8             while(str1[i]!='\0')
```

可见程序在第 8 行时暂停运行了，因为在第 8 行对它设置了断点，如果想继续执行一行源代码但不进入函数内部，用 `next` 命令即可。

```
(gdb) next
10                     str2[i]=str1[i];
(gdb) watch str2[i]
Watchpoint 2: str2[i]
(gdb) next
Watchpoint 2: str2[i]

Old value = 0 '\0'
New value = 72 'H'
main () at test.c:11
11                     i++;
```

上述调试过程中，用 `watch` 命令来监视 `str2[i]` 的值，其中可以看出 `str2[i]` 的以前存放的值和新赋得值，然后继续用 `next` 命令察看 `str2[i]` 的值，发现 `while` 循环过程中执行没有问题，然后给 14 和 15 行设置了断点，结果在执行第 15 行时程序终止，并且报错如下：

```
Program received signal SIGSEGV, Segmentation fault.
0x080483ae in main () at test.c:15
15             printf("The string2 is:%s.\n",str2);
(gdb) next

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

说明执行在第 15 行时出错，经分析是由于 `str2` 数组越界造成的，也就是说 `str2` 的数组没有 `'\0'` 结束符，这样在程序中会经常输出异常的值，该程序改正的方法就是将第 5 行代码改为：`char str2[12]="";` 或者数组的大小大于 12 即可。

以上通过一个非常简单的例子讲述 GDB 在实际中的应用，其实在实际中不会像例子中那么简单的程序，但是操作方式都非常类似，请读者具体使用时参考 GDB 指南。

### 1.4.3 ARM Linux 系统开发流程

不同于常见的桌面系统开发软件，在开发嵌入式系统时，常常把所有的软件模块最终都生成一个单一的文件，我们把这个单一的文件称为 `image`（通常叫映像文件），它一般布局

如下图 1.15 所示：

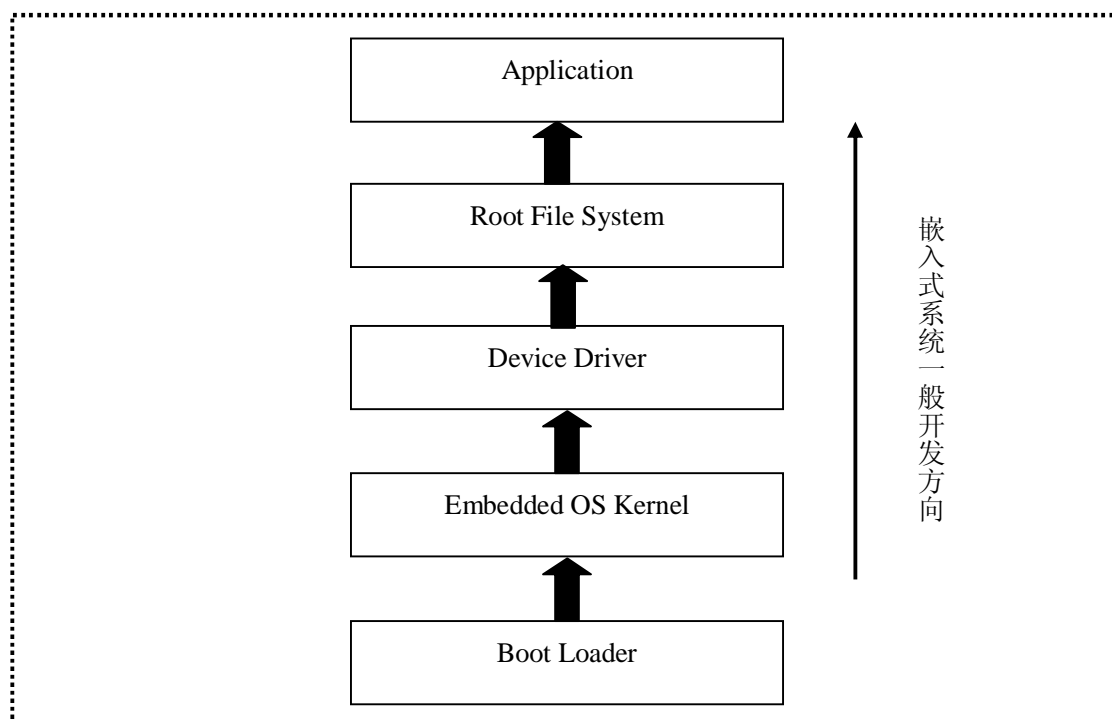


图 1.15 典型嵌入式系统软件 image 的逻辑布局

其中最底层是 BootLoader（启动加载程序），接着是嵌入式操作系统内核（如 Linux 内核），内核之上就是设备驱动，然后就是根文件系统和应用程序。这里只是给出一般情况下映像文件的逻辑组成，通常还有 DSP 等其他程序。

一般嵌入式开发方法如图 1.16 所示，当程序员开始开发一个基于嵌入式系统应用的时候，首先用该嵌入式相关的开发工具在宿主主机上进行开发，例如通常使用 ARM 的 ADS 工具编写程序，使用 ARMulator 或者在评估板上调试，最后将在宿主主机上开发生成的 image 文件烧写到独立的嵌入式应用设备中去。

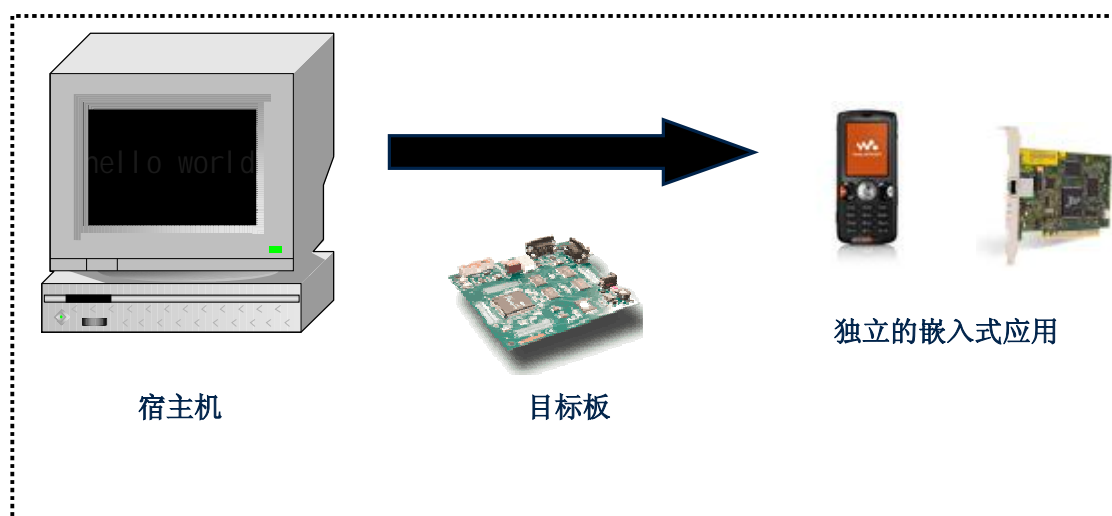


图 1.16 嵌入式系统开发的一般方法

通常基于 ARM 系统的 Linux 开发步骤如下：

- a) 开发目标硬件系统：如选择微处理器，flash 及其它外设等。

- b) 建立交叉编译工具：一般的 GCC 工具都是针对 X86 体系的，为了能够产生目标板执行的代码必须建立交叉编译工具。
- c) 开发 Bootloader：建立启动系统的主引导程序。
- d) 移植 Linux 内核：如基于 ARM 的 Linux 2.6 内核移植。
- e) 开发一个根文件系统：如 rootfs 的制作。
- f) 开发相关硬件的驱动程序：如 LCD，Keypad 等。
- g) 开发上层的应用程序：如 QT GUI 开发。

## 1.5 Linux 内核介绍

Linux 内核是 Linux 系统的核心，它实现了操作系统五大主要功能模块：进程管理、内存管理、文件系统、设备控制和网络。Linux 内核的功能模块如图 1.17 所示[1]：

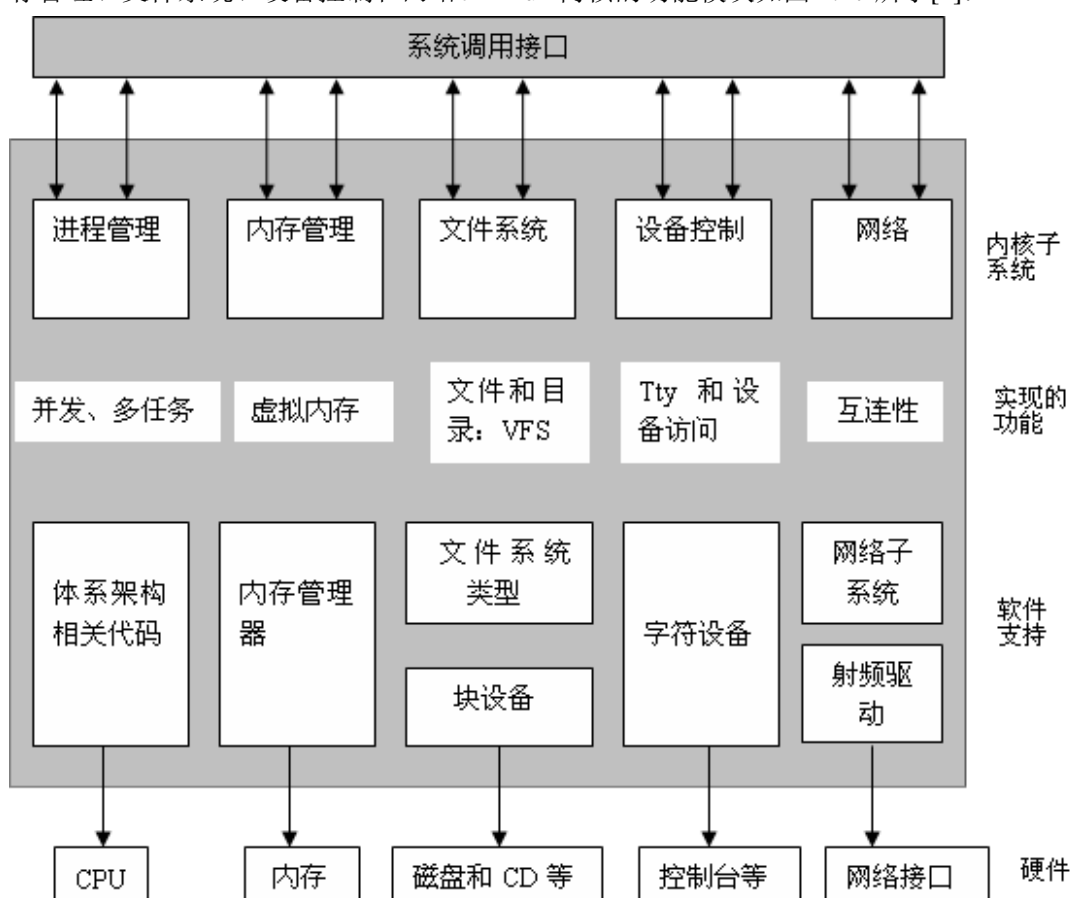


图 1.17 Linux 内核的功能模块划分

进程管理模块可以说是 Linux 内核的心脏模块，它负责创建和终止进程，并且处理它们和外部世界的联系（输入和输出）。对整个系统功能来讲，不同进程之间的通信（通过信号，管道，进程间通信原语）是基本的，这也是由内核来处理的。另外，调度器应该是整个操作系统中最关键的例程，是进程管理中的一部分。更广义的说，内核的进程管理活动实现了在一个 CPU 上多个进程的抽象概念。内存管理模块的作用是用于确保所有进程能够安全地共享计算机主内存区，此外，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多的内存容量，并可以利用文件系统把暂时不用的内存数据块交换到外部存储设备中去，等需要时再交换回来，这样大大提高了内存使用效率，节省了内存空间。文件系统模块用于支持对外部设备的驱动和存储，虚拟文件系统通过向所有的外部存储设备

提供一个通用的文件系统接口，从而隐藏了各种硬件设备的不同细节。网络模块提供对多种网络通信标准的访问，并支持许多网络硬件设备[1]。

总之，本书不是讲述操作系统内核原理的书籍，读者可以参考相关的书籍来学习它。但是对于本书中所介绍的内容常常是需要跟内核打交道，所以希望读者对操作系统内核原理有所了解。

### 1.5.1 Linux 内核目录结构

在解压后的 Linux 内核（这里以 Linux 2.6.10 内核为例）根目录下输入如下 `tree` 命令，将会显示如下目录信息，其中 `tree` 命令是由 `tree` 工具实现的，用来显示文件目录信息，它的下载站点是 [ftp://mama.indstate.edu/linux/tree/](http://mama.indstate.edu/linux/tree/)。

```
# tree -L 1
.
|-- COPYING
|-- CREDITS
|-- Documentation
|-- MAINTAINERS
|-- Makefile
|-- README
|-- REPORTING-BUGS
|-- arch
|-- crypto
|-- drivers
|-- fs
|-- include
|-- init
|-- ipc
|-- kernel
|-- lib
|-- mm
|-- net
|-- scripts
|-- security
|-- sound
`-- usr
```

16 directories, 6 files

内核根目录下的主要目录和文件的意义介绍如下：

**COPYING:** 该文件主要是对 Linux 内核代码的版权声名。

**CREDITS:** 该文件是对该版本和之前版本 Linux 内核所做贡献的所有成员列表。

**Documentation:** 该目录存放所有内核相关的技术文档，是学习内核原理的很好参考资料。

**MAINTAINERS:** 该文件记录所有维护内核人员列表以及讲述如何提交一个内核的改变。

**Makefile:** 该文件是编译内核的最上层 Makefile 文件，也是编译内核的入口文件。

**README:** 该文件是编译内核的帮助文件，编译前一定要阅读该文件，该文件对于编译内核很有帮助。

**REPORTING-BUGS:** 该文件是有关提交内核 Bug 的一些要求和建议。

**Arch:** 该目录包括了所有和体系结构相关的核心代码。它下面的每一个子目录都代表一种 Linux 支持的体系结构，例如 i386 就是 Intel CPU 及与之相兼容体系结构的子目录。arm 就代表是 ARM 体系结构相关的代码。

**Drivers:** 该目录包含内核中所有硬件相关的驱动实现代码，它又进一步划分成几类设备驱动，比如 char 目录为字符设备，block 目录为块设备等。

**Fs:** 该目录存放 Linux 支持的文件系统代码。不同的文件系统有不同的子目录对应，如 ext3 文件系统对应的就是 ext3 子目录。

**Include:** 该目录包括编译内核所需要的大部分头文件，例如与平台无关的头文件在 include/linux 子目录下。

**Init:** 该目录包含内核的初始化代码（不是系统的引导代码）。这是研究内核如何工作的好起点。

**Ipc:** 该目录包含了内核进程间的通信代码。

**Kernel:** 该目录包含内核管理的核心代码。同时与处理器结构相关代码都放在 arch/\*/kernel 目录下。

**Lib:** 该目录包含了内核的库代码，与处理器结构相关的库代码被放在 arch/\*/lib/目录下。

**Mm:** 该目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/\*/mm 目录下。

**Net:** 该目录里是内核的网络部分代码，其每个子目录对应于网络的一个方面。

**Scripts:** 该目录包含用于配置核心的脚本文件。

## 1.5.2 如何阅读 Linux 内核源代码

Linux 0.01 版是在 1991 年出生的，是 Linux 内核的第一版，该内核的大小是 158K 字节，6975 行代码。Linux 内核在全球热衷于开源项目的计算机高手努力下已经发展为一个庞大成熟的操作系统，到目前为止，最新的内核版本是 2.6.18，其大小超过 200M 字节，代码行超过 400 万行，如此庞大的源代码如果没有合适的工具或方法去阅读它，那么想在短时间学习它基本上是不可能的。技术的发展总会使人类不断简化所作的工作，同样现在可以利用有效的工具来辅助我们去阅读和研究如此庞大的系统。目前最流行的阅读内核源代码工具有两个，UltraEdit 和 SourceInsight，这两个工具在实际工作中应用非常广泛。下面简单介绍一下这两个重要工具。

**UltraEdit:** 它是一套功能强大的文本编辑器，可以编辑文本、十六进制、ASCII 码，可以取代记事本，内建英文单字检查、C、C++ 及 VB 指令突显，可同时编辑多个文件，而且即使开启很大的文件速度也是相当的快。软件附有 C/C++ 标签颜色显示、搜寻替换以及无限制的还原功能，一般大家喜欢用其来修改 EXE、DLL 和源文件等。比如用 UE (UltraEdit 简称) 打开 Linux 2.6.10 内核的一个名为 cpu.c 的源文件，如图 1.18 所示。用 UE 阅读源代码可以高亮显示源代码的关键字，查找方便，尤其是可以方便编辑。关于 UE 的具体使用请参考其帮助文档。

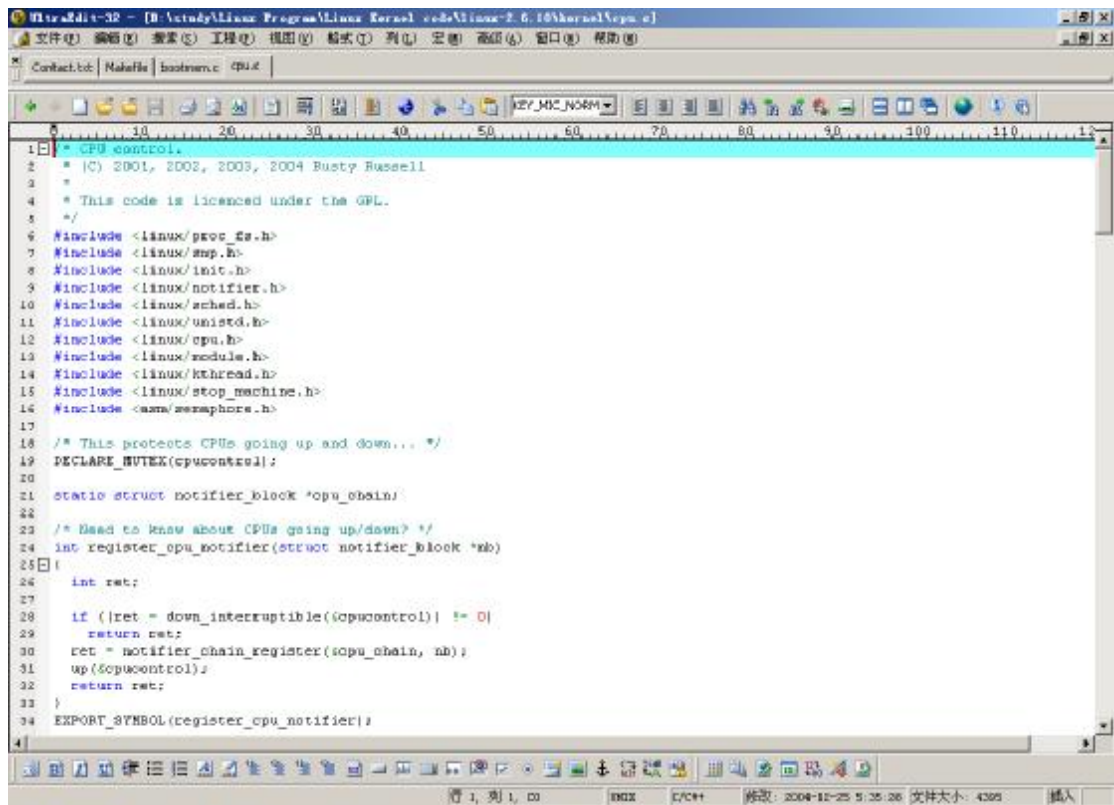


图 1.18: UltraEdit 查看源代码文件

**SourceInsight:** 它实质上是一个支持多种开发语言（java,c,c++等等）的编辑器，只不过由于其查找、定位、彩色显示等功能的强大，而被我们当成源代码阅读工具使用。它和 UltraEdit 相比较增加了许多功能，比如提供了代码之间调用关系的显示，以及文件之间的关系查找。比如用 SourceInsight 建立一个 Linux 2.6.10 内核代码的项目工程，如图 1.19 所示。关于 SourceInsight 的具体使用方法请参考其帮助文档。

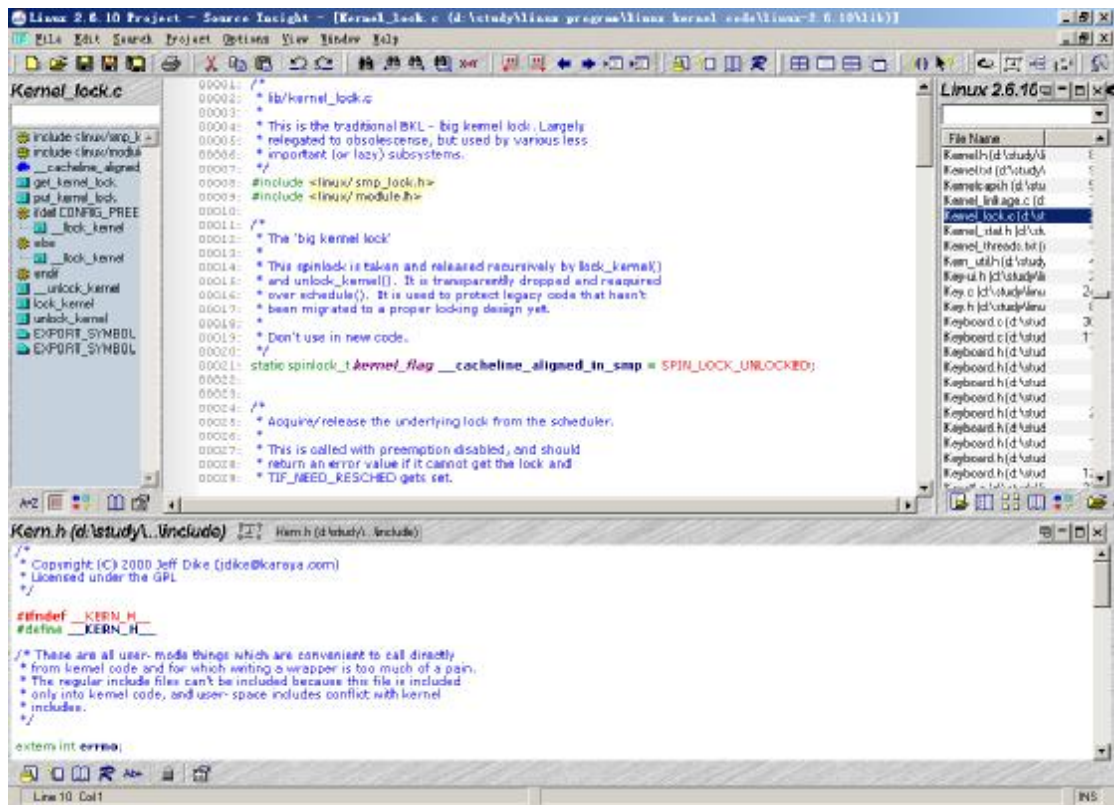


图 1.19: 用 SourceInsight 建立的 Linux 2.6.10 内核代码工程

注意: UltraEdit 和 SourceInsight 这两个工具通常使用在 Windows 操作系统下使用。

## 1.6 本章小节

本章是嵌入式系统开发的最基础部分,通过本章的学习读者可以了解嵌入式系统的基本概念,嵌入式系统的基本组成,ARM 处理器的基本知识,ADS 工具的基本使用方法, Linux 开发环境, ARM Linux 系统开发的基本流程,以及 Linux 内核目录结构和阅读 Linux 内核代码的方法。通过学习本章的内容,为以后更深入的学习嵌入式开发打下良好的基础。下一章将开始介绍如何自己构建交叉编译工具链。

## 1.7 常见问题

### 1. 嵌入式微处理器有哪些特点？

参考答案：

通常情况下，嵌入式微处理器都具有以下四个基本特点：

1. 对实时多任务有很强的支持能力，能完成多任务并且有较短的中断响应时间，从而使内部的代码和实时内核的执行时间减少到最低限度。
2. 具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断。
3. 可扩展的处理器结构，以能最迅速地开展出满足应用的最高性能的嵌入式微处理器。
4. 嵌入式微处理器必须功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此。

### 2. 嵌入式系统的组成部分？

参考答案：

嵌入式系统一般由硬件平台和软件平台两部分组成，其中硬件平台由嵌入式微处理器和外围硬件设备组成，而软件平台由嵌入式操作系统和应用软件组成。

### 3. 嵌入式 ARM Linux 系统的一般开发步骤？

参考答案：

通常基于 ARM 平台的嵌入式 Linux 开发步骤如下：

1. 开发目标硬件系统
2. 建立交叉编译工具
3. 开发 Bootloader
4. 开发 Linux 内核
5. 开发一个根文件系统
6. 开发特定硬件的驱动程序
7. 开发上层的应用程序。

### 4. 选择 ARM 处理器的准则有哪些？

参考答案：

选择 ARM 处理器一般需要关注以下 4 个主要方面：

1. ARM 微处理器内核的选择
2. 系统的工作频率
3. 晶片内部存储体的容量
4. 晶片内部周围电路选择

### 5. 在 makefile 文件中，特殊符号\$@、\$\*、\$?、\$^和\$<分别代表什么？

参考答案：

- \$@ -- 完整的目标文件，包括扩展名
- \$\* -- 目标文件去掉后缀的部分
- \$^ -- 所有的依赖文件
- \$< -- 比目标文件更新的依赖文件
- \$? -- 表示被修改的文件



## 第 2 章 交叉编译工具链的构建

本章学习目标：

- 1 了解交叉编译工具链
- 1 理解分步构建交叉编译工具链的方法
- 1 学会使用 Crosstool 工具构建交叉编译工具链

### 2.1 交叉编译工具链介绍

在介绍交叉编译工具链之前读者会有个疑问，为什么要用交叉编译器？交叉编译通俗地讲就是在一种平台上编译出能运行在体系结构不同的另一种平台上，比如在 PC 平台 (X86 CPU) 上编译出能运行在 ARM 为内核 CPU 平台上的程序，编译得到的程序在 X86 CPU 平台上是不能运行的，必须放到 ARM CPU 平台上才能运行，当然两个平台用的都是 Linux 系统。这种方法在异平台移植和嵌入式开发时非常普遍。相对与交叉编译，平常做的编译叫本地编译，也就是在当前平台编译，编译得到的程序也是在本地执行。用来编译这种跨平台程序的编译器就叫交叉编译器，相对来说，用来做本地编译的工具就叫本地编译器。所以要生成在目标板上运行的程序，必须要用交叉编译工具链来完成。在裁减和定制 Linux 内核用于嵌入式系统之前，由于一般嵌入式开发系统存储大小有限，通常都要在性能优越的 PC 机上建立一个用于目标机的交叉编译工具链，用该交叉编译工具链在 PC 机上编译目标板上要运行的程序。交叉编译工具链是一个由编译器、连接器和解释器组成的综合开发环境。交叉编译工具链主要由 binutils、gcc 和 glibc 三个部分组成。有时出于减小 libc 库大小的考虑，你也可以用别的 c 库来代替 glibc，例如 uClibc、dietlibc 和 newlib。建立一个交叉编译工具链是一个相当复杂的过程，如果你不想自己经历复杂繁琐的编译过程，网上有一些编译好的可用的交叉编译工具链可以下载，但就学习为目的来说读者有必要学习自己制作一个交叉编译工具链。本章通过具体的实例讲述基于 ARM 的嵌入式 Linux 交叉编译工具链的制作过程。

### 2.2 ARM Linux 交叉编译工具链的构建

构建交叉编译器的第一个步就是确定目标平台。在 GNU 系统中，每个目标平台都有一个明确的格式，这些信息用于在构建过程中识别要使用的不同工具的正确版本。因此，当你在一个特定目标机器下运行 GCC 时，GCC 便在目录路径中查找包含该目标规范的应用程序路径。GNU 的目标规范格式为 CPU-PLATFORM-OS。例如 x86/i386 目标机名为：i686-pc-linux-gnu。本章目的是讲述建立基于 ARM 平台的交叉工具链，所以目标平台名：arm-linux-gnu。

通常构建交叉工具链有三种方法：

方法一：分步编译和安装交叉编译工具链所需要的库和源代码，最终生成交叉编译工具链。该方法相对比较困难，适合想深入学习构建交叉工具链的读者。如果只是想使用交叉工具链，建议使用方法二或方法三构建交叉工具链。

方法二：通过 Crosstool 脚本工具来实现一次编译生成交叉编译工具链，该方法相对于方法一要简单许多，并且出错的机会也非常少，建议大多数情况下使用该方法构建交叉工具链。

方法三：直接通过网上（<ftp.arm.kernel.org.uk>）下载已经制作好的交叉编译工具链。该方法的优点不用多说了，当然是简单省事了，与此同时该方法有一定的弊端就是局限性太大，因为毕竟是别人构建好的，也就是固定的没有灵活性，构建所用的库以及编译器的版本也许并不适合你要编译的程序，同时也许会在使用时出现许多莫名的错误，建议读者慎用此方法。

为了让读者真正的学习交叉编译工具链的构建，下面将重点详细地介绍前两种构建 ARM Linux 交叉编译工具链的方法。

### 2.2.1 分步构建交叉编译链

分步构建，顾名思义就是一步一步的建立交叉编译链，不同于下一节中讲述的 Crosstool 脚本工具一次编译生成的方法，该方法建议适合那些希望深入学习了解构建交叉编译工具链的读者。该方法相对来说难度较大，通常情况下困难重重，犹如唐僧西天取经，不过本节尽可能详细地介绍构建的每一个步骤，读者完全可以根据本节的内容自己独立去实践，构建自己的交叉工具链。该过程所需的时间较长，希望读者有较强的耐心和毅力去学习和实践它，通过实践可以使读者更加清楚交叉编译器构建过程以及各个工具包的作用。该方法所需资源如表 2.1 所示。

表 2.1 所需资源

安装包	下载地址
linux-2.6.10.tar.gz	<a href="ftp.kernel.org">ftp.kernel.org</a>
binutils-2.15.tar.bz2	<a href="ftp.gnu.org">ftp.gnu.org</a>
gcc-3.3.6.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>
glibc-2.3.2.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>
glibc-linuxthreads-2.3.2.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>

以上资源通过相关站点下载后，就可以开始建立交叉编译工具链了。

#### 2.2.1.1 建立工作目录

首先建立工作目录，工作目录就是在什么目录下构建交叉工具链，目录的构建一般没有特别的要求，根据个人喜好建立。以下所建立的目录是作者自定义的，当前的用户定义为 mike，因此用户目录为：/home/mike，在用户目录下首先建立一个工作目录（armlinux）。建立工作目录的命令行操作如下：

```
# cd /home/mike
# mkdir armlinux
```

再在这个工作目录 armlinux 下建立三个目录 build-tools、kernel 和 tools。具体操作如下：

```
# cd armlinux
# mkdir build-tools kernel tools
```

其中各目录的作用是：

build-tools：用来存放你下载的 binutils、gcc、glibc 等源代码和用来编译这些源代码的目录。

kernel：用来存放你的内核源代码。

tools：用来存放编译好的交叉编译工具和库文件。

### 2.2.1.2 建立环境变量

该步骤的目的是为了方便重复输入路径,因为重复操作每件相同的事情总会让人觉得很麻烦,如果读者不习惯使用环境变量就可以略过该步,直接输入绝对路径就可以。声明以下环境变量的目的就是在之后编译工具库的时候会用到,并且很方便输入,尤其是可以降低输错路径的风险。

```
# export PRJROOT=/home/mike/armlinux
# export TARGET=arm-linux
# export PREFIX=$PRJROOT/tools
# export TARGET_PREFIX=$PREFIX/$TARGET
# export PATH=$PREFIX/bin:$PATH
```

注意,用 `export` 声明的变量是临时的变量,也就是当你注销或更换了控制台,这些环境变量就消失了,如果还需要使用这些环境变量就必须重复 `export` 操作,所以有时会挺麻烦。值得庆幸的是,环境变量也可以定义在 `bashrc` 文件中,这样当注销或更换控制台时,这些变量就一直有效,就不用老是 `export` 这些变量了。

### 2.2.1.3 编译、安装 Binutils

Binutils 是 GNU 工具之一,它包括连接器,汇编器和其他用于目标文件和档案的工具,它是二进制代码的处理维护工具。安装 Binutils 工具包含的程序有: `addr2line`, `ar`, `as`, `c++filt`, `gprof`, `ld`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings`, `strip`, `libiberty`, `libbfd` 和 `libopcodes`。对这些程序的简单解释如下:

**addr2line:** 把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行文件名,它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。

**ar:** 建立、修改、提取归档文件。归档文件是包含多个文件内容的一个大文件,其结构保证了可以恢复原始文件内容。

**as:** 主要用来编译 GNU C 编译器 `gcc` 输出的汇编文件,产生的目标文件由连接器 `ld` 连接。

**c++filt:** 连接器使用它来过滤 C++ 和 Java 符号,防止重载函数冲突。

**gprof:** 显示程序调用段的各种数据。

**ld:** 是连接器,它把一些目标和归档文件结合在一起,重定位数据,并链接符号引用。通常,建立一个新编译程序的最后一步就是调用 `ld`。

**nm:** 列出目标文件中的符号。

**objcopy:** 把一种目标文件中的内容复制到另一种类型的目标文件中。

**objdump:** 显示一个或者更多目标文件的信息。使用选项来控制其显示的信息。它所显示的信息通常只有编写编译工具的人才感兴趣。

**ranlib:** 产生归档文件索引,并将其保存到归档文件中。在索引中列出了归档文件各成员所定义的可重分配目标文件。

**readelf:** 显示 `elf` 格式可执行文件的信息。

**size:** 列出目标文件每一段的大小以及总体的大小。默认情况下,对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

**strings:** 打印某个文件的可打印字符串,这些字符串最少 4 个字符长,也可以使用选项 `-n` 设置字符串的最小长度。默认情况下,它只打印目标文件初始化和可加载段中的可打印

字符；对于其它类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。

**strip:** 丢弃目标文件中的全部或者特定符号。

**libiberty:** 包含许多 GNU 程序都会用到的函数，这些程序有：getopt, obstack, strerror, strtol 和 strtoul。

**libbfd:** 二进制文件描述库。

**libopcode:** 用来处理 opcodes 的库，在生成一些应用程序的时候也会用到它。

Binutils 工具安装依赖于：Bash, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo 等工具。

介绍完 Binutils 工具后，下面将分步介绍安装 binutils-2.15 的过程。

首先解压 binutils-2.15.tar.bz2 包，命令如下：

```
# cd $PRJROOT/build-tools
# tar -xjvf binutils-2.15.tar.bz2
```

接着配置 Binutils 工具，建议建立一个新的目录用来存放配置和编译文件，这样可以使源文件和编译文件独立开，具体操作如下：

```
# cd $PRJROOT/build-tools
# mkdir build-binutils
# cd build-binutils
# ../binutils-2.15/configure --target=$TARGET --prefix=$PREFIX
```

其中选项 `--target` 的意思是制定生成的是 `arm-linux` 的工具，`--prefix` 是指出可执行文件安装的位置。执行上述操作会出现很多 `check` 信息，最后产生 `Makefile` 文件。接下来执行 `make` 和安装操作，命令如下：

```
# make
# make install
```

该编译过程较慢，需要数十分钟，安装完成后察看 `/home/mike/armlinux/tools/bin` 目录下文件，察看结果如下，表明此时 Binutils 工具已经安装结束。

```
# ls $PREFIX/bin
arm-linux-addr2line  arm-linux-ld          arm-linux-ranlib      arm-linux-strip
arm-linux-ar         arm-linux-nm          arm-linux-readelf
arm-linux-as         arm-linux-objcopy     arm-linux-size
arm-linux-c++filt    arm-linux-objdump     arm-linux-strings
```

#### 2.2.1.4 获得内核头文件

编译器需要通过系统内核的头文件来获得目标平台所支持的系统函数调用所需要的信息。对于 Linux 内核，最好的方法是下载一个合适的内核，然后拷贝获得头文件。需要对内核做一个基本的配置来生成正确的头文件；不过，不需要编译内核。对于本例中的目标，`arm-linux`，需要以下步骤：

在 `kernel` 目录下解压 `linux-2.6.10.tar.gz` 内核包，执行命令如下：

```
# cd $PRJROOT/kernel
# tar -xvzf linux-2.6.10.tar.gz
```

接下来配置编译内核使其生成正确的头文件，执行命令如下：

```
# cd linux-2.6.10
```

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

其中 ARCH=arm 表示是以 arm 为体系结构, CROSS\_COMPILE=arm-linux- 表示以 arm-linux- 为前缀的交叉编译器。也可以用 config 和 xconfig 来代替 menuconfig, 推荐大家用 make menuconfig, 这也是内核开发人员用的最多的配置方法。注意在配置时一定要选择处理器的类型, 这里选择三星的 S3C2410 (System Type->ARM System Type->/Samsung S3C2410), 如图 2.1 所示。配置完退出并保存, 检查一下的内核目录中的 include/linux/version.h 和 include/linux/autoconf.h 文件是不是生成了, 这是编译 glibc 是要用到的, 如果 version.h 和 autoconf.h 文件存在, 这说明生成了正确的头文件。



图 2.1 Linux 2.6.10 内核配置界面

拷贝头文件到交叉工具链的目录, 首先需要在 /home/mike/armlinux/tools/arm-linux 目录下建立工具的头文件目录 include, 然后拷贝内核头文件到此目录下, 具体操作如下:

```
# mkdir -p $TARGET_PREFIX/include
# cp -r $PRJROOT/kernel/linux-2.6.10/include/linux $TARGET_PREFIX/include
# cp -r $PRJROOT/kernel/linux-2.6.10/include/asm-arm $TARGET_PREFIX/include/asm
```

### 2.2.1.5 编译安装 boot-trap gcc

这一步的目的主要是建立 arm-linux-gcc 工具, 注意这个 gcc 没有 glibc 库的支持, 所以只能用于编译内核, bootloader 等不需要 C 库支持的程序, 后面创建 C 库也要用到这个编译器, 所以创建它主要是为创建 C 库做准备, 如果只想编译内核和 Bootloader, 那么安装完这个就可以到此结束。安装命令如下:

```
# cd $PRJROOT/build-tools
# tar -xvzf gcc-3.3.6.tar.gz
# mkdir build-gcc
# cd gcc-3.3.6
# vi gcc/config/arm/t-linux
```

由于第一次安装 ARM 交叉编译工具, 那么支持的 libc 库的头文件也没有, 所以在 gcc/config/arm/t-linux 文件中给变量 TARGET\_LIBGCC2\_CFLAGS 增加操作参数选项 -D\_\_gthr\_posix\_h 来屏蔽使用头文件, 否则一般默认会使用 /usr/include 头文

件。

将 TARGET\_LIBGCC2-CFLAGS = -fomit-frame-pointer -fPIC 改为以下：

TARGET\_LIBGCC2-CFLAGS=-fomit-frame-pointer-fPIC -D\_\_gthr\_posix\_h  
修改完 t-linux 文件后保存，紧接着执行配置操作，如下命令：

```
# cd build-gcc
# ./ build-gcc /configure --target=$TARGET --prefix=$PREFIX --enable-languages=c
--disable-threads --disable-shared
```

其中选项--enable-languages=c 表示只支持 C 语言，--disable-threads 表示去掉 thread 功能，这个功能需要 glibc 的支持。--disable-shared 表示只进行静态库编译，不支持共享库编译。

接下来执行编译和安装操作，命令如下：

```
# make
# make install
```

安装完成后，在/home/mike/armlinux/tools/bin 下查看，arm-linux-gcc 等工具已经生成，boot-trap gcc 工具已经安装成功。

### 2.2.1.6 建立 glibc 库

glibc 是 GUN C 库，它是编译 Linux 系统程序很重要的组成部分。安装 glibc-2.3.2 版本之前推荐先安装以下的工具：

- l GNU make 3.79 或更新
- l GCC 3.2 或更新
- l GNU binutils 2.13 或更新

首先解压 glibc-2.2.3.tar.gz 和 glibc-linuxthreads-2.2.3.tar.gz 源代码，操作如下：

```
# cd $PRJROOT/build-tools
# tar -xvzf glibc-2.2.3.tar.gz
# tar -xvzf glibc-linuxthreads-2.2.3.tar.gz --directory=glibc-2.2.3
```

然后进行编译配置，glibc-2.2.3 配置前必须新建一个编译目录，否则在 glibc-2.2.3 目录下不允许进行配置操作，此处在\$PRJROOT/build-tools 目录下建立名为 build-glibc 的目录，配置操作如下：

```
# cd $PRJROOT/build-tools
# mkdir build-glibc
# cd build-glibc
# CC=arm-linux-gcc ../glibc-2.2.3 /configure --host=$TARGET --prefix="/usr"
--enable-add-ons --with-headers=$TARGET_PREFIX/include
```

选项 CC=arm-linux-gcc 是把 CC (Cross Compiler) 变量设成刚编译完的 gcc，用它来编译 glibc。--prefix="/usr" 定义一个目录用于安装一些与机器无关的数据文件，默认情况下是/usr/local 目录。--enable-add-ons 是告诉 glibc 用 linuxthreads 包，在上面已经将它放入了 glibc 源码目录中，这个选项等价于 -enable-add-ons=linuxthreads。--with-headers 告诉 glibc linux 内核头文件的目录位置。

配置完后就可以编译和安装 glibc 了，具体操作如下：

```
# make
# make install
```

### 2.2.1.7 编译安装完整的 gcc

由于第一次安装的 gcc 没有交叉 glibc 的支持，现在已经安装了 glibc，所以需要重新编译来支持交叉 glibc。并且上面的 gcc 也只支持 C 语言，现在可以让它不仅支持 C 语言还要让它支持 C++ 语言。具体操作如下：

```
# cd $PRJROOT/build-tools/gcc-2.3.6
# ./configure --target=arm-linux --enable-languages=c,c++ --prefix=$PREFIX
# make
# make install
```

安装完成后会发现在 \$PREFIX/bin 目录下又多了 arm-linux-g++ 、 arm-linux-c++ 等文件。

```
# ls $PREFIX/bin
arm-linux-addr2line  arm-linux-g77          arm-linux-gnatbind  arm-linux-ranlib
arm-linux-ar         arm-linux-gcc          arm-linux-jcf-dump  arm-linux-readelf
arm-linux-as         arm-linux-gcc-3.3.6   arm-linux-jv-scan   arm-linux-size
arm-linux-c++        arm-linux-gccbug       arm-linux-ld        arm-linux-strings
arm-linux-c++filt    arm-linux-gcj          arm-linux-nm        arm-linux-strip
arm-linux-cpp        arm-linux-gcjh         arm-linux-objcopy   grepjar
arm-linux-g++        arm-linux-gcov         arm-linux-objdump   jar
```

### 2.2.1.8 测试交叉编译工具链

到此为止，已经介绍完了用分步构建的方法建立交叉工具链。下面通过一个简单的程序可以测试刚刚建立的交叉工具链看是否能够正常工作。写一个最简单的 hello.c 源文件，内容如下：

```
#include <stdio.h>
int main()
{
    printf("Hello,world!\n");
    return 0;
}
```

通过以下命令进行编译，编译后生成名为 hello 的可执行文件。通过 file 命令可以查看文件的类型。具体操作如下，当显示以下信息时表明交叉工具链正常安装了，通过以下的编译生成了 ARM 体系可执行的文件。注意，通过该交叉编译链编译的可执行文件只能在 ARM 体系下执行，不能在基于 X86 的普通 PC 上执行该文件。

```
# arm-linux-gcc -o hello hello.c
# file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.4.3,
dynamically linked (uses shared libs), not stripped
```

## 2.2.2 用 Crosstool 工具构建交叉工具链

Crosstool 是一组脚本工具集，构建和测试不同版本的 gcc 和 glibc，用于那些支持 glibc

的许多体系结构。它也是一个开源项目，下载地址是：<http://kegel.com/crosstool>。用 Crosstool 构建交叉工具链要比上述的分步编译要容易得多，并且也方便许多，对于仅仅为了工作需要构建交叉编译工具链的读者建议使用此方法。用 Crosstool 工具构建所需资源如表 2.2 所示。

表 2.2 所需资源

安装包	下载地址
crosstool-0.42.tar.gz	<a href="http://kegel.com/crosstool">http://kegel.com/crosstool</a>
linux-2.6.10.tar.gz	<a href="ftp.kernel.org">ftp.kernel.org</a>
binutils-2.15.tar.bz2	<a href="ftp.gnu.org">ftp.gnu.org</a>
gcc-3.3.6.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>
glibc-2.3.2.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>
glibc-linuxthreads-2.3.2.tar.gz	<a href="ftp.gnu.org">ftp.gnu.org</a>
linux-libc-headers-2.6.12.0.tar.bz2	<a href="ftp.gnu.org">ftp.gnu.org</a>

### 2.2.2.1 准备资源文件

首先从网上下载所需资源文件：linux-2.6.10.tar.gz，binutils-2.15.tar.bz2，gcc-3.3.6.tar.gz，glibc-2.3.2.tar.gz，glibc-linuxthreads-2.3.2.tar.gz 和 linux-libc-headers-2.6.12.0.tar.bz2。然后将这些工具包文件放在新建的/home/mike/downloads 目录下，最后在/home/mike 目录下解压 crosstool-0.42.tar.gz，命令如下：

```
# cd /home/mike
# tar -xvzf crosstool-0.42.tar.gz
```

### 2.2.2.2 建立脚本文件

接着需要建立自己的编译脚本，起名为 arm.sh，为了简化编写 arm.sh，寻找一个最接近的脚本文件 demo-arm.sh 作为模版，然后将该脚本的内容复制到 arm.sh，修改 arm.sh 脚本，具体操作如下：

```
# cd crosstool-0.42
# cp demo-arm.sh arm.sh
# vi arm.sh
```

修改后的 arm.sh 的脚本内容如下：

```
#!/bin/sh
set -ex
TARBALLS_DIR=/home/mike/downloads # 定义工具链源码所存放位置。
RESULT_TOP=/opt/crosstool          # 定义工具链的安装目录
export TARBALLS_DIR RESULT_TOP
GCC_LANGUAGES="c,c++"              # 定义支持 C, C++语言
export GCC_LANGUAGES
# 创建/opt/crosstool 目录
mkdir -p $RESULT_TOP
# 编译工具链，该过程需要数小时完成。
eval `cat arm.dat gcc-3.3.6-glibc-2.3.2.dat` sh all.sh --notest
echo Done.
```



### 2.2.2.3 建立配置文件

在 arm.sh 脚本文件中需要注意 arm.dat 和 gcc-3.3.6-glibc-2.3.2.dat 两个文件，这两个文件是作为 crosstool 的编译的配置文件。其中 arm.dat 文件内容如下，主要用于定义配置文件，定译生成编译工具链的名称以及定义编译选项等。

```
KERNELCONFIG=`pwd`/arm.config # 内核的配置
TARGET=arm-linux- # 编译生成的工具链名称
TARGET_CFLAGS="-O" # 编译选项
```

gcc-3.3.6-glibc-2.3.2.dat 文件内容如下，该文件主要定义编译过程中所需要的库以及它定义的版本，如果当在编译过程中发现有些库不存在时，crosstool 会自动在相关网站上下载，该工具在这点上相对非常智能，也非常有用。

```
BINUTILS_DIR=binutils-2.15
GCC_DIR=gcc-3.3.6
GLIBC_DIR=glibc-2.3.2
GLIBCTHREADS_FILENAME=glibc-linuxthreads-2.3.2
LINUX_DIR=linux-2.6.10
LINUX_SANITIZED_HEADER_DIR=linux-libc-headers-2.6.12.0
```

### 2.2.2.4 执行脚本

将 Crosstool 的脚本文件和配置文件准备好之后，开始执行 arm.sh 脚本来编译交叉变异工具。具体执行命令如下：

```
# cd crosstool-0.42
# ./arm.sh
```

经过数小时的漫长编译之后，会在 /opt/crosstool 目录下生成新的交叉编译工具，其中包括以下内容：

arm-linux-addr2line	arm-linux-g++	arm-linux-ld	arm-linux-size
arm-linux-ar	arm-linux-gcc	arm-linux-nm	arm-linux-strings
arm-linux-as	arm-linux-gcc-3.3.6	arm-linux-objcopy	arm-linux-strip
arm-linux-c++	arm-linux-gccbug	arm-linux-objdump	fix-embedded-paths
arm-linux-c++filt	arm-linux-gcov	arm-linux-ranlib	
arm-linux-cpp	arm-linux-gprof	arm-linux-readelf	

### 2.2.2.5 添加环境变量

然后将生成的编译工具链路径添加到环境变量 PATH 上去，添加的方法是在系统 /etc/bashrc 文件中添加下面一行在文件的最后，如图 2.2 所示。

```
export PATH=/opt/crosstool/gcc-3.3.6-glibc-2.3.2/arm-linux/bin:$PATH
```

```
root@localhost:/etc
File Edit View Terminal Go Help

;;
screen)
    if [ -e /etc/sysconfig/bash-prompt-screen ]; then
        PROMPT_COMMAND=/etc/sysconfig/bash-prompt-screen
    else
        PROMPT_COMMAND='echo -ne "\033_${USER}@${HOSTNAME%.*}:${PWD/#$HOME/~}\033\\ "'
    fi
    ;;
*)
    [ -e /etc/sysconfig/bash-prompt-default ] && PROMPT_COMMAND=/etc/sysconfig/bash-prompt-de
fault
    ;;
esac
# Turn on checkwinsize
shopt -s checkwinsize
[ "$PS1" = "\s-\v\\\$ " ] && PS1="[u@h \W]\\$ "

if [ "$SHLVL" != "1" ]; then # We're not a login shell
    for i in /etc/profile.d/*.sh; do
        if [ -r "$i" ]; then
            . $i
        fi
    done
fi
fi
# vim:ts=4:sw=4
export PATH=/opt/crostooll/gcc-3.3.6-glibc-2.3.2/arm-linux/bin:$PATH
"bashrc" 57L, 1566C 57,1 Bot
```

图 2.2 用 Vi 编辑器在 bashrc 文件中添加环境变量

设置完环境变量，也就意味着交叉编译工具链已经构建完成，然后就可以用 2.2.1.8 节中的方法进行测试刚刚建立的工具链，此处就不用再赘述。

## 2.3 本章小节

本章讲述的内容非常有实用价值，因为交叉工具链的构建是嵌入式系统开发必不可少的一部分，也是嵌入式系统开发的基础。本章首先对交叉工具链进行了大体地介绍，然后分别介绍两种构建交叉工具链的方法：分步构建法和 Crosstool 工具构建法。这两种构建交叉工具链的方法在实际应用中非常广泛，相信读者通过学习本章的内容可以构建一套自己的交叉编译工具链。下一章将介绍嵌入式系统的启动程序——Bootloader。

## 2.4 常见问题

1. 编译 boot-trap gcc 时出现如下图 2.3 错误，提示：crti.o: No such file: No such file or directory collect2: ld returned 1 exit status，为什么？

```

map -o libgcc_s.so.1 libgcc/./_udivsi3.o libgcc/./_divsi3.o libgcc/./_umodsi3.
o libgcc/./_modsi3.o libgcc/./_dvm_d_lnx.o libgcc/./_muldi3.o libgcc/./_negdi2.o
libgcc/./_lshrdi3.o libgcc/./_ashldi3.o libgcc/./_ashrdi3.o libgcc/./_ffsdi2.o l
ibgcc/./_clz.o libgcc/./_cmpdi2.o libgcc/./_ucmpdi2.o libgcc/./_floatdidf.o libg
cc/./_floatdisf.o libgcc/./_fixunsdfsi.o libgcc/./_fixunssfsi.o libgcc/./_fixuns
dfdi.o libgcc/./_fixdfdi.o libgcc/./_fixunssfdi.o libgcc/./_fixsfdi.o libgcc/./_
fixxfdi.o libgcc/./_fixunssfdi.o libgcc/./_floatdixf.o libgcc/./_fixunssfdi.o li
bgcc/./_fixtfdi.o libgcc/./_fixunstfdi.o libgcc/./_floatditf.o libgcc/./_clear_c
ache.o libgcc/./_trampoline.o libgcc/./_main.o libgcc/./_exit.o libgcc/./_absvs
i2.o libgcc/./_absvdi2.o libgcc/./_addvsi3.o libgcc/./_addvdi3.o libgcc/./_subvs
i3.o libgcc/./_subvdi3.o libgcc/./_mulvsi3.o libgcc/./_mulvdi3.o libgcc/./_negvs
i2.o libgcc/./_negvdi2.o libgcc/./_ctors.o libgcc/./_divdi3.o libgcc/./_moddi3.o
libgcc/./_udivdi3.o libgcc/./_umoddi3.o libgcc/./_udiv_w_sdiv.o libgcc/./_udivm
oddi4.o libgcc/./_unwind-dw2.o libgcc/./_unwind-dw2-fde-glibc.o libgcc/./_unwind-s
jlj.o libgcc/./_unwind-c.o -lc && rm -f libgcc_s.so && ln -s libgcc_s.so.1 libgcc
_s.so
/home/mike/armlinux/tools/arm-linux/bin/ld: crt1.o: No such file: No such file o
r directory
collect2: ld returned 1 exit status
make[2]: *** [libgcc_s.so] Error 1
make[2]: Leaving directory `/home/mike/armlinux/build-tools/gcc-3.3.6/gcc'
make[1]: *** [libgcc.a] Error 2
make[1]: Leaving directory `/home/mike/armlinux/build-tools/gcc-3.3.6/gcc'
make: *** [all-gcc] Error 2
[root@localhost gcc-3.3.6]# _

```

图 2.3: gcc 工具编译出错界面

参考答案: 由于在配置时没有选择 `--disable-shared` 选项, 该选项的意思是只编译静态库。默认选项为 `--enable-shared`, 而 `libf2c` 和 `libiberty` 不支持共享库。

## 2. Glibc 里面静态库和共享库有什么区别?

参考答案: 应用程序在链接静态库时, 会把引用到的数据和代码放到生成的可执行文件中, 程序运行时就不再需要库了。应用程序链接共享库时, 连接器不会把引用到的数据和代码放到可执行文件中, 而仅仅做一个标记, 当程序运行时, 系统会去加载相应的共享库。链接共享库时, 可执行文件的大小会小一些, 但运行时依赖于共享库。起动态态库和共享库的方法分别是在配置时用 `--disable-shared` 和 `--enable-shared` 选项。

## 3. 本地编译器与交叉编译器的作用?

参考答案: 编译器可以生成用来在与编译器本身所在的计算机和操作系统(平台)相同的环境下运行的目标代码, 这种编译器叫做本地编译器。另外, 编译器也可以生成用来在其它平台上运行的目标代码, 这种编译器又叫做交叉编译器。交叉编译器在生成新的硬件平台时非常有用。

## 第 3 章 嵌入式系统的 BootLoader

本章学习目标：

- | 了解 BootLoader 的作用
- | 熟悉常见的嵌入式 Linux BootLoader
- | 熟悉 S3C2410 开发板
- | 学会基于嵌入式系统的 U-Boot 移植
- | 学会自己编写 BootLoader

### 3.1 BootLoader 概述

一个嵌入式 Linux 系统从软件的角度看通常分为四个层次：引导加载程序、Linux 内核、文件系统、用户应用程序。

引导加载程序，是系统加电后运行的第一段代码。大家熟悉的 PC 中的引导程序一般由 BIOS 和位于 MBR 的操作系统 BootLoader（例如 LILO 或者 GRUB）一起组成。然而在嵌入式系统中通常没有像 BIOS 那样的固件程序，因此整个系统的加载启动任务就完全由 BootLoader 来完成。在嵌入式 Linux 中，引导加载程序即等效为 BootLoader。简单地说，BootLoader 就是在操作系统内核运行前执行的一段小程序。通过这段小程序，我们可以初始化必要的硬件设备，创建内核需要的一些信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，真正起到引导和加载内核的作用。

BootLoader 是依赖于硬件而实现的，特别是在嵌入式系统中。不同体系结构需求的 BootLoader 是不同的；除了体系结构，BootLoader 还依赖于具体的嵌入式板级设备的配置。也就是说，对于两块不同的嵌入式板而言，即使它们基于相同的 CPU 构建，运行在其中一块电路板上的 BootLoader，未必能够运行在另一块电路开发板上。

Bootloader 的启动过程可以是单阶段的，也可以是多阶段的。大多数单阶段的 BootLoader 应用于简单的系统，比如没有操作系统的系统。通常多阶段的 BootLoader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 BootLoader 大多数是两阶段的启动过程，也就是启动过程可以分为 stage 1 和 stage 2 两部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。而 stage2 则通常用 C 语言来实现，这样可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。

大多数 BootLoader 都包含两种不同的操作模式。启动加载（Boot loading）模式和下载（Down loading）模式，这种区别仅对于开发人员才有意义。但从最终用户的角度看，BootLoader 的作用就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载模式：这种模式也称为自主（Autonomous）模式，即 BootLoader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是 BootLoader 的正常工作模式。因此在嵌入式产品发布的时候，BootLoader 显然必须工作在这种模式下。

下载模式：在这种模式下目标机上的 BootLoader 将通过串口连接或网络连接等通信手

段从主机上下载文件，比如：下载应用程序、数据文件、内核映像等。从主机下载的文件通常首先被 BootLoader 保存到目标机的 RAM 中然后再被 BootLoader 写到目标机上的固态存储设备中。BootLoader 的这种模式通常在系统更新时使用。工作于这种模式下的 BootLoader 通常都会向它的终端用户提供一个简单的命令行接口。比如 U-Boot, Blob, vivi 等。

## 3.2 常用的嵌入式 Linux BootLoader

从上一节的内容可以了解到 BootLoader 是嵌入式系统中非常重要的一部分，也是系统运行工作的必要组成部分。在嵌入式系统中常见的 BootLoader 有以下几种：

### 3.2.1 U-Boot

U-Boot 是德国 DENX 小组的开发用于多种嵌入式 CPU 的 BootLoader 程序，它可以运行在基于 PowerPC, ARM, MIPS 等多种嵌入式开发板上。从 <http://u-boot.sourceforge.net/> 或 <ftp://ftp.denx.de/pub/u-boot/> 站点都可以下载 U-Boot 的源代码。U-Boot 源代码主要目录解释如下：

- board: 目标板相关文件，主要包含 SDRAM、FLASH 驱动；
- common: 独立于处理器体系结构的通用代码，如内存大小探测与故障检测；
- cpu: 与处理器相关的文件。如 mpc8xx 子目录下含串口、网口、LCD 驱动及中断初始化等文件；
- driver: 通用设备驱动，如 CFI FLASH 驱动（目前对 INTEL FLASH 支持较好）；
- doc: U-Boot 的说明文档；
- examples: 可在 U-Boot 下运行的示例程序；如 hello\_world.c, timer.c；
- include: U-Boot 头文件；尤其 configs 子目录下与目标板相关的配置头文件是移植过程中经常要修改的文件；
- lib\_XXX: 处理器体系结构的文件，如 lib\_ppc, lib\_arm 目录分别包含与 PowerPC、ARM 体系结构相关的文件；
- net: 与网络功能相关的文件目录，如 bootp, nfs, tftp；
- post: 上电自检文件目录。尚有待于进一步完善；
- rtc: RTC（Real Time Clock，实时时钟）驱动程序；
- tools: 用于创建 U-Boot S-RECORD 和 BIN 镜像文件的工具。

### 3.2.2 VIVI

VIVI 是由韩国 MIZI 公司开发的专门用于 ARM 产品线的一种 BootLoader。因为 VIVI 目前只支持使用串口和主机通信，所以您必须使用一条串口电缆来连接目标板和主机。VIVI 的源代码下载地址为：<http://www.mizi.com/developer/s3c2410x/download/vivi.html>。VIVI 一般有如下作用：

- 1)、把内核(kernel)从 flash 复制到 RAM，然后启动它
- 2)、初始化硬件
- 3)、下载程序并写入 flash
- 4)、检测目标板

vivi.tar.bz2 源代码包解压后的目录结构如下所示:

```
# tree -L 1
.
|-- COPYING
|-- CVS
|-- Documentation
|-- Makefile
|-- Rules.make
|-- arch
|-- drivers
|-- include
|-- init
|-- lib
|-- scripts
|-- test
`-- util

10 directories, 3 files
```

其中 VIVI 主要目录介绍:

CVS: 存放 CVS 工具相关的文件

Documentation: 存放一些使用 VIVI 的帮助文档

arch: 存放一些平台相关的代码文件

drivers: 存放 VIVI 相关的驱动代码

include: 存放所有 VIVI 源码的头文件

init: 存放 VIVI 初始化代码

lib: 存放 VIVI 实现的库函数文件

scripts: 存放 VIVI 脚本配置文件

test: 存放一些测试代码文件

util: 存放一些 Nand Flash 写 Image 相关的实用文件。

### 3.2.3 Blob

Blob 是 Boot Loader Object 的缩写, 是一款功能强大的 BootLoader。其源码在 <http://sourceforge.net/projects/blob> 可以获取。Blob 最初是由 Jan-Derk Bakker 和 Erik Mouw 两人一块名为 LART (Linux Advanced Radio Terminal) 的板子写的, 该板使用的处理器是 StrongARM SA-1100, 现在 Blob 已经被成功地移植到许多基于 ARM 的 CPU 上。

### 3.2.4 RedBoot

RedBoot 是一个专门为嵌入式系统定制的引导启动工具, 最初由 Redhat 开发, 它是基于 eCos (Embedded Configurable Operating System) 的硬件抽象层, 同时它继承了 eCos 的高可靠性, 简洁性, 可配置性和可移植性等特点。RedBoot 集 Bootloader、调试、Flash 烧写于一体。支持串口、网络下载, 执行嵌入式应用程序。既可以用在产品的开发阶段 (调试功能),

也可以用在最终的产品上（Flash 更新、网络启动）。RedBoot 支持下载和调试应用程序，开发板可以通过 BOOTP/DHCP 协议动态配置 IP 地址，支持跨网段访问。用户可以通过 tftp 协议下载应用程序和 image。或者通过串口用 x-modem/y-modem 下载。Redboot 支持用 GDB(the GNU debugger)通过串口或者网卡调试嵌入式程序。可对 gcc 编译的程序进行源代码级的调试。相比于简易 jtag 调试器，可靠、高速（CPU 的 Cache 打开后，通过网卡 tftp 下在能达到 1M bytes，GDB 下载的速度能达到 2M bps）、稳定。用户可通过串口或网卡，以命令行的形式管理 Flash 上的 image，下载 image 到 flash。动态配置 RedBoot 启动的各种参数、启动脚本。上电后 Redboot 可自动从 flash 或 tftp 服务器上下载应用程序执行。RedBoot 在 <http://sourceware.org/redboot> 站点可以下载其源码，同时可以了解更多地关于 RedBoot 详细信息，它在嵌入式系统应用中也非常广泛。

### 3.2.5 ARMboot

ARMboot 是一个基于 ARM 或 StrongARM 为内核 CPU 的嵌入式系统 BootLoader 固件程序。该软件的主要目标是使新的平台更容易被移植并且尽可能发挥其强大性能。它只基于 ARM 固件，但是它支持多种类型启动，比如 flash，网络下载通过 bootp，dhcp，tftp 等。它也是开源项目，可以从 <http://www.sourceforge.net/projects/armboot> 网站可以获得最新的 ARMboot 源码和详细资料。它在 ARM 处理器方面应用非常广泛。

### 3.2.6 DIY

DIY(Do It Yourself)，即自己制作。以上 U-Boot，VIVI，Blob，RedBoot 和 ARMboot 等成熟工具移植起来简单快捷，但同时他们都存在着一定的局限性，首先是因为它们是面向大部分硬件的工具，所以说在功能上要满足大部分硬件的需求，但一般情况下我们只需要特定的开发板相关的实现代码，其他型号开发板的实现代码对它来说是没有用的，所以通常它们的代码量较大。其次它们在使用上不够灵活，比如在这些工具上添加自己的特有功能相对比较困难，因为你必须熟悉该代码的组织关系，以及了解它的配置编译等文件。所以用 DIY 的方式自己编写针对目标板的 BootLoader 不但代码量短小，同时灵活性很大，最重要的是将来好维护。所以在实际嵌入式产品开发时大都选择 DIY 的方式编写 BootLoader。

## 3.3 基于 S3C2410 开发板的 BootLoader 实现

本节将以实例讲述基于 S3C2410 开发板的 BootLoader 的具体实现，主要分两个方面进行介绍，一是介绍基于 U-Boot 的移植，二是介绍 DIY 方式开发 BootLoader。要移植或开发 BootLoader 首先要清楚具体的硬件系统，在这里就是要了解我们使用的目标板——S3C2410 开发板。

### 3.3.1 S3C2410 开发板介绍

本书中所设计的开发板相关的实例都是基于 S3C2410 开发板设计和测试的。S3C2410 开发板是非常通用的一款 ARM 9 开发板，读者使用任何类型的 ARM 9 开发板都能参考书中的实例。对于 S3C2410 开发板基本配置如下：

## Ø CPU

采用三星的 S3C2410 ARM920T, 主频 203MHz。集成有 SDRAM 内存控制器、NAND Flash 控制器、SD 卡控制器、USB Host、USB Device 控制器、LCD 控制器、IIC 总线控制器、IIS 控制器、SPI 接口等多种接口。

## Ø 存储器

64Mbyte 的 SDRAM

64Mbyte 的 NAND flash\* (注 1)

## Ø 以太网控制器

10M 网口, CS8900Q3, 带联接和传输指示灯

## Ø 串行接口

系统提供两个串行收发 DB9 母口连接器, 上面分别表示 COM0、COM1

## Ø USB Host 接口

两个 USB1.1HOST 接口

一个 USB 1.1Device 接口

## Ø 存储接口

一个 SD 卡接口

一个十针的 AD 接口

一个 IDE 接口

## Ø LCD 和触摸屏接口

一个 50 芯 LCD 接口引出了 LCD 控制器和触摸屏的全部信号

TFT 真彩 LCD 接口

提供真彩 LCD 的接口, LCD 模块不需要外接电源等, 插入该接口直接可以使用。接口另外还带触摸屏的接口

## Ø 调试及下载接口

20 针 Multi-ICE 标准 JTAG 接口, 支持 SGT2.51 和 ADS1.2 调试

## Ø 音频接口

采用 IIS 接口芯片 UDA1341, 一路立体声音频输出接口可接耳机或音箱; 支持录音, 板子自带主机话筒可直接录音, 另有一路话筒输入接口可接麦克风

## Ø 电源接口

5V 电源供电, 带电源开关和指示灯

## Ø 操作系统

支持 Linux 2.4 或以上系统, 支持 WinCE4.2.net

开发板上包括 1 片 64M × 8 位数据宽度的 NAND Flash(K9F1208)。和 2 片 16M × 16 位数据宽度的 SDRAM, 地址范围为 0x30000000~0x34000000。S3C2410 将系统的存储空间分为 8 组 (Bank), 每组大小为 128MB, 共 1GB。Bank0 到 Bank5 之间的开始地址是固定的, 用于 ROM 或 SRAM。Bank6 和 Bank7 用于 ROM, SRAM 或 SDRAM, 这两个组是可编程且大小相同。S3C2410 具有三种启动方式, 通过 OM[1:0]管脚进行选择:

OM[1:0] = 00 时, 处理器通过 NAND Flash 启动

OM[1:0] = 01 时, 处理器通过 16 位宽的 ROM 启动

OM[1:0] = 10 时, 处理器通过 32 位宽的 ROM 启动

\*注 1: NOR Flash 和 NAND Flash 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR flash 技术, 彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着, 1989 年, 东芝公司发表了 NAND flash 结构, 强调降低每比特的成本, 更高的性能, 并且象磁盘一样可以通过接口轻松升级。NOR Flash 的特点是芯片内执行(XIP, eXecute In Place), 这样应用程序可以直接在 flash 闪存内运行, 不必再把代



码读到系统 RAM 中。NOR 的传输效率很高，在 1~4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 flash 的管理和需要特殊的系统接口。通常 NOR 的读速度比 NAND 稍快一些，而 NAND 的写入速度比 NOR 快很多。所以在设计中应该考虑这些情况。

由于 NAND Flash 容量大，比 Nor Flash 便宜等优势，所以经常选择 NAND Flash 启动。当从 Nor Flash 启动时，要把 flash 芯片的首地址映射到 0x00000000 位置，系统启动后，启动程序本身把自己从 flash 搬运到 RAM 中去。当从 NAND Flash 启动时，S3C2410 会自动把 NAND Flash 的前 4K 数据搬到自己内部的 RAM 中去，并把内部 RAM 的首地址设为 0x00000000，CPU 从 0x00000000 地址开始运行。本章选择的实现启动方式就是通过 NAND Flash 启动。图 3.1 所示为通过 Nor Flash 启动和 NAND Flash 启动两种方式存储空间分配。其中图 a) 是 nGCS0 片选的 Nor Flash 启动模式存储分配图，图 b) 是 NAND Flash 启动模式的存储分配图。其中 SFR 为 Special Function Register 的缩写，即特殊功能寄存器。

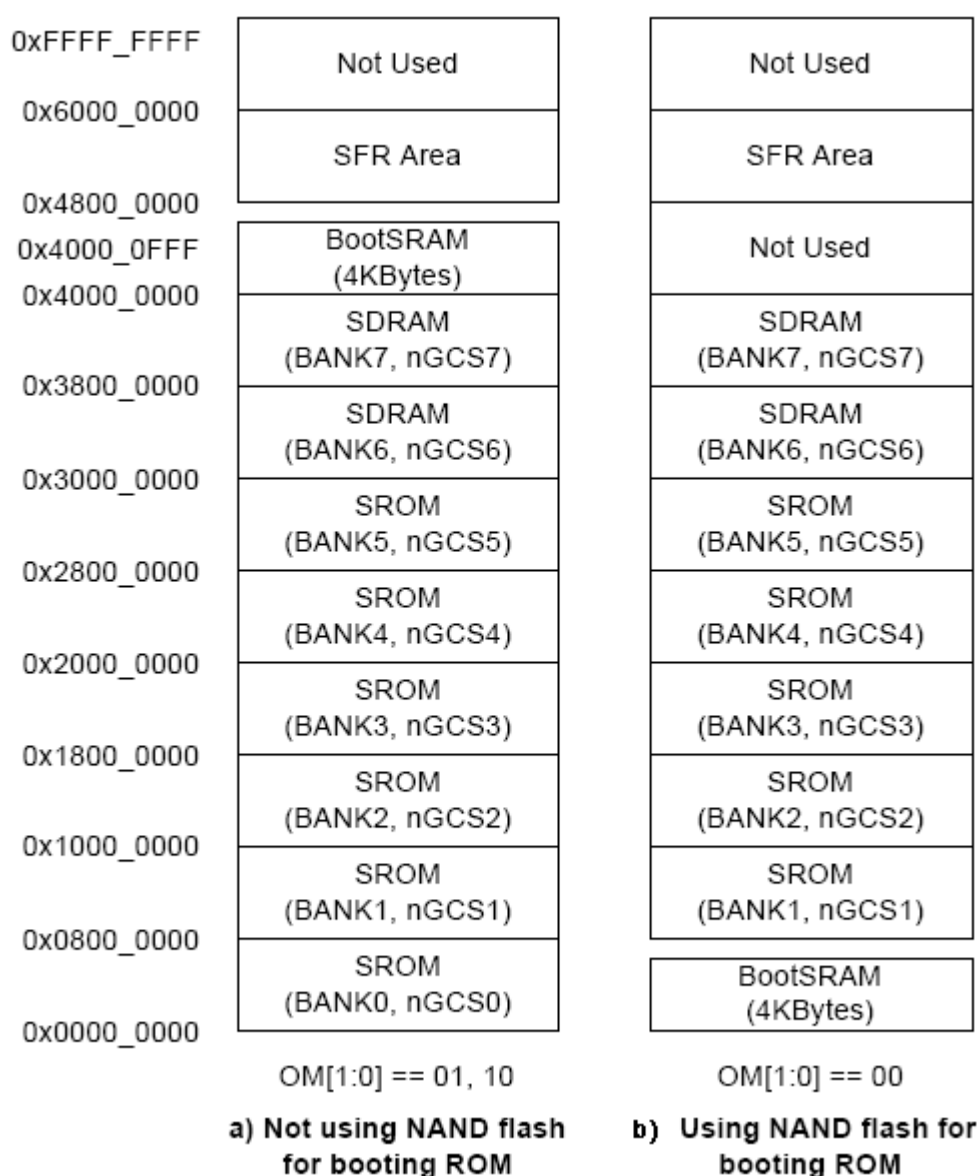


图 3.1 NAND Flash 的内存映射

### 3.3.2 U-Boot 分析与移植

本章以应用非常广泛的 U-Boot 为例讲述基于 S3C2410 开发板的 BootLoader 分析与移植。解压 u-boot-1.1.6.tar.bz2 包，查看其目录结构如下所示：

```
# tree -L 1 -d
.
|-- board
|-- common
|-- cpu
|-- disk
|-- doc
|-- drivers
|-- dtb
|-- examples
|-- fs
|-- include
|-- lib_arm
|-- lib_avr32
|-- lib_blackfin
|-- lib_generic
|-- lib_i386
|-- lib_m68k
|-- lib_microblaze
|-- lib_mips
|-- lib_nios
|-- lib_nios2
|-- lib_ppc
|-- nand_spl
|-- net
|-- post
|-- rtc
`-- tools

26 directories
```

大多数 BootLoader 都包含“启动加载”模式和“下载”模式，U-Boot 作为一款强大的 BootLoader 也支持这两种工作模式，并且常常允许用户在这两种模式之间切换。同时 U-Boot 也分为 Stage1 和 Stage2 两个阶段。其中依赖于 CPU 体系结构的代码通常都放在 Stage1 里，并且通常用汇编语言实现。Stage2 通常用 C 语言实现，可以实现更复杂的功能，并且有更好的移植性和可读性。

#### 3.3.2.1 U-Boot Stage1 分析

U-Boot 的 Stage1 通常是在 start.S 文件中实现，并且都是用汇编语言编写。一个可执行

性 image 文件必须有一个入口点，并且只能有一个全局入口点，通常这个入口点的地址放在 ROM (Flash) 0x0 位置，因此必须使编译器知道这个入口地址，该过程通常修改连接器的脚本文件来完成。此处以三星的 smdk2410 开发板为例，该开发板的 U-Boot 实现代码已经包含在里面。打开 board/smdk2410/ u-boot.lds 文件，该脚本文件的内容如下所示：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
    . = ALIGN(4);
    .text      :
    {
        cpu/arm920t/start.o  (.text)
        *(.text)
    }
    . = ALIGN(4);
    .rodata : { *(.rodata) }
    . = ALIGN(4);
    .data : { *(.data) }
    . = ALIGN(4);
    .got : { *(.got) }
    . = .;
    __u_boot_cmd_start = .;
    .u_boot_cmd : { *(.u_boot_cmd) }
    __u_boot_cmd_end = .;
    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) }
    _end = .;
}
```

其中，ENTRY(\_start)定义了入口点在 cpu/arm920t/start.S 文件，入口地址为 0x00000000。在 cpu/arm920t/config.mk 文件中定义了代码区基地址：TEXT\_BASE = 0x33F80000。接下来分析 U-Boot 的 Stage1 的核心文件 start.S。

#### Ø 设置异常向量表

对于 ARM 处理器一般包括复位、未定义指令、SWI、预取终止、数据终止、IRQ、FIQ 等异常，关于 ARM 处理器这些异常在后面会有专门的介绍，其中 U-Boot 中关于异常向量的定义如下，当发生异常时执行 cpu/arm920t/ interrupts.c 文件。

```
.globl _start
_start:    b        reset
    ldr    pc, _undefined_instruction
    ldr    pc, _software_interrupt
    ldr    pc, _prefetch_abort
    ldr    pc, _data_abort
```

```
ldr pc, _not_used
ldr pc, _irq
ldr pc, _fiq
```

#### Ø 设置 CPU 模式为 SVC 模式

Reset，即复位，在系统中经常会用到，该操作是异常处理的第一个操作，其主要目的是设置 CPU 模式为 SVC 模式。在此有必要介绍一下 ARM 处理器的 7 种工作模式：

- ┆ 用户模式 (usr)：ARM 处理器正常的程序执行状态
- ┆ 快速中断模式 (fiq)：用于高速数据传输或通道处理
- ┆ 外部中断模式 (irq)：用于通用的中断处理
- ┆ 管理模式 (svc)：操作系统使用的保护模式
- ┆ 数据访问终止模式(abt)：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- ┆ 系统模式 (sys)：运行具有特权的操作系统任务。
- ┆ 未定义指令中止模式 (und)：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM 微处理器共有 37 个 32 位寄存器，其中 31 个为通用寄存器，6 个为状态寄存器。但是这些寄存器不能被同时访问，具体哪些寄存器是可编程访问的，取决微处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器 R0~R14、程序计数器 PC、一个或两个状态寄存器都是可访问的。通用寄存器包括 R0~R15，可以分为三类：

- ┆ 未分组寄存器 R0~R7：在所有的运行模式下，未分组寄存器都指向同一个物理寄存器，他们未被系统用作特殊的用途，因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在进行程序设计时应引起注意。
- ┆ 分组寄存器 R8~R14：对于分组寄存器，他们每一次所访问的物理寄存器与处理器当前的运行模式有关。对于 R8~R12 来说，每个寄存器对应两个不同的物理寄存器，当使用 fiq 模式时，访问寄存器 R8\_fiq~R12\_fiq；当使用除 fiq 模式以外的其他模式时，访问寄存器 R8\_usr~R12\_usr。对于 R13、R14 来说，每个寄存器对应 6 个不同的物理寄存器，其中的一个是用户模式与系统模式共用，另外 5 个物理寄存器对应于其他 5 种不同的运行模式。
- ┆ 程序计数器 PC(R15)：在 ARM 状态下，位[1:0]为 0，位[31:2]用于保存 PC；在 Thumb 状态下，位[0]为 0，位[31:1]用于保存 PC；虽然可以用作通用寄存器，但是有一些指令在使用 R15 时有一些特殊限制，若不注意，执行的结果将是不可预料的。在 ARM 状态下，PC 的 0 和 1 位是 0，在 Thumb 状态下，PC 的 0 位是 0。注意，Thumb 状态下的寄存器集是 ARM 状态下寄存器集的一个子集，程序可以直接访问 8 个通用寄存器 (R7~R0)、程序计数器 (PC)、堆栈指针 (SP)、连接寄存器 (LR) 和 CPSR。同时，在每一种特权模式下都有一组 SP、LR 和 SPSR。

设置 CPU 模式为 SVC 模式操作的具体实现代码如下，其中 CPSR 是 Current Program Status Register 的缩写，即当前程序状态寄存器，寄存器 R16 用作 CPSR，CPSR 可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为 SPSR 是 Saved

Program Status Register 的缩写，即备份程序状态寄存器，当异常发生时，SPSR 用于保存 CPSR 的当前值，从异常退出时则可由 SPSR 来恢复 CPSR。关于 CPU 相关的寄存器设置需要参考 S3C2410 的用户手册来实现。

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

#### Ø 关闭看门狗

看门狗，即 watchdog timer，是一个定时器电路，一般有一个输入叫喂狗，一个输出到 MCU（Micro Controller Unit，多点控制单元）的 RST 端（复位端），MCU 正常工作的时候，每隔一段时间输出一个信号到喂狗端，给 WDT（watchdog timer 的简写）清零，如果超过规定的时间不喂狗，一般在程序跑飞时 WDT 定时超过，就会给出一个复位信号到 MCU，然后 MCU 复位。看门狗的作用就是防止程序发生死循环，或者说程序跑飞。根据 S3C2410 的用户手册，关闭看门狗的具体实现如下：

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
    ldr    r0, =pWTCON
    mov    r1, #0x0
    str    r1, [r0]
```

#### Ø 禁止所有中断

在 SVC 模式下，不允许有任何中断发生，根据 S3C2410 的用户手册，通过设置相应的寄存器值的位来禁止中断，具体实现如下：

```
mov r1, #0xffffffff
ldr r0, =INTMSK
str r1, [r0]
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
    str r1, [r0]
# endif
```

#### Ø 设置 CPU 的频率

S3C2410 的用户手册推荐 FCLK:HCLK:PCLK = 1:2:4，其中 FCLK 默认是 120MHz，通常 FCLK 用于 CPU，HCLK 用于 AHB 总线，PCLK 用于 APB 总线。

```
ldr r0, =CLKDIVN
mov r1, #3
str r1, [r0]
```

#### Ø 设置 CP15 寄存器

CP15 是系统控制协处理器寄存器，用于连接在内存中的页表描述符，此外还用于决定对 MMU 的操作。设置 CP15 寄存器的目的是失效 ICache（指令 Cache）和 DCache（数据 Cache），然后禁止 MMU 和 Cache。

```
cpu_init_crit:
    /* 失效 I/D caches */
    mov r0, #0
    mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
    mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
```

```

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

```

#### Ø 配置内存控制寄存器

配置内存控制寄存器一般是和开发板紧密相关的，寄存器的具体值由开发板商或硬件工程师提供。如果你对总线周期和外围芯片非常熟悉，也可以自己定义。在 U-Boot 中的设置文件是 board/smdk2400/lowlevel\_init.S，该文件包含 lowlevel\_init 程序段用于内存控制配置。在 start.S 中的相关实现如下：

```

mov ip, lr
bl lowlevel_init
mov lr, ip
mov pc, lr

```

#### Ø 配置栈空间

配置代码段的开始地址，动态内存区长度，全局数据的大小以及分配 IRQ 和 FRQ 的栈空间。

```

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

```

#### Ø BSS 段清零

BSS (Block Started by Symbol 的简称) 段是可执行性文件中的一种数据段。通常 ARM 编译器生成的可执行性文件由两部分数据组成，分别是代码段和数据段。代码段又分为可执行代码段 (text) 和只读数据段 (rodata)。数据段又分为初始化数据段 (data) 和未初始化数据段 (bss)。清除 BSS 段的具体实现如下：

```

clear_bss:
    ldr r0, _bss_start    /* find start of bss segment */
    ldr r1, _bss_end    /* stop here */
    mov r2, #0x00000000    /* clear */
clbss_1:
    str r2, [r0]    /* clear loop... */
    add r0, r0, #4
    cmp r0, r1
    ble clbss_1

```

#### Ø 拷贝 NAND Flash 代码到 RAM

从 NAND Flash 把数据拷贝到 RAM，在 U-Boot 中没有相关的实现，这里可以参考 VIVI 源代码的实现，利用 copy\_myself 函数实现，具体实现将在 U-Boot 移植一节中详细介绍。

#### Ø 进入 C 代码

进入 C 代码的实现很简单，利用 `ldr` 指令实现到 C 代码地址的装载，具体实现如下：

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

到这里已经介绍完了 U-Boot Stage1 的所有主要实现，接下来讲述 U-Boot Stage2 实现的内容。

### 3.3.2.2 U-Boot Stage2 分析

Stage2 部分全是用 C 语言实现，可读性较强，并且可以实现较为复杂的功能。通过上节可以看出 Stage1 最后调用的函数名是 `start_armboot`，同时这个函数也就是 Stage2 的入口函数，其定义在 `lib_arm/board.c` 文件中，该文件主要实现的内容有：

#### Ø 定义初始化函数表

```
init_fnc_t *init_sequence[] = {
    cpu_init,      /* 基本 CPU 相关的设置*/
    board_init,    /*基本开发板相关的设置*/
    interrupt_init, /* 设置异常*/
    env_init,      /*初始化环境变量*/
    init_baudrate, /* 初始化波特率*/
    serial_init,   /* 串行通信的设置*/
    console_init_f, /*初始化控制台的 stage 1 */
    display_banner, /* 通知代码已经运行在何处*/
#ifdef CONFIG_DISPLAY_CPUINFO
    print_cpuinfo, /* 打印 CPU 相关的信息*/
#endif
#ifdef CONFIG_DISPLAY_BOARDINFO
    checkboard,   /* 现实开发板相关的信息*/
#endif
    dram_init,    /* 配置有效地 ram 组*/
    display_dram_config,
    NULL,
};
```

#### Ø 配置可用的 Flash 区

利用 `flash_init()` 函数来配置可用的 Flash 区

#### Ø 初始化内存分配

利用 `mem_malloc_init()` 函数来初始化内存分配

#### Ø NAND Flash 初始化

利用 `nand_init()` 函数初始化 NAND Flash

#### Ø 初始化环境变量

利用 `env_relocate()` 函数来初始化环境变量

#### Ø 初始化外围设备

利用 `devices_init()` 函数来初始化外围设备

#### Ø 使能中断

- 利用 `enable_interrupts()` 函数来使能中断
- Ø 初始化网卡  
利用 `cs8900_get_enetaddr()` 函数来初始化 cs8900 网卡
  - Ø 初始化 I2C 总线  
I2C 是飞利浦公司研发的一种片内通信总线技术，利用 `i2c_init()` 函数来初始化，具体实现在 `cpu/arm920t/s3c24x0/i2c.c` 文件中。
  - Ø 初始化 LCD  
利用 `drv_lcd_init()` 函数来实现 LCD 的初始化，具体实现在 `common/lcd.c` 文件中。
  - Ø 进入 U-Boot 的命令循环  
该过程进入 U-Boot 的命令循环，接受用户输入的命令，然后进行相应的工作。具体实现如下：

```
for (;;)
{
    main_loop(); /*具体实现参见 common/main.c 文件*/
}
```

上述讲述了 U-Boot 在 Stage2 主要所做的工作，当然上述给出的只是一般情况下的，可以根据具体的开发板相应的增加或减少实现代码。下面将介绍基于 S3C2410 开发板的 U-Boot 移植。

### 3.3.2.3 U-Boot 的移植过程

前面两节对 U-Boot 已经有了较为详细地介绍，本节开始讲述一个具体的 U-Boot 移植实例，该实例的硬件环境是基于前面所述的 S3C2410 开发板。

软件环境：

- l u-boot-1.1.3.tar.bz2（从 <http://sourceforge.net/projects/u-boot> 下载）
- l arm-linux-gcc 3.3.6（根据第二章内容自己构建或从 <ftp.arm.kernel.org.uk> 直接下载）

准备好软、硬件环境后就开始真正的 U-Boot 移植工作了，下面将一步一步介绍移植的具体过程。

#### 1. 修改 Makefile

首先给要建立的 S3C2410 开发板取名为 `mike2410`，因为 U-Boot 的移植过程需要它。修改 Makefile 的具体操作如下：

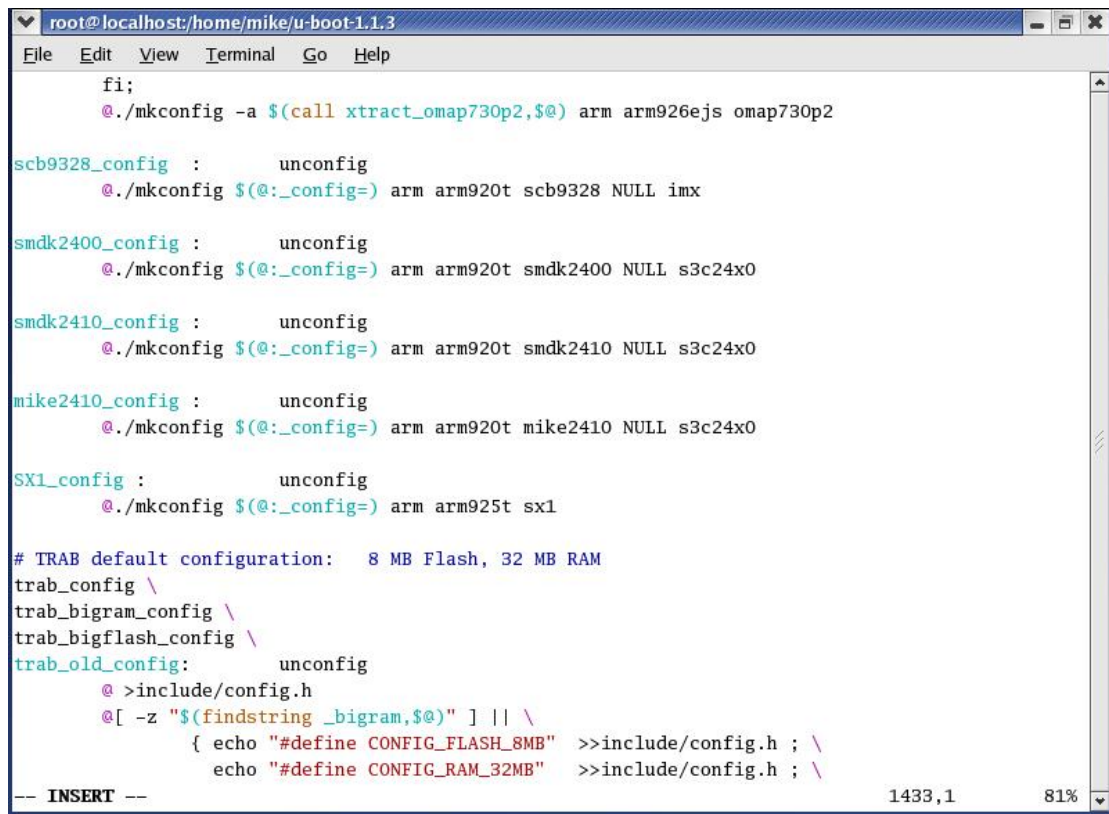
```
# tar -xvjf u-boot-1.1.3.tar.bz2
# cd u-boot-1.1.3
# vi Makefile
```

由于 `mike2410` 开发板和三星的 `smdk2410` 开发板很相似，所以在移植 U-Boot 时大部分源代码都以 `smdk2410` 为模版，然后在此基础上进行修改。此处修改 Makefile 如图 3.2 所示，在 Makefile 中添加了如下内容：

```
mike2410_config :   unconfig
@$(MKCONFIG) $(@:_config=) arm arm920t mike2400 NULL s3c24x0
```

其中，`mike2400_config:unconfig` 意思是为 `mike2410` 开发板建立一个编译项；第二行中 `arm` 的意思是 CPU 的架构是基于 ARM 体系的；`arm920t` 的意思是 CPU 的类型是 `arm920t`；`mike2410` 的意思是开发版的型号；`NULL` 的意思是开发者或经销商的名称为空；`s3c24x0` 的意思是基于 `s3c24x0` 的片上系统。





```
root@localhost:/home/mike/u-boot-1.1.3
File Edit View Terminal Go Help

fi;
@./mkconfig -a $(call xtract_omap730p2,$@) arm arm926ejs omap730p2

scb9328_config :      unconfig
@./mkconfig $@:_config= arm arm920t scb9328 NULL imx

smdk2400_config :      unconfig
@./mkconfig $@:_config= arm arm920t smdk2400 NULL s3c24x0

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

mike2410_config :      unconfig
@./mkconfig $@:_config= arm arm920t mike2410 NULL s3c24x0

SX1_config :          unconfig
@./mkconfig $@:_config= arm arm925t sx1

# TRAB default configuration:  8 MB Flash, 32 MB RAM
trab_config \
trab_bigram_config \
trab_bigflash_config \
trab_old_config:      unconfig
@ >include/config.h
@[ -z "$(findstring _bigram,$@)" ] || \
{ echo "#define CONFIG_FLASH_8MB" >>include/config.h ; \
  echo "#define CONFIG_RAM_32MB" >>include/config.h ; \

-- INSERT --
1433,1      81%
```

图 3.2: 修改 U-Boot 的 Makefile

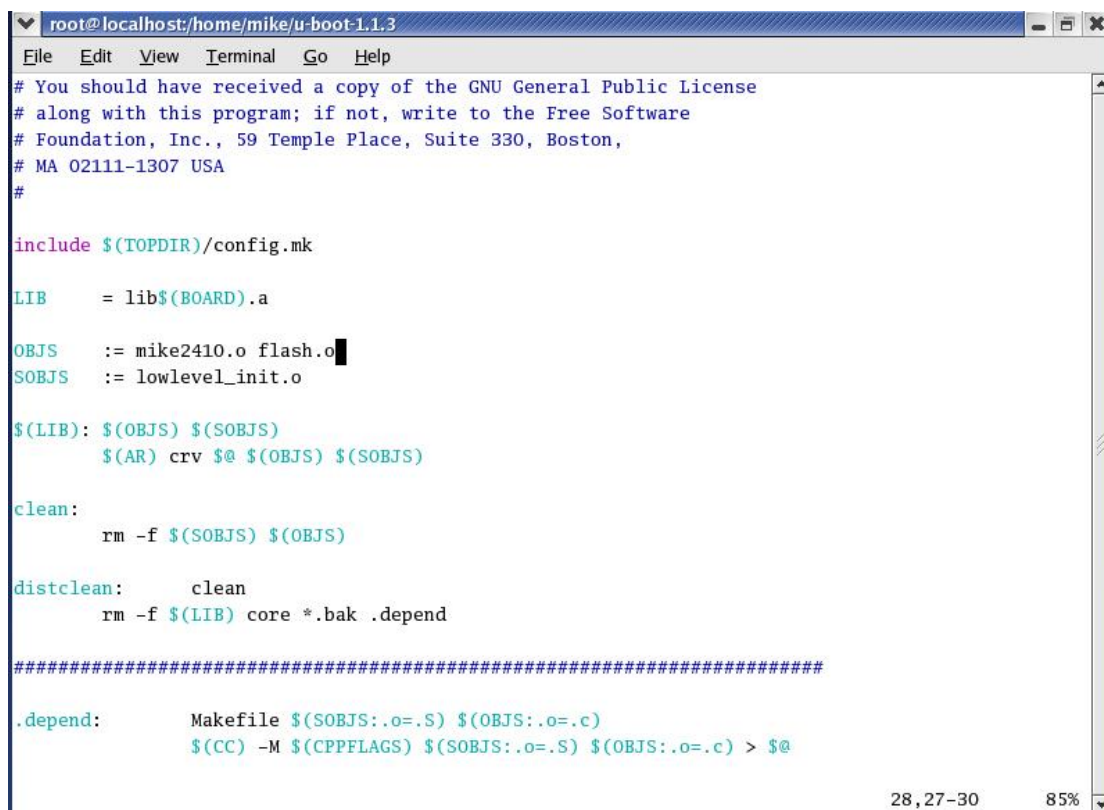
## 2. 建立 mike2410 开发板目录

在 board 目录下建立 mike2410 开发板子目录, 目前 board 目录下已经有两百多种针对不同开发板的子目录, 由于 mike2410 开发板最接近 smdk2410 开发板, 所以可以用下面的简单方法建立 mike2410 开发板目录:

```
# cp -fr board/ smdk2410 board/ mike2410
# cd board/ mike2410
# mv smdk2410.c mike2410.c
```

同时需要修改 board/mike2410/Makefile 文件, 修改如图 3.3 所示, 修改的内容是将 COBJS := smdk2410.o flash.o 改为 COBJS := mike2410.o flash.o

由于 board/mike2410 目录下的源代码文件为 mike2410.c, 所以编译中间文件为 mike2410.o, 如果没有修改这个 Makefile 文件编译就会出错。



```
root@localhost:/home/mike/u-boot-1.1.3
File Edit View Terminal Go Help
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston,
# MA 02111-1307 USA
#
include $(TOPDIR)/config.mk

LIB      = lib$(BOARD).a

OBJS     := mike2410.o flash.o
SOBJS    := lowlevel_init.o

$(LIB): $(OBJS) $(SOBJS)
         $(AR) crv $@ $(OBJS) $(SOBJS)

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

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

#####

.depend:        Makefile $(SOBJS:.o=.S) $(OBJS:.o=.c)
                 $(CC) -M $(CPPFLAGS) $(SOBJS:.o=.S) $(OBJS:.o=.c) > $@

28,27-30 85%
```

图 3.3 修改 mike2410 目录下的 Makefile 文件

### 3. 建立配置头文件

在 include/configs 目录下建立 mike2410.h 头文件，建立方法如下：

```
# cd include/configs
# cp -fr smdk2410.h mike2410.h
```

### 4. 指定交叉编译器的路径

在/etc/bashrc 文件中添加下面一行来指定交叉编译器的路径，这个路径不是绝对的，要根据交叉编译工具链所放的位置决定。

```
export PATH=/opt/crosstool/gcc-3.3.6-glibc-2.3.2/arm-linux/bin:$PATH
```

### 5. 测试编译

这个步骤的目的一是为了检查交叉编译器是否正常工作，二是为了测试新建的 mike2410 配置项是否能够正常编译，具体操作如下：

```
# cd u-boot-1.1.6
# make mike2410_config
# make CROSS_COMPILE=arm-linux-
```

如果编译正确，将在 u-boot-1.1.6 目录下生成 u-boot、u-boot.bin 和 u-boot.srec 三个映像文件。其中：u-boot 是 ELF 格式二进制的 image 文件，u-boot.bin 是原始的二进制 image 文件，u-boot.srec 是 Motorola S-Record 格式的 image 文件。到这一步说明建立好了 mike2410 的 U-Boot 编译项，但是具体的实现部分还需要修改，因为现在的实现代码还是完全和 smdk2410 开发板一样的，而不能工作在 mike2410 开发板上，接下来的步骤就是根据 mike2410 开发板的硬件配置和设计要求来进一步移植。

### 6. 修改 start.S 文件

首先在 ldr pc, \_start\_armboot 一行之前添加下面内容，由于 U-Boot 中没有支持从 NAND Flash 启动，所以将程序自己复制到 DRAM 里面去需要新加代码实现，一般通过

copy\_myself 函数来实现，其参考 VIVI 的 copy\_myself 代码。

```
#ifdef CONFIG_S3C2410_NAND_BOOT
    bl    copy_myself

    @ jump to ram
    ldr    r1, =on_the_ram
    add    pc, r1, #0
    nop
    nop
1:  b      1b          @ infinite loop
on_the_ram:
#endif
```

然后在\_start\_armboot: .word start\_armboot 一行之后添加下面内容，该部分的内容基本上都参考 VIVI 代码实现，这段代码的主要目的是搬运 NAND Flash 数据到 DRAM 里面。

```
#ifdef CONFIG_S3C2410_NAND_BOOT
copy_myself:
    mov r10, lr
    @ reset NAND
    mov r1, #NAND_CTL_BASE
    ldr    r2, =0xf830          @ initial value
    str    r2, [r1, #oNFCNF]
    ldr    r2, [r1, #oNFCNF]
    bic    r2, r2, #0x800        @ enable chip
    str    r2, [r1, #oNFCNF]
    mov r2, #0xff              @ RESET command
    strb r2, [r1, #oNFCMD]
    mov r3, #0                  @ wait

1:  add    r3, r3, #0x1
    cmp r3, #0xa
    blt    1b
2:  ldr    r2, [r1, #oNFSTAT]    @ wait ready
    tst    r2, #0x1
    beq    2b
    ldr    r2, [r1, #oNFCNF]
    orr    r2, r2, #0x800        @ disable chip
    str    r2, [r1, #oNFCNF]

    @ get read to call C functions (for nand_read())
    ldr    sp, DW_STACK_START    @ setup stack pointer
    mov fp, #0                  @ no previous frame, so fp=0

    @ copy vivi to RAM
    ldr    r0, =UBOOT_RAM_BASE
```

```

        mov     r1, #0x0
mov r2, #0x30000
        bl      nand_read_ll
tst     r0, #0x0
        beq     ok_nand_read

#ifdef CONFIG_DEBUG_LL
bad_nand_read:
        ldr     r0, STR_FAIL
        ldr     r1, SerBase
        bl      PrintWord
1:      b       1b          @ infinite loop
#endif

ok_nand_read:
#ifdef CONFIG_DEBUG_LL
        ldr     r0, STR_OK
        ldr     r1, SerBase
        bl      PrintWord
#endif

@ verify
        mov     r0, #0
        ldr     r1, =UBOOT_RAM_BASE
        mov     r2, #0x400    @ 4 bytes * 1024 = 4K-bytes
go_next:
        ldr     r3, [r0], #4
        ldr     r4, [r1], #4
        teq     r3, r4
        bne     notmatch
        subs    r2, r2, #4
        beq     done_nand_read
        bne     go_next

notmatch:
#ifdef CONFIG_DEBUG_LL
        sub     r0, r0, #4
        ldr     r1, SerBase
        bl      PrintHexWord
        ldr     r0, STR_FAIL
        ldr     r1, SerBase
        bl      PrintWord
#endif
1:      b       1b

```

```

done_nand_read:
#ifdef CONFIG_DEBUG_LL
    ldr    r0, STR_OK
    ldr    r1, SerBase
    bl     PrintWord
#endif

    mov pc, r10
@ clear memory
@ r0: start address
@ r1: length
mem_clear:
    mov r2, #0
    mov r3, r2
    mov r4, r2
    mov r5, r2
    mov r6, r2
    mov r7, r2
    mov r8, r2
    mov r9, r2

clear_loop:
    stmia   r0!, {r2-r9}
    subs r1, r1, #(8 * 4)
    bne     clear_loop
    mov pc, lr

#endif @ CONFIG_S3C2410_NAND_BOOT

```

接着在 start.S 文件最后添加下面几行的内容，用于定义栈地址变量。到这里 start.S 文件已经修改完毕，接着添加 NAND Flash 读函数。

```

#ifdef CONFIG_S3C2410_NAND_BOOT
    .align    2
    DW_STACK_START:
    .word     STACK_BASE+STACK_SIZE-4
#endif

```

## 7. 添加 nand\_read.c 和 nandflash.h 源文件

在 start.S 文件中调用了 nand\_read\_ll 函数，该函数用于 NAND Flash 读操作，在 U-Boot 中没有定义，需要新加该函数的实现，该函数的实现可以参考 VIVI 源代码，在 board/mike2410 目录下新建 nand\_read.c 源文件，文件内容下：

```

#include <config.h>

#define __REGb(x)    (*(volatile unsigned char *)(x))
#define __REGi(x)    (*(volatile unsigned int *)(x))
#define NF_BASE      0x4e000000
#define NFCONF       __REGi(NF_BASE + 0x0)

```

```

#define NFCMD      __REGb(NF_BASE + 0x4)
#define NFADDR     __REGb(NF_BASE + 0x8)
#define NFDATA     __REGb(NF_BASE + 0xc)
#define NFSTAT     __REGb(NF_BASE + 0x10)

#define BUSY 1
inline void wait_idle(void) {
    int i;
    while(!(NFSTAT & BUSY))
        for(i=0; i<10; i++);
}

#define NAND_SECTOR_SIZE    512
#define NAND_BLOCK_MASK    (NAND_SECTOR_SIZE - 1)

/* low level nand read function */
int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
        return -1;    /* invalid alignment */
    }
    /* chip Enable */
    NFCONF &= ~0x800;
    for(i=0; i<10; i++);

    for(i=start_addr; i < (start_addr + size);) {
        /* READ0 */
        NFCMD = 0;

        /* Write Address */
        NFADDR = i & 0xff;
        NFADDR = (i >> 9) & 0xff;
        NFADDR = (i >> 17) & 0xff;
        NFADDR = (i >> 25) & 0xff;

        wait_idle();

        for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
            *buf = (NFDATA & 0xff);
            buf++;
        }
    }
    /* chip Disable */

```

```
NFCONF |= 0x800;    /* chip disable */  
return 0;  
}
```

新建完 `nand_read.c` 文件后，需要修改相同目录下的 `Makefile` 文件，否则编译会出错，修改内容为：

将 `COBJS` := `mike2410.o flash.o`

改为：`COBJS` := `mike2410.o flash.o nand_read.o`

同时需要在 `board/mike2410` 目录下添加 `nandflash.h` 文件，该文件主要定义了 NAND Flash 的一些芯片配置函数，具体代码如下：

```
#include <s3c2410.h>  
  
#if (CONFIG_COMMANDS & CFG_CMD_NAND)  
typedef enum {  
    NFCE_LOW,  
    NFCE_HIGH  
} NFCE_STATE;  
  
static inline void NF_Conf(u16 conf)  
{  
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();  
    nand->NFCONF = conf;  
}  
  
static inline void NF_Cmd(u8 cmd)  
{  
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();  
    nand->NFCMD = cmd;  
}  
  
static inline void NF_CmdW(u8 cmd)  
{  
    NF_Cmd(cmd);  
    udelay(1);  
}  
  
static inline void NF_Addr(u8 addr)  
{  
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();  
    nand->NFADDR = addr;  
}  
  
static inline void NF_SetCE(NFCE_STATE s)  
{  
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();  
    nand->NFCE = s;  
}
```

```

switch (s) {
    case NFCE_LOW:
        nand->NFCONF &= ~(1<<11);
        break;
    case NFCE_HIGH:
        nand->NFCONF |= (1<<11);
        break;
}

static inline void NF_WaitRB(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    while (!(nand->NFSTAT & (1<<0)));
}

static inline void NF_Write(u8 data)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFDATA = data;
}

static inline u8 NF_Read(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    return(nand->NFDATA);
}

static inline void NF_Init_ECC(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFCONF |= (1<<12);
}

static inline u32 NF_Read_ECC(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    return(nand->NFECC);
}

#endif

```

#### 8. 修改 mike2410.c 文件

修改这个文件的主要目的有两个，一是初始化 CPU 相关寄存器来支持 USB 主、从设备；二是初始化 NAND Flash 设备。添加代码如下：

```
#include "nandflash.h"    //添加该头文件
```



```

int board_init(void)
{
    .....
    /* 根据 S3C2410 用户手册，设置相应的寄存器值来支持 USB*/
    gpio->MISCCR |= (1<<3);
    gpio->MISCCR &= ~((1<<12)|(1<<13));
    .....
}
/* 添加以下代码实现 NAND flash 的初始化*/
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
extern ulong nand_probe(ulong physadr);
static inline void NF_Reset(void)
{
    int i;
    NF_SetCE(NFCE_LOW);
    NF_Cmd(0xFF);      /* reset command */
    for(i = 0; i < 10; i++); /* tWB = 100ns. */
    NF_WaitRB();        /* wait 200~500us; */
    NF_SetCE(NFCE_HIGH);
}

static inline void NF_Init(void)
{
    #if 0 /* a little bit too optimistic */
    #define TACLS    0
    #define TWRPH0   3
    #define TWRPH1   0
    #else
    #define TACLS    0
    #define TWRPH0   4
    #define TWRPH1   2
    #endif

    NF_Conf((1<<15)|(0<<14)|(0<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0));
    NF_Conf((1<<15)|(0<<14)|(0<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0));
}

void nand_init(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    NF_Init();
}

```

```

#ifdef DEBUG
    printf("NAND flash probing at 0x%.8IX\n", (ulong)nand);
#endif

    printf ("%4lu KB\n", nand_probe((ulong)nand) >> 10);
}

#endif

```

#### 9. 修改头文件 mike2410.h

修改 include/configs/ mike2410.h 文件, 在该文件中添加如下内容, 其中定义了栈的基地址和、栈的大小、RAM 的基地址以及定义 NAND Flash 设置参数等内容。

```

.....
#define CONFIG_CMDLINE_TAG    1
#define CONFIG_SETUP_MEMORY_TAGS 1
#define CONFIG_INITRD_TAG    1
.....
#define CONFIG_COMMANDS \
    (CONFIG_CMD_DFL    |\
     CFG_CMD_CACHE    |\
     CFG_CMD_ENV       |\
     CFG_CMD_PING      |\
     CFG_CMD_NAND      |\
     /*CFG_CMD_EEPROM */\
     /*CFG_CMD_I2C    */\
     CFG_CMD_REGINFO   |\
     CFG_CMD_ELF)

.....
/* Nandflash Boot */
#define CONFIG_S3C2410_NAND_BOOT 1
#define STACK_BASE    0x33ff8000
#define STACK_SIZE    0x8000
#define UBOOT_RAM_BASE    0x33f00000

/* NAND Flash Controller */
#define NAND_CTL_BASE    0x4E000000
#define bINT_CTL(Nb)    __REG(INT_CTL_BASE + (Nb))

/* Offset */
#define oNFCNF    0x00
#define oNFCMD    0x04
#define oNFADDR    0x08
#define oNFDATA    0x0c
#define oNFSTAT    0x10
#define oNFECC    0x14
/* 以下代码是定义 NAND flash 设置参数*/
#if (CONFIG_COMMANDS & CFG_CMD_NAND)

```

```

#define CFG_MAX_NAND_DEVICE 1    /* Max number of NAND devices    */
#define SECTORSIZE 512

#define ADDR_COLUMN 1
#define ADDR_PAGE 2
#define ADDR_COLUMN_PAGE 3

#define NAND_ChipID_UNKNOWN 0x00
#define NAND_MAX_FLOORS 1
#define NAND_MAX_CHIPS 1

#define NAND_WAIT_READY(nand) NF_WaitRB()

#define NAND_DISABLE_CE(nand) NF_SetCE(NFCE_HIGH)
#define NAND_ENABLE_CE(nand) NF_SetCE(NFCE_LOW)
#define WRITE_NAND_COMMAND(d, adr) NF_Cmd(d)
#define WRITE_NAND_COMMANDW(d, adr) NF_CmdW(d)
#define WRITE_NAND_ADDRESS(d, adr) NF_Addr(d)
#define WRITE_NAND(d, adr) NF_Write(d)
#define READ_NAND(adr) NF_Read()
/* the following functions are NOP's because S3C24X0 handles this in hardware */
#define NAND_CTL_CLRALE(nandptr)
#define NAND_CTL_SETALE(nandptr)
#define NAND_CTL_CLRCLE(nandptr)
#define NAND_CTL_SETCLE(nandptr)

#define CONFIG_MTD_NAND_VERIFY_WRITE 1
#define CONFIG_MTD_NAND_ECC_JFFS2 1

#endif    /* CONFIG_COMMANDS & CFG_CMD_NAND */

```

然后修改以下各宏的值，包括内存相关的地址、启动提示字母和网络的设置。

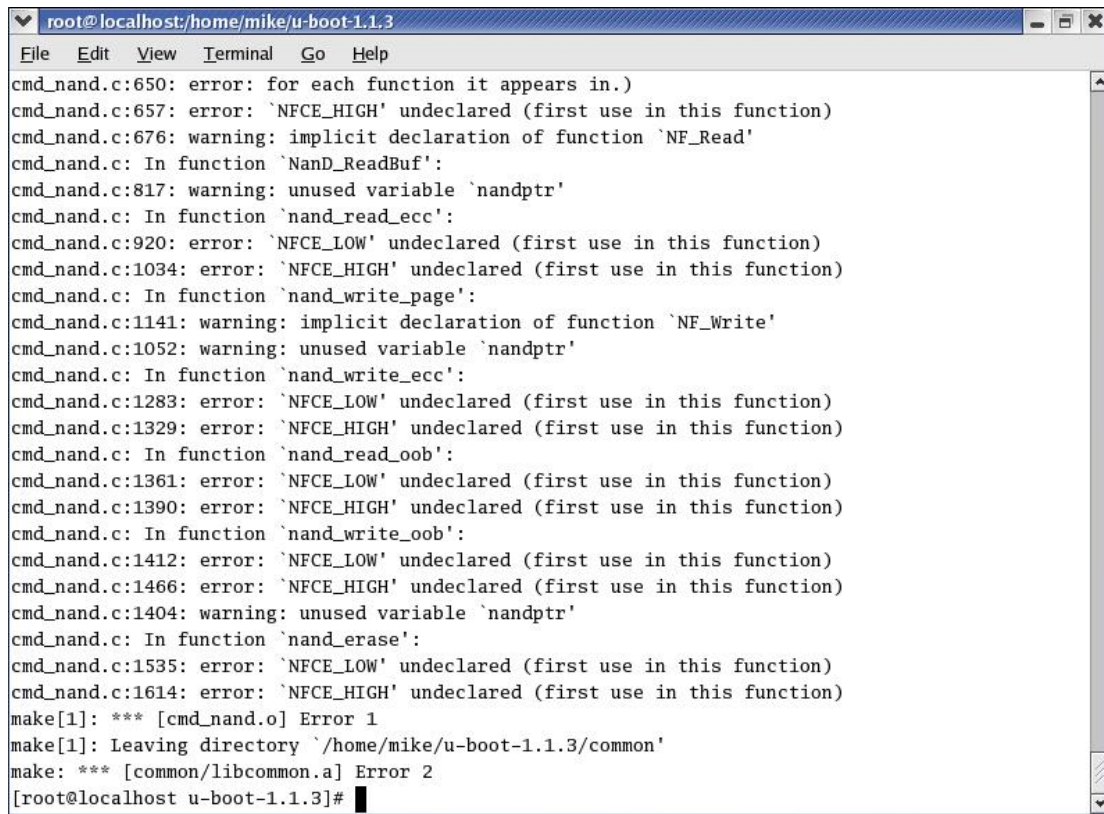
```

#define CFG_MEMTEST_END        0x33F00000    /* 64 MB in DRAM    */
#define CFG_LOAD_ADDR          0x30008000    /* default load address    */
#define PHYS_SDRAM_1_SIZE      0x04000000 /*64 MB */
#define CFG_PROMPT              "MIKE2410 # " /* Monitor Command Prompt    */
#define CONFIG_IPADDR          192.168.1.10
#define CONFIG_SERVERIP        192.168.1.1
#define CFG_ENV_IS_IN_NAND      1
#define CFG_ENV_OFFSET         0x30000

```

#### 10. 修改 cmd\_nand.c 文件

cmd\_nand.c 文件在 commom 目录下，修改该文件的目的是添加包含 nandflash.h 的头文件，因为如果没有包含该头文件，编译时将出错如图 3.4 所示，正是因为该文件调用了 nandflash.h 头文件中的变量而没有包含该头文件引起的，包含该头文件后将编译正确。



```
root@localhost/home/mike/u-boot-1.1.3
File Edit View Terminal Go Help
cmd_nand.c:650: error: for each function it appears in.)
cmd_nand.c:657: error: `NFCE_HIGH' undeclared (first use in this function)
cmd_nand.c:676: warning: implicit declaration of function `NF_Read'
cmd_nand.c: In function `Nand_ReadBuf':
cmd_nand.c:817: warning: unused variable `nandptr'
cmd_nand.c: In function `nand_read_ecc':
cmd_nand.c:920: error: `NFCE_LOW' undeclared (first use in this function)
cmd_nand.c:1034: error: `NFCE_HIGH' undeclared (first use in this function)
cmd_nand.c: In function `nand_write_page':
cmd_nand.c:1141: warning: implicit declaration of function `NF_Write'
cmd_nand.c:1052: warning: unused variable `nandptr'
cmd_nand.c: In function `nand_write_ecc':
cmd_nand.c:1283: error: `NFCE_LOW' undeclared (first use in this function)
cmd_nand.c:1329: error: `NFCE_HIGH' undeclared (first use in this function)
cmd_nand.c: In function `nand_read_oob':
cmd_nand.c:1361: error: `NFCE_LOW' undeclared (first use in this function)
cmd_nand.c:1390: error: `NFCE_HIGH' undeclared (first use in this function)
cmd_nand.c: In function `nand_write_oob':
cmd_nand.c:1412: error: `NFCE_LOW' undeclared (first use in this function)
cmd_nand.c:1466: error: `NFCE_HIGH' undeclared (first use in this function)
cmd_nand.c:1404: warning: unused variable `nandptr'
cmd_nand.c: In function `nand_erase':
cmd_nand.c:1535: error: `NFCE_LOW' undeclared (first use in this function)
cmd_nand.c:1614: error: `NFCE_HIGH' undeclared (first use in this function)
make[1]: *** [cmd_nand.o] Error 1
make[1]: Leaving directory `/home/mike/u-boot-1.1.3/common'
make: *** [common/libcommon.a] Error 2
[root@localhost u-boot-1.1.3]#
```

图 3.4 没有包含 nandflash.h 引起的错误

## 11. 重新编译

如果以上步骤都正确的话，执行下面的编译命令，将会在 U-Boot 的根目录下重新生成 u-boot.bin 可执行性文件。

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

## 12. 用 Jtag 烧写 u-boot.bin 文件

用开发板上自带的 Jtag 线和开发板连接，使用三星公司提供的 SJF(Sec Jtag Flash)工具将上面生成的 u-boot.bin 文件烧写到开发板上，烧写过程如图 3.5 所示。



```

bootp    - boot image via network using BootP/TFTP protocol
bootvx   - Boot vxWorks from an ELF image
cmp      - memory compare
coninfo  - print console devices and information
cp       - memory copy
crc32    - checksum calculation
dcache   - enable or disable data cache
echo     - echo args to console
erase    - erase FLASH memory
flinfo   - print FLASH memory information
go       - start application at address 'addr'
help     - print online help
icache   - enable or disable instruction cache
iminfo   - print header information for application image
imls     - list all images found in flash
itest    - return true/false on integer compare
loadb    - load binary file over serial line (kermit mode)
loads    - load S-Record file over serial line
loop     - infinite loop on address range
md       - memory display
mm       - memory modify (auto-incrementing)
mtest    - simple RAM test
mw       - memory write (fill)
nand     - NAND sub-system
nboot    - boot from NAND device
nfs      - boot image via network using NFS protocol
nm       - memory modify (constant address)
ping     - send ICMP ECHO_REQUEST to network host
printenv - print environment variables
protect  - enable or disable FLASH write protection
rarpboot - boot image via network using RARP/TFTP protocol
reset    - Perform RESET of the CPU
run      - run commands in an environment variable
saveenv  - save environment variables to persistent storage
setenv   - set environment variables
sleep    - delay execution for some time
tftpboot - boot image via network using TFTP protocol
version  - print monitor version

```

以下将介绍 U-Boot 常用命令：

1. ? : 得到所有命令列表，和 help 命令作用相同
2. bdfinfo: 打印开发板信息，以本文开发板为例，使用该命令可以看到如下信息：

```

MIKE2410# bdfinfo
arch_number = 0x000000C1      //CPU 体系结构编号
env_t       = 0x00000000      //环境变量

```

```

boot_params = 0x30000100      //启动引导参数
DRAM bank   = 0x00000000      //内存区
-> start     = 0x30000000      //SDRAM 起始地址
-> size      = 0x04000000      //SDRAM 大小
ethaddr     = 01:23:45:67:89:AB //以太网地址（自己可以设定）
ip_addr     = 192.168.1.10     //IP 地址
baudrate    = 115200 bps      //波特率大小

```

3. tftp (tftpboot): 利用 tftp 协议将 PC 本地上的映象文件下载到指定地址的 SDRAM 中, 执行该命令的前提是所用 PC 上已经安装了 tftp 服务。具体使用方法如下:

```

MIKE2410# tftp 0x30008000 zImage
TFTP from server 192.168.1.5; our IP address is 192.168.1.10
Filename 'zImage'.
Load address: 0x30008000
Loading: #####
          #####
          #####
done
Bytes transferred = 890752 (d9780 hex)

```

其中, 0x30008000 为指定下载到 SDRAM 的地址, zImage 为下载映象的文件名。

4. bootm: 从内核的入口地址引导内核。具体使用方法如下:

```

MIKE2410# bootm 0x30008000
## Booting image at 30008000 ...
Starting kernel ...
Uncompressing Linux.....done, booting the
kernel.

```

5. go: 直接跳转到可执行性文件的入口地址, 执行可执行性文件。使用方法如下:

```

MIKE2410# go 0x30008000
## Starting application at 0x30008000 ...

```

6. printenv: 打印环境变量信息。使用方法如下:

```

MIKE2410# printenv
bootdelay=3
baudrate=115200
ipaddr=192.168.1.10
serverip=192.168.1.5
netmask=255.255.255.0
stdin=serial
stdout=serial
stderr=serial
Environment size: 132/65532 bytes

```

7. setenv: 设置环境变量, 使用方法如下:

```

MIKE2410# setenv ipaddr 192.168.1.2

```

其中, ipaddr 为要设置的环境变量名, 192.168.1.2 为要设置的环境变量值。

8. nand read: 从 NAND Flash 的具体地址读数据到内存中去, 具体使用如下:

```

MIKE2410# nand read 0x30008000 0 0x100000

```

```
NAND read: device 0 offset 0, size 1048576 ... 1048576 bytes read: OK
```

其中，0x30008000 为要读到内存的地址，0 为要读取的 NAND Flash 的地址，0x100000 为要读取数据的大小。

9. nand erase: 擦除 NAND Flash 指定地址的数据，具体使用如下：

```
MIKE2410# nand erase 0x100000 0x20000
```

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

其中，0x100000 为指定擦除 NAND Flash 的起始地址，0x20000 擦除数据块的大小。

10. nand write: 给 NAND Flash 中写数据，具体使用如下：

```
MIKE2410# nand write 0x30200000 0x100000 0x20000
```

```
NAND write: device 0 offset 1048576, size 131072 ... 131072 bytes written: OK
```

其中，0x30200000 为内存的起始地址，0x100000 为要写的 NAND Flash 的起始地址，0x20000 要写数据的大小。注意，在给 NAND Flash 写数据前必须先执行擦除操作，否则给 NAND Flash 写数据会失败。

## 3.4 基于 S3C2410 开发板自己编写 BootLoader

相对于 U-Boot 移植来说，这种 DIY 的方法要复杂一些，不过和第二章中介绍分步构建交叉编译工具链一样，通过自己编写 BootLoader 更能清楚的知道它的工作原理，从而对于深入的学习嵌入式系统开发非常有好处。为了让读者全面学习 BootLoader，在本书附件的光盘中附有本节讲述的参考 BootLoader 实现代码，对于类似的开发板只需要做简单的修改就可以使用。

### 3.4.1 设计系统的启动流程

系统加电复位后，几乎所有的 CPU 都从由复位地址上取指令。比如，基于 ARM920T 或 ARM7TDMI 内核的 CPU 在复位时通常都从地址 0x00000000 处取它的第一条指令。而以微处理器为核心的嵌入式系统通常都有某种类型的固态存储设备（比如 EEPROM、FLASH 等）被映射到这个预先设置好的地址上。因此在系统加电复位后，处理器将首先执行存放在复位地址处的程序。通过集成开发环境可以将 Bootloader 定位在复位地址开始的存储空间内，如图 3.4 用 ADS 集成开发工具中的 ARM Linker 设置选项定义 RO Base 为 0x30200000，它的意思就是定义了 BootLoader 的入口地址，即\_ENTRY 地址。因此 Bootloader 在系统加电后、操作系统内核或应用程序运行之前，首先必须运行。对于嵌入式系统来说，有的使用操作系统，也有的不使用操作系统，比如功能简单仅包括应用程序的系统，但是无论你是否使用操作系统，在系统启动时都必须执行 Bootloader，为系统运行准备好软硬件运行环境。



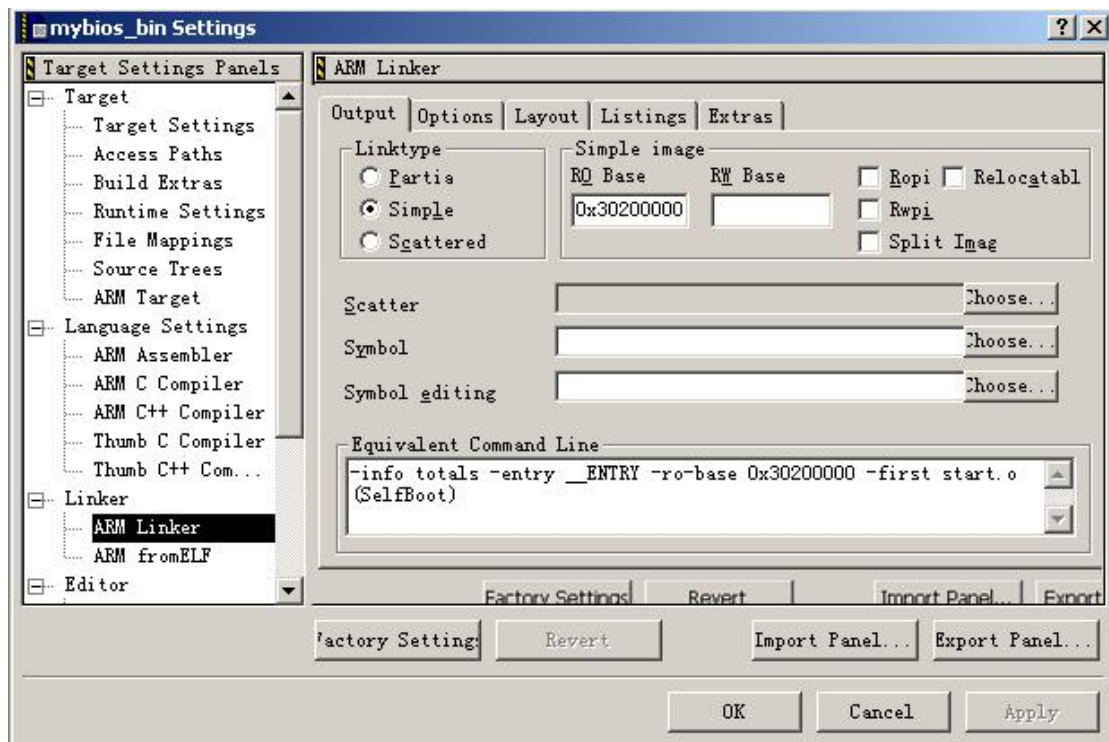


图 3.4 用 ADS 工具定义 BootLoader 的复位地址

系统的启动通常有两种方式：一种是可以直接从 Nor Flash 启动，另一种是可以将压缩的内存映像文件从 Flash（为节省 Flash 资源、提高速度）中复制、解压到 RAM，再从 RAM 启动。这里还是以第二种方式为例进行讲解，当电源打开时，一般的系统会去执行 ROM（应用较多的是 Flash）里面的启动代码。这些代码是用汇编语言编写的，其主要作用在于初始化 CPU 和板上的必备硬件如内存、中断控制器等。有时候用户还必须根据自己板子的硬件资源情况做适当的调整与修改。

BootLoader 完成基本软硬件环境初始化后，对于有操作系统的情况下，启动操作系统、启动内存管理、任务调度、加载驱动程序等，最后执行应用程序或等待用户命令；对于没有操作系统的系统直接执行应用程序或等待用户命令。在商业的实时操作系统中，启动代码部分一般被称为板级支持包，英文缩写为 BSP（Board Support Package）。它的主要功能就是：电路初始化和为高级语言编写的软件运行做准备。基于 S3C2410 开发板系统的启动流程设计如图 3.5 所示：

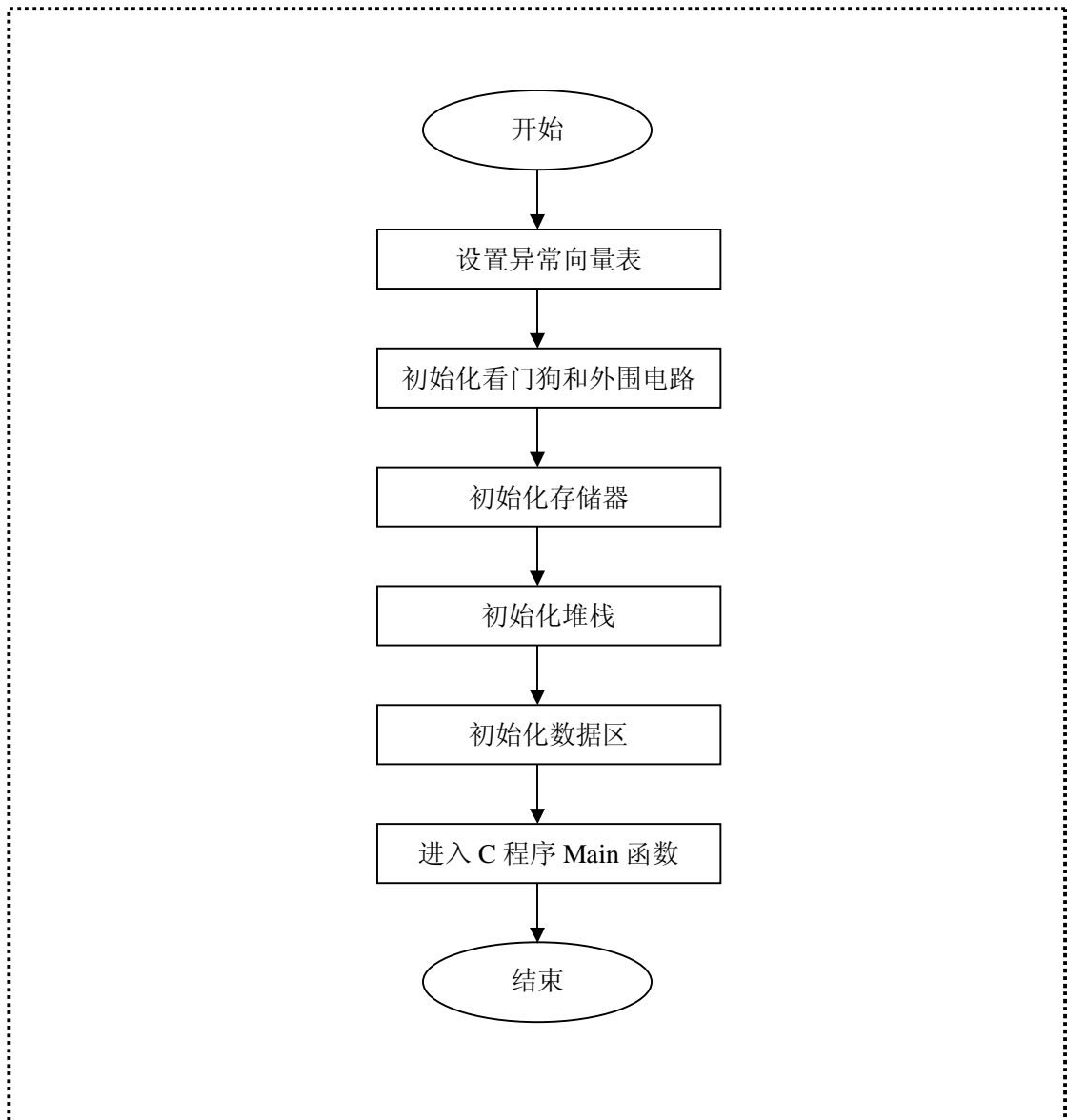


图 3.5 基于 S3C2410 开发板系统的启动流程设计图

接下来讲述基于 S3C2410 开发板系统的 BootLoader 具体实现，将根据设计的启动流程逐步进行具体讲述。

### 3.4.2 BootLoader 的具体实现

本例中 BootLoader 的实现和 U-Boot 类似，也是分为 Stage1 和 Stage2 两个阶段实现。首先建立一个名为 start.s 的文件在 src 目录下，通过观察这个文件名读者就可以知道 start.s 是一个汇编源文件，通常 BootLoader 的启动代码都是用汇编语言来实现的，因为它的执行效率高，并且代码量小，所以它是开发嵌入式系统 BootLoader 的首选开发语言。这个文件实现 Stage1 部分，其主要完成以下几个功能：

1. 设置异常向量表
2. 初始化看门狗和外围电路
3. 初始化存储器
4. 初始化堆栈

5. 初始化数据区
6. 跳转到 C 程序 Main 函数

Stage1 的任务结束后就要开始 Stage2 了，此时定义一个 C 源文件为 bios.c，它主要完成一些高级复杂功能的初始化，包括 I/O 端口、中断、串口、MMU（内存管理单元）等初始化工作，以及实现装载操作系统的功能。下面将分别介绍这些功能的具体实现。

### 3.4.2.1 设置异常向量表

ARM 通常要求异常向量表必须放置在从 0 地址开始，且放在连续 8\*4 字节的空间内。每当一个中断发生以后，ARM 处理器便强制把 PC 指针置为向量表中对应中断类型的地址值。因为每个中断只占据向量表中 1 个字的存储空间，所以只能放置一条 ARM 指令，使程序跳转到存储器的其他地方，再执行中断处理。

本系统的异常向量表的程序实现如下：

AREA	SelfBoot, CODE, READONLY
ENTRY	
b	ResetHandler ;handler for Reset
b	HandlerUndef ;handler for Undefined mode
b	HandlerSWI ;handler for SWI interrupt
b	HandlerPabort ;handler for Prefetch Abort
b	HandlerDabort;handler for DAbort
b	. ;reserved
b	HandlerIRQ ;handler for IRQ interrupt
b	HandlerFIQ ;handler for FIQ interrupt

其中第一行定义了代码区域的属性，表示了是只读属性的自启动代码区。关键字 ENTRY 指定编译器保留这段代码，因为编译器可能会认为这是一段冗余代码而加以优化。链接的时候要确保这段代码被链接在 0 地址处或集成开发工具自定义的地址，并且作为整个程序的入口。这段程序定了 ARM 处理器常见的 7 种异常向量，具体含义如下：

- ü Reset：即复位异常，通常是系统上电复位或通过软件实现复位。
- ü Undefined Instruction：即未定义指令异常，当出现一个既不能被主处理器识别又不能被协处理器识别的执行指令发生时。
- ü Software Interrupt (SWI)：即软件中断，这是用户定义的同步中断指令，它允许程序运行在用户模式。
- ü Prefetch Abort：即预取中止异常，当处理器试图执行一个还没有取到的指令时发生，是因为试图执行一个不合法的地址。
- ü Data Abort：即数据中止异常，当一个数据传输指令试图装载或保存一个数据在一个不合法的地址时发生。
- ü IRQ：即中断请求，当处理器的外部中断请求针被设置时发生，此时 CPSR 状态寄存器的第 I 位是被清除的。
- ü FIQ：即快速中断请求，当处理器外部快速中断请求针被设置时发生，此时 CPSR 状态寄存器的第 F 位是被清除的。

通常当几个异常同时发生时，处理器必须知道先处理那个再处理那个，所以必须定义一个规则来确定它们的先后顺序，这就是异常的优先级。ARM 处理器定义了异常的优先级如表 3.1 所示：

向量地址	异常类型	异常模式	优先级(1=high,
------	------	------	-------------

			6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor(SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	Reserved	没有定义	没有定义
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

### 3.4.2.2 初始化看门狗和外围电路

这一步参考 S3C2410 用户手册实现了看门狗、中断、PLL（Phase Lock Loop）锁时间计数器（Lock time counter）和 MPLL 配置寄存器的初始化。具体实现请参考以下代码和注释：

```

ldr r0,=WTCON          ; 关闭看门狗
ldr r1,=0x0
str r1,[r0]

ldr r0,=INTMSK   ; 屏蔽所有第一级中断
ldr r1,=0xffffffff
str r1,[r0]

ldr r0,=INTSUBMSK ; 屏蔽所有第二级中断
ldr r1,=0x3ff
str r1,[r0]

; 为了减少 PLL 锁时间通过调节 LOCKTIME 寄存器.
ldr r0,=LOCKTIME
ldr r1,=0xffffffff
str r1,[r0]

; 配置 MPLL 控制寄存器
ldr r0,=MPLLCON
ldr r1,=((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV) ;Fin=12MHz,Fout=50MHz
str r1,[r0]

```

### 3.4.2.3 初始化存储器

这一步用来设置内存控制寄存器，具体实现参考以下代码：

```

; 设置内存控制器
adr r0, SMRDATA      ; 不能使用 ldr r0, =xxxx
ldr r1, =BWSCON      ; BWSCON 为总线宽度和等待状态控制寄存器
add r2, r0, #52      ; SMRDATA 结束地址

```

```

0
ldr r3, [r0], #4
str r3, [r1], #4
cmp r2, r0
bne %B0

```

其中 SMRDATA 的定义如下，其作用是定义内存区域控制寄存器值。

```

SMRDATA DATA
; 1) 即使在 HCLK=75Mhz，内存设置的都是安全的参数
; 2) SDRAM 刷新周期是用于 HCLK=75Mhz 时。
DCD
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5
_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
DCD
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tapc<<2)+
(B0_PMC)) ;GCS0
DCD
((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tapc<<2)+
(B1_PMC)) ;GCS1
DCD
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tapc<<2)+
(B2_PMC)) ;GCS2
DCD
0x1f7c;((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tapc<<2)+(B3_PMC)) ;GCS3
DCD
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tapc<<2)+
(B4_PMC)) ;GCS4
DCD
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tapc<<2)+
(B5_PMC)) ;GCS5
DCD ((B6_MT<<15)+(B6_Tred<<2)+(B6_SCAN)) ;GCS6
DCD ((B7_MT<<15)+(B7_Tred<<2)+(B7_SCAN)) ;GCS7
DCD
((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)

DCD 0x32 ; SCLK 电压保护模式 BANKSIZE 是 128M
DCD 0x30 ; MRSR6 CL=3clk
DCD 0x30 ; MRSR7

```

### 3.4.2.4 初始化堆栈

接下来要初始化堆栈，一般 ARM 有 7 种工作模式：用户模式 (usr)，快速中断模式 (fiq)，外部中断模式 (irq)，管理模式 (svc)，数据访问终止模式 (abt)，系统模式 (sys) 和未定义指令中止模式 (und)。关于这 7 种模式在前面的章节中已经介绍，这里不再重复。初始化堆

栈也就是初始化这 7 种模式下的堆栈。

```
bl  InitStacks      ; 调用初始化堆栈函数

InitStacks          ; 实现初始化堆栈
; 不要使用 DRAM, 如 stmfd,ldmfd.....
mrs  r0,cpsr
bic  r0,r0,#MODEMASK
orr  r1,r0,#UNDEFMODE|NOINT
msr  cpsr_cxsf,r1    ; 未定义指令中止模式 (und)
ldr  sp,=UndefStack

orr  r1,r0,#ABORTMODE|NOINT
msr  cpsr_cxsf,r1    ; 数据访问终止模式(abt)
ldr  sp,=AbortStack

orr  r1,r0,#IRQMODE|NOINT
msr  cpsr_cxsf,r1    ; 外部中断模式 (irq)
ldr  sp,=IRQStack

orr  r1,r0,#FIQMODE|NOINT
msr  cpsr_cxsf,r1    ; 快速中断模式 (fiq)
ldr  sp,=FIQStack

bic  r0,r0,#MODEMASK|NOINT
orr  r1,r0,#SVCMODE
msr  cpsr_cxsf,r1    ; 管理模式 (svc)
ldr  sp,=SVCStack

; 用户模式 (usr) 和系统模式 (sys) 没有被初始化
mov  pc,lr ; 如果当前模式不适 SVC 模式, LR 寄存器将无效

LTORG
```

### 3.4.2.5 初始化数据区

内核映像一开始总是存储在 ROM 或 Flash 里面的, 其 RO 部分既可以在 ROM 或 Flash 里面执行, 也可以转移到速度更快的 RAM 中执行; 而 RW 和 ZI\* (注 2) 这两部分是必须转移到可写的 RAM 里去。所谓数据区的初始化, 就是完成必要的从 ROM 到 RAM 的数据传输和内容清零。这步的代码实现如下:

```
InitRam
ldr  r2, BaseOfBSS  ; BaseOfBSS 定义 RW 部分的基地址
ldr  r3, BaseOfZero ; BaseOfZero 定义 ZI 部分的基地址
0
cmp  r2, r3
```

```
ldrccr1, [r0], #4
strccr1, [r2], #4
bcc %B0

mov r0, #0
ldr r3, EndOfBSS ; EndOfBSS 定义 ZI 部分的末地址
1 cmp r2, r3
strccr0, [r2], #4
bcc %B1
```

\*注2：当可执行性文件被装载到RAM中后，它在RAM中存放的位置如下：



图 3.6 可执行性文件在装载前后的分布

其中 BaseOfROM, TopOfROM, BaseOfBSS, BaseOfZero 和 EndOfBSS 的定义如下：

BaseOfROM	DCD	Image\$\$RO\$\$Base
TopOfROM	DCD	Image\$\$RO\$\$Limit
BaseOfBSS	DCD	Image\$\$RW\$\$Base
BaseOfZero	DCD	Image\$\$ZI\$\$Base
EndOfBSS	DCD	Image\$\$ZI\$\$Limit

其中：

- |Image\$\$RO\$\$Base|：表示 RO 区开始地址
- |Image\$\$RO\$\$Limit|：表示 RO 区末地址后面的地址，即 RW 数据源的起始地址
- |Image\$\$RW\$\$Base|：表示 RW 区在 RAM 里的执行区起始地址，也就是编译器选项 RW\_Base 指定的地址
- |Image\$\$ZI\$\$Base|：表示 ZI 区在 RAM 里面的起始地址
- |Image\$\$ZI\$\$Limit|：表示 ZI 区在 RAM 里面的结束地址后面的一个地址

上述程序先把 ROM 里|Image\$\$RO\$\$Limit|开始的 RW 初始数据拷贝到 RAM 里面 |Image\$\$RW\$\$Base|开始的地址，当 RAM 这边的目标地址到达|Image\$\$ZI\$\$Base|后就表示 RW 区的结束和 ZI 区的开始，接下去就对这片 ZI 区进行清零操作，直到遇到结束地址 |Image\$\$ZI\$\$Limit|为止。

### 3.4.2.6 跳转到 C 程序 Main 函数

这一步是进入 C 程序入口的操作，也是 BootLoader 必不可少的一个过程。它的具体实现如下：

```
    ; 发送复位状态信息到 Main 函数
    ldr r1, =GSTATUS2
    ldr r0, [r1]
    str r0, [r1]    ; 清除复位信息

    ldr pc, GotoMain ; 调用 Main 函数，此时 GotoMain 代表 Main。
```

### 3.4.2.7 Main 函数的具体实现

该部分可以说实现 BootLoader 的 Stage2 的功能，该函数主要完成以下工作：

1. 初始化系统频率
2. 初始化 I/O 端口
3. 初始化中断处理程序表
4. 串口初始化
5. 其他部分的初始化
6. 主程序循环

具体实现代码如下：

```
int Main(U32 RstStat)
{
    int i;
    SetClockDivider(0, 1); //设置 FCLK:HCLK:PCLK= 1:1:2
    SetSysFclk(FCLK_96M);   //设置系统 FCLK=96MHz
    Port_Init();           //I/O 端口初始化
    Isr_Init();            //中断处理程序表初始化

    Uart_Init(0, Console_Baud); //串口初始化
    Uart_Select(Console_Uart);

    MMU_Init(); //MMU 初始化
    //使能 GPIO,UART0,PWM TIMER,NAND FLASH 等模块时钟
    EnableModuleClock(CLOCK_ALL);

    .....

    NandLoadRun(); //定义从 NAND Flash 装载操作系统
    while(1)
    {
        ;
    }
}
```



## 3.5 本章小节

BootLoader 是嵌入式系统非常重要的一部分，本章一开始讲述了 BootLoader 的基本概念，以及它的主要作用；紧接着讲述了常见的几种 BootLoader: U-Boot, VIVI, Blob, RedBoot, ARMboot 和 DIY；最后重点讲述了基于 S3C2410 开发板的 U-Boot 移植和编写自己的 BootLoader。通过学习本章内容，使读者可以清楚地了解嵌入式系统的启动过程，为进一步学习嵌入式系统开发奠定良好的基础。下一章将介绍 Linux 内核移植。

## 3.6 常见问题

### 1. 什么是 BootLoader?

参考答案：简单地说，BootLoader 就是在操作系统内核运行前运行的一段小程序。通过这段小程序，我们可以初始化必要的硬件设备，创建内核需要的一些信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，真正起到引导和加载内核的作用。

### 2. 常用的嵌入式 BootLoader 有哪些?

参考答案：有 U-Boot, Blob, VIVI, RedBoot 和 ARMboot 等。

### 3. ARM 处理器的有哪几种工作模式?

参考答案：ARM 处理器有以下 7 种工作模式：

- ┆ 用户模式 (usr)： ARM 处理器正常的程序执行状态
- ┆ 快速中断模式 (fiq)：用于高速数据传输或通道处理
- ┆ 外部中断模式 (irq)：用于通用的中断处理
- ┆ 管理模式 (svc)：操作系统使用的保护模式
- ┆ 数据访问终止模式(abt)：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- ┆ 系统模式 (sys)：运行具有特权的操作系统任务。
- ┆ 未定义指令中止模式(und)：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

### 4. 通常 BootLoader 的 Stage1 都实现哪些功能?

参考答案：通常情况下完成以下几个任务：

1. 设置异常向量表
2. 初始化看门狗和外围电路
3. 初始化存储器
4. 初始化堆栈
5. 初始化数据区
6. 跳转到 C 程序 Main 函

## 第4章 嵌入式 Linux 内核移植

本章学习目标：

- l 了解移植的基本概念
- l 熟悉 Linux 内核的配置过程
- l 熟悉 Linux 内核的编译过程
- l 了解根文件系统的作用
- l 学会构建 Cramfs 文件系统
- l 学会 BusyBox 工具的使用

### 4.1 移植的基本概念

在第三章中我们已经接触了真正的移植，但并没有解释什么是移植，在这节里有必要给读者解释一下在嵌入式开发中应用非常广泛的一个概念——移植，英文为 **Porting**。从广义上讲，移植包括软件移植和硬件移植。从狭义上讲，移植就是指软件移植，即将一个软件从一个平台迁移到另外一个与其不同的平台上工作。通常情况下，移植分为以下三种情况：

ü 从一个硬件平台移植到另外一个硬件平台

在 Linux 内核代码中，可以看到 arch 目录下有许多子目录，其中每一个子目录代表一种硬件平台，也就是说 Linux 内核 arch 目录下有多少个子目录就代表其支持多少种硬件平台。这里以 Linux 2.6.10 内核为例，其 arch 目录下的内容如下：

alpha	cris	ia64	m68knommu	ppc	sh	sparc64	x86_64
arm	h8300	m32r	mips	ppc64	sh64	um	
arm26	i386	m68k	parisc	s390	sparc	v850	

以上每一种体系结构里又包含许多的子体系，以 arm 体系结构为例，它又包含 mach-h720x，mach-l7200，mach-sa1100，mach-epxa10db，mach-ixp2000，mach-rpc，mach-s3c2410 等子体系结构。

这种形式的移植最常见的就是 Linux 操作系统移植。比如将基于 x86 体系的 Linux 移植到基于 ARM 体系的嵌入式 Linux。首先是工具链的移植，因为基于 x86 体系的 gcc 就不能用在基于 ARM 的体系中，所以在 PC 机上编译时要建立交叉编译工具链。同时还要考虑 binutils、glibc 等移植。其次是内核移植，内核移植主要包括两方面的工作，一是 arch 目录下的体系结构的移植，如从 i386 移植到 arm，二是移植 drivers 目录下的许多硬件驱动程序。最后是应用程序的移植，如 C 代码的重新编译，实现一些缺少的库，如 Qt/embedded 库的移植等。

由于硬件平台对 C 语言有一定的影响，移植时必须要考虑。例如处理器的字长，定义处理器一次能处理数据位的个数，通常是 32 位，处理器字长会影响 C 语言的数据类型的长度，如 int，long 等。数据对齐也是这种形式移植时必须考虑的，数据对齐的意思是数据块的地址是某个特定数大小的整数倍，如 32 位处理器要求是 4 对齐的，即必须是 4 的整数倍。字节顺序也是必须要考虑的，字节顺序 (byte order) 是指一个字中字节排列的顺序。不同硬件可能采用不同的排列顺序，例如 x86 是 little-endian，ppc 是 big-endian。软件中与时间相关的代码也会影响移植过程，所以建议采用与时间无关的代码可以提高移植性，比如 Linux 内核里用 HZ 来表示每秒钟的嘀嗒数。内存页面的大小也是移植过程应该考虑的，不同的体

系结构可能会有不同的页面大小，常用的 32 位处理器用 4KB 的页面大小，有些体系结构可以支持多种页面大小。

ü 从一个操作系统移植到另一个操作系统

这种形式的移植也是最常见的。比如将 Windows 系统下运行的程序移植到 Linux/Unix 系统中，这时需要考虑操作系统提供的 API，以及所调用的函数库等。

ü 从一种软件库环境移植到另一种软件库环境

这种类型的移植也是比较常见的，例如基于 Qt 3.0 库的应用程序移植到 Qt 4.0 库环境中去。再如基于 glibc 库环境的程序移植到基于 uclibc 库环境。

## 4.2 内核移植的准备

为什么要选择移植 Linux 内核？首先让我们来看一下 Linux 内核都具有哪些特点：

- l 开放性：Linux 内核遵循 GNU GPL (General Public License) \* (注 1)，所以其源代码都是免费公开的
- l 可移植性：支持几乎所有硬件平台
- l 可定制性：不但能够运行在高性能计算机上，也可以运行在资源有限的嵌入式设备中
- l 互操作性：兼容许多标准
- l 网络支持：支持许多网络协议
- l 安全性：实现许多的安全协议，并且开发人员都非常重视它的安全性
- l 稳定性：经过多年许多产品的使用已经表明它具有很好的稳定性
- l 模块化：可以使内核只包含系统必须的东西，其他的都可以使用模块化来完成
- l 方便编程：可以通过学习已有的代码和网络上的丰富的资源。

\*注 1：GNU GPL，即 GNU 通用公共许可证，是由自由软件基金会发行的用于计算机软件的许可证。最初由 Richard Stallman 为 GNU 计划而撰写。目前大多数的 GNU 程序和超过半数的自由软件使用此许可证。此许可证最新版本为“版本 3”，1991 年发布。GNU LGPL(Lesser General Public License)，即宽通用公共许可证是由 GPL 衍生出的许可证，被用于一些 GNU 程序库。

由于 Linux 内核具有以上许多特点，这些特点正都是它的优点。除此之外，它和其他操作系统相比还有许多优点。一是平台独立性，它不依赖于某个特定的硬件平台，通常选择一个操作系统也许就会锁定你仅能使用特定的硬件平台，而 Linux 却可以真正的实现平台独立性。二是快速上市，因为使用 Linux 系统很容易移植许多硬件到系统中去，所以会大大减少开发时间，从而加快产品上市。三是低成本，它不但可以节约开发成本，而且也可以节约培训成本。四是遵循 POSIX (Portable Operating System Interface) \* (注 2) 标准，POSIX 的目的就是提升软件的可移植性在 UNIX 系统上，因此遵循这个标准可以使开发更容易。五是代码开放性，Linux 之所以变得如此流行，这应该是一个非常重要的原因。六是支持多种硬件，Linux 不但支持最新的高性能硬件，同时也支持低价格和早期的微处理和 I/O 设备[7]。总之，Linux 还有许多独特的优点，所以在选择嵌入式操作系统时被大多数硬件平台选用。它现在已经成为嵌入式系统中的一个主流操作系统。

\*注 2：POSIX 表示可移植操作系统接口 (Portable Operating System Interface，缩写为 POSIX 是为了读音更像 UNIX)。电气和电子工程师协会 (Institute of Electrical and Electronics Engineers, IEEE) 最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。它已被应用许多其它的操作系统，例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990 (1995 年修订) 或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应

用编程接口 (API) 给操作系统的服务程序, 例如读写文件。POSIX.1 已经被国际标准化组织 (International Standards Organization, ISO) 所接受, 被命名为 ISO/IEC 9945-1:1990 标准。POSIX 现在已经发展成为一个非常庞大的标准族, 某些部分新的功能正处在开发过程中。

移植内核首先要确保编译内核的工具是否准备好, Linux 内核归根结底也是一个程序, 所以它必须通过编译器编译后才能在硬件上执行, 由于我们的目标板是基于 ARM920T 内核的 S3C2410 处理器, 需要能够编译出在 ARM 处理器上可以运行的程序, 这时就需要交叉编译链了, 好在我们已经未雨绸缪, 在第二章已经介绍了交叉编译工具的制作, 同时已经构建了自己的交叉编译工具链, 在此直接使用就可以了。

其次, 从 [ftp.kernel.org](http://ftp.kernel.org) 站点下载要移植的内核代码, 这里下载的内核代码为 linux-2.6.10.tar.gz, 可以看出它的版本为 Linux 2.6.10。

最后检查要移植的开发板硬件是否准备就绪, 当所有基本条件都准备好了, 下面我们就可以正式移植内核了。

## 4.3 内核移植

内核是所有嵌入式 Linux 系统的核心软件, 内核移植是一个比较复杂的任务, 当然也是嵌入式系统开发中非常重要的一个过程。内核移植一般包括内核配置、内核编译和内核下载三大部分。下面将具体介绍内核移植的每一个步骤。

### 4.3.1 内核配置

配置内核是构建一个嵌入式 Linux 系统内核的第一步, 有好几种配置内核的方式, 同时有很多内核的配置选项。下面将按执行的步骤讲述内核配置所要做的内容。

#### 4.3.1.1 修改 Makefile

这一步的作用是修改内核根目录下的 Makefile 文件, 从而指明要用的编译器为 arm-linux-交叉编译器, 使用的体系结构为 ARM。具体操作如下:

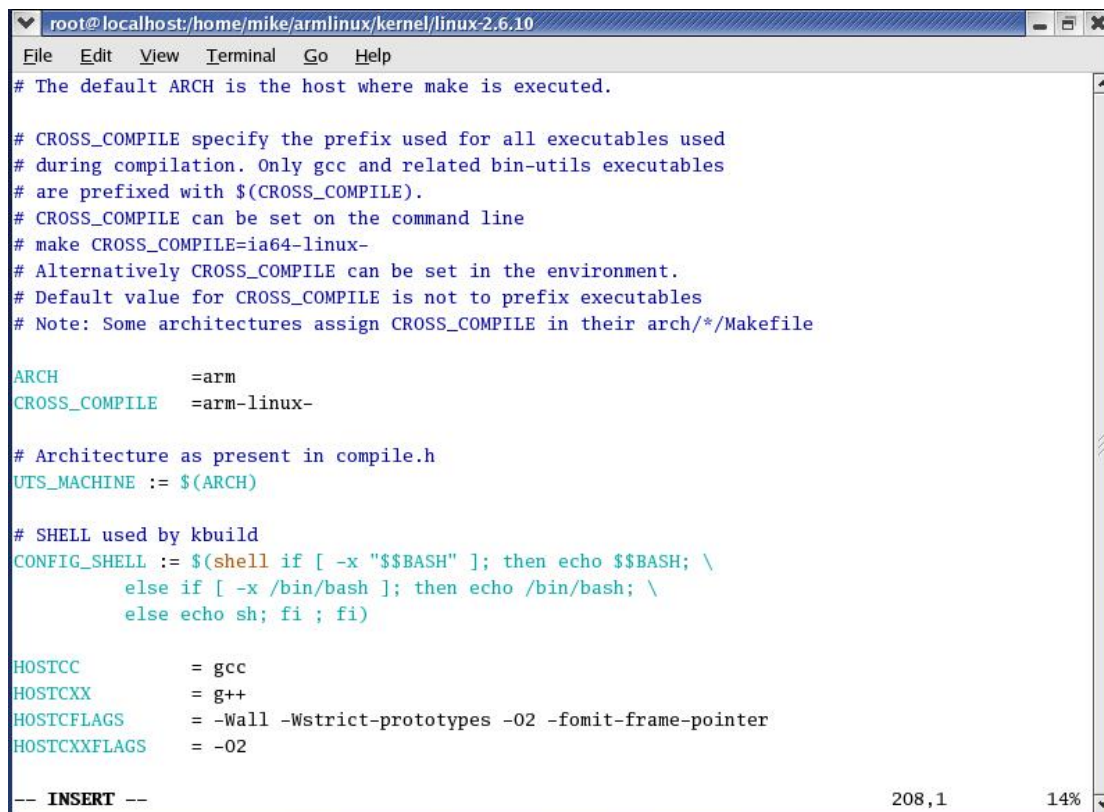
```
# cd linux-2.6.10
# vi Makefile
```

修改内容如图 4.1 所示, 其中修改两行内容如下:

```
ARCH = arm
CROSS_COMPILE = arm-linux-
```

其含义:

ARCH = arm 说明目标是 ARM 体系结构, 默认的 ARCH 是指宿主机的体系, 如 i386;  
CROSS\_COMPILE=arm-linux- 说明使用交叉编译器前缀为 arm-linux-, 默认情况下 CROSS\_COMPILE 为空, 即没有前缀。注意, 如果这个时候你还没有把交叉编译工具链的路径添加到环境变量中, 那么这里的 CROSS\_COMPILE 必须设置为宿主机上交叉编译工具链的绝对路径。



```
root@localhost:/home/mike/armlinux/kernel/linux-2.6.10
File Edit View Terminal Go Help
# The default ARCH is the host where make is executed.

# CROSS_COMPILE specify the prefix used for all executables used
# during compilation. Only gcc and related bin-utils executables
# are prefixed with $(CROSS_COMPILE).
# CROSS_COMPILE can be set on the command line
# make CROSS_COMPILE=ia64-linux-
# Alternatively CROSS_COMPILE can be set in the environment.
# Default value for CROSS_COMPILE is not to prefix executables
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile

ARCH                =arm
CROSS_COMPILE       =arm-linux-

# Architecture as present in compile.h
UTS_MACHINE := $(ARCH)

# SHELL used by kbuild
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
    else if [ -x /bin/bash ]; then echo /bin/bash; \
    else echo sh; fi ; fi)

HOSTCC              = gcc
HOSTCXX             = g++
HOSTCFLAGS          = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer
HOSTCXXFLAGS        = -O2

-- INSERT --
208,1 14%
```

图 4.1 修改 Makefile 的交叉编译器变量

#### 4.3.1.2 设置 NAND Flash 分区

由于我们的目标板使用的是 64M NAND Flash 作为 Flash 存储器，所以首先我们需要建立一个 NAND Flash 分区表，该分区表用来定义开发板上 64M 空间划分，以及定义各分区存放的起始地址以及大小等。该部分的实现在<arch/arm/mach-s3c2410/devs.c>源文件中，故修改该文件，修改内容如下：

```
//首先添加相应的头文件
#include <linux/mtd/partitions.h>
#include <linux/mtd/nand.h>
#include <asm/arch/nand.h>

.....
/*新建 64M 的 Nand Flash 分区表*/
static struct mtd_partition partition_info[]={
    /*1MB*/
        name:"bootloader",
        size:0x00100000,
        offset:0x0,
    },
    /*3MB*/
        name:"kernel",
        size:0x00300000,
        offset:0x00100000,
```

```

    },
    { /*40MB*/
        name:"rootfs",
        size:0x02800000,
        offset:0x00400000,
    },
    { /*20MB*/
        name:"user",
        size:0x01400000,
        offset:0x02c00000,
    }
};
//定义一个 NAND Flash 分区数据结构
struct s3c2410_nand_set nandset={
    nr_partitions: 4 ,           //定义分区数
    partitions: partition_info , //定义分区表
};
... ..

```

上述代码的作用是建立一个 64M 的 Nand Flash 分区表，将其分为：bootloader（启动程序），kernel（内核），rootfs（根文件系统）和 user（用户空间）四个分区。其中：name：代表分区的名称；size：代表分区的大小；offset：代表分区在 Flash 中的起始地址。同时后面定义 NAND Flash 的分区数据结构。

紧接着要建立内核对 NAND Flash 芯片的支持，同时加入对 NAND Flash 芯片的支持代码到 NAND Flash 的驱动程序。具体代码实现如下<arch/arm/mach-s3c2410/devs.c>：

```

//建立 NAND Flash 的芯片支持数据结构
struct s3c2410_platform_nand superlpplatform={
    tacs:0,
    twrph0:1,
    twrph1:0,
    sets: &nandset,
    nr_sets: 1,
};
... ..
//修改 s3c_device_nand 结构体增加对 superlpplatform 设备的支持
struct platform_device s3c_device_nand = {
    .name = "s3c2410-nand",
    .id = -1,
    .num_resources = ARRAY_SIZE(s3c_nand_resource),
    .resource = s3c_nand_resource,
    //添加以下内容用来支持 NAND Flash 芯片
    .dev = {
        .platform_data = &superlpplatform
    }
};

```

... ..

上述代码的主要作用是完成内核对 NAND Flash 芯片驱动的支持，需要指出的是 TACLS、TWRPH0 和 TWRPH1，参考 S3C2410 用户手册，可以看到这三个参数控制的是 NAND Flash 信号线 CLE/ALE 与写控制信号 nWE 的时序关系。我们设的值为 TACLS=0，TWRPH0=1，TWRPH1=0，其含义为：TACLS=1 个 HCLK 时钟，TWRPH0=2 个 HCLK 时钟，TWRPH1=1 个 HCLK 时钟。其中 sets 定义支持的分区集，nr\_sets 定义分区集数。最后修改 s3c\_device\_nand 结构体变量，添加对 NAND Flash 设备驱动的支持。其中：name：为设备名称；id：有效设备编号，如果只有一个定义为-1，如果多个则从 0 开始计算；num\_resources：定义有几个 NAND Flash 芯片资源；resource：定义 NAND Flash 芯片资源的首地址。

虽然我们已经实现了对新加的 NAND Flash 芯片的支持，但是现在内核还不能正常工作，因为在内核启动时还没有添加对 NAND Flash 分区表初始化配置，所以还需要修改 <arch/arm/mach-s3c2410/ mach-smdk2410.c>源文件，修改内容如下：

```
... ..
//修改 smdk2410_devices[]的初始化项中增加最后一项
static struct platform_device *smdk2410_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
    &s3c_device_nand //添加此行来初始化新增的 NAND Flash 分区
};
... ..
```

接下来还有一个问题就是要禁止 Flash ECC\*（注 3）校验，由于通常我们使用的 BootLoader 通过软件已经产生 ECC 校验码，这与内核 ECC 校验码不一致，所以这里选择禁止内核 ECC 校验。完成这个步骤需要修改的文件是 drivers/mtd/nand/s3c2410.c，具体操作如下：

\*注 3：在普通的内存上，常常使用一种技术，即 Parity，同位检查码（Parity check codes）被广泛地使用在侦错码（error detection codes）上，它们增加一个检查位给每个资料的字元（或字节），并且能够侦测到一个字符中所有奇（偶）同位的错误，但 Parity 有一个缺点，当计算机查到某个 Byte 有错误时，并不能确定错误在哪一个位，也就无法修正错误。基于上述情况，产生了一种新的内存纠错技术，那就是 ECC，英文全称是 Error Checking and Correcting，即错误检查和纠正，从这个名称我们就可以看出它的主要功能就是“发现并纠正错误”，它比奇偶校正技术更先进的方面主要在于它不仅能发现错误，而且能纠正这些错误，这些错误纠正之后计算机才能正确执行下面的任务，确保服务器的正常运行。ECC 本身并不是一种内存型号，也不是一种内存专用技术，它是一种广泛应用于各种领域的计算机指令中，是一种指令纠错技术。之所以说它不是一种内存型号，那是因为并不是一种影响内存结构和存储速度的技术，可以应用到不同的内存类型之中，就象前讲到的“奇偶校正”内存，它也不是一种内存，最开始应用这种技术的是 ED0 内存，现在的 SD 也有应用，而 ECC 内存主要是从 SD 内存开始得到广泛应用，而新的 DDR、RDRAM 也有相应的应用，目前主流的 ECC 内存其实是一种 SD 内存。

... ..

```
//修改 s3c2410 的 NAND Flash 芯片初始化函数
static void s3c2410_nand_init_chip(struct s3c2410_nand_info *info,
```

```

        struct s3c2410_nand_mtd *nmt_d,
        struct s3c2410_nand_set *set)
    {
        struct nand_chip *chip = &nmt_d->chip;

        chip->IO_ADDR_R      = (char *)info->regs + S3C2410_NFDATA;
        chip->IO_ADDR_W      = (char *)info->regs + S3C2410_NFDATA;
        chip->hwcontrol       = s3c2410_nand_hwcontrol;
        chip->dev_ready       = s3c2410_nand_devready;
        chip->cmdfunc         = s3c2410_nand_command;
        chip->write_buf       = s3c2410_nand_write_buf;
        chip->read_buf        = s3c2410_nand_read_buf;
        chip->select_chip     = s3c2410_nand_select_chip;
        chip->chip_delay      = 50;
        chip->priv            = nmt_d;
        chip->options         = 0;
        chip->controller      = &info->controller;

        nmt_d->info           = info;
        nmt_d->mtd.priv        = chip;
        nmt_d->set            = set;

        if (hardware_ecc) {
            chip->correct_data = s3c2410_nand_correct_data;
            chip->enable_hwecc = s3c2410_nand_enable_hwecc;
            chip->calculate_ecc = s3c2410_nand_calculate_ecc;
            chip->eccmode       = NAND_ECC_HW3_512;
            chip->autooob       = &nand_hw_eccoob;
        } else {
            chip->eccmode       = NAND_ECC_NONE; //修改此行的赋值
        }
    }
    ...

```

到此已经完成了新的内核对 NAND Flash 分区的支持，下面将介绍配置内核的主要选项。

#### 4.3.1.3 配置内核选项

配置内核选项是移植内核过程中很重要的一步，也是非常复杂的一步，配置时一定要细心。由于我们的开发板很接近 Linux 内核中提供的 smdk2410 开发板，所以可以参考 smdk2410 开发板的配置文件，将其默认的配置文​​件拷贝到内核代码的根目录下，然后开始配置内核，操作如下：

```

# cd linux-2.6.10
# cp arch/arm/configs/smdk2410_defconfig .config

```



## # make menuconfig

通常有 4 种主要的配置内核方法：

### 1. make config

提供了一个命令行接口方式来配置内核，它会一个接着一个的询问关于每一个选项，这个方式相对非常繁琐，因为有太多的选项要进行配置，并且你不知道什么时候才能配置结束直到配置完最后一个选项才知道，所以在实践中很少应用该方法。如果已经有了.config 配置文件，它将根据配置文件来设置询问选项的默认值。

### 2. make oldconfig

它会使用一个已有的.config 配置文件，提示行会提示你的是哪些之前你还没有配置过的选项，它与 make config 相比会简单许多，因为它需要配置的选项不再是所有的了，而是.config 文件中还没有配置的选项。

### 3. make menuconfig

显示一个基于文本的图形化终端配置菜单，目前被公认为是使用最广泛的配置内核方式。如果一个.config 文件存在，它将使用该文件设置那些默认的值。

### 4. make xconfig

显示一个基于 X 窗口的配置菜单，用户可以通过图形用户界面(GUI)和鼠标来对内核进行配置，使用该方法时必须支持 X Window 系统。如果一个.config 文件存在，它将使用该文件设置那些默认的值[8]。

下面列出了 Linux2.6.10 内核的主菜单配置选项：

```
| Code maturity level options --->
| General setup --->
| Loadable module support --->
| System Type --->
| General setup --->
| Parallel port support --->
| Memory Technology Devices (MTD) --->
| Plug and Play support --->
| Block devices --->
| Multi-device support (RAID and LVM) --->
| Networking support --->
| ATA/ATAPI/MFM/RLL support --->
| SCSI device support --->
| Fusion MPT device support --->
| IEEE 1394 (FireWire) support --->
| I2O device support --->
| ISDN subsystem --->
| Input device support --->
| Character devices --->
| I2C support --->
| Multimedia devices --->
| File systems --->
| Profiling support --->
| Graphics support --->
| Sound --->
```

- | Misc devices --->
- | USB support --->
- | MMC/SD Card support --->
- | Kernel hacking --->
- | Security options --->
- | Cryptographic options --->
- | Library routines --->

关于内核中每个选项的含义由于内容太多，所以这里就不再介绍，请参考相关的帮助文档。由于我们是基于 smdk2410 开发板的配置选项，所以我们需要在此开发板上修改部分的配置选项来配置我们的内核。

首先，配置可加载模块的支持，具体配置选项如下：

Loadable module support --->

- [\*] Enable loadable module support # 该选项的目的是使内核支持可加载模块，使用 modprobe, lsmod, modinfo, insmod, rmmod 等工具需要，所以必须选择。
- [\*] Module unloading # 卸载模块选项，这里选择该选项。
- [\*] Forced module unloading # 强制性卸载模块选项，如用 rmmod -f 命令强制卸载。
- [ ] Module versioning support (EXPERIMENTAL) # 一般地，我们编译的模块是用于当前运行的内核，选择该选项可以针对其他版本内核编译模块，这里可以不选择。
- [ ] Source checksum for all modules # 检查所有模块的代码，一般不选择。
- [\*] Automatic kernel module loading # 内核在任务中要使用一些被编译为模块的驱动或特性时，先使用 modprobe 命令来加载它，然后该选项自动调用 modprobe 加载需要的模块，所以该选项一定要选择！

注意：在每个选项前有一个方括号，其中 [\*]表示该选项加入内核编译；[ ]表示不选择该选项；[M]表示该选项作为模块编译内核，也就是说可以动态的加载和卸载该模块。

接着加入内核对 S3C2410 DMA（Direct Memory Access，直接内存访问）的支持，具体配置选项如下：

System Type -->

- [\*] S3C2410 DMA support # 该选项来支持S3C2410 DMA特性。

然后在General setup ---> Default kernel command string 菜单下修改默认的内核启动参数，修改后内容如下：

```
noinitrd root=/dev/mtdblock2 init=/linuxrc console= ttySAC0,115200 mem=64M
```

对上面的参数解释一下，mtdblock2代表使用第3个flash分区（也就是我们建立的rootfs分区），用来作根文件系统；console=ttySAC0,115200使kernel启动期间的信息全部输出到串口0上，波特率为115200；Linux 2.6内核对于串口的命名改为ttySAC0，在2.4内核中，串口名为ttyS0，使用时请注意。mem=64M表示内存大小是64M。

同时还需要添加对浮点算法的支持，在菜单下选择：

General setup --->

- [\*] NWFPE math emulation # 支持NWFPE浮点库，在许多情况下要使用，所以最好选上。

接下来做对MTD设备(如flash, ram等芯片)的配置，如下选择配置：

Memory Technology Devices (MTD) -->

- [\*] MTD partitioning support

RAM/ROM/Flash chip drivers -->

- [\*] Detect flash chips by Common Flash Interface (CFI) probe

[\*] Detect nonCFI AMD/JEDECcompatible flash chips

[\*] Support for Intel/Sharp flash chips

[\*] Support for AMD/Fujitsu flash chips

[\*] Support for ROM chips in bus mapping

NAND Flash Device Drivers -->

[\*] NAND Device Support

[\*] NAND Flash support for S3C2410/S3C2440 SoC

为了我们要移植的内核支持devfs (Device Filesystem, 设备文件系统), 以及在启动时能自动加载/dev为devfs。需要针对文件系统的设置, 由于我们目标板上要用的文件系统是cramfs, 所以需要如下配置:

File systems -->

[ ] Second extended fs support #去除对ext2的支持

Pseudo filesystems -->

[\*] /proc file system support

[\*] Virtual memory file system support (former shm fs)

[\*] /dev file system support (OBSOLETE)

[\*] automatically mount at boot (NEW)

Miscellaneous filesystems -->

[\*] Compressed ROM file system support (cramfs)

Network File Systems >

[\*] NFS file system support

除此之外, 还需要配置以下选项来支持S3C2410 RTC、USB和MMC/SD卡驱动, 具体选项如下:

Character devices -->

[\*] Nonstandard serial port support

[\*] S3C2410 RTC Driver

USB Support -->

[\*] Support for Host-side USB

MMC/SD Card Support -->

[\*] MMC Support

[\*] MMC block device driver

以上完成了所有内核相关的基本配置, 然后选择保存, 默认会保存到.config文件。保存结束显示如图4.2所示, 下一步就可以进行编译内核了。

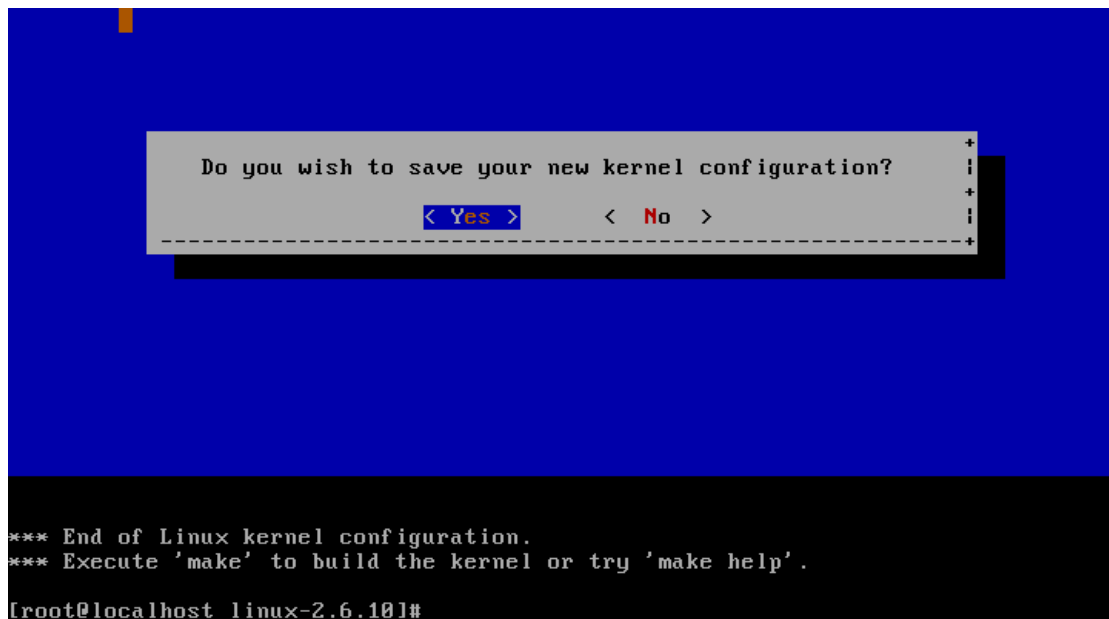


图4.2 内核配置保存

## 4.3.2 内核编译

编译内核通常也需要几个步骤，一是清除以前编译过的残留文件，二是编译内核 image 文件和可加载模块，三是安装模块。在编译内核之前，一定要阅读内核根目录下的 README 文件和 Documentation/Changes 文件，其中该 README 文件告诉我们一个通用的安装内核的方法，Changes 文件主要告诉我们编译和运行该内核需要的最低工具软件列表。由于我们是基于 ARM 处理器平台移植，所以我们还需要阅读 Documentation/arm/README 文件，该文档主要告诉我们编译 ARM Linux 内核的基本方法。通过阅读这些文档，可以让我们更多的了解 Linux 内核的使用方法，下面将具体介绍编译内核的基本方法。

### 4.3.2.1 清除冗余文件

首先进入内核根目录，执行以下操作，其实这个步骤现在是可以不用的，它目的是清除原先编译过而残留的.config 和.o (object 文件)，如果我们是刚下载的源码，那么这一步就可以省了，但是如果你已经编译过多次内核的话，这一步可是一定要的，不然以后可能会出现很多莫名其妙的小问题。

```
# cd linux-2.6.10
# make mrproper
```

注意：该步骤一定要在执行配置命令（make menuconfig/xconfig/config 等）之前做，否则你的.config 配置文件会被清除掉，这样你的内核就不能正确编译了。

### 4.3.2.2 编译内核映像和模块

接下来就要编译内核 image (映像) 和可加载模块了，这里有必要介绍一下常见的 Linux 内核编译相关的命令：

ü make dep: 该命令应用在内核 2.4 或之前，用于建立源文件之间的依赖关系，在执行

内核配置命令之后使用，不过在 2.6 内核中已经取消该命令。该功能被内核配置命令实现了。

- ü **make clean:** 删除前面留下的中间文件，该命令不会删除.config 等配置文件。
- ü **make zImage:** 编译生成 gzip 压缩形式的 image。
- ü **make bzImage:** bzImage 是 big zImage 的缩写，与 make zImage 不同的是它可以生成较大一点的内核。
- ü **make modules:** 编译在配置时选择为模块的，即选项前为[M]的。
- ü **make modules\_install:** 将 make modules 生成的模块文件拷贝到相应的目录。

以上这些命令在内核 2.4 版本之前大部分都需要用，然而在内核 2.6 时，只需要在命令行简单得输入 make 将完成内核 image 的生成和模块的编译。执行完 make 操作，会在 arch/arm/boot 目录下生成 Image 和 zImage 两个内核映像文件，其中 Image 为正常大小的映像文件，而 zImage 为压缩后的映像文件。就编译过程来说，内核 2.6 和以前版本相比，现在 Makefile 的功能已经变得越来越智能，从而使得编译内核和模块变得越来越简单。

### 4.3.2.3 安装模块

由于上面的步骤已经编译了可加载模块，现在需要将编译好的模块进行安装到相应的位置，需要执行的操作如下，默认情况下模块被安装到/lib/modules。由于我们利用交叉编译器编译，并且应用在目标板上，所以需要安装的模块一般不在默认的路径下。通常可以利用选项：INSTALL\_MOD\_PATH=\$TARGETDIR，来指定要安装的位置，此处就是 TARGETDIR 所定义的位置。

```
# make modules_install
```

### 4.3.3 内核下载

内核下载就是将内核映像文件烧写到目标板上，内核下载的前提是已经在目标板上下载了相应的 BootLoader 程序，这里我们以 U-Boot 为例，来讲述下载内核映像的过程。首先在开发使用的宿主机 (PC) 上建立一个 tftp 服务，然后使用超级终端或 DNW 工具启动目标板，然后在 U-Boot 的命令行输入以下命令，具体操作如下：

```
MIKE2410# tftp 0x30008000 zImage
TFTP from server 192.168.1.5; our IP address is 192.168.1.10
Filename 'zImage'.
Load address: 0x30008000
Loading: #####
          #####
          #####
done
Bytes transferred = 890752 (d9780 hex)
```

其中，0x30008000 为指定下载到内存的地址，zImage 就是我们在上节中生成的内核映像文件。以上显示信息表明内核下载是正确的。下载完内核可以通过 bootm 命令来启动内核，具体如下：

```
MIKE2410# bootm 0x30008000
## Booting image at 30008000 ...
```

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

注意，此时下载的内核文件在断电后不能保存，要想固化内核和模块到开发板需要添加根文件系统，所以下一节专门介绍根文件系统的使用。

## 4.4 建立 Linux 根文件系统

根文件系统是 Linux/Unix 系统启动的一个重要组成部分，也是操作系统正常工作时的必要组成部分，在启动时内核需要根文件系统来挂载。在现代 Linux 操作系统中，内核代码映像文件保存在根文件系统中。系统引导启动程序会从这个根文件系统设备上把内核执行代码加载到内存中去运行。本节首先介绍根文件系统的基本知识，然后介绍如何构建自己的根文件系统。

### 4.4.1 根文件系统的基本介绍

本节主要讲述根文件系统的一些基本概念，包括根文件系统的一般目录结构，常见的根 filesystem 和如何选择根文件系统。

#### 4.4.1.1 根文件系统的基本目录结构

在根文件系统的最顶层目录中，每一个目录都有其具体目的和用途。一般建立一个正式的文件系统结构是根据 FHS（Filesystem Hierarchy Standard）定义，FHS，即文件系统结构（层次）标准，它在 Unix /Linux 操作系统的文件系统中用于确定在何处存储何种文件的标准。例如，在 / bin 目录下存放基本命令，在 / etc 目录下存放设置文件等等。表 4.1 将提供一个完整的 FHS 定义的根文件系统顶层目录。

目录名	内 容
<i>bin</i>	提供基本的用户命令库
<i>boot</i>	用于 BootLoader 的静态文件
<i>dev</i>	设备或其他特殊文件
<i>etc</i>	系统配置文件，包括启动文件
<i>home</i>	多个用户的主目录
<i>lib</i>	基本的系统库，例如 C 库，内核模块等
<i>mnt</i>	用于临时挂载的文件系统
<i>opt</i>	可选的软件包
<i>proc</i>	内核虚拟文件系统和进程信息
<i>root</i>	根用户的主目录

目录名	内 容
<i>sbin</i>	基本的系统管理二进制库
<i>tmp</i>	临时文件
<i>usr</i>	它的二级目录里包含许多应用程序和许多有用的文档，包括 X server
<i>var</i>	一些变化的实例和工具等

对于经常使用 Linux 系统的读者来说，这些目录大部分应该很熟悉了。对于一般基于 PC 的 Linux 系统都是多用户的，而嵌入式系统通常都不是针对多用户的，所以根文件系统目录会有较大的不同。例如/boot 目录，这个目录取决于你所使用的 BootLoader 是否能够重新获得内核映象从你的根文件系统在内核启动之前。/home 这个目录在一般嵌入式 Linux 中就很少用到。但通常目录：/bin，/dev，/etc，/lib，/proc，/sbin，/usr 这些都是必须有的。

通常这几个目录对初学者来说容易混淆，那就是/bin，/sbin，/usr/bin 和/usr/sbin。由于这些目录有相似的目的，所以经常会混淆。/bin 目录一般存放对于用户和系统来说都是必须的二进制文件，而/sbin 目录要存放的是只针对系统管理的二进制文件，该目录的文件将不会被普通用户使用。相反，那些不是必要的用户二进制文件存放在/usr/bin 下面，那些不是非常必要的系统管理工具放在/usr/sbin 下。此外，对于一些本地的库也非常类似，对于那些要求启动系统和运行的必须命令要存放在/lib 目录下，而对于其他不是必须的库存放在/usr/lib 目录就可以。

#### 4.4.1.2 常见的根文件系统

常见的根文件系统有：RomFS、JFFS2、NFS、EXT2、RAMDISK、Cramfs 等。下面将分别介绍各自文件系统的特点。

- ü **RomFS:** 是一个空间利用有效的只读文件系统，最初用于 Linux 和基于 Linux 的许多项目。它是一个块文件系统，即利用块（或扇区）访问存储驱动（如磁盘，CD 和 ROM 驱动）。内核 2.4，2.5 和 2.6 代码都支持 RomFS。它有两个特点：一是只读属性，如果你要使用一个 RomFS 文件系统的磁盘，必须预先写该磁盘的驱动程序，让它看起来是一个块设备。二是要求存储空间最小，它是根文件系统中存储空间要求最小的一个，因为没有修改日期，没有访问权限等属性。
- ü **JFFS2:** JFFS2 是 The Journalling Flash File System, version 2 的缩写，JFFS2 是 Flash 嵌入式系统上应用最广的一个日志结构的文件系统。它提供的垃圾回收机制，使得我们不需要马上对擦写越界的块进行擦写，而只需要将其设置一个标志，标明为脏块。当可用的块数不足时，垃圾回收机制才开始回收这些节点。同时，由于 JFFS2 基于日志结构，在意外掉电后仍然可以保持数据的完整性，而不会丢失数据。JFFS2 的不足之处有挂载时间过长和扩展性较差等缺点。
- ü **NFS:** NFS 是 Network File System 的缩写，即网络文件系统。它是 FreeBSD 支持的文件系统中的一种。NFS 允许一个系统在网络上与它人共享目录和文件。通过使用 NFS，用户和程序可以象访问本地文件一样访问远端系统上的文件。它的优点有：一是本地工作站使用更少的磁盘空间，因为通常的数据可以存放在一台机器上而且可以通过网络访问到；二是用户不必在每个网络上机器里头都有一个 home 目录。Home 目录可以被放在 NFS 服务器上并且在网络上处处可用；三是软驱，CDROM，和 Zip® 之类的存储设备可以在网络上面被别的机器使用。这可以减少整个网络上的可移动介质设备的数量。

- ü **EXT2:** EXT2 是 The Second Extended Filesystem 的缩写, 它最初发布于 1993 年 1 月。它仍旧是当前一个主要的文件系统在 Linux 中, 它还可用于 NetBSD, FreeBSD, the GNU HURD, Windows 95/98/NT, OS/2 和 RISC 操作系统等。其特点为存取文件的性能极好, 对于中小型的文件更显示出优势, 这主要得利于其簇块取层的优良设计。其单一文件大小与文件系统本身的容量上限与文件系统本身的簇大小有关, 在一般常见的 x86 电脑系统中, 簇最大为 4KB, 则单一文件大小上限为 2048GB, 而文件系统的容量上限为 16384GB。至于 Ext3 文件系统, 它属于一种日志文件系统, 是对 EXT2 系统的扩展。它兼容 EXT2, 并且从 EXT2 转换成 EXT3 并不复杂。
- ü **RAMDISK:** RAMDISK 存在于 RAM 中, 其存取功能类似块设备。使用 RAMDISK 文件系统, 在系统启动时, 首先把外存(Flash)上的映像文件解压缩到内存中, 构造起 RAMDISK 环境, 然后可以开始运行程序。内核可以在同一时间支持多个活动的 RAMDISK, 在 RAMDISK 上可以使用任何磁盘文件系统。RAMDISK 通常会从经压缩的磁盘文件系统(例如 ext2)加载其内容, 因此内核必须具备从存储设备取出 initrd (initial RAM disk) 映像作为它的根文件系统的能力。启动时, 内核会确认引导选项是否指示有 initrd 的存在, 如果有就会从所选定的存储设备取出文件系统映像放入 RAMDISK, 并且将它安装成根文件系统。
- ü **Cramfs:** Cramfs 被设计为简单且非常小的可压缩的文件系统, 它主要用于较小 ROM 的嵌入式系统。在嵌入式的环境之下, 内存和外存资源都需要节约使用。如果使用 RAMDISK 方式来使用文件系统, 那么在系统运行之后, 首先要把外存(Flash)上的映像文件解压缩到内存中, 构造起 RAMDISK 环境, 才可以开始运行程序。但是它也有很致命的弱点。在正常情况下, 同样的代码不仅在外存中占据了空间(以压缩后的形式存在), 而且还在内存中占用了更大的空间(以解压缩之后的形式存在), 这违背了嵌入式环境下尽量节省资源的要求。使用 Cramfs 就是一种解决这个问题的方式。Cramfs 是一个压缩式的文件系统, 它并不需要一次性地将文件系统的所有内容都解压缩到内存之中, 而只是在系统需要访问某个位置的数据的时候, 马上计算出该数据在 Cramfs 中的位置, 将其实时地解压缩到内存之中, 然后通过对内存的访问来获取文件系统中需要读取的数据。Cramfs 中的解压缩以及解压缩之后的内存中数据存放位置都是由 Cramfs 文件系统本身进行维护的, 用户并不需要了解具体的实现过程, 因此这种方式增强了透明度, 对开发人员来说, 既方便, 又节省了存储空间。

总之, 在实际开发应用中, 根文件系统还有许多种类, 这里就不再一一介绍, 下面将讲述如何选择根文件系统。

#### 4.4.1.3 选择根文件系统

选择一个文件系统用于根文件系统是一个取舍的过程, 最后的决定往往是一个文件系统性能和目标用途的折中。通常选择一个文件系统需要注意以下几个特点:

- ü **可写:** 是否该文件系统能被写数据, 只有当一个文件系统发现有更新的数据时需要可写的文件系统, 通常嵌入式根文件系统并不需要可写的功能。
- ü **可保存:** 是否该文件系统在在重启后能保存修改后的东西, 一般是在有可写功能的基础上才会有该功能。
- ü **可压缩:** 是否挂载的文件系统内容可被压缩, 通常情况下该功能对于嵌入式系统非常有用, 因为它可以节省宝贵的存储空间。
- ü **存在 RAM:** 是否可以在挂载之前将该文件系统的内容第一次从存储设备解压到 RAM 中, 通常许多文件系统被直接从存储设备挂载。文件系统挂载在 RAM 磁盘必须首先



从外存储设备解压到 RAM 中，然后执行挂载。

ü 可恢复：当突然断电时能否恢复文件系统的修改

表 4.2 列出了常见的文件系统特点，在选择文件系统时一定要考虑到这些特点，从而选择最适合的根文件系统。

表 4.2 常见文件系统特点

常见的文件系统	可写	可保存	可恢复	可压缩	存在 RAM
CRAMFS	No	N/A	N/A	Yes	No
JFFS2	Yes	Yes	Yes	Yes	No
JFFS	Yes	Yes	Yes	No	No
Ext2 over NFTL	Yes	Yes	No	No	No
Ext3 over NFTL	Yes	Yes	Yes	No	No
Ext2 over RAM disk	Yes	No	No	No	Yes

总之，在选择根文件系统时，如果你的系统有非常小的 flash，但是有相对比较大的 RAM，建议选择 RAMDISK 作为根文件系统，因为该文件系统可以存在 RAM 磁盘被完全的压缩在外存设备上。如果你的系统有稍微多的 flash 或者你希望保存尽可能多的 RAM 在实际的应用程序运行时，Cramfs 根文件系统是一个不错的选择，尽管 Cramfs 的压缩率低于 RAMDISK，但是它的性能通常对于许多嵌入式应用来说非常充分，因为它不要求永久保存数据。如果你的目标系统需要能够更新在文件系统中，那么 Cramfs 将不是一个可行的选择。如果你需要一个能够经常改变的文件系统，JFFS2 文件系统将是一个很好的选择，尽管 JFFS2 没有 Cramfs 那么高的压缩率，因为 Cramfs 没有垃圾回收和元数据结构，但是 JFFS2 提供了断电可恢复的机制，这点是非常重要的在依靠 flash 存储的设备中。但是当你使用 NAND flash 设备时，JFFS2 是不可行的。

下面将以 Cramfs 根文件系统为例，讲述如何构建自己的根文件系统。

## 4.4.2 建立根文件系统

本小节以 Cramfs 根文件系统为例，讲述 Cramfs 工具包和构建 Cramfs 根文件系统，首先讲述 Cramfs 工具包的使用。

### 4.4.2.1 Cramfs 工具包的使用

从 <http://sourceforge.net/projects/cramfs/> 下载 cramfs-1.1.tar.gz。然后解压并且查看解压后的目录结构如下：

```
# tar xvzf cramfs-1.1.tar.gz
# cd cramfs-1.1
# tree
.
|-- COPYING
|-- GNUmakefile
|-- NOTES
```

```
|-- README
|-- cramfsck.c
|-- linux
|   |-- cramfs_fs.h
|   `-- cramfs_fs_sb.h
`-- mkcramfs.c
1 directory, 8 files
```

利用 **cramfs** 工具包主要是为了生成 **mkcramfs** 和 **cramfsck** 两个工具，其中 **mkcramfs** 工具是用来创建 **cramfs** 文件系统的，而 **cramfsck** 工具则用来进行 **cramfs** 文件系统的释放以及检查。通过执行 **make** 命令将生成 **mkcramfs** 和 **cramfsck** 工具，具体过程如下：

```
# make
# tree
.
|-- COPYING
|-- GNUmakefile
|-- NOTES
|-- README
|-- cramfsck
|-- cramfsck.c
|-- linux
|   |-- cramfs_fs.h
|   `-- cramfs_fs_sb.h
|-- mkcramfs
`-- mkcramfs.c
1 directory, 10 files
```

很明显可以从上面的目录结构中看出编译后生成了 **mkcramfs** 和 **cramfsck** 两个可执行性文件。

#### 4.2.2.2 构建 Cramfs 根文件系统

**Cramfs** 是 Linux Torvalds 编写的只具备最基本特性的文件系统，它非常简单、经过压缩并且只读，主要用于嵌入式系统，它具有以下限制。

- ü 每个文件最大不超过 16MB
- ü 不提供当前目录“.”和上级目录“..”
- ü 文件的 UID 字段只有 16 位，GID 字段只有 8 位
- ü 所有文件的时间戳为 Unix epoch（00:00:00 GMT, January 1, 1970）
- ü 内存分页大小必须是 4096 字节
- ü 文件链接计数器永远是 1

由于构建 **Cramfs** 映射需要使用 **mkcramfs** 和 **cramfsck** 工具，所以首先介绍以下这两个工具的使用方法。

下面是 **mkcramfs** 的命令格式：

```
mkcramfs [-h] [-e edition] [-i file] [-n name] dirname outfile
```

其中个参数解释如下：

-h: 显示帮助信息

- e edition: 设置生成的文件系统版本号
- i file: 将一个文件映像插入这个文件系统之中(只能在 Linux2.4.0 以后的内核版本中使用)
- n name: 设定 cramfs 文件系统的名字
- dirname: 指明需要被压缩的整个目录树
- outfile: 最终输出的文件

cramfsck 的命令格式如下:

```
cramfsck [-hv] [-x dir] file
```

其中个参数解释如下:

- h: 显示帮助信息
- x dir: 释放文件到 dir 所指出的目录中
- v: 输出信息更加详细
- file: 希望测试的目标文件

在本地建立自己根文件系统目录结构, 首先建立名为 myrootfs 的根目录, 然后在其目录下建立所需要的子目录, 具体操作如下:

```
# mkdir myrootfs
# mkdir bin dev etc lib proc sbin tmp usr var
# mkdir usr/bin usr/lib usr/sbin
```

目录建立好以后, 然后就要给各相应的目录复制相应的文件或库, 例如给 lib 目录下要拷贝 glibc 库和内核模块, 给 etc 目录下建立一些系统配置文件, bin 目录下放置常用的命令工具, 下面将介绍在嵌入式系统中常用的 BusyBox 工具来制作命令工具集。

BusyBox 是很多标准 Linux 工具的单个可执行实现。BusyBox 包含了一些简单的工具, 例如 cat 和 echo, 还包含了一些更大、更复杂的工具, 例如 grep、find、mount 以及 telnet; 有些人将 BusyBox 称为 Linux 工具里的瑞士军刀。BusyBox 提供一个公平、完整的 POSIX 环境用于许多小系统, 它是一个可配置的工具, 可以根据需要配置所需要的工具, 目前它已经提供了七十多种 Linux 上标准的工具程序, 仅需要几百 KB 的磁盘空间, 在嵌入式系统中经常被使用。BusyBox 通过将很多必需的工具放入到一个可执行程序, 并让它们可以共享代码中相同的部分, 从而对它们的大小进行了很大程度的缩减, BusyBox 对于嵌入式系统来说是一个非常有用的工具。从 <http://busybox.net/downloads/> 站点可以下载 busybox-1.3.2.tar.bz2, 首先解压该源码包, 然后进行配置, 具体操作如下:

```
# tar xjvf busybox-1.3.2.tar.bz2
# cd busybox-1.3.2
```

关于 BusyBox 的配置方法有以下几种:

- ü make allnoconfig —禁止所有的配置选项在.config 文件
- ü make allyesconfig —启动所有的配置选项在.config 文件
- ü make allbareconfig —启动所有 applets 程序不包括任何子目录下的选项
- ü make config —基于文本的配置方式, 很少用这种方式
- ü make defconfig —设置.config 配置文件为最大通用配置
- ü make menuconfig —交互式图形化配置方式, 该方法应用最广
- ü make oldconfig —配置哪些还没有被配置的选项在.config 配置文件中

最常用的配置方法是 make menuconfig, 当执行该命令时将显示图 4.3 的配置界面, 如果你希望选择尽可能多的配置项, 那么就可以直接使用 make defconfig 命令, 它会自动配置为最大通用的配置选项, 从而使得配置过程变得更加简单、快速。

配置完 BusyBox 后, 接下来就要编译它, 编译之前修改 BusyBox 源代码根目录下的

Makefile 文件，修改的内容是：ARCH=arm; CROSS\_COMPILE=arm-linux-，修改的目的和编译内核时修改是一样的，由于都是基于 ARM 平台，并且都是使用交叉编译环境制作。编译过程非常简单，在命令行输入 make 即可，该编译过程一般需要几分钟完成。编译正确完成后，执行安装命令，即执行 make install 命令。安装完成后，默认情况下，会在\_install 目录下生成 bin, sbin, usr/bin 和 usr/sbin 四个目录，并且在每个目录下都会有许多 busybox 可执行文件的符合连接，busybox 可执行文件存在 bin 目录下。最后将这四个目录下的文件分别拷贝到我们要构建的根文件系统的相应目录下即：myrootfs/bin, myrootfs/sbin, myrootfs/usr/bin 和 myrootfs/sbin 目录下。在准备好要构建的文件系统下所有库和文件后，下面将使用 mkcramfs 工具来制作 Cramfs 根文件系统的映像。

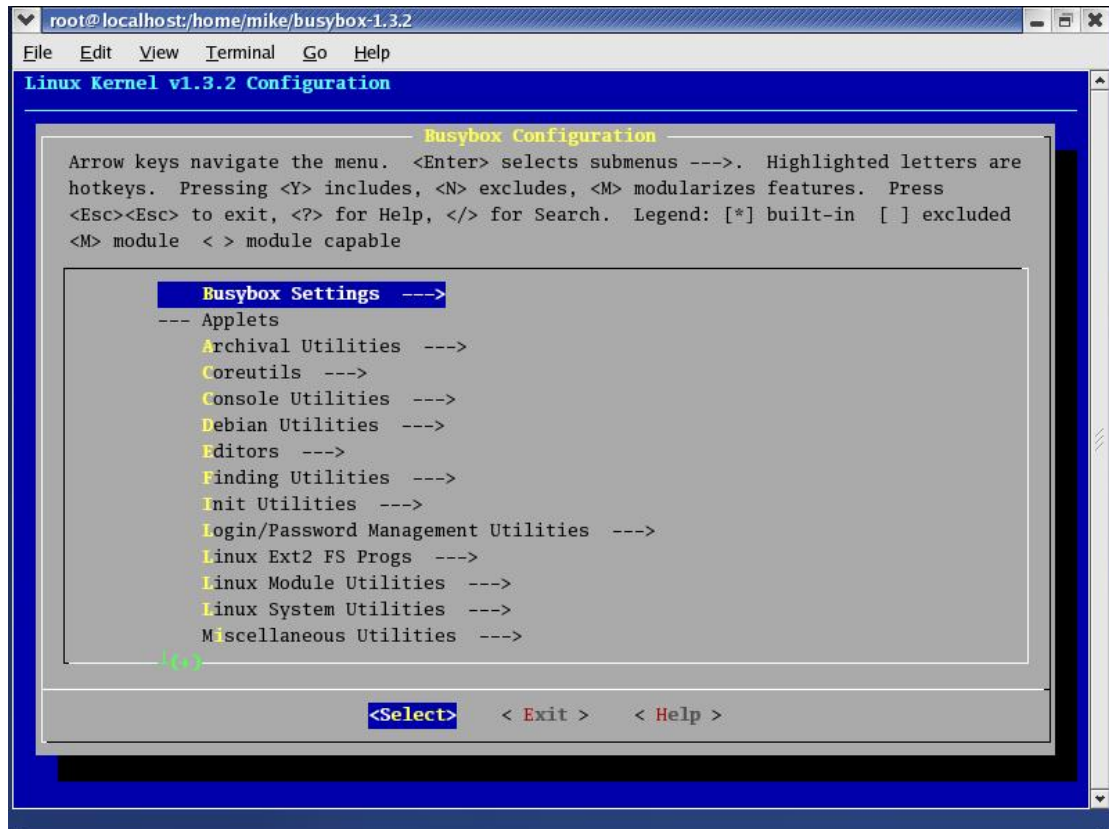


图 4.3 BusyBox 的配置界面

执行以下命令将生成根文件系统的映像文件，下面的命令将生成名为 myrootfs.cramfs 的映像文件。

```
# mkcramfs myrootfs myrootfs.cramfs
```

然后通过 mount 命令来检查添加的根文件系统目录是否正确，通过以下命令实现，下面的显示结果表明，我们构建的根文件系统目录结构正确，接着进行下载我们的 myrootfs.cramfs 根文件映像到开发板上，使用 DNW 或其他工具下载。

```
# mount -o loop myrootfs.cramfs /mnt
# cd /mnt
# tree -L 2 -d
.
|-- bin
|-- dev
|-- etc
|-- lib
```

```
|-- proc
|-- sbin
|-- tmp
|-- usr
|   |-- bin
|   `-- sbin
`-- var

11 directories
```

到此，关于根文件系统的构建已经完成，读者可以阅读内核中 `Documentation/filesystems` 目录下的文档，从而了解更多关于文件系统的知识。

## 4.5 本章小节

本章主要讲述了嵌入式 Linux 系统内核移植的主要过程，本章的内容在实际中应用非常广泛，并且涉及到的内容也非常多。本章首先讲述移植的基本概念，接着讲述内核移植前需要哪些准备工作，然后重点讲述了内核的移植（内核配置、内核编译和内核下载），最后讲述了如何建立自己的根文件系统（包括 **Cramfs** 和 **BusyBox** 工具介绍）。到此为止，第一部分 ARM Linux 系统移植已经介绍完毕，其中主要讲述了嵌入式系统地基础概念，Linux 开发环境，交叉编译工具链的制作，BootLoader 的开发和移植以及 Linux 内核移植，相信读者通过学习和实践，已经对嵌入式系统的构建有了较深入的了解，如果读者希望再更加深入的学习，请参考其它更高级的开发资料。下面将进入本书的第二部分 ARM Linux 系统驱动程序设计，这部分内容是实际工作中非常重要并且必不可少的组成部分，相信很多读者对它会感兴趣。

## 4.6 常见问题

1. 常见移植的几种情况？

参考答案：

- a) 从一个硬件平台移植到另外一个硬件平台
- b) 从一个操作系统移植到另一个操作系统
- c) 从一种软件库环境移植到另一种软件库环境

2. 通常 Linux 内核移植有哪些基本过程？

参考答案：

- a) 对内核进行配置
- b) 对内核进行编译
- c) 将内核下载到目标板

3. 通常根文件系统的目录有哪些？

参考答案：

目录名	内 容
<i>bin</i>	提供基本的用户命令库
<i>boot</i>	用于 BootLoader 的静态文件

目录名	内 容
<i>dev</i>	设备或其他的特殊文件
<i>etc</i>	系统配置文件，包括启动文件
<i>home</i>	多个用户的主目录
<i>lib</i>	基本的系统库, 例如 C 库, 内核模块等
<i>mnt</i>	用于临时挂载的文件系统
<i>opt</i>	可选的软件包
<i>proc</i>	内核虚拟文件系统和进程信息
<i>root</i>	根用户的主目录
<i>sbin</i>	基本的系统管理二进制库
<i>tmp</i>	临时文件
<i>usr</i>	它的二级目录里包含许多应用程序和许多有用的文档，包括 X server
<i>var</i>	一些变化的实例和工具等

#### 5. Cramfs 工具包的作用？

参考答案：

利用 **cramfs** 工具包主要是为了生成 **mkcramfs** 和 **cramfsck** 两个工具，其中 **mkcramfs** 工具是用来创建 **cramfs** 文件系统的，而 **cramfsck** 工具则用来进行 **cramfs** 文件系统的释放以及检查。

#### 6. BusyBox 工具的作用？

参考答案：

**BusyBox** 是很多标准 **Linux** 工具的单个可执行实现。**BusyBox** 包含了一些简单的工具，例如 **cat** 和 **echo**，还包含了一些更大、更复杂的工具，例如 **grep**、**find**、**mount** 以及 **telnet**；有些人将 **BusyBox** 称为 **Linux** 工具里的瑞士军刀。**BusyBox** 提供一个公平、完整的 **POSIX** 环境用于许多小系统，它是一个可配置的工具，可以根据需要配置所需要的工具，目前它已经提供了七十多种 **Linux** 上标准的工具程序，仅需要几百 **KB** 的磁盘空间，在嵌入式系统中经常被使用。**BusyBox** 通过将很多必需的工具放入到一个可执行程序，并让它们可以共享代码中相同的部分，从而对它们的大小进行了很大程度的缩减，**BusyBox** 对于嵌入式系统来说是一个非常有用的工具。

## 第二部分 ARM Linux 设备驱动程序开发

ARM Linux 设备驱动程序开发是本书第二部分要讲述的内容，也是嵌入式系统开发中非常重要的组成部分。通常 Linux 内核把设备驱动程序划分为 3 种类型：字符设备，块设备和网络设备，本书主要讲述这三种类型的设备驱动程序开发。这部分由第 5 章到第 8 章组成，首先第 5 章讲述了 ARM Linux 设备驱动程序的基础知识，包括驱动程序作用、分类以及驱动程序中的一些重要概念，对于初学者来说是非常重要的入门级的一章；接着第 6 章开始讲述 Linux 设备驱动程序中应用最广的一种类型——字符设备驱动，利用触摸屏设备的驱动开发为实例进行了讲述，使读者对 Linux 字符设备驱动的开发有全面的了解；然后第 7 章讲述 Linux 设备驱动程序中应用最广的另一种类型——块设备驱动，以 MMC/SD 卡驱动为例，讲述块设备驱动在 Linux 系统中的实现，使读者对 Linux 块设备驱动开发有较深入的理解；最后第 8 章讲述 Linux 设备驱动程序中应用较多的另一种类型——网络设备驱动，以 CS8900A 以太网卡驱动为例，讲述网络设备驱动在 Linux 系统中的具体实现，使读者对 Linux 网络设备驱动开发有深入的学习。

总之，由于篇幅关系，不可能将每一种 Linux 设备驱动程序都拿来讲述，并且读者也没有必要了解所有驱动程序的实现过程，希望读者通过这些典型实例的分析与学习，对 Linux 驱动开发有较深的理解。

## 第 5 章 ARM Linux 驱动程序开发入门

本章学习目标：

- | 了解驱动程序的作用
- | 熟悉 Linux 设备驱动程序的分类
- | 了解 Linux 内核模块的基本框架
- | 熟悉 Linux 内存与 I/O 端口概念
- | 理解 Linux 并发控制概念，包括自旋锁与信号量等
- | 理解 Linux 阻塞与非阻塞概念
- | 理解 Linux 中断处理过程
- | 熟悉 Linux 内核调试工具 KDB 的使用

### 5.1 嵌入式 Linux 驱动程序介绍

在讲述嵌入式 Linux 驱动程序开发之前，需要对 Linux 驱动程序有个大概了解。本节主要讲述两个方面，一是驱动程序的作用，二是 Linux 驱动程序的分类。首先让我们来看一下驱动程序的作用。

#### 5.1.1 驱动程序的作用

驱动程序从字面上就可以理解为一类程序，而这类程序的目的一般就是驱动硬件正常工作，所以通常所说的驱动程序都是针对特定的硬件来编写的。驱动程序既可以工作在有操作系统的环境下，也可以工作在无操作系统的环境中。通常在做一些简单的硬件控制时，由于功能比较单一，不需要操作系统来管理，所以针对这种情况下的驱动程序相对来说也比较简单，因为它只完成控制特定硬件的功能而不需要考虑其他的并发任务等情况。但是往往作为一个嵌入式系统，它要实现的任务相对比较多，并且比较复杂，所以需要操作系统来对它进行管理，所以在这种情况下，编写驱动程序就要考虑到许多其他任务的并发，任务的优先级以及出现中断情况的处理等，所以通常在带有操作系统的环境下编写驱动程序相对比较复杂，但是这也是实际中应用最广的类型，所以对想从事驱动程序开发的读者来说，这部分的内容是很有必要掌握的。本书是以应用最广范的 Linux 操作系统为参考，讲述如何在 Linux 系统下开发驱动程序。通常驱动程序在 Linux 系统中的位置如图 5.1 所示，在最底层为 Linux 内核，在其上面有文件系统，网络栈和设备驱动，也就是说这三部分都是直接工作在 Linux 内核之上，在最上面一层是我们常见的应用程序，也就是说应用程序的运行是在基于文件系统，网络栈和设备驱动程序之上。所以从图中可以看出设备驱动程序为上层应用程序提供了控制硬件设备的接口，同时它直接与 Linux 内核打交道，对上层应用程序来说，设备驱动程序封装了 Linux 内核，所以应用程序不能直接访问内核程序，从而增加了内核的安全性。



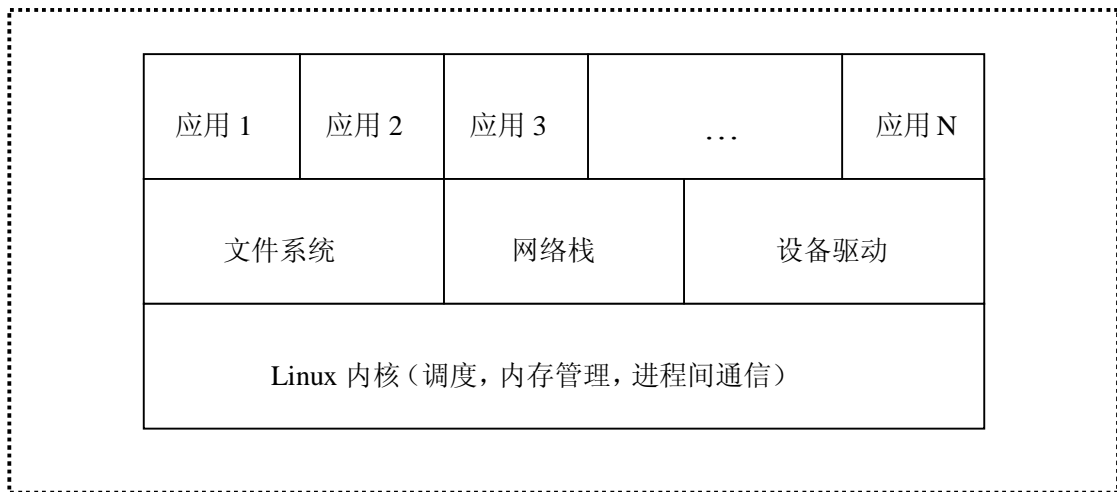


图 5.1 设备驱动程序在 Linux 系统中的位置

从嵌入式驱动开发人员角度看，Linux 驱动程序是直接操控硬件的软件，它通常完成以下功能：

- ü 直接读写硬件寄存器来控制硬件
- ü 读写存储设备，如 Flash，硬盘等
- ü 操作设备缓冲区设备
- ü 操作输入/输出设备，如键盘、打印机等

从嵌入式应用程序开发人员角度看，Linux 驱动程序为应用程序提供了访问硬件设备的应用编程接口（API，Application Programming Interface），它主要提供以下功能：

- ü 应用程序通过驱动程序安全有效的访问硬件设备
- ü 驱动程序作为嵌入式系统的中间一层软件，它隐藏了底层的细节，从而提高了软件的可移植性和可复用性
- ü 驱动程序文件节点可以方便的提供访问权限控制

总之，驱动程序作为嵌入式系统开发的一部分，具有非常重要的意义，通常情况下，Linux 设备驱动程序属于 Linux 操作系统的一部分。

## 5.1.2 Linux 设备驱动程序分类

Linux 系统将设备驱动程序一般分为三类：字符设备、块设备和网络设备，不过他们之间的区别并不严格，下面将介绍这三类驱动程序的各自特点：

### ü 字符设备

字符设备是能够象字节流（文件）一样被访问的设备，字符设备驱动程序通常会实现 open, close, read 和 write 等系统调用函数。系统控制台和并口就是字符设备的例子。通过文件系统节点可以访问字符设备，例如/dev/tty1 和/dev/lp1。字符设备和普通文件系统之间的唯一区别是：普通文件允许在其上来回读写，而大多数字符设备仅仅是数据通道，只能顺序读写。此外，字符设备驱动程序不需要缓冲而且不以固定大小进行操作，它直接从用户进程传输数据，或传输数据到用户进程。

### ü 块设备

所谓块设备是指对其信息的存取以“块”为单位，如常见的光盘、硬磁盘、软磁盘、磁带等，块长取 512 字节或 1024 字节或 4096 字节。块设备和字符设备一样可以通过文件系统节点来访问。在大多数 Unix/Linux 系统中，只能将块设备看作多个块进行访问，一个块设备通常是 1024 字节数据。Linux 允许大家象字符设备那样读取块设

备，即允许一次传输任意数目的字节。块设备和字符设备只在内核内部的管理上有所区别，其中应用程序对于字符设备的每一个 I/O 操作都会直接传递给系统内核对应的驱动程序；而应用程序对于块设备的操作要经过系统的缓冲区管理间接的传递给驱动程序处理。

## ü 网络设备

网络设备驱动在 Linux 系统中是比较特殊的，它不像字符设备和块设备通常实现 read 和 write 等操作，而通常是通过一种套接字（Socket）\*（注 1）等接口来实现。任何网络事务处理都是通过接口实现的，即可以和其他宿主交换数据的设备。通常接口是一个硬件设备，但也可以象 loopback（回路）接口一样是软件工具。网络接口是由内核网络子系统驱动的，它负责发送和接收数据包，而且无需了解每次事务是如何映射到实际被发送的数据包。尽管“telnet”和“ftp”连接都是面向流的，它们使用同样的设备进行传输；但设备并没有看到任何流，仅看到数据报。由于不是面向流的设备，所以网络接口不能象/dev/tty1 那样简单地映射到文件系统的节点上。Unix/Linux 调用这些接口的方式是给它们分配一个独立的名称（如 eth0）。这样的名称在文件系统中并没有对应项。内核和网络设备驱动程序之间的通信与字符设备驱动程序和块设备驱动程序与内核间的通信是完全不一样的。

\*注 1：套接字是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。可以将套接字看作不同主机间的进程进行双向通信的端点，它构成了单个主机内及整个网络间的编程界面。套接字存在于通信域中，通信域是为了处理一般的线程通过套接字通信而引进的一种抽象概念。套接字通常和同一个域中的套接字交换数据（数据交换也可能穿越域的界限，但这时一定要执行某种解释程序）。各种进程使用这个相同的域互相之间用 Internet 协议簇来进行通信。

其实除了这三种类型的驱动程序类型外，还有许多比较特殊的设备驱动程序，如 IIC，USB 等，只不过在实际中大部分驱动程序都属于这三种类型，所以通常所说的 Linux 设备驱动程序开发也就是这三类驱动的开发。

## 5.2 最简单的内核模块举例

Linux 设备驱动模块属于内核的一部分，这在前面已经介绍，Linux 内核的一个模块可以以两种方式被编译和加载，方法一是直接配置编译进 Linux 内核，随同 Linux 启动时加载；方法二是编译成一个可加载和卸载的模块，使用 insmod 加载（modprobe 和 insmod 命令类似，但依赖于相关的配置文件）模块，而 rmmod 用来卸载模块。通过方法二可以控制内核的大小，而模块一旦被插入内核，它就和内核其他部分一样被使用，此外方法二是动态的加载和卸载模块而不需要重新编译整个内核，所以方便调试。

### 5.2.1 编写 Hello World 模块

下面以一个最简单的“Hello World”模块程序为例介绍 Linux 内核模块程序的框架结构，在准备运行该模块程序之前，建议你已安装了 2.6 的内核，并且最好安装的内核版本与你所使用的内核版本是一致的。

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/moduleparam.h>
4
```

```

5  MODULE_LICENSE("Dual BSD/GPL");
6
7  static char *who = "world";
8  static int times = 1;
9  module_param(times, int, S_IRUGO);
10 module_param(who, charp, S_IRUGO);
11
12 static int hello_init(void)
13 {
14     int i;
15     for (i = 0; i < times; i++)
16         printk(KERN_ALERT "(%d) Hello, %s!\n", i, who);
17     return 0;
18 }
19
20 static void hello_exit(void)
21 {
22     printk(KERN_ALERT "Goodbye, %s!\n", who);
23 }
24
25 module_init(hello_init);
26 module_exit(hello_exit);

```

看到上述代码读者会问这是 Linux 内核模块程序吗？回答是肯定的，只不过这个模块程序没有任何实际意义，但是它能够说明 Linux 内核驱动模块程序的一些基部特点。第 1 行包含的头文件是定义 `__init`、`__exit`、`module_init` 和 `module_exit` 必须的宏。第 2 行包含的头文件定义所有模块相关的宏，如 `MODULE_LICENSE`。第 3 行包含的头文件是用来定义模块参数所必须的。现在来具体分析一下这段程序，该内核模块程序定义了两个函数，当该模块被装载进内核时调用 `hello_init` 函数，当该模块被卸载时调用 `hello_exit` 函数。其中 `module_init` 和 `module_exit` 为内核特殊的宏，用来定义模块被装载和卸载时依次分别调用的函数。其中第 5 行用 `MODULE_LICENSE` 宏来声明该模块的许可协议，该模块声明为 BSD（Berkly Software Distribution）和 GPL（General Public License）双重许可协议。内核函数 `printk` 被定义在 Linux 内核中，它行为简单类似于标准 C 库函数 `printf`。也许读者会问为什么不用 `printf` 函数呢？细心的读者应该记得在第二章我们讲述交叉编译工具链的时候就说过，编译 Linux 内核不需要标准 C 库和其他函数库的支持，所以这里就不能使用 `printf` 库函数，然而内核需要自己的打印函数即 `printk`，它通过自身内核运行而不需要 C 库的帮助。内核模块之所以能够调用 `printk` 函数，是因为它是在 `insmod` 装载之后，此时内核模块可以与内核公共函数和变量进行链接，从而可以使用 `printk` 函数。其中 `KERN_ALERT` 宏是标记 `printk` 打印出的字符串优先等级，通常有 8 种消息等级，定义在 `<include/linux/kernel.h>` 文件中，具体定义如下：

```

#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 */
```

这些宏的具体说明如下：

- ü **KERN\_EMERG**  
用于紧急事件发生时的消息说明，一般用在系统崩溃之前的说明消息
- ü **KERN\_ALERT**  
用于需要立即执行的情况
- ü **KERN\_CRIT**  
用在临界状态下的情况，如硬件或软件的故障时
- ü **KERN\_ERR**  
用于报告一个错误条件，设备驱动经常使用它来报告硬件错误
- ü **KERN\_WARNING**  
用于有问题情况下的警告，通常不会警告一个严重的系统问题
- ü **KERN\_NOTICE**  
用于正常的通知，许多安全相关的条件由这个级别的宏报告
- ü **KERN\_INFO**  
用于打印一些相关信息消息，许多驱动程序用这个级别宏打印一些硬件的启动时信息
- ü **KERN\_DEBUG**  
用于调试信息。

上面的程序中还应用到一个重要的概念，即模块参数。在实际设备驱动模块开发中，经常需要传递给硬件设备一些不同的指令来控制不同的功能，其中模块参数就是用来传递给硬件设备指令的。该程序中利用参数 `who` 来指定一个对象，用 `times` 来指定打印“Hello, who!”信息的次数。其中 `module_param` 宏是 Linux 2.6 内核中新增的，该宏被定义在 `<include/linux/moduleparam.h>` 文件中，具体定义如下：

```
#define module_param(name, type, perm) \
    module_param_named(name, name, type, perm)
```

其中，参数 `name` 为要传递的参数变量，参数 `type` 为变量的数据类型，参数 `perm` 为访问参数的权限。

## 5.2.2 编写 Hello World 模块的 Makefile

接下来编写一个 Makefile 文件来编译“Hello World”模块程序，此处 Makefile 的内容编写如下：

```
EXEC = hello
OBJS = hello.o
SRC  = hello.c

INCLUDE = /usr/src/linux-2.6.10/include
CC = arm-linux-gcc
LD = arm-linux-ld
MODCFLAGS = -O2 -Wall -D__KERNEL__ -DMODULE -I$(INCLUDE)
-march=armv4t -c -o
LDFLAGS = -r
```

```
all: $(EXEC)
$(EXEC): $(OBJS)
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

%.o: %.c
    $(CC) $(MODCFLAGS) -mapcs -c $< -o $@

clean:
    -rm -f $(EXEC) *.o *~ core
```

在上面的 Makefile 文件中，INCLUDE 来指定要编译的程序需要的头文件路径，这个路径要根据你的内核所放的位置来定。其中 CC 和 LD 定义使用的编译器和连接器，由于我们希望在 S3C2410 开发板上运行，所以需要使用交叉编译器来执行。MODCFLAGS 变量来指定编译该模块时所用的编译选项，LDFLAGS 变量定义了连接选项。

### 5.2.3 加载和卸载 Hello World 模块

接下来在存放 Makefile 和 “Hello World” 源程序的目录下执行 make 命令，经过编译会在该目录下生成 hello 和 hello.o 的基于 ARM 体系的可执行文件。然后将这些可执行性文件下载到开发板上执行以下命令：

```
# insmod hello who= "world" times=5
```

执行这条命令的作用是，装载该模块到内核，并且传递了两个参数，who 参数传递为 world，times 参数传递为 5。执行这条命令将在控制台显示 5 次 “Hello world!”。

接下来执行卸载模块操作，通过执行以下命令来卸载刚才加载的模块，具体操作如下：

```
# rmmmod hello
Goodbye, world!
```

通过 “Hello World” 一个完整模块程序的学习，读者应该会觉得 Linux 内核模块开发并不是一件难事。的确，Linux 内核开发并不像许多人想象的那样深奥，其实一个庞大的体系结构往往都是由简单的模块组合而成，我相信读者只要下定决心去学，一定能够成为 Linux 开发高手。下面将介绍 Linux 驱动程序开发一些要点。

## 5.3 Linux 驱动程序开发要点

对于初学 Linux 驱动开发的读者来说，首先需要掌握一些 Linux 驱动程序开发相关的基本知识，比如内存与 I/O 端口，并发控制，阻塞与非阻塞操作，中断处理和内核调试等，下面将分别具体讲述这些重要的概念。

### 5.3.1 内存与 I/O 端口

内存与 I/O 端口是 Linux 驱动设备开发经常用到的两个概念，编写驱动程序大多数情况下都是对内存和 I/O 端口的操作。下面将分别介绍内存和 I/O 端口在 Linux 设备驱动程序中的使用。

### 5.3.1.1 内存

对于运行标准的 Linux 内核平台需要提供对 MMU（内存管理单元）的支持，并且 Linux 内核提供了复杂的存储管理系统，使得进程能够访问的内存达到 4GB。这 4GB 的空间被人们分为两个部分，一是用户空间，二是内核空间。用户空间的地址分布是从 0 到 3GB，3GB 到 4GB 空间定义为内核空间。编写 Linux 驱动程序必须知道如何在内核中申请内存，内核中最常用的内存分配和释放函数是 `kmalloc` 和 `kfree`，这两个函数非常类似标准 C 库中的 `malloc` 和 `free`。这两个函数原型如下：

```
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

这两个函数被声明在内核源码 `<include/linux/slab.h>` 文件中，设备驱动程序作为内核的一部分，不能使用虚拟内存，必须利用内核提供的 `kmalloc` 与 `kfree` 来申请和释放内核存储空间。`Kmalloc` 带两个参数，第一个参数（`size`）是要申请的是内存数量；第二个参数（`flags`）用来控制 `kmalloc` 的优先权。其中 `flags` 参数的经常有以下几种：

- ü **GFP\_KERNEL**：它的意思是该内存分配（内部是通过调用 `get_free_pages` 来实现的，所以名字中带 **GFP**）是由运行在内核态的进程调用的。也就是说，调用它的函数是属于某个进程的，使用 **GFP\_KERNEL** 优先允许 `kmalloc` 函数在系统空闲内存低于水平线 `min_free_pages` 时延迟分配函数的返回。当空闲内存太少时，`kmalloc` 函数会使当前进程进入 `sleep`（睡眠）\*（注 2）状态，等待空闲页的出现。
- ü **GFP\_ATOMIC**：并非使用 **GFP\_KERNEL** 优先权后一定正确，有时 `kmalloc` 是在进程上下文之外调用的——比如在中断处理，任务队列处理和内核定时器处理时发生。这些情况下，当前的进程就不应该进入 `sleep` 状态，这时应该就使用优先权 **GFP\_ATOMIC**。原子性（`atomic`）的内存分配允许使用内存的空闲位，而与 `min_free_pages` 值无关。实际上，这个最低水平线值的存在就是为了能满足原子性的请求。但由于内核并不允许通过换出数据或缩减文件系统缓冲区来满足这种分配请求，所以必须还有一些真正可以获得的空闲内存。
- ü **GFP\_USER**：用来分配内存给用户空间的页，当空闲内存太少时，`kmalloc` 函数会使当前进程进入 `sleep` 状态，等待空闲页的出现。
- ü **GFP\_HIGHUSER**：类似 **GFP\_USER**，只是从高处分配内存。
- ü **GFP\_NOIO**：类似 **GFP\_KERNEL**，只是增加了禁止任何 I/O 初始化的限制。
- ü **GFP\_NOFS**：类似 **GFP\_KERNEL**，不允许执行任何文件系统的调用。

\*注 2：睡眠（`sleep`）是一个非常重要的概念在设备驱动程序开发中，当一个进程被置入 `sleep` 状态时，它会被标记为一种特殊状态并从调度器的运行队列中移走，直到某些情况下修改了这个状态，进程才会在 CPU 上调度，也就是运行该进程。进入 `sleep` 状态的进程会被搁置在一边，等待将来的某个事件发生。

关于 `kmalloc` 与 `kfree` 的具体实现，可参考内核源程序中的 `<include/linux/slab.h>` 文件。如果希望分配大一点的内存空间，内核会利用一个更好的面向页的机制，分配页的相关函数有以下三个，这三个函数的定义在 `<mm/page_alloc.c>` 文件中。

- ü `get_zeroed_page(unsigned int gfp_mask);`  
该函数的作用是申请一个新的页，初始化该页的值为零，并返回页的指针。
- ü `_get_free_page(unsigned int flags);`  
该函数类似于 `get_zeroed_page`，但是它不初始化页的值为零。
- ü `_get_free_pages(unsigned int flags, unsigned int order);`  
该函数类似于 `_get_free_page`，但是它可以申请多个页，并且返回的是第一个页的指针。

再介绍一个内存相关的重要概念，虚拟内存空间的分配，以上介绍的内存分配函数都是针对实际的物理内存而言的，但在 Linux 系统中经常会使用虚拟内存的技术，虚拟内存通俗的说就是系统在硬盘上建立的缓冲区，它并不是真正的实际内存，是计算机使用的临时存储器，用来运行所需内存大于计算机具有的内存的程序。说到虚拟内存，首先需要说明一下 Linux 的地址类型，Linux 通常有以下几种地址类型：

ü 用户虚拟地址

这类地址是用户空间编程的常规地址，该地址通常是 32 或 64 位，它依赖于使用的硬件体系结构，并且每一个进程有其自己的用户空间。

ü 物理地址

这类地址是用在处理器和系统内存之间的地址，该地址通常是 32 或 64 位，在有些情况下，32 位系统可以使用更大的物理地址。

ü 总线地址

这类地址用在外围总线和内存之间，通常它们和被 CPU 使用的物理地址一样。一些体系结构可以提供一个 I/O 内存管理单元 (I/OMMU)，它可以重新映射地址在总线和主存之间。总线地址是与体系结构密不可分的。

ü 内核逻辑地址

该类地址是由普通的内核地址空间组成，这些地址映射一部分或全部主存，并且经常被对待如同物理地址。在许多体系结构下，逻辑地址和物理地址之间只是差别一个恒定的偏移量。逻辑地址通常储存一些变量类型，如 long, int, void\* 等。利用 kmalloc 可以申请返回一个内核逻辑地址。

ü 内核虚拟地址

在从内核空间地址映射到物理地址时，内核虚拟地址与内核逻辑地址类似。内核虚拟地址并不一定是线性的，一对一的映射到物理地址。所有的逻辑地址都是内核虚拟地址，但是许多内核虚拟地址却不是逻辑地址。内核虚拟地址通常储存在指针变量中。

这五种地址类型经常被使用在 Linux 系统中，如图 5.2 给出了这几种地址在系统中的逻辑关系。

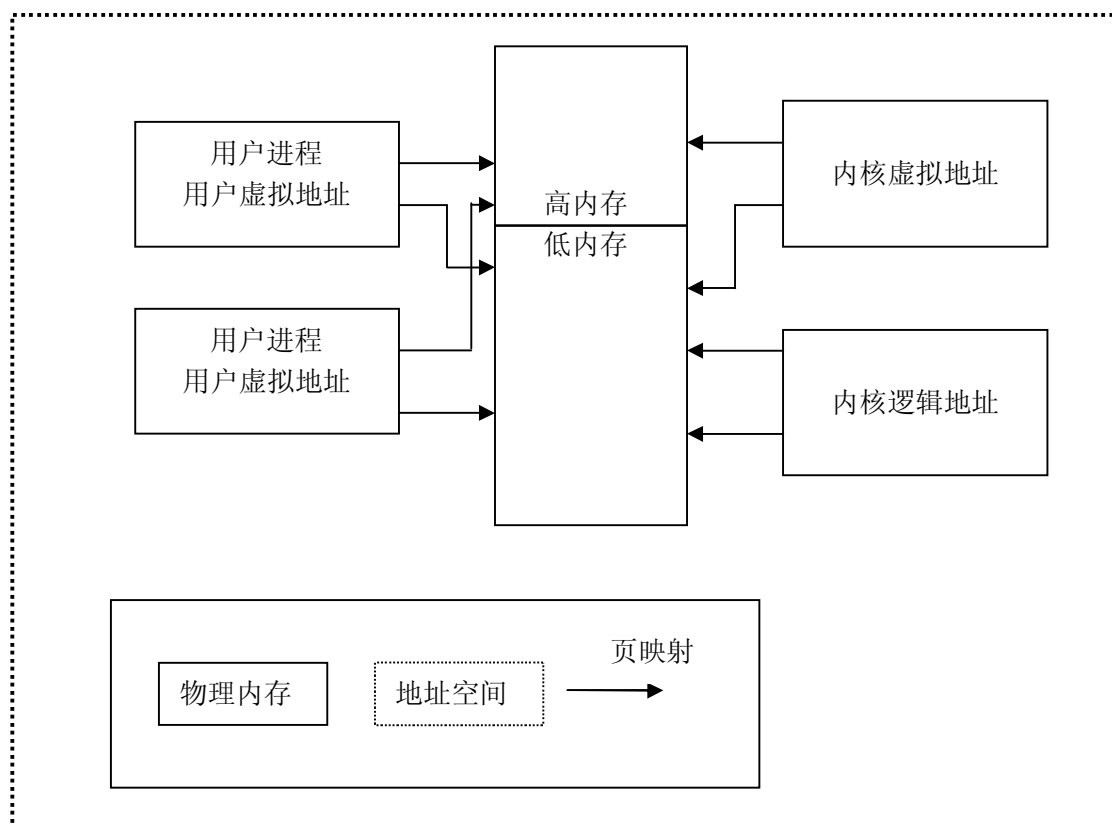


图 5.2 Linux 系统中的地址类型

如果你有一个逻辑地址，通过宏 `__pa(x)` 可以得到相应的物理地址，另外，通过宏 `__va(x)` 可以计算获得物理地址对应的逻辑地址。这两个宏被定义在 `<include/asm/page.h>` 中，具体定义如下：

```
#define __pa(x)          ((unsigned long) (x) - PAGE_OFFSET)
#define __va(x)          ((void *)((unsigned long) (x) + PAGE_OFFSET))
```

其中 `PAGE_OFFSET` 会根据平台的不同定义也就会不同。

现在让我们来学习虚拟内存分配相关的知识，虚拟内存分配函数通常是 `vmalloc`（也有 `vmalloc_32` 和 `__vmalloc`），它分配虚拟地址空间的连续区域。尽管这段区域在物理上可能是不连续的，内核却认为它们在地址上是连续的。分配的内存空间被映射进入内核数据段中，对用户空间是不可见的，这一点上与其他分配技术不同。`Vmalloc` 和其相关函数的原型如下所示，这些函数被包含在 `<include/linux/vmalloc.h>` 头文件中。

- ü `void* vmalloc(unsigned long size);`  
该函数的作用是申请 `size` 大小的虚拟内存空间，发生错误时返回 0，成功时返回一个指向一个大小为 `size` 的线性地址空间的指针。参数 `size` 为申请内存的大小。
- ü `void vfree(void* addr);`  
该函数的作用是释放一个由 `vmalloc`（`vmalloc_32` 或 `__vmalloc`）函数申请内存，释放的内存基地址为 `addr`。
- ü `void *vmap(struct page **pages, unsigned int count, unsigned long flags, pgprot_t prot);`  
该函数的作用是映射一个数组（其内容为页）到连续的虚拟空间中。第一个参数 `pages` 为指向页数组的指针，第二个参数 `count` 为要映射页的个数，第三个参数 `flags` 为传递 `vm_area->flags` 值，第四个参数 `prot` 为映射时页保护。
- ü `void vunmap(void *addr);`  
该函数的作用是释放由 `vmap` 映射的虚拟内存，释放从 `addr` 地址开始的连续的虚拟区



域。

与其他内存分配函数不同的是，`vmalloc` 返回很“高”的地址值，这些地址要高于物理内存的顶部。由于 `vmalloc` 对页表调整后允许用连续的“高”地址访问分配得到的页面，因此处理器是可以访问返回得到的内存区域的。内核能和其他地址一样地使用 `vmalloc` 返回的地址，但程序中用到的这个地址与地址总线上的地址并不相同。用 `vmalloc` 分配得到的地址是不能在微处理器之外使用的，因为它们只有在处理器的分页单元之上才有意义。但驱动程序需要真正的物理地址时，就不能使用 `vmalloc` 了。正确使用 `vmalloc` 函数的场合是为软件分配一大块连续的用于缓冲的内存区域。注意 `vmalloc` 的开销要比 `__get_free_pages` 大，因为它处理获取内存还要建立页表。因此，不值得用 `vmalloc` 函数只分配一页的内存空间。

### 5.3.1.2 I/O 端口

接下来介绍一下经常会用到的 I/O 端口概念，也叫 I/O 寄存器。和硬件打交道离不开 I/O 端口，在 linux 下，操作系统没有对 I/O 端口屏蔽，也就是说，任何驱动程序都可以对任意的 I/O 端口操作，这样就很容易引起混乱。每个驱动程序应该自己避免误用端口。I/O 端口有点类似内存位置，可以用访问内存芯片相同的电信号对它进行读写。但这两者实际上并不一样，端口操作是直接对外设进行的，和内存相比更不灵活，而且有 8 位的端口，也有 16 位的端口和 32 位的端口，不能相互混淆使用。C 语言程序必须调用不同的函数来访问大小不同的端口。

有两个重要的内核函数可以保证驱动程序使用正确的端口，以下两个函数定义在 `<include/linux/ioport.h>` 中。

ü `check_region(unsigned long s, unsigned long n);`

该函数作用是察看系统的 I/O 表，看是否有别的驱动程序占用某一段 I/O 口。第一个参数 `s` 是 I/O 端口的基地址；第二个参数是 I/O 端口占用的范围。返回值为 0 时表示没有占用，非 0 时表示已经被占用。

ü `request_region(start,n,name);`

该函数的作用是如果这段 I/O 端口没有被占用，在我们的驱动程序中就可以使用它。在使用之前，必须向系统注册，以防止被其他程序占用。注册后，在 `/proc/ioports` 文件中可以看到你注册的 I/O 端口。第一个参数 `start` 是 I/O 端口的基地址。第二个参数是 I/O 端口占用的范围。第三个参数是使用这段 I/O 地址的设备名。

根据 CPU 体系结构的不同，CPU 对 I/O 端口的编址方式通常有两种：第一种是 I/O 映射（I/O-mapped）方式，如 X86 处理器为外设专门实现了一个单独的地址空间，称为 I/O 地址空间或 I/O 端口空间，CPU 通过专门的 I/O 指令（如 X86 的 `IN` 和 `OUT` 指令）来访问这一空间的地址单元；第二种是内存映射（Memory-mapped）方式，RISC 指令系统的 CPU（如 ARM，PowerPC 等）通常只实现一个物理地址空间，外设 I/O 端口成为了内存的一部分，此时 CPU 访问 I/O 端口就像访问一个内存单元不需要单独的 I/O 指令。这两种方式在硬件实现上的差异对软件来说是完全可见的，驱动程序开发人员可以将内存映射方式的 I/O 端口和外设内存统一看作是 I/O 内存资源。

I/O 端口的主要作用是用来控制硬件，如何控制硬件？其实就是对 I/O 端口进行具体操作，内核中对 I/O 端口进行操作的函数同样也分为两类，第一类是基于 I/O 端口映射方式的，其定义在体系结构相关的 `<asm/io.h>` 文件中，常用函数如下：

ü `inb(unsigned long port);`

按字节(8 位宽度)读端口。参数 `port` 为要读取的端口号。参数 `port` 在一些平台上定义为 `unsigned long`，而在另一些平台上定义为 `unsigned short`。不同平台上 `inb` 返回值的

类型也不相同。

- ü `outb(unsigned long value, unsigned long port);`  
按字节(8 位宽度)写端口。第一个参数 `value` 是要写进端口的值，第二个参数 `port` 为要写的端口号。
- ü `inw(unsigned long port);`  
按字(16 位宽度)读端口。参数 `port` 为要读取的端口号。
- ü `outw(unsigned long value, unsigned long port);`  
按字(16 位宽度)写端口。第一个参数 `value` 是要写进端口的值，第二个参数 `port` 为要写的端口号。
- ü `inl(unsigned long port)`  
按双字(32 位宽度)读端口。参数 `port` 为要读取的端口号。
- ü `outl(unsigned long value, unsigned long port)`  
按双字(32 位宽度)写端口。第一个参数 `value` 是要写进端口的值，第二个参数 `port` 为要写的端口号。

第二类 I/O 端口操作函数是基于内存映射方式的，常用函数如下：

- ü `readb(const volatile void __iomem *addr);`  
按字节(8 位宽度)读指定地址单元。参数 `addr` 为要读取的内存地址。
- ü `writeb(unsigned char b, volatile void __iomem *addr);`  
按字节(8 位宽度)写数据到指定地址单元。第一个参数 `b` 为要写的值，第二个参数 `addr` 为要写到的内存地址。
- ü `readw(const volatile void __iomem *addr)`  
与 `readb` 类似，只是按字(16 位宽度)读指定地址单元。参数 `addr` 为要读取的内存地址。
- ü `writew(unsigned short b, volatile void __iomem *addr);`  
与 `writeb` 类似，只是按字(16 位宽度)写到指定地址单元。第一个参数 `b` 为要写的值，第二个参数 `addr` 为要写到的内存地址。
- ü `readl(const volatile void __iomem *addr);`  
与 `readb` 类似，只是按双字(32 位宽度)读指定地址单元。参数 `addr` 为要读取的内存地址。
- ü `writel(unsigned int b, volatile void __iomem *addr);`  
与 `writeb` 类似，只是按双字(32 位宽度)写到指定地址单元。第一个参数 `b` 为要写的值，第二个参数 `addr` 为要写到的内存地址。

总之，内存与 I/O 端口是 Linux 驱动程序开发中非常重要的两个概念，所以希望读者对他们有较深入的理解，以便更好的学习 Linux 设备驱动程序开发。

### 5.3.2 并发控制

在驱动程序中经常会出现这种情况，即多个进程同时访问相同的资源时可能会出现竞态（race condition），即竞争资源状态，因此我们必须对共享资料进行并发控制。Linux 内核中解决并发控制最常用的方法是自旋锁（Spinlocks）和信号量（Semaphores）。下面将分别介绍这两种方法。

### 5.3.2.1 自旋锁 (Spinlocks)

自旋锁作为保护数据并发访问的一种重要方法，在 Linux 内核及驱动程序编写中经常采用。自旋锁的名字来自它的特性，在试图加锁的时候，如果当前锁已经处于“锁定”状态，加锁进程就进行“旋转”，用一个死循环测试锁的状态，直到成功的取得锁。自旋锁的这种特性避免了调用进程的挂起，用“旋转”来取代进程切换。而我们知道上下文切换需要一定时间，并且会使高速缓冲失效，对系统性能影响是很大的，所以自旋锁在多处理器环境中非常方便。当然，被自旋锁所保护的“临界代码”一般都比较短，否则就会浪费过多的 CPU 资源。自旋锁是一个互斥现象的设备，它只能有两个值：locked（锁定）或 unlocked（解锁）。它通常作为一个整型值的单个位来实现。在任何时刻，自旋锁只能有一个保持者，即在同一时刻只能有一个进程获得锁。

自旋锁在内核中的相关函数如下所示：

自旋锁的实现函数有：

ü spin\_lock(spinlock\_t \*lock);

该函数用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则它将自旋在那里，直到该自旋锁的保持者释放，这时它获得锁并返回。总之，只有它获得锁才返回。

ü spin\_lock\_irqsave(spinlock\_t \*lock, unsigned long flags);

该函数获得自旋锁的同时把标志寄存器的值保存到变量 flags 中并失效本地中断。

ü spin\_lock\_irq(spinlock\_t \*lock);

该函数类似于 spin\_lock\_irqsave，只是该函数不保存标志寄存器的值。禁止本地中断并获取指定的锁。

ü spin\_lock\_bh(spinlock\_t \*lock);

类似 spin\_lock，该函数在获得自旋锁之前要求禁止软件中断，而使硬件中断开启着。

此外，还有两个非阻塞（nonblocking）自旋锁函数，关于非阻塞的概念将在后面的小节介绍。

ü spin\_trylock(spinlock\_t \*lock);

该函数尽力获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待 lock 被释放。

ü spin\_trylock\_bh(spinlock\_t \*lock);

该函数如果获得了自旋锁，它也将失效本地软中断。如果得不到锁，它什么也不做。因此如果得到了锁，它等同于 spin\_lock\_bh，如果得不到锁，它等同于 spin\_trylock。如果该宏得到了自旋锁，需要使用 spin\_unlock\_bh 来释放。

类似的还有以下释放自旋锁的函数：

ü spin\_unlock(spinlock\_t \*lock);

该函数释放自旋锁 lock，它与 spin\_trylock 或 spin\_lock 配对使用。如果 spin\_trylock 返回假，表明没有获得自旋锁，因此不必使用 spin\_unlock 释放。

ü spin\_unlock\_irqrestore(spinlock\_t \*lock, unsigned long flags);

该函数释放自旋锁 lock 的同时，也恢复标志寄存器的值为变量 flags 保存的值。它与 spin\_lock\_irqsave 配对使用。

ü spin\_unlock\_irq(spinlock\_t \*lock);

该函数释放自旋锁 lock 的同时，并激活本地中断。它与 spin\_lock\_irq 配对应用。

ü spin\_unlock\_bh(spinlock\_t \*lock);

该函数释放自旋锁 lock 的同时，也使能本地的软中断。它与 spin\_lock\_bh 配对使用。

关于读自旋锁的 API 函数有：

- ü read\_lock(rwlock\_t \*lock);
- ü read\_lock\_irqsave(rwlock\_t \*lock, unsigned long flags);
- ü read\_lock\_irq(rwlock\_t \*lock);
- ü read\_lock\_bh(rwlock\_t \*lock);
- ü read\_unlock(rwlock\_t \*lock);
- ü read\_unlock\_irqrestore(rwlock\_t \*lock, unsigned long flags);
- ü read\_unlock\_irq(rwlock\_t \*lock);
- ü read\_unlock\_bh(rwlock\_t \*lock);

关于写自旋锁的 API 函数有：

- ü write\_lock(rwlock\_t \*lock);
- ü write\_lock\_irqsave(rwlock\_t \*lock, unsigned long flags);
- ü write\_lock\_irq(rwlock\_t \*lock);
- ü write\_lock\_bh(rwlock\_t \*lock);
- ü write\_trylock(rwlock\_t \*lock);
- ü write\_unlock(rwlock\_t \*lock);
- ü write\_unlock\_irqrestore(rwlock\_t \*lock, unsigned long flags);
- ü write\_unlock\_irq(rwlock\_t \*lock);
- ü write\_unlock\_bh(rwlock\_t \*lock);

总之，关于自旋锁的 API 函数还有许多，这里就不再一一介绍了，下面我们可以看一下自旋锁的通常用法。

```
.....
spinlock_t my_slock = SPIN_LOCK_UNLOCKED;
spin_lock(&my_slock);
    /*临界区*/
spin_unlock(&my_slock);
.....
```

因为自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区，这就为多处理器提供了防止并发访问所需的保护机制，但是在单处理器上，编译的时候不会加入自旋锁。它仅仅被当作一个设置内核抢占机制是否被启用的开关。注意，Linux 内核实现的自旋锁是不可递归的，这一点不同于自旋锁在其他操作系统中的实现，如果你得到一个你正持有的锁，你必须自旋，等待你自己释放这个锁，但是你处于自旋忙等待中，所以永远没有机会释放锁，于是你就被自己锁死了，一定要注意这种情况的发生。

由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁可以用在中断处理程序中，但是在使用时一定要在获取锁之前，首先禁止本地中断（当前处理器上的中断），否则中断处理程序就可能打断正持有锁的内核代码，有可能会试图力争用这个已经被持有的自旋锁。这样一来，中断处理程序就会自旋，等待该锁重新可用，但是锁的持有者在这个中断处理程序执行完毕之前不可能运行，这就会造成双重请求死锁。

自旋锁与下半部（中断程序下半部，在本章后面会介绍），由于下半部可以抢占进程上下文中的代码，所以当下半部和进程上下文共享数据时，必须对进程上下文中的共享数据进行保护，所以需要加锁的同时还要禁止下半部执行。同样，由于中断处理程序可以抢占下半部，所以如果中断处理程序和下半部共享数据，那么就必须在获取恰当的锁的同时还要禁止中断。对于软中断，无论是否同种类型，如果数据被软中断共享，那么它必须得到锁的保护，因为同种类型的两个软中断也可以同时运行在一个系统的多个处理器上。但是同一个处理器

上的一个软中断绝不会抢占另一个软中断，因此这种情况下根本不需要禁止下半部。

### 5.3.2.2 信号量（Semaphores）

现在介绍信号量，信号量是一个很好理解的概念，一个信号量是一个结合一对函数的整型值，这对函数通常被称为 P 操作和 V 操作。一个进程希望进入一个临界区域时将调用 P 操作在相应的信号量上，如果这个信号量的值大于 0，这个值将被减 1 同时该进程继续进行。相反如果这个信号量的值等于或小于 0，则该进程将等待特别的进程释放该信号量，然后才能执行。解锁一个信号量通过调用 V 操作来完成，这个函数的作用正好与 P 操作相反，调用 V 操作时信号量的值将增加 1，如果需要，同时唤醒哪些等待的进程。当信号量用于互斥现象（多个进程在同时运行一个相同的临界区域）时，此时信号量的值被初始化为 1。信号量只能在一个时刻被一个进程或线程拥有，一个信号量使用在这种模式下通常被称为互斥体（mutex，mutual exclusion 的缩写）体。几乎所有的信号量在 Linux 内核中都是用于互斥现象。

信号量和互斥体的实现相关函数有：

- ü `sema_init(struct semaphore *sem, int val);`  
该函数用来初始化一个信号量。其中第一个参数 `sem` 为指向信号量的指针，`val` 为赋给该信号量的初始值。
- ü `DECLARE_MUTEX(name);`  
该宏声明一个信号量 `name` 并初始化它的值为 1，即声明一个互斥锁。
- ü `DECLARE_MUTEX_LOCKED(name);`  
该宏声明一个互斥锁 `name`，但把它的初始值设置为 0，即锁在创建时就处在已锁状态。因此对于这种锁，一般是先释放后获得。
- ü `init_MUTEX(struct semaphore *sem);`  
该函数被用在运行时初始化（如在动态分配互斥体的情况下），其作用类似 `DECLARE_MUTEX`，即它把信号量 `sem` 的值设置为 1。
- ü `init_MUTEX_LOCKED(struct semaphore *sem);`  
该函数也用于初始化一个互斥锁，但它把信号量 `sem` 的值设置为 0，即一开始就处在已锁状态。
- ü `down(struct semaphore *sem);`  
该函数用于获得信号量 `sem`，它会导致睡眠，因此不能在中断上下文（包括 IRQ 上下文和软中断上下文）使用该函数。该函数将把 `sem` 的值减 1，如果信号量 `sem` 的值非负，就直接返回，否则调用者将被挂起，直到别的任务释放该信号量才能继续运行。
- ü `down_interruptible(struct semaphore *sem);`  
该函数功能与 `down` 类似，不同之处为 `down` 不会被信号（signal）打断，但 `down_interruptible` 能被信号打断，因此该函数用返回值来区分是正常返回还是被信号中断，如果返回 0，表示获得信号量正常返回，如果被信号打断，返回 `-EINTR`。
- ü `down_trylock(struct semaphore *sem);`  
该函数试着获得信号量 `sem`，如果能够立刻获得，它就获得该信号量并返回 0，否则，表示不能获得信号量 `sem`，返回值为非 0 值。因此它不会导致调用者睡眠，可以在中断上下文使用。
- ü `up(struct semaphore *sem);`  
该函数释放信号量 `sem`，即把 `sem` 的值加 1，如果 `sem` 的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者。

自旋锁和信号量有很多相似之处又一些本质的不同，其相同之处有：一是它们对互斥来说都是非常有用的工具。二是在任何时刻最多只能有一个线程获得自旋锁或信号量。不同之处有：一是自旋锁可在不能 `sleep` 的代码中使用，如在中断服务程序（ISR）中使用，而信号量不可以。二是自旋锁和信号量的实现机制不一样。三是通常自旋锁被用在多处理器系统。总之，通常自旋锁适合保持时间非常短的情况，它可以在任何上下文中使用，而信号量用于保持时间较长的情况，只能在进程上下文中使用。如果被包含的共享资源需要在中断上下文访问时，就只能使用自旋锁。

针对读者和写者信号量的相关函数有：

ü `init_rwsem(struct rw_semaphore *sem);`

其中 `rwsem` 是 `reader/writer semaphore` 的缩写，它是 Linux 内核提供了一种特殊信号量类型，通过这个函数在运行时初始化 `rwsem` 的对象。

ü `down_read(struct rw_semaphore *sem);`

读者调用该函数来得到读写信号量 `sem`。该函数会导致调用者睡眠，因此只能在进程上下文使用。

ü `down_read_trylock(struct rw_semaphore *sem);`

该函数类似于 `down_read`，只是它不会导致调用者睡眠。它尽力得到读写信号量 `sem`，如果能够立即得到，它就得到该读写信号量，并且返回 1，否则表示不能立刻得到该信号量，返回 0。因此，它也可以在中断上下文使用。

ü `up_read(struct rw_semaphore *sem);`

读者使用该函数释放读写信号量 `sem`。它与 `down_read` 或 `down_read_trylock` 配对使用。如果 `down_read_trylock` 返回 0，不需要调用 `up_read` 来释放读写信号量，因为根本就没有获得信号量。

ü `down_write(struct rw_semaphore *sem);`

写者使用该函数来得到读写信号量 `sem`，它也会导致调用者睡眠，因此只能在进程上下文使用。

ü `down_write_trylock(struct rw_semaphore *sem);`

该函数类似于 `down_write`，只是它不会导致调用者睡眠。该函数尽力得到读写信号量，如果能够立刻获得，就获得该读写信号量并且返回 1，否则表示无法立刻获得，返回 0。它可以在中断上下文使用。

ü `up_write(struct rw_semaphore *sem);`

写者调用该函数释放信号量 `sem`。它与 `down_write` 或 `down_write_trylock` 配对使用。如果 `down_write_trylock` 返回 0，不需要调用 `up_write`，因为返回 0 表示没有获得该读写信号量。

ü `downgrade_write(struct rw_semaphore *sem);`

该函数用于把写者降级为读者，这有时是必要的。因为写者是排他性的，因此在写者保持读写信号量期间，任何读者或写者都将无法访问该读写信号量保护的共享资源，对于那些当前条件下不需要写访问的写者，降级为读者将使得等待访问的读者能够立刻访问，从而不但增加了并发性而且提高了效率。

一个读者和写者信号量（`rwsem`）允许一个写入者或多个读取者拥有该信号量。写者拥有更高的优先级，当某个给定写者试图进入临界区域，在所有写者完成其工作之前，不允许读者获得访问该信号量。如果有大量写者竞争该信号量，则会导致读者长时间被拒绝访问。所以最好在很少需要写访问且写者只会短暂拥有信号量的时候使用 `rwsem`。

### 5.3.3 阻塞（Blocking）与非阻塞（Nonblocking）

阻塞（Blocking）和非阻塞（Nonblocking）是开发设备驱动程序时必须考虑的两个方面，下面将具体介绍这两个概念，同时也会介绍相关的异步通知（Asynchronous Notification）概念。

#### 5.3.3.1 阻塞（Blocking）与非阻塞（Nonblocking）操作

阻塞操作是指在执行设备操作时，若不能获得资源则进程挂起，直到满足可操作的条件再进行操作。被挂起的进程进入 `sleep` 状态，被从调度器的运行队列中移走，直到等待的条件被满足。非阻塞操作是在不能进行设备操作时并不挂起，它会立即返回，使得应用程序可以快速查询状态。在处理非阻塞型文件时，应用程序在调用 `stdio` 函数时必须小心，因为很容易把一个非阻塞操作返回值误认为是 EOF（End of File，文件结束符），所以必须始终检查 `errno`（错误类型）。在内核中定义了一个非阻塞标志的宏，即 `O_NONBLOCK`，通常只有 `read`、`write` 和 `open` 文件操作受非阻塞标志的影响。在 Linux 驱动程序中，我们可以使用等待队列（`wait queue`）来实现阻塞操作。等待队列很早就被用在 Linux 内核中了，它以队列为基础数据结构，与进程调度机制紧密结合，能够实现重要的异步通知（Asynchronous Notification）机制。下面将具体介绍异步通知的概念。

#### 5.3.3.2 异步通知（Asynchronous Notification）

什么是异步通知？异步通知是指一旦设备准备就绪，则该设备会主动通知应用程序，这样应用程序就不需要不断的查询设备状态，通常把异步通知称为信号驱动的异步 I/O（SIGIO），这点类似于硬件上的中断。下面将给出一段应用异步通知的简单例子，光盘中有该程序的参考代码，文件名为 `sigio.c`。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>

#define MAX_LEN 200
void input_event(int num) //定义一个输入字符的异步事件
{
    char data[MAX_LEN];
    int len;

    len=read(STDIN_FILENO,&data,MAX_LEN); //读取 STDIN_FILENO 上的输入
    data[len]=0;
    printf("Input string is :%s\n",data);
}
main()
```

```

{
    int oflags;

    signal(SIGIO,input_event); //启动信号驱动机制
    fcntl(STDIN_FILENO,F_SETOWN,getpid());
    oflags=fcntl(STDIN_FILENO,F_GETFL);
    fcntl(STDIN_FILENO,F_SETFL,oflags|FASYNC);

    while(1); //该循环是非常必要的，如果没有该程序将立即执行结束。
}

```

这段程序的作用就是实现一个简单的异步通知机制，当你在控制台输入一段字符串时，它将显示你输入的字符串，当你没有输入任何字符时，它将一直在执行循环操作，也就是说只有等你发送一个异步事件时，它将执行该异步事件。

使用非阻塞 I/O 的应用程序也经常使用 poll、select 和 epoll 系统调用，这三个函数的功能是一样的，即都允许进程决定是否可以对一个或多个打开的文件做非阻塞的读取和写入。这些调用也会阻塞进程，直到给定的文件描述符集合中的任何一个可读取或写入。poll、select 和 epoll 用于查询设备的状态，以使用户程序是否能对设备进行非阻塞的访问，他们都需要设备驱动程序中的 poll 函数支持。驱动程序中 poll 函数最主要的一个 API 函数是 poll\_wait，其原型如下：

Ü poll\_wait(struct file \* filp, wait\_queue\_head\_t \* wait\_address, poll\_table \*p);

该函数并不阻塞，而是把当前任务添加到指定的一个等待列表中，真正的阻塞动作是在 select/poll 函数中完成的。该函数的作用是把当前进程添加到 p 参数指定的等待列表（poll\_table）中，第一个参数 filp 是文件指针，第二个参数 wait\_address 是睡眠队列的头指针地址，第三个参数 p 是指定的等待队列。

阻塞与非阻塞操作是驱动程序开发中经常要考虑的两个操作，关于阻塞与非阻塞的具体使用会在后面章节的实例中出现，希望在这里读者对这些常用的概念有所了解。

### 5.3.4 中断处理

设备的许多工作通常与处理器的速度完全不同，并且总是要比处理器慢，这种让处理器等待外部事件的情况会明显降低处理器的效率，所以必须有一种方法可以让设备在产生某个事件时通知处理器，这种方法被称为中断。一个中断在 Linux 系统中仅仅是一个信号，当硬件需要获得处理器对它的关注时就可以发送这个中断信号。Linux 处理中断的方式在很大程度上与它在用户空间处理信号是一样的，通常一个驱动程序只需要为它自己设备的中断注册一个处理例程，并且在中断到达时进行正确的处理。

#### 5.3.4.1 Linux 中断及其相关函数

与 Linux 设备驱动程序中断处理相关的函数首先是申请和释放 IRQ（中断请求）函数，即 request\_irq 和 free\_irq，这两个非常重要的中断函数原型如下，在头文件 <include/linux/interrupt.h>文件中声明。

Ü int request\_irq(unsigned int irq, irqreturn\_t (\*handler)(int irq, void \*dev\_id, struct pt\_regs \*regs), unsigned long flags, const char \*dev\_name, void \*dev\_id);



该函数的作用是注册一个 IRQ，其中参数 `irq` 是要申请的硬件中断号，参数 `handler` 是向系统登记的中断处理函数，是一个回调函数，中断发生时系统调用这个函数，参数 `dev_id` 是设备的 ID，参数 `flags` 是中断处理的属性，若设置为 `SA_INTERRUPT`，表明中断处理程序是 FRQ（快速中断请求），FRQ 程序被调用时屏蔽所有中断，而 IRQ 程序被调用时不屏蔽 FRQ。若设为 `SA_SHIRQ`，则多个设备共享中断，`dev_id` 在中断共享时会用到。参数 `dev_name` 是定义传递给 `request_irq` 的字符串，用来在 `/proc/interrupts` 中显示中断的拥有者。

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

该函数的作用是释放一个 IRQ，一般是在退出设备或关闭设备时调用。

Linux将中断分为两个部分：上半部和下半部。上半部的功能是注册中断，当一个中断发生时，它进行相应地硬件读写后就把中断处理函数的下半部挂到该设备的下半部执行队列中去。因此上半部执行速度很快，可以服务更多的中断请求。但是仅有中断注册是不够的，因为中断事件可能很复杂，因此引出了下半部，用它来完成中断事件的绝大多数任务。上半部和下半部最大的不同是下半部是可中断的，而上半部是不可中断的，下半部完成了中断处理程序的大部分工作，所以通常比较耗时，因此下半部由系统自行安排运行，不在中断服务上下文中执行。Linux实现下半部的机制主要是tasklet和工作队列。Tasklet是一个可以在由系统决定的安全时刻在软件中断上下文被调用运行的特殊函数，它们可以被多次调用运行，但tasklet的调用并不会累积，也就是说只会运行一次，即使在激活tasklet的运行之前重复请求该tasklet的运行也是这样。Tasklet运行时可以有其他中断发生，因此在tasklet和中断服务程序之间的锁还是需要的。必须使用宏`DECLARE_TASKLET(name, func, data)`来声明tasklet，其中`name`是给tasklet起的名字，`func`是执行tasklet时调用的函数，`data`是一个用来传递给tasklet函数的值。一些设备可以在很短时间内产生多次中断，例如键盘扫描中断，所以在下半部被执行前肯定会有多次中断发生，驱动程序必须对这种情况有所准备，通常利用tasklet可以记录自从上次被调用产生多少次中断，从而让系统知道还有多少工作需要完成。工作队列函数运行在进程上下文中，因此可在必要时sleep，工作队列的中断服务程序和tasklet版本非常类似，唯一不同就是它调用`schedule_work`来调度下半部处理，而tasklet使用`tasklet_schedule`来调度下半部处理。

### 5.3.4.2 ARM 中断处理

通常 ARM 中将中断又叫异常（Exception），当中断或异常发生时，系统执行完当前指令后，将跳转到相应的异常中断处理程序处执行。当异常中断处理程序执行完成后，程序返回到发生中断的指令的下一条指令处执行。在进入异常中断处理程序时，要保存被中断的程序的执行现场。从异常中断处理程序退出时，要恢复被中断的程序的执行现场。ARM 体系中通常在存储地址的低端固化了一个 32 字节的硬件中断向量表，用来指定各种异常中断及其处理程序的对应关系。当一个异常出现以后，ARM 微处理器会执行以下几步操作：

- ❏ 保存处理器当前状态、中断屏蔽位以及各条件标志位；
- ❏ 设置当前程序状态寄存器（CPSR）中相应的位；
- ❏ 将寄存器 `lr_mode`（当异常出现或调用函数时保存下一条指令的位置）设置成返回地址；
- ❏ 将程序计数器(PC)值设置成该异常中断的中断向量地址，从而跳转到相应的异常中断处理程序处执行。

在接收到中断请求以后，ARM 处理器内核会自动执行以上四步，程序计数器 PC 总是跳转到相应的固定地址。然而从异常中断处理程序中返回也会有一些操作，主要包括下面两

- ü 恢复被屏蔽的程序的处理器状态；
- ü 返回到发生异常中断的指令的下一条指令处继续执行。

The diagram illustrates the interrupt flow process:

- 多个外围中断源** (Multiple peripheral interrupt sources) send signals to the **地址映射中断控制器** (Address mapping interrupt controller).
- The **地址映射中断控制器** sends **nIRQ** and **nFIQ** signals to the **ARM** processor.
- The **ARM** processor reads the controller registers to find the **IRQ/FIQ** interrupt source.
- The **ARM** processor writes to the peripheral registers to clear the corresponding interrupt source.

当多个中断产生时，FIQ 优先级高于 IRQ，处理 FIQ 时禁止 IRQs，IRQs 将不会被响应直到 FIQ 处理完成。FIQs 的设计使中断处理尽可能的快，FIQ 模式有 5 个额外的私有寄存器 (r8-r12)，中断处理必须保护其使用的非私有寄存器，可以有多个 FIQ 中断源，但是考虑到系统性能应避免嵌套。

```

AREA SWI_Area, CODE, READONLY ;声明该代码的属性

EXPORT SWI_Handler
IMPORT C_SWI_Handler
T_bit EQU 0x20      ;定义一个位来判断指令模式

SWI_Handler
    STMFD    sp!, {r0-r3, r12, lr}  ; 保存相应的寄存器值
    MOV      r1, sp
    MRS      r0, spsr                ; 获得 spsr
    STMFD    sp!, {r0}              ; 保存 spsr 到栈中
    TST      r0, #T_bit              ; 判断是否在 Thumb 指令模式?
    LDRNEH   r0, [lr, #-2]           ; 如果是 Thumb 指令模式的话, 装载半字长指令

    BICNE    r0, r0, #0xFF00
    LDREQ    r0, [lr, #-4]           ; 如果是 ARM 指令模式的话, 装载字长指令
    BICEQ    r0, r0, #0xFF000000

    ; r0 寄存器现在包含了 SWI 号

```

; r1 包含了指向栈寄存器地址

```
BL      C_SWI_Handler      ; 调用高级 C 语言的中断处理函数
LDMFD   sp!, {r0}          ; 从栈中获得 spsr
MSR      spsr_cf, r0        ; 恢复 spsr
LDMFD   sp!, {r0-r3, r12, pc}^; 恢复其他相应的寄存器并且返回
END
```

其中第一行代码是用来声明该代码的属性，必须声明为 **CODE** 和 **READONLY** 属性，否则编译不能通过，**CODE** 和 **READONLY** 属性分别表示该代码属于代码区，并且为只读属性。本实例（第 1 章中的 **SWI** 例子）用来实现一个 **SWI**（Soft Ware Interrupter，软件中断）服务程序，其实现由两部分来完成，一部分就是上面的 **ARM** 汇编代码实现的中断服务程序，还有一部分就是其调用的 **C\_SWI\_Handler** 函数实现的中断服务程序，该函数是用高级 C 语言实现，增加了程序的可读性，并且便于实现更加复杂的功能，这里给出了 **C\_SWI\_Handler** 函数的一个简单实现，如下代码所示：

```
//用高级 C 语言实现了软件中断服务程序的复杂功能
void C_SWI_Handler( int swi_num, int *regs )
{
    switch( swi_num )
    {
        case 0:
            regs[0] = regs[0] * regs[1];
            break;
        case 1:
            regs[0] = regs[0] + regs[1];
            break;
        case 2:
            regs[0] = (regs[0] * regs[1]) + (regs[2] * regs[3]);
            break;
        default:
            break;
    }
}
```

现在读者应该对 **ARM** 中断处理有了一定的了解，下面将介绍一个具体的 **Linux** 设备中断实现的实例。

#### 5.3.4.3 一个 Linux 中断相关的实例

关于中断处理通常都是针对硬件设备而产生的概念，所以离开了硬件谈中断都是没有任何意义的，所以在此给出了一个关于中断的具体实例——键盘扫描实例，通常在写一个硬件设备相关的驱动时，需要了解相关硬件的知识，最重要的是参考芯片相关的 **datasheet**（也就是用户手册），同时还需要能看懂基本的电路图。本实例使用 8 个按键来实现 4 个外部中断响应的功能，电路原理图如 5.4，本书的重点讲述软件相关的知识，关于硬件的知识不会进行深入的分析。但是希望读者一定要牢记，要想成为一名优秀的驱动开发人员，掌握硬件的基本知识是必不可少的。通过 5.4 原理图可以看到，该模块使用了 4 个外部中断源，分别是：

EINT0, EINT2, EINT11 (KBDINT) 和 EINT19, 通过查阅 S3C2410 的 datasheet, 可以看到这四个中断所对应的 I/O 端口依次是 GPF0, GPF2, GPG3 和 GPG11。同时每两个按键共享一个中断, 那么当一个中断被触发时如何区分是这两个键中的哪一个呢? 其实很简单, 就是利用 nSS KBD 和 nXDACK1 信号线来区别在同一中断情况下使用哪一个按键。其中这两个信号线所对应的端口分别是 GPB6 和 GPB7。该键盘实例的硬件工作原理是: 首先配置 nSS KBD 和 nXDACK1 对应的端口 GPB6 和 GPB7 为输出, 同时配置这四个中断都是下降沿触发, 正常工作时, 当 GPB6 配置为低电平和 GPB7 为高电平时, 左边一列的键盘被按下时有效, 右边按键无效, 当 GPB6 配置为高电平和 GPB7 为低电平时, 右边一列按键有效, 而左边按键无效。所以正常情况下将 GPB6 和 GPB7 之间电平互相高低切换, 由于他们之间切换的时间非常短, 所以一般用户不会感觉到他们在处理按键上是切换响应的。

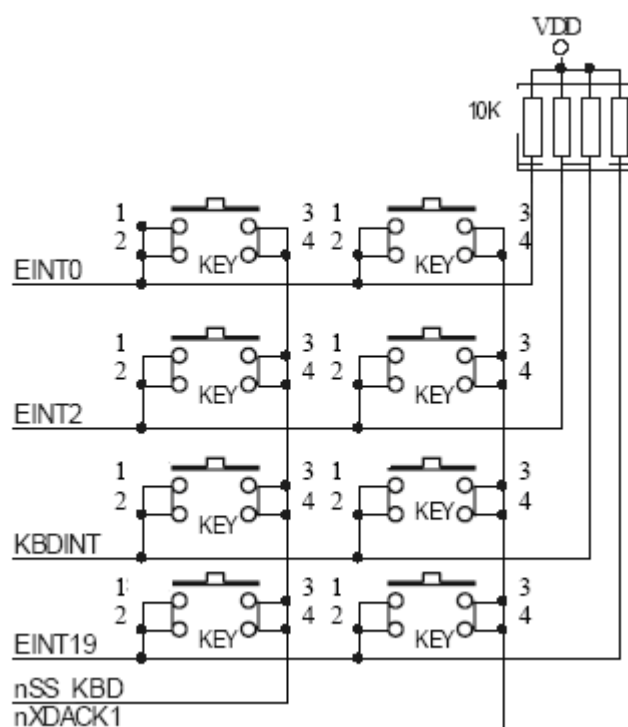


图 5.4 键盘实例的硬件电路图

下面将编写出针对该键盘设备组成的驱动模块代码, 如下所示, 完整的代码可参考附件光盘内容。

```
.....
U8 Key_Scan( void )    //按键扫描函数
{
    Delay(300);

    GPIO_BDAT &=~(1<<6);
    GPIO_BDAT |=1<<7;

    if(      (GPIO_FDAT&(1<<0)) == 0 )      return 1 ;
    else if( (GPIO_FDAT&(1<<2)) == 0 )      return 3 ;
    else if( (GPIO_G&(1<<3)) == 0 )      return 5 ;
    else if( (GPIO_G&(1<<11)) == 0 )      return 7 ;
}
```

```

        Delay(300) ;

        GPIO_BDAT &=~(1<<7);
        GPIO_BDAT |=1<<6;
        if(      (GPIO_FDAT&(1<< 0)) == 0 )      return 2 ;
        else if( (GPIO_FDAT&(1<< 2)) == 0 )      return 4 ;
        else if( (GPIO_GDAT&(1<< 3)) == 0 )      return 6 ;
        else if( (GPIO_GDAT&(1<<11)) == 0 )      return 8 ;
        else return 0xff ;
    }

static void key_irq_handle(int irq, void *dev_id, struct pt_regs *reg) //中断服务程序.
{

    if(INTPND==BIT_EINT8_23)
    {
        SRCPND=INTPND=BIT_EINT8_23;
    }
    if(INTPND==BIT_EINT0) {
        SRCPND=INTPND=BIT_EINT0;
    }
    if(INTPND==BIT_EINT2)
    {
        SRCPND=INTPND=BIT_EINT_2;
    }
    ready = 1;
    key_value=Key_Scan();
    printk("key is%d.\n",key_value);

    GPIO_BCON &=~((3<<12)|(3<<14));
    GPIO_BCON |=((1<<12)|(1<<14));
    GPIO_BUP &=~(3<<6);
    GPIO_BDAT &=~(3<<6);

    wake_up_interruptible(&key_wait);
}

static int key_open(struct inode *inode, struct file *filp)
{
    int result;
    int i = 0;
    struct key_info *key_info;

    ready = 0;

```

```

        key_info = key;

        for(i=0; i<4; i++)
        {
            set_external_irq(key_info[i].irq_num,          EXT_FALLING_EDGE,
GPIO_PULLUP_DIS);
            result = request_irq(key_info[i].irq_num, key_irq_handle, SA_INTERRUPT,
DEVICE_NAME, NULL);
            if(result)
            {
                printk(KERN_INFO DEVICE_NAME " Failed to request irq.\n");
                return result;
            }
        }

        MOD_INC_USE_COUNT;
        printk(KERN_INFO DEVICE_NAME ": opened.\n");
        return 0;
    }

    .....
    static struct file_operations key_fops =
    {
        owner:    THIS_MODULE,
        read:     key_read,
        open:     key_open,
        release:  key_release,
    };

    static devfs_handle_t devfs_handle;
    static int __init key_init(void)
    {
        devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
KEY_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &key_fops, NULL);

        return 0;
    }

    static void __exit key_exit(void)
    {
        devfs_unregister(devfs_handle);
    }

    module_init(key_init);
    module_exit(key_exit);

```

由于篇幅的原因，这里只列出了我们所关心的中断处理部分代码，其中最重要的就是 `key_irq_handle` 函数，该函数就是按键时产生的中断处理函数，每当按键发生时调用这个函数，这个函数会根据寄存器的值来判断所按键是哪一个，然后再做相应的操作。`key_irq_handle` 函数是在 `key_open` 函数中由 `request_irq` 函数调用。所以应用程序在想使用该键盘相关的功能时，必须先调用 `open` 函数来注册键盘的中断请求。通过这个实例，可以让读者接触到了一个有实际应用价值的设备驱动模块程序，虽然读者可能还不理解其具体的实现，不过不用担心，在后面的章节中会具体讲述设备驱动模块程序的编写，从而会加深理解。

## 5.3.5 内核调试

内核调试是驱动开发人员必须掌握的一项技能，和调试其他程序一样，内核也属于一种特殊的程序。调试内核问题时，能够跟踪内核执行情况并查看其内存和数据结构是非常有用的。Linux 中的内置内核调试器 KDB (Kernel Debugger 的缩写) 提供了这种功能。Linux 内核调试器 (KDB) 允许你调试 Linux 内核。这个恰如其名的工具实质上是内核代码的补丁，它允许开发人员访问内核内存和数据结构。KDB 的主要优点之一就是它只需要用一台机器就可以进行调试，比如你可以调试正在运行的内核。设置一台用于 KDB 的机器需要花费一些工作，因为需要给内核打补丁并进行重新编译。KDB 的用户应当熟悉 Linux 内核的编译（在一定程度上还要熟悉内核内部机理），KDB 项目是由 Silicon Graphics 维护的，需要从它的 [ftp://oss.sgi.com/www/projects/kdb/download/](http://oss.sgi.com/www/projects/kdb/download/) 站点下载与内核版本有关的补丁。

### 5.3.5.1 准备内核调试环境

首先从上述的站点中下载并应用两个补丁，一个是公共的补丁，包含了对通用内核代码的更改，另一个是特定于体系结构的补丁。目前可用最新的 KDB 版本是 4.4，以运行 2.6.10 内核的 x86 机器为例，需要下载 `kdb-v4.4-2.6.10-common-1.bz2` 和 `kdb-v4.4-2.6.10-i386-1.bz2` 文件。

接下来将这两个压缩补丁拷贝到 `/usr/src/linux` 目录下，也就是你放内核源码的目录。然后解压。

```
# bzip2 -d kdb-v4.4-2.6.10-common-1.bz2
# bzip2 -d kdb-v4.4-2.6.10-i386-1.bz2
```

执行上面的压缩命令将会生成 `kdb-v4.4-2.6.10-common-1` 和 `kdb-v4.4-2.6.10-i386-1` 两个补丁文件。然后应用这两个补丁文件，具体命令如下：

```
# patch -p1 <kdb-v4.4-2.6.10-common-1
# patch -p1 <kdb-v4.4-2.6.10-i386-1
```

接下来需要构建内核以支持 KDB。第一步是设置 `CONFIG_KDB` 选项，使用你喜欢的配置方法（`make xconfig` 和 `make menuconfig` 等）来完成这一步。选择“Kernel hacking”选项并选择“Built-in Kernel Debugger support”选项。还可以根据自己的偏好选择其它两个选项。选择“Compile the kernel with frame pointers”选项（如果有的话）则设置 `CONFIG_FRAME_POINTER` 标志。这将产生更好的堆栈回溯，因为帧指针寄存器被用作帧指针而不是通用寄存器。还可以选择“KDB off by default”选项。这将设置 `CONFIG_KDB_OFF` 标志，并且在缺省情况下将关闭 KDB。如果编译期间没有选中 `CONFIG_KDB_OFF`，那么在缺省情况下 KDB 是活动的。否则，你需要显式地激活它，可以通过在引导期间将 `kdb=on` 标志传递给内核或者通过在挂装了 `/proc` 之后执行以下命令

激活:

```
# echo "1" >/proc/sys/kernel/kdb
```

相反, 如果缺省情况下 KDB 是打开的, 那么将 `kdb=off` 标志传递给内核或者执行下面这个操作将会取消激活 KDB:

```
# echo "0" >/proc/sys/kernel/kdb
```

最后, 保存配置, 然后退出。重新编译内核。建议在构建内核之前执行 “make clean”。用常用方式安装内核并引导它。到此为止, 关于 KDB 的内核调试环境已经建立, 下面讲述 KDB 的一般用法。

### 5.3.5.2 KDB 的基本用法

KDB 是一个功能非常强大的工具, 它允许进行多个操作, 比如内存和寄存器修改、应用断点和堆栈跟踪等。根据这些操作可以将 KDB 命令分成几个类别。下面将介绍每一类中最常用命令的基本用法。

#### Ø 内存显示和修改

最常用的命令有 `md`、`mdr`、`mm` 和 `mmW`。

`md` 命令以一个地址/符号和行计数为参数, 显示从该地址开始的 `count` 行的内存。如果没有指定 `count`, 那么就使用环境变量所指定的缺省值。如果没有指定地址, 那么 `md` 就从上一次打印的地址继续。地址打印在开头, 字符转换打印在结尾。使用格式 1: `md [vaddr [line-count [output-radix]]]` 其中显示地址为 `vaddr` 的内存的内容。`line-count` 为要显示的内存的行数, `output-radix` 指定以 8 进制、10 进制或者 16 进制显示。如果省略 `line-count` 和 `output-radix`, 那么将以设置的环境变量 `MDCOUNT` 和 `RADIX` 方式显示。如果不带任何参数, `md` 命令将接着上次 `md` 命令的后续地址显示内存内容。使用格式 2: `mdWcn` 在缺省情况下, `md` 以当前环境变量 `BYTESPERWORD` (确定字的长度) 的值读取数据, 在读取硬件寄存器的时候, 需要指定数据的宽度。这是可以使用 `mdWcn` 来进行读取, `W` 是读取的宽度, 单位是字节, `cn` 为要读取的数目。例如, 显示从 `0x1000000` 开始的 10 行内存, 具体使用如下:

```
[0]kdb> md 0x1000000 10
```

`mdr` 命令带有地址或符号以及字节计数, 显示从指定的地址开始的 `count` 字节数的初始内存内容。它本质上和 `md` 一样, 但是它不显示起始地址并且不在结尾显示字符转换。`mdr` 命令较少使用。使用格式: `mdr <vaddr> <count>`

`mm` 命令修改内存内容。它以地址 / 符号和新内容作为参数, 用 `new-contents` 替换地址处的内容。使用格式: `mm <vaddr> <new content>`

`mmW` 命令更改从地址开始的 `W` 个字节。请注意, `mm` 更改一个机器字。使用格式: `mmW <vaddr> <new content>`

#### Ø 寄存器显示和修改

常用命令有 `rd`、`rm` 和 `ef` 等。

`rd` 命令 (不带任何参数) 显示处理器寄存器的内容。它可以有选择地带三个参数。如果传递了 `c` 参数, 则 `rd` 显示处理器的控制寄存器; 如果带有 `d` 参数, 那么它就显示调试寄存器; 如果带有 `u` 参数, 则显示上一次进入内核的当前任务的寄存器组。使用格式: `rd [c|d|u]`

`rm` 命令修改寄存器的内容。它以寄存器名称和 `new-contents` 作为参数, 用 `new-contents` 修改寄存器。寄存器名称与特定的体系结构有关。目前, 不能修改控



制寄存器。使用格式：rm <register-name> <register-content> 例如，修改 eax 寄存器内容为 0x10，具体操作如下：

```
[0]kdb> rm %eax 0x10
```

ef 命令以一个地址作为参数，它显示指定地址处的异常帧。使用格式：ef <vaddr>

#### Ø 断点

常用的断点命令有 bp 、 bc 、 bd 、 be 和 bl。

bp 命令以一个地址 / 符号作为参数，它在地址处应用断点。当遇到该断点时则停止执行并将控制权交予 KDB。该命令有几个有用的变体。bpa 命令对 SMP 系统中的所有处理器应用断点。bph 命令强制在支持硬件寄存器的系统上使用它。bpha 命令类似于 bpa 命令，差别在于它强制使用硬件寄存器。使用格式：bp [<vaddr>]  
bc 命令从断点表中除去断点。它以具体的断点号或 \* 作为参数，在后一种情况下它将除去所有断点。使用格式：bc <bpnum>

bd 命令禁用特殊断点。它接收断点号作为参数。该命令不是从断点表中除去断点，而只是禁用它。断点号从 0 开始，根据可用性顺序分配给断点。使用格式：bd <bpnum>

be 命令用来启用断点。该命令的参数也是断点号。使用格式：be <bpnum>

bl 命令列出当前的断点集。它包含了启用的和禁用的断点。该命令的操作与 bp 命令相同。

#### Ø 堆栈跟踪

堆栈跟踪命令主要有 bt 、 btp 、 btc 和 bta。

bt 命令设法提供有关当前线程的堆栈的信息。它可以有选择地将堆栈帧地址作为参数。如果没有提供地址，那么它采用当前寄存器来回溯堆栈。否则，它假定所提供的地址是有效的堆栈帧起始地址并设法进行回溯。如果内核编译期间设置了 CONFIG\_FRAME\_POINTER 选项，那么就用帧指针寄存器来维护堆栈，从而就可以正确地执行堆栈回溯。如果没有设置 CONFIG\_FRAME\_POINTER ，那么 bt 命令可能会产生错误的结果。使用格式：bt [<stack-frame addr>]

btp 命令将进程标识作为参数，并对这个特定进程进行堆栈回溯。使用格式：btp <pid>

btc 命令对每个活动 CPU 上正在运行的进程执行堆栈回溯。它从第一个活动 CPU 开始执行 bt ，然后切换到下一个活动 CPU，以此类推。

bta 命令对处于某种特定状态的所有进程执行回溯。若不带任何参数，它对所有进程执行回溯。可以有选择地将各种参数传递给该命令。将根据参数处理处于特定状态的进程。选项以及相应的状态如下[9]：

- l D: 不可中断状态
- l R: 正运行
- l S: 可中断休眠
- l T: 已跟踪或已停止
- l Z: 僵死
- l U: 不可运行

#### Ø 其他用法

id 命令用于反汇编。使用格式：id <vaddr> 从 vaddr 开始的地址反汇编指令。

cpu 命令用于切换到另一个 CPU。使用格式：cpu <cpunum> 这条命令仅仅在 SMP 结构下有用，它切换到由 cpunum 指定的 CPU。

ps 命令用于显示所有活动的进程。使用格式：ps 显示当前的活动的进程。包括 pid、

ppid、CPU 号、当前状态，以及对应的线程。

**reboot** 命令用来重新启动机器。使用格式：**reboot** 在某些情况下，内核无法返回到正常工作状态，这时可以利用 **reboot** 重新启动机器。注意在重启机器前，它不进行任何状态保存的工作。

**sections** 命令列出内核中所有已知的段的信息。使用格式：**sections** 列出模块和内核的所有已知的段的信息。首先是模块信息，最后是内核信息。包括模块名和一个或者多个段的信息。段信息包括段名、段起始地址、段结束地址和段标识。本命令仅仅是为外部调试器而设立的。

**sr** 命令激活 SysRq 代码，也就是调用 **MAGIC\_SYSRQ** 函数。格式：**sr <sysrq key>** 将 **sysrq key** 字符作为参数传递给 SysRq 函数进行处理，就像你已经键入了 SysRq 键和该字符一样。如果要使用这个命令，需要在配置内核时，选择 Magic SysRq Key。然后在新内核启动后，使用如下命令激活 SysRq 功能。

```
# echo "1" > /proc/sys/kernel/sysrq
```

这是一个功能强大的命令，它使得在 **kdb** 中可以使用操作系统提供的 SysRq 处理函数。

**lsmod** 命令列出内核中加载的所有模块。使用格式：**lsmod** 显示所有模块的信息。包括模块名、模块大小、模块结构地址、引用计数，以及被哪个模块所引用。

**rmmod** 命令卸载一个模块。使用格式：**rmmod <modname>** 将由 **modname** 指定的模块从内核中卸载。

**ll** 命令对链表中的每个元素重复执行命令。使用格式：**ll <addr> <link-offset> <cmd>** 它对以地址 **addr** 开头的链表的头 **link-offset** 个元素，重复执行 **cmd** 命令。

**help** 和 **?** 命令显示帮助信息。使用格式：**help** 或者 **?** 显示 **kdb** 的命令以及简单的用法。

总之，**KDB** 是一个强大的内核调试工具，通常 **GDB** 需要两台机器通过串口才能进行调试，而 **KDB** 只需要一台机器即可进行调试，对于普通用户来说，是非常方便的。对于编写内核程序（如可加载模块）的程序员来说，**KDB** 提供的这些命令使得调试工作难度大大降低，使得调试效率得以提高。另外对于内核感兴趣的读者可以使用 **KDB** 来查看内核的数据结构和运行状态，从而加深对内核的理解。不足之处是 **KDB** 无法提供源码级的调试，要求程序员有一定的汇编程序基础。但总的来说，**KDB** 提供了一种强有力的内核调试手段，值得读者去学习和使用。

## 5.4 本章小结

本章是 Linux 驱动程序开发中最基础的一章，也是最适合初学者入门的一章。本章首先对驱动程序的作用和分类进行了介绍，让读者对驱动程序有个大概的了解；接着用一个最简单的“Hello World”实例讲述了 Linux 内核模块的实现框架，通过这个简单的实例，使读者对 Linux 内核模块有个初步的了解；最后重点介绍了 Linux 驱动程序开发的几个要点，包括内存与 I/O 端口，并发控制（自旋锁与信号量），阻塞与非阻塞，中断处理，以及内核调试工具 **KDB** 等重要概念，其中还通过一些典型的代码来结合讲述这些概念。总之，通过对本章的学习，可以为后面章节的学习奠定坚实的基础，下一章将讲述 Linux 设备驱动中最常见的类型——字符设备驱动程序。

## 5.5 常见问题

1. 在 Linux 系统中，CPU 对 I/O 端口的编址方式有哪两种？

参考答案：一是 I/O 映射（I/O-mapped）方式，如 X86 处理器为外设专门实现了一个单独的地址空间，称为 I/O 地址空间或 I/O 端口空间，CPU 通过专门的 I/O 指令（如 X86 的 IN 和 OUT 指令）来访问这一空间的地址单元；二是内存映射（Memory-mapped）方式，RISC 指令系统的 CPU（如 ARM，PowerPC 等）通常只实现一个物理地址空间，外设 I/O 端口成为了内存的一部分，此时 CPU 访问 I/O 端口就像访问一个内存单元不需要单独的 I/O 指令。这两种方式在硬件实现上的差异对软件来说是完全可见的，驱动程序开发人员可以将内存映射方式的 I/O 端口和外设内存统一看作是 I/O 内存资源。

2. 自旋锁和信号量的基本概念？

参考答案：自旋锁的概念非常简单，自旋锁是一个互斥现象的设备，它只能有两个值：locked（锁定）或 unlocked（解锁）。它通常作为一个整型值的单个位来实现。如果自旋锁已经被别的进程占用，调用者就一直循环查看是否该自旋锁被释放，在任何时刻，自旋锁只能有一个保持者，即在同一时刻只能有一个进程获得锁。一个信号量是一个结合一对函数的整型值，这对函数通常被称为 P 操作和 V 操作。一个进程希望进入一个临界区域时将调用 P 操作在相应的信号量上，如果这个信号量的值大于 0，这个值将被减 1 同时该进程继续进行。相反如果这个信号量的值等于或小于 0，则该进程将等待别的进程释放该信号量，然后才能执行。解锁一个信号量通过调用 V 操作来完成，这个函数的作用正好与 P 操作相反，调用 V 操作时信号量的值将增加 1，如果需要，同时唤醒哪些等待的进程。当信号量用于互斥现象——多个进程在同时运行一个相同的临界区域，此时信号量的值被初始化为 1。信号量只能在一个时刻被一个进程或线程拥有，一个信号量使用在这种模式下通常被称为互斥体（mutex，mutual exclusion 的缩写）体。

3. 当一个异常出现以后，ARM 微处理器会执行哪几个步操作？

参考答案：当一个异常出现以后，ARM 微处理器会执行以下几步操作：

- ü 保存处理器当前状态、中断屏蔽位以及各条件标志位；
- ü 设置当前程序状态寄存器（CPSR）中相应的位；
- ü 将寄存器 lr\_mode（当异常出现或调用函数时保存下一条指令的位置）设置成返回地址；
- ü 将程序计数器(PC)值设置成该异常中断的中断向量地址，从而跳转到相应的异常中断处理程序处执行。

4. 使用 KDB 工具的优缺点？

参考答案：KDB 是一个强大的内核调试工具，通常 GDB 需要两台机器通过串口才能进行调试，而 KDB 只需要一台机器即可进行调试，对于普通用户来说，是非常方便的。对于编写内核程序（如可加载模块）的程序员来说，KDB 提供的这些命令使得调试工作难度大大降低，使得调试效率得以提高。另外对于内核感兴趣的读者可以使用 KDB 来查看内核的数据结构和运行状态，从而加深对内核的理解。不足之处是 KDB 无法提供源码级的调试，要求程序员有一定的汇编程序基础。

## 第 6 章 字符设备驱动程序

本章学习目标：

- | 熟悉字符设备驱动程序概念
- | 熟悉字符设备驱动相关的三个结构：file\_operations, file, inode
- | 熟悉主、次设备号作用和使用
- | 理解触摸屏设备的硬件实现原理
- | 理解触摸屏设备的驱动程序实现

### 6.1 字符设备驱动介绍

系统调用是操作系统内核和应用程序之间的接口，设备驱动程序是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。本章介绍最常见的设备驱动程序——字符设备驱动，在介绍字符设备驱动程序之前，首先需要学习几个重要的概念：字符设备相关的数据结构和主、次设备号等。

#### 6.1.1 字符设备驱动相关的重要结构

编写 Linux 字符设备驱动程序首先要熟悉这三个结构，即 file\_operations（文件操作）、file（文件）和 inode（节点）。这三个数据结构非常重要，被定义在<include/linux/fs.h>文件中。首先来介绍 file\_operations 结构体。

##### 6.1.1.1 file\_operations（文件操作）结构

由于用户进程是通过设备文件同硬件打交道，所以对设备文件的操作方式不外乎就是一些系统调用，如 open, read, write, close 等，注意，不是 fopen, fread, fwrite, fclose。但是如何把系统调用和驱动程序关联起来呢？这时需要一个非常关键的数据结构，即 file\_operations，它用来存储驱动内核模块提供的对设备进行各种操作的函数的指针。该数据结构体在 Linux 2.6.10 内核中具体定义如下：

```
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 (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, 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);
```

```

int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
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 (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
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 *);
};

```

该结构体的每个成员都对应着驱动内核模块用来处理某个被请求事务的函数的地址。下面对这个定义相对复杂且非常重要的数据结构成员进行解释：

- n** struct module \*owner;  
该成员是 file\_operations 结构中唯一一个不是声明操作的成员，它是一个指向拥有这个模块的指针，该成员用来在它的操作还在使用时不允许卸载该模块，通常情况下，都被简单的初始化为 THIS\_MODULE。
- n** loff\_t (\*llseek) (struct file \*, loff\_t, int);  
该成员为 file\_operations 结构的一个操作，用来改变当前文件的读写位置，并且将新位置作为返回值。其中 loff\_t 为 long long 类型的别名。
- n** ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);  
该操作用来从设备中获取数据，如果这个成员为一个空指针，则系统调用 read 将返回一个-EINVAL（Invalid argument，即无效参数）错误，正常情况下，返回一个非负整数代表读取的字节数。其中 ssize\_t 为 int 或 long 型，和平台相关。\_\_user 用来声明为用户空间。
- n** ssize\_t (\*aio\_read) (struct kiocb \*, char \_\_user \*, size\_t, loff\_t);  
该操作用来初始化一个异步的读操作，即当一个读操作还没有完成时也许这个函数已经返回。当这个操作为空时，它将由 read（同步）操作代替。
- n** ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);  
该操作用来发送数据给设备，当为空时，系统调用 write 将返回-EINVAL 错误。正常情况下，返回一个非负整数代表成功写的字节数。
- n** ssize\_t (\*aio\_write) (struct kiocb \*, const char \_\_user \*, size\_t, loff\_t);  
该操作用来初始化一个异步写操作，当该操作为空时，调用 write 操作。
- n** int (\*readdir) (struct file \*, void \*, filldir\_t);  
该操作用于文件系统，用来读取目录。对于设备文件时，该操作为空。
- n** unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  
该操作用来查询一个或多个文件描述符的读或写是否会阻塞。Poll 方法返回一个位掩码来指示是否非阻塞的读或写是可能的，并且提供给内核信息用来使调用进程 sleep 直到 I/O

端口变为可用。如果一个设备驱动的 poll 方法为空，则设备默认为不阻塞的可读和可写。

- n** int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);  
该操作用来提供发出设备特定命令的方法。对于一个未定义 ioctl 操作的设备，当系统调用时将返回-ENOTTY 错误。
- n** int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  
该操作用来请求将设备内存映射到进程的地址空间。如果这个方法为空，mmap 系统调用将返回-ENODEV 错误。
- n** int (\*open) (struct inode \*, struct file \*);  
该操作用来打开设备文件，也是对设备文件进行的第一个操作。如果这个操作为空，则设备打开操作一直成功，但是你的驱动程序将不会被调用。
- n** int (\*flush) (struct file \*);  
该操作用来执行和等待设备未完成的操作，目前 flush 很少使用，不过 SCSI 磁带驱动使用了它，用来确保所有写的数据在设备关闭前已经写到磁带上。如果 flush 为空，内核简单的忽略应用程序的请求。
- n** int (\*release) (struct inode \*, struct file \*);  
该操作用来释放文件结构，该操作可以为空。
- n** int (\*fsync) (struct file \*, struct dentry \*, int datasync);  
该操作用来刷新任何等待处理的数据，如果这个操作为空，则系统调用 fsync 将返回-EINVAL。
- n** int (\*aio\_fsync) (struct kiocb \*, int datasync);  
该操作是 fsync 的异步版本。
- n** int (\*fasync) (int, struct file \*, int);  
该操作用来通知设备 FASYNC 标志的改变。如果该操作为空，则说明该驱动不支持异步通知。
- n** int (\*lock) (struct file \*, int, struct file\_lock \*);  
该操作用来对文件实行加锁，加锁对常规文件是必不可少的特性，但是设备驱动很少有实现该操作的。
- n** ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
该操作用来实现多个内存区的单个 read 操作，该操作不必对数据进行额外拷贝。如果该操作为空，则会调用 read 方法。
- n** ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
该操作用来实现多个内存区的单个 write 操作，该操作不必对数据进行额外拷贝。如果该操作为空，则会调用 write 方法。
- n** ssize\_t (\*sendfile) (struct file \*, loff\_t \*, size\_t, read\_actor\_t, void \*);  
该操作用来实现使用最少的拷贝从一个文件描述符搬移数据到另一个。通常设备驱动使 sendfile 为空。
- n** ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  
该操作用来由内核调用来发送数据，一次一页到对应的文件。设备驱动程序实际上不实现 sendpage 方法。
- n** unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  
该操作用来在进程地址空间找一个合适的位置来映射在底层设备上的内存段中。该方法是使驱动能强制满足特殊设备的对齐请求。通常情况下，设置该方法为空。
- n** int (\*check\_flags)(int);

该操作允许模块检查传递给 `fcntl(F_SETFL...)` 调用的标志。通常情况下，设置该方法为空。

**n** `int (*dir_notify)(struct file *filp, unsigned long arg);`

该操作只对文件系统有用，该方法在应用程序使用 `fcntl` 函数来请求目录改变通知时调用。设备驱动程序不需要实现 `dir_notify` 方法。

**n** `int (*flock) (struct file *, int, struct file_lock *);`

该操作用来对文件设备加锁，但是基本上没有设备驱动程序实现该操作。

结构体 `file_operations` 的确包含了很多操作，但在实际设备驱动程序中只会用到其中的很少一部分，大部分操作将不会被用到。例如在 5.3.4.3 小节中的 Linux 中断实例中的文件操作定义如下：

```
static struct file_operations key_fops =
{
    owner:    THIS_MODULE,
    read:     key_read,
    open:     key_open,
    release:  key_release,
};
```

通过这个实例可以看出，该设备驱动模块只实现了 `read`、`open` 和 `release` 三个操作，这三个操作所对应的实现函数分别为：`key_read`、`key_open` 和 `key_release`，其他的操作都没有实现。

### 6.1.1.2 file（文件）结构

设备驱动程序中第二个非常重要的数据结构，即 `file`（文件）结构，它不同于应用程序空间的 `FILE` 指针，`FILE` 指针定义在 C 库中而不会出现在内核代码中，而 `struct file` 只出现在内核代码中，从不出现在用户程序中。结构体 `file` 在 Linux 2.6.10 版本中的定义如下：

```
struct file {
    struct list_head    f_list;
    struct dentry       *f_dentry;
    struct vfsmount     *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t            f_count;
    unsigned int        f_flags;
    mode_t              f_mode;
    int                 f_error;
    loff_t              f_pos;
    struct fown_struct  f_owner;
    unsigned int        f_uid, f_gid;
    struct file_ra_state f_ra;

    unsigned long       f_version;
    void                *f_security;

    /* needed for tty driver, and maybe others */
};
```

```

void          *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head    f_ep_links;
    spinlock_t          f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space *f_mapping;
};

```

下面解释一下 `file` 结构体成员，在内核源码中，指向 `struct file` 的指针通常被称为 `file` 或 `flip`，为了和它结构本身区别，我们这里将该文件指针称为 `flip`，这样就可以和 `file` 结构本身区别开。由于 `file` 结构体中很多成员并不对设备驱动程序有用，所以在这里只介绍一些常用的重要成员。

- n** `struct dentry *f_dentry;`  
该成员是文件对应的目录项结构。除了用 `flip->f_dentry->d_inode` 的方法访问索引节点结构之外，设备驱动开发人员一般无需关心 `dentry` 结构。
- n** `struct file_operations *f_op;`  
这个成员是定义与文件关联的操作，也就是上面讲的文件操作。内核在执行 `open` 操作时对这个指针赋值，以后需要处理这些操作时就读取这个指针。
- n** `unsigned int f_flags;`  
该成员是文件标志，如 `O_RDONLY`（只读），`O_NONBLOCK`（非阻塞）和 `O_SYNC`（同步）。驱动程序应该检查 `O_NONBLOCK` 标志看是否是非阻塞操作请求，其他标志很少用到。特别的是，读写权限通过 `f_mode` 成员检查而不是 `f_flags`。
- n** `mode_t f_mode;`  
该成员用于确定文件是可读的或可写的（或者两者都是），通过位 `FMODE_READ` 和 `FMODE_WRITE` 实现。当文件还没有打开时，试图读或写操作将被拒绝，驱动程序可能都不知道这个情况。
- n** `loff_t f_pos;`  
该成员用来确定当前的读写位置。如果需要知道当前在文件中的位置，驱动程序可以读这个值，但是不应该改变这个值。读和写操作使用它们的最后一个参数指针确定文件位置而不是直接用 `flip->f_pos` 实现。
- n** `void *private_data;`  
该成员是跨系统调用时保存状态信息非常有用的资源。驱动程序可以用这个字段指向已分配的数据，但一定要在内核销毁 `file` 结构前在 `release` 方法中释放内存。

文件（`file`）结构代表一个打开的文件描述符，它不是专门给设备驱动使用，系统中每一个打开的文件在内核中都有一个关联的 `struct file`。它由内核在 `open` 时创建，并传递给在文件上操作的任何函数，直到最后关闭。当文件的所以实例都关闭后，内核释放这个数据结构。

### 6.1.1.3 inode（节点）结构

在 Linux 设备驱动开发中，还有一个非常重要的数据结构，即 `inode` 结构。内核中用 `inode` 结构表示具体的文件，而用 `file` 结构表示打开的文件描述符。对于单个文件，可能会有许多



个表示打开的文件描述符 `file` 结构，但是他们都指向了单个的 `inode` 结构，所以 `file` 结构和 `inode` 结构是不同的。在 Linux 2.6.10 内核中，`inode` 结构体的具体定义如下：

```
struct inode {
    struct hlist_node    i_hash;
    struct list_head     i_list;
    struct list_head     i_dentry;
    unsigned long        i_ino;
    atomic_t             i_count;
    umode_t              i_mode;
    unsigned int         i_nlink;
    uid_t                i_uid;
    gid_t                i_gid;
    dev_t                i_rdev;
    loff_t               i_size;
    struct timespec      i_atime;
    struct timespec      i_mtime;
    struct timespec      i_ctime;
    unsigned int         i_blkbits;
    unsigned long        i_blksize;
    unsigned long        i_version;
    unsigned long        i_blocks;
    unsigned short       i_bytes;
    unsigned char        i_sock;
    spinlock_t           i_lock; /* i_blocks, i_bytes, maybe i_size */
    struct semaphore     i_sem;
    struct rw_semaphore  i_alloc_sem;
    struct inode_operations *i_op;
    struct file_operations *i_fop; /* former ->i_op->default_file_ops */
    struct super_block    i_sb;
    struct file_lock      *i_flock;
    struct address_space  *i_mapping;
    struct address_space  i_data;
#ifdef CONFIG_QUOTA
    struct dquot          *i_dquot[MAXQUOTAS];
#endif
    /* These three should probably be a union */
    struct list_head      i_devices;
    struct pipe_inode_info *i_pipe;
    struct block_device    i_bdev;
    struct cdev            *i_cdev;
    int                   i_cindex;
    __u32                 i_generation;
#ifdef CONFIG_DNOTIFY
    unsigned long          i_dnotify_mask; /* Directory notify events */
#endif
}
```

```

        struct dnotify_struct    *i_dnotify; /* for directory notifications */
    #endif

    unsigned long    i_state;
    unsigned long    dirtied_when; /* jiffies of first dirtying */
    unsigned int     i_flags;
    atomic_t         i_writecount;
    void             *i_security;
    union {
        void         *generic_ip;
    } u;
    #ifdef __NEED_I_SIZE_ORDERED
        seqcount_t    i_size_seqcount;
    #endif
};

```

可以看到 `inode` 结构包含了大量有关文件的信息，但通常情况下对设备驱动程序开发有用的成员有下面两个：

**n** `dev_t i_rdev;`

该成员表示设备文件的 `inode` 结构，它包含了真正的设备编号(将在下一节介绍设备编号)。

**n** `struct cdev *i_cdev;`

该成员表示字符设备的内核的内部结构，当 `inode` 指向一个字符设备文件时，该成员包含了指向 `struct cdev` 结构的指针，其中 `cdev` 结构是字符设备结构体。

其实关于 Linux 字符设备驱动开发还有一些其它的数据结构，这里只介绍了最经常使用的三个结构体，在后面的章节中还会介绍其他的数据结构。

## 6.1.2 主、次设备号

对字符设备的访问是通过文件系统内的设备名称进行的，那些文件被称为设备文件，他们通常位于 `/dev` 目录下。在 `/dev` 目录下，通过 `ls -l` 命令可以查看系统中的字符设备和块设备。例如在系统中利用 `ls -l` 命令查看设备显示如下：

```

brw-rw---- 1 root    disk      15,  0 Jan 30  2003 cdu31a
brw-rw---- 1 root    disk      24,  0 Jan 30  2003 cdu535
crw-rw---- 1 root    disk      67,  0 Jan 30  2003 cfs0
brw-rw---- 1 root    disk      30,  0 Jan 30  2003 cm205cd
brw-rw---- 1 root    disk      32,  0 Jan 30  2003 cm206cd
drwxr-xr-x 2 root    root          4096 Oct 24 17:43 compaq
crw----- 1 root    root         5,  1 Feb 11 15:15 console
crw----- 1 root    root         1,  6 Jan 30  2003 core
drwxr-xr-x 18 root    root          4096 Oct 24 17:43 cpu
crw-rw---- 1 root    uucp         5,  64 Jan 30  2003 cua0
crw-rw---- 1 root    uucp         5,  65 Jan 30  2003 cua1
crw-rw---- 1 root    uucp         5,  66 Jan 30  2003 cua2
crw-rw---- 1 root    uucp         5,  67 Jan 30  2003 cua3

```

crw-rw----	1 root	uucp	205,	16 Jan 30	2003 cuam0
crw-rw----	1 root	uucp	205,	17 Jan 30	2003 cuam1
crw-rw----	1 root	uucp	205,	18 Jan 30	2003 cuam2
crw-rw----	1 root	uucp	205,	19 Jan 30	2003 cuam3
crw-rw----	1 root	uucp	44,	0 Jan 30	2003 cui0

上面只列举了部分的设备，其中第一列中第一个字符为 **c** (character) 的行代表的是字符设备，为 **b** (block) 的行代表为块设备。在日期的前面有两个数字，并且用逗号分开，这两个数字就是相应设备的主设备号和次设备号。读者会问，用主设备号和次设备号干什么？通常主设备号用来标识该设备对应的驱动程序，而次设备号是由内核使用，用于正确确定设备文件所指的设备。上述设备中 `/dev/ console`, `/dev/ cua0`, `/dev/ cua1`, `/dev/ cua2`, `/dev/ cua3` 的主设备号都是 5，所以这些设备都由驱动程序 5 管理。现代的 Linux 内核允许多个驱动程序共享主设备号，但是大多数设备仍然按照“一个主设备号对应一个驱动程序”的原则安排。

### 6.1.2.1 主、次设备号的内部表示

在内核中，用 `dev_t` 类型来保存设备编号（包括主设备号和次设备号）。在 Linux 2.6.10 版本中，`dev_t` 是一个 32 位的数，其中用 12 位表示主设备号，而其余 20 位用来表示次设备号。要获得 `dev_t` 的主设备号和次设备号，应使用以下宏：

**n**     `MAJOR(dev)`

**n**     `MINOR(dev)`

这两个宏定义在 `<include/linux/kdev_t.h>` 文件中，具体定义如下：

```
#define MAJOR(dev)  ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)  ((unsigned int) ((dev) & MINORMASK))
```

通过定义可以看出，获得主、次设备号的方法就是通过左移和位与获得，相反，如果需要将主设备号和次设备号转换成 `dev_t` 类型，则使用以下宏：

**n**     `MKDEV(int major, int minor)`

这个宏的定义也在 `<include/linux/kdev_t.h>` 文件中，具体定义如下：

```
#define MKDEV(ma,mi)  (((ma) << MINORBITS) | (mi))
```

这个宏的实现通过对主设备号移位后然后与次设备号或而产生。关于主设备号的定义在 `<include/linux/major.h>` 文件中，关于设备号的分配可参考内核中 `<Documentation/devices.txt>` 文件。

在内核 2.6 以前，限定 255 个主设备号和 255 个次设备号，在计算机硬件飞速发展的时代，这些设备号已经不能满足大量设备的需求，所以在内核 2.6 版本中，它可以容量大于 255 个的设备。

### 6.1.2.2 静态分配和释放主设备号

在建立一个字符设备之前，首先要获得一个设备编号，在 Linux 2.6.10 内核中，该工作通过 `register_chrdev_region` 函数完成。该函数在 `include/linux/fs.h` 文件中声明，该函数的原型如下：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

其中，参数 `from` 是要分配的设备编号范围的起始值；参数 `count` 是所请求的连续设备编号的个数；如果 `count` 值非常大，则所请求的范围可能和下一个主设备号重叠，但是只要

所申请的编号范围是可用的，则不会带来任何问题。参数 **name** 是和该编号范围关联的设备名称。该函数在分配字符设备号成功时返回 0，在错误情况下，将返回一个负的错误码，并且不能使用所请求的编号区域。

无论采用 **register\_chrdev\_region** 静态分配设备号还是下一节介绍的 **alloc\_chrdev\_region** 函数动态分配设备编号，都必须在不再使用这些设备编号时释放它们，释放设备编号使用 **unregister\_chrdev** 函数，该函数的原型如下：

```
int unregister_chrdev(unsigned int major, const char *name)
```

其中，参数 **major** 为要释放设备的主设备号，参数 **name** 为相关的设备名称。

### 6.1.2.3 动态分配主设备号

如果我们提前知道所需要的设备编号，那么使用 **register\_chrdev\_region** 函数非常合适，但是经常我们不知道设备要使用哪些主设备号，在运行过程中通常使用 **alloc\_chrdev\_region** 函数，内核将会恰当地动态分配所需主设备号，该函数原型如下：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
                        const char *name)
```

其中，参数 **dev** 是仅用于输出的参数，在成功完成调用后将保存已分配范围的第一个编号；参数 **baseminor** 是要使用的被请求的第一个次设备号，它通常是 0；参数 **count** 为所请求的连续设备编号的个数；参数 **name** 是和该编号范围关联的设备名称。

对于一个新的驱动程序，笔者强烈建议不要随意选择一个当前没有被使用的设备号作为主设备号，而应该使用动态分配机制获得主设备号。也就是说，建议读者经常使用 **alloc\_chrdev\_region** 函数而取代 **register\_chrdev\_region** 函数。动态分配相对方便且安全，但是它也有缺点，由于分配的主设备号不能保证始终一致，所以无法预先创建设备节点，对于驱动程序的一般用法是从系统 **/proc/devices** 中读取得到。因此，为了加载一个使用动态主设备号的设备驱动程序，通常用 **insmod** 的调用替换为一个脚本，该脚本在调用 **insmod** 之后读取 **/proc/devices** 以获得新分配的主设备号，然后创建对应的设备文件。但是通过编写脚步来实现动态主设备号的方法还是有些麻烦，然而在实际中有另外一种更为简单的方法，默认采用动态分配，同时保留在加载或编译时指定主设备号的余地。具体实现是，定义一个全局主设备号变量 **major**，初始化该变量的值为 **MY\_MAJOR**，**MY\_MAJOR** 的默认值取 0，也就是选择动态分配。用户可以使用这个默认值或选择某个特定的主设备号，而且既可以在编译前修改宏定义，也可以通过 **insmod** 命令行指定 **major** 的值，以下是 **TTY** 设备利用这种方式分配主设备号的一段代码：

```
.....
if (!major)
{
    error = alloc_chrdev_region(&dev, minor_start, num, name);
    if (!error)
    {
        major = MAJOR(dev);
        driver->minor_start = MINOR(dev);
    }
}
else
{

```

```

        dev = MKDEV(major, minor_start);
        error = register_chrdev_region(dev, num, name);
    }
    if (error < 0)
    {
        kfree(p);
        return error;
    }
    .....

```

到此为止，对字符设备的一些基本知识有了一个大的介绍，下面让我们进入最为期待的字符驱动程序的实例开发。

## 6.2 字符设备驱动开发实例

为了使读者对字符设备驱动的开发有更深刻的认识，这里选择典型且实用的例子——Touch screen（触摸屏）设备驱动的开发。对于触摸屏的应用大家应该都比较熟悉，几乎大家都曾用到过。由于触摸屏设备使用简单并且美观，它的应用如今随处可见，工业控制系统、消费电子产品，甚至医疗设备上很多都装备了触摸屏输入装置。我们平时不经意间都会用到触摸屏设备，如在 ATM 机上取款、签署包裹，办理登机手续或查找电话号码时都可能会用到触摸屏。目前触摸屏技术不但应用到大型设备，而且现在也向小型移动终端发展，比如手机、MP3、MP4、掌上游戏机等等。随着技术的不断发展，应用触摸屏技术的消费类电子产品将会越来越多。常见的触摸屏分为：电阻式触摸屏，电容式触摸屏，声表面波式触摸屏和红外线扫描式触摸屏等。S3C2410 芯片内部包含了触摸屏接口，所以需要选择一种触摸屏来工作，我们这里使用应用最广泛的四线电阻式触摸屏。

### 6.2.1 四线电阻式触摸屏原理

四线电阻式触摸屏是电阻式触摸屏中应用最广、最普及的一种。其结构由下线路（玻璃或薄膜材料）导电 ITO（Indium Tin Oxides）\*（注 1）层和上线路（薄膜材料）导电 ITO 层组成。中间有细微绝缘点隔开，当触摸屏表面无压力时，上下线路成开路状态。一旦有压力施加到触摸屏上，上下线路导通，控制器通过下线路导电 ITO 层在 X 坐标方向上施加驱动电压，通过上线路导电 ITO 层上的探针，侦测 X 方向上的电压，由此推算出触点的 X 坐标。通过控制器改变施加电压的方向，同理可测出触点的 Y 坐标，从而明确触点的位置。四线电阻式触摸屏的等效电路如图 6.1 所示。

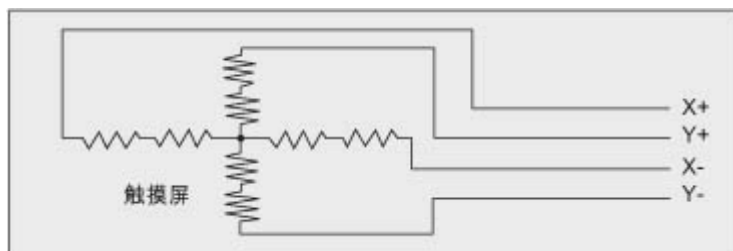


图 6.1 电阻式触摸屏的等效电路

\*注 1：ITO（Indium Tin Oxides）作为纳米铟锡金属氧化物，具有很好的导电性和透明性，可以切断对人

体有害的电子辐射，紫外线及远红外线。因此，喷涂在玻璃，塑料及电子显示屏上后，在增强导电性和透明性的同时切断对人体有害的电子辐射及紫外、红外线。

## 6.2.2 S3C2410 触摸屏工作原理

S3C2410 芯片支持触摸屏接口，其包含触摸屏控制器，四个外部晶体管，还有一个外部电压源。触摸屏接口控制，选择控制信号（nYPON, YMON, nXPON, 和 XMON）和模拟垫脚（analog pads）与触摸屏面板的垫脚和外部晶体管相连。触摸屏接口包含一个外部晶体管控制逻辑和一个 ADC（Analog-to-Digital Converter，模数转换器）接口逻辑（包含中断发生器逻辑）。图 6.2 给出了 S3C2410 的 A/D 转换和触摸屏接口的功能框图，注意，这里的 A/D 转化器是可复用类型的。一个上拉电阻连接在 AIN[7]和 VDDA\_ADC 之间，所以触摸屏面板的 XP 垫脚应该与 AIN[7]连接，触摸屏面板的 YP 垫脚应与 AIN[5]连接。

S3C2410 芯片提供的触摸屏接口模式有四种，这四种模式分别是：

### ü 正常转化模式

这种模式一般用于通用目的的 ADC 转化，其中（AUTO\_PST=0, XY\_PST=0）。该模式通过设置 ADCCON 和 ADCTSC 寄存器来初始化。

### ü 单独的 X/Y 位置转化模式

该模式由两个子模式组成：X 位置模式和 Y 位置模式。X 位置模式（AUTO\_PST=0, XY\_PST=1）写 X 位置转化数据到 ADCDAT0 寄存器的 XPDATA 位。转化之后，触摸屏接口产生中断源（INT\_ADC）到中断控制器。Y 位置模式（AUTO\_PST=0, XY\_PST=2）写 Y 位置转化数据到 ADCDAT1 寄存器的 YPDATA 位。转化之后，触摸屏接口产生中断源（INT\_ADC）到中断控制器。触摸屏面板在单独的 X/Y 位置转化模式的条件是：

单独的 X/Y 位置转化模式	XP	XM	YP	YM
X 位置转化	外部电压	GND（接地）	AIN[5]	Hi-Z（高阻抗）
Y 位置转化	AIN[7]	Hi-Z（高阻抗）	外部电压	GND（接地）

### ü 自动（有序的）X/Y 位置转化模式

这种模式（AUTO\_PST=0, XY\_PST=0）通常被操作在这些方式下：触摸屏控制器自动地转化 X 位置和 Y 位置。触摸屏控制器写 X 测量数据到 ADCDAT0 寄存器的 XPDATA 位，并且写 Y 测试数据到 ADCDAT1 寄存器的 YPDATA 位。在自动位置转化之后，触摸屏控制器产生中断源（INC\_ADC）到中断控制器。触摸屏面板在自动（有序）X/Y 位置转化模式的条件是：

自动（有序的）X/Y 位置转化模式	XP	XM	YP	YM
X 位置转化	外部电压	GND（接地）	AIN[5]	Hi-Z（高阻抗）
Y 位置转化	AIN[7]	Hi-Z（高阻抗）	外部电压	GND（接地）

### ü 等待中断模式

当触摸屏控制器在等待中断模式时，它将等待触摸笔按下。当触摸笔被按下时，控制器将产生一个中断信号（INT\_TC）。在中断发生之后，X 和 Y 位置被合适的转化模式（单独的 X/Y 位置转化模式或自动（有序）X/Y 位置转化模式）读取。触摸屏面板在等待中断模式的条件是：

N/A	XP	XM	YP	YM
-----	----	----	----	----

等待中断模式	上拉	Hi-Z（高阻抗）	AIN[5]	GND（接地）
--------	----	-----------	--------	---------

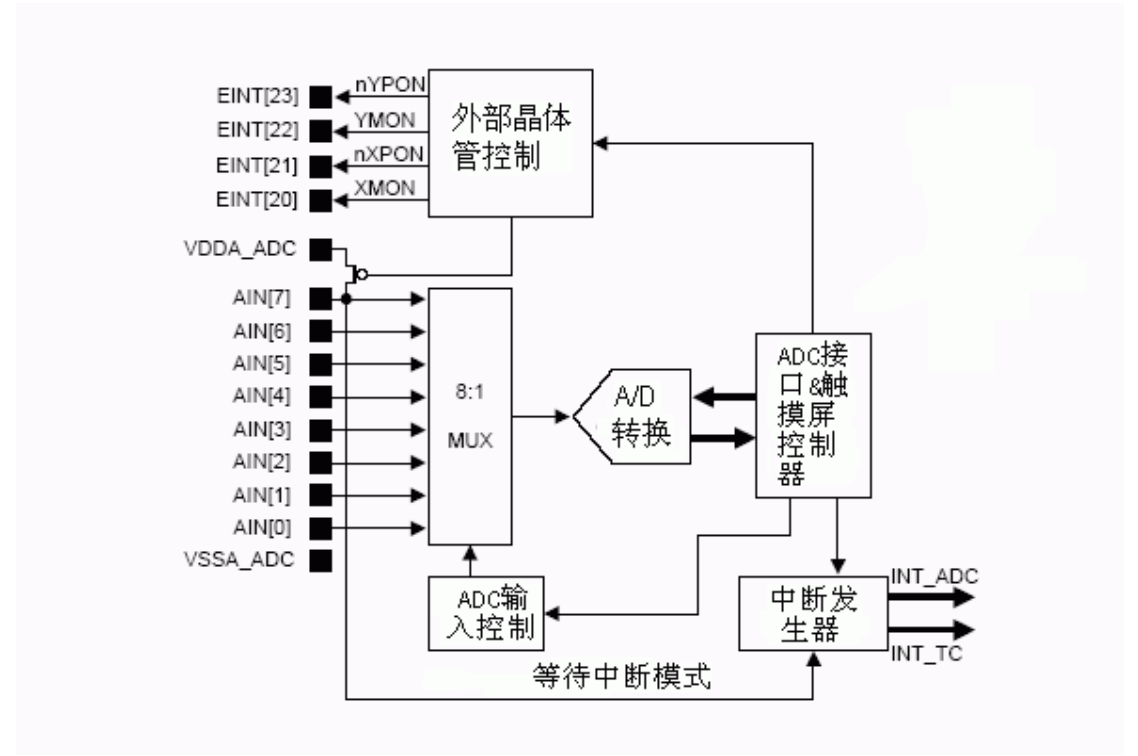


图 6.2 ADC 和触摸屏接口的功能框图

6.2.3 S3C2410 的 ADC 和触摸屏接口特殊寄存器

编写驱动程序最重要的一个过程就是阅读芯片用户手册中相关的寄存器，因为设备驱动程序的作用通常都是对相应寄存器的控制，所以现在让我们通过查看 S3C2410 芯片的用户手册来获得 ADC 和触摸屏接口的特殊寄存器。

6.2.3.1 ADC 控制（ADCCON）寄存器

寄存器	地址	R/W	描述	复位值
ADCCON	0x58000000	R/W	ADC控制寄存器	0x3FC4

ADCCON	位	描述	初始状态
ECFLG	[15]	结束转化标志（只读）。如果等于0，A/D转化正在处理，如果等于1，A/D转化结束	0
PRSCEN	[14]	使能A/D转化器的预分频器 如果等于0，禁止，如果等于1，则使能	0
PRSCVL	[13:6]	A/D转化器的预分频器值。数据值：1 ~ 255 之间，注意，当预分频器值是N时，分频因子是（N+1）。	0xFF

SEL_MUX	[5:3]	模拟输入信道选择。 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = AIN 4 101 = AIN 5 110 = AIN 6 111 = AIN 7 (XP)	0
STDBM	[2]	备用模式选择。 0 = 正常操作模式 1 = 备用模式	1
READ_START	[1]	读的A/D转化开始 0 = 禁止读操作的A/D转化开始 1 = 启动读操作的A/D转化开始	0
ENABLE_START	[0]	A/D 转化开始根据设定这个位。 如果READ_START 是启动的，这个值将无效。 0 = 无操作 1 = A/D 转化开始并且这个位将被清除在启动之后	0

### 6.2.3.2 ADC 触摸屏控制（ADCTSC）寄存器

寄存器	地址	R/W	描述	复位值
ADCTSC	0x58000004	R/W	ADC触摸屏控制寄存器	0x058

ADCTSC	位	描述	初始状态
保留	[8]	这位应该为0	0
YM_SEN	[7]	选择YMON的输出值。 0 = YMON 输出是0 (YM = 高阻抗). 1 = YMON 输出是1 (YM = 接地)	0
YP_SEN	[6]	选择nYPON的输出值。 0 = nYPON 输出是0 (YP = 外部电压) 1 = nYPON 输出是1 (YP 与AIN[5]连接)	1
XM_SEN	[5]	选择XMON的输出值。 0 = XMON输出是0 (XM = 高阻抗) 1 = XMON 输出是1 (XM =接地)	0
XP_SEN	[4]	选择nXPON的输出值 0 = nXPON 输出是0 (XP =外部电压) 1 = nXPON 输出是1 (XP与AIN[7]连接)	1
PULL_UP	[3]	选择上拉 0 = XP 启动上拉 1 = XP 上拉禁止	1
AUTO_PST	[2]	自动按顺序转化X位置和Y位置 0 = 正常的ADC 转化 1 = 自动（有序的）X/Y位置转化模式	0



XY_PST	[1:0]	手动测量X位置或Y位置 00 = 无操作模式 01 = X位置测量 10 = Y位置测量 11 = 等待中断模式	0
--------	-------	--	---

注意：在自动（有序的）X/Y位置转化模式，ADCTSC寄存器应该被重新配置在开始读操作之前。

### 6.2.3.3 ADC 开始延迟（ADCDLY）寄存器

寄存器	地址	R/W	描述	复位值
ADCDLY	0x58000008	R/W	ADC开始或间隔延迟寄存器	0x00ff

ADCDLY	位	描述	初始状态
DELAY	[15:0]	1. 正常转化模式，单独的X/Y位置转化模式和自动（有序的）X/Y位置转化模式。→X/Y位置转化延迟值。 2. 等待中断模式。当触摸笔按下发生在等待中断模式时，这个寄存器为自动X/Y位置转化而产生中断信号（INT_TC）在每间隔几毫秒中。 注意不要使用0值	0x00ff

注意：

1. 在 ADC 转化之前，触摸屏使用 X-tal 时钟或外部始终（等待中断模式）
2. 在 ADC 转化期间，触摸屏使用 PCLK 时钟。

### 6.2.3.4 ADC 转化数据 (ADCDAT0) 寄存器

寄存器	地址	R/W	描述	复位值
ADCDAT0	0x5800000C	R	ADC转化数据寄存器	—

ADCDAT0	位	描述	初始状态
UPDOWN	[15]	在等待中断模式，触摸笔的抬起或按下状态 0=触摸笔按下状态 1=触摸笔抬起状态	—
AUTO_PST	[14]	自动有序的转化X位置和Y位置 0=正常ADC转化 1=按顺序自动的测量X位置和Y位置	—
XY_PST	[13:12]	手动测量X位置或Y位置 00=无操作模式 01=X位置测量 10=Y位置测量 11=等待中断模式	—
保留	[11:10]	保留	
XPDATA	[9:0]	X位置转化数据值（包括正常ADC转化数据值）	—

(正常的ADC)		数据值: 0 ~ 3FF	
----------	--	--------------	--

### 6.2.3.5 ADC 转化数据(ADCDAT1)寄存器

寄存器	地址	R/W	描述	复位值
ADCDAT1	0x58000010	R	ADC转化数据寄存器	-

ADCDAT1	位	描述	初始状态
UPDOWN	[15]	在等待中断模式, 触摸笔的抬起或按下状态 0=触摸笔按下状态 1=触摸笔抬起状态	—
AUTO_PST	[14]	自动有序的转化X位置和Y位置 0=正常ADC转化 1=按顺序自动的测量X位置和Y位置	—
XY_PST	[13:12]	手动测量X位置或Y位置 00=无操作模式 01=X位置测量 10=Y位置测量 11=等待中断模式	—
保留	[11:10]	保留	
YPDATA	[9:0]	Y位置转化数据值 (包括正常ADC转化数据值) 数据值: 0 ~ 3FF	—

## 6.2.4 触摸屏驱动概要设计

基于 S3C2410 芯片设计触摸屏驱动接口是一个很实用的例子, 首先让我们来看一下该系统的硬件接口连接。

### 6.2.4.1 触摸屏硬件接口

对于本节中所列举的触摸屏实例的接口连接图如 6.3 所示, AIN[7]与 XP 连接, AIN[5]与 YP 连接, 为了控制触摸屏面板的垫脚 (XP, XM, YP 和 YM), 使用四个外部晶体管和四个控制信号 (nYPON, YMON, nXPON 和 XMON), 这四个控制信号分别与四个外部晶体管相连。获得 X/Y 位置坐标时选择单独的 X/Y 位置转化模式或自动 (有序的) X/Y 位置转化模式; 一般情况下设置触摸屏接口为等待中断模式, 如果中断发生, 合适的触摸屏接口模式将被激活; 在获得正确的 X/Y 位置坐标值后, 系统将返回到等待中断模式。注意, 外部电压 (VCC) 应该为 3.3V, 外部晶体管的电阻值应该小于 5 欧姆。

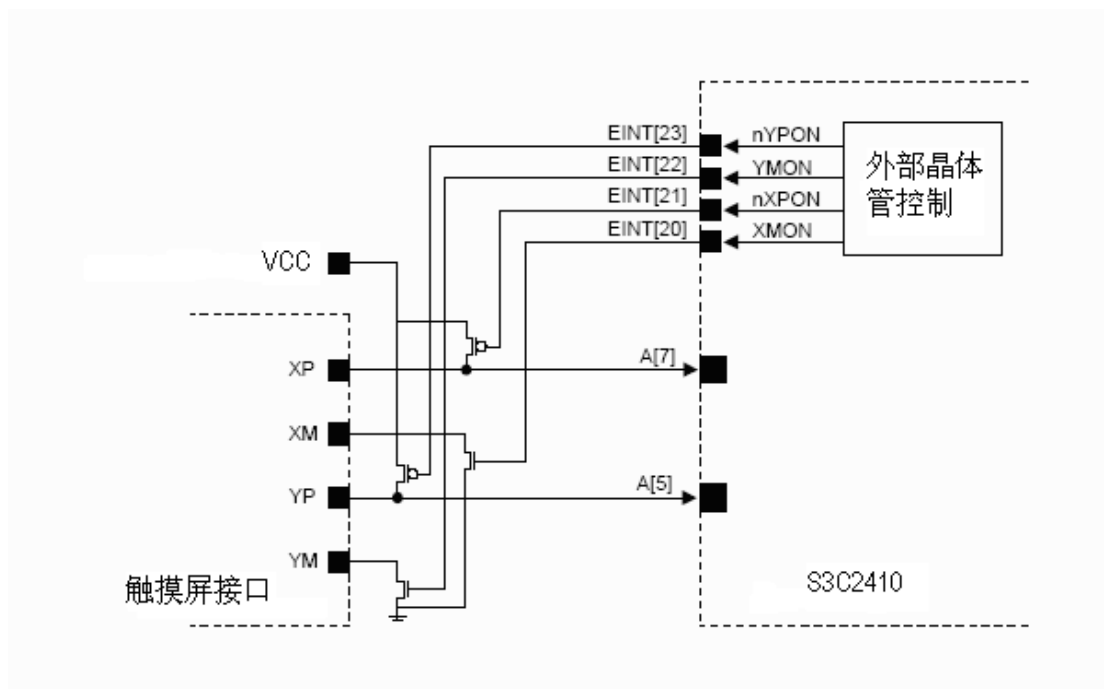


图 6.3 S3C2410 触摸屏接口连接图

#### 6.2.4.2 触摸屏驱动程序流程设计

接下来需要设计驱动程序所需要实现的主要功能，该驱动程序需要实现以下 5 个主要任务：

1. 配置触摸屏控制器硬件
2. 判断屏幕是否被触摸
3. 获得稳定的、去抖动的位置测量数据
4. 校准触摸屏
5. 将触摸状态和位置变化信息发送给更高层的图形软件

确定驱动程序所要实现的任务之后，接下来需要设计它的工作流程，本实例的流程图如图 6.4 所示。该设备驱动程序的工作流程是：首先初始化触摸屏控制器为等待中断模式，同时初始化计时器为延迟 10 毫秒后检查一次，映射 INC\_ADC、INC\_TC 和定时器中断向量到相应的中断服务程序。然后使能中断，并且使计时器准备就绪，当触摸笔按下时，触摸屏中断开始工作，同时启动定时器，等待 10 毫秒后查看是否有按下事件发生，如果是则确定触摸笔按下，获得触摸点坐标信息，并进行相应的处理。如果没有按下返回继续进行判断。

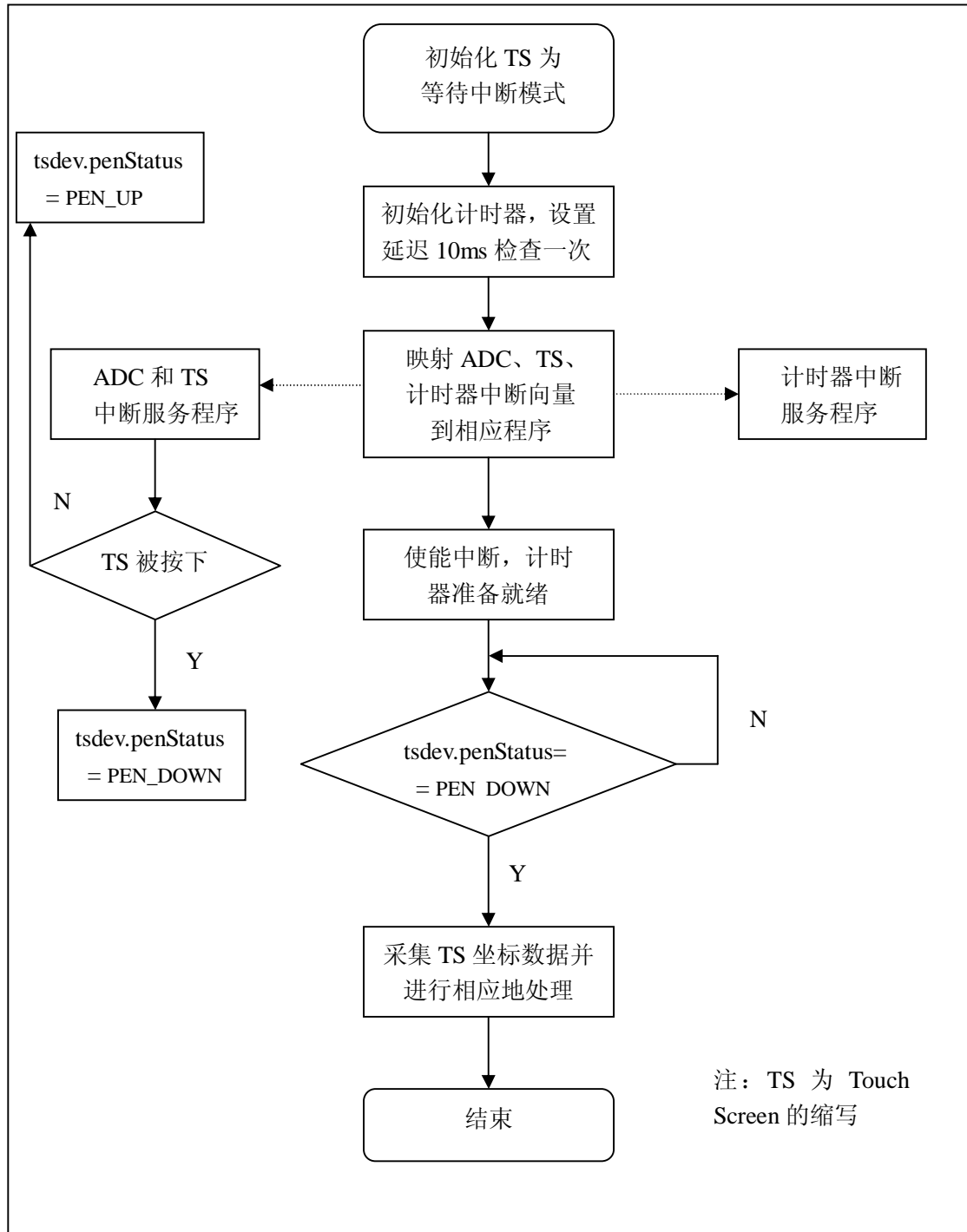


图 6.4 触摸屏工作流程

### 6.2.5 触摸屏驱动程序分析

下面就要看到触摸屏驱动实现的具体代码了，为了便于读者理解，这里按照实现该驱动程序的逻辑顺序讲解，首先讲述触摸屏设备初始化。

### 6.2.5.1 触摸屏设备初始化

由于触摸屏设备具有字符设备的特点，所以将它作为字符设备类型来实现，即对触摸屏设备注册就是对字符设备注册。关于触摸屏的初始化是通过调用 `s3c2410_ts_init` 模块加载函数实现的，该函数的实现代码如下：

```
int __init s3c2410_ts_init(void)
{
    return driver_register(&s3c2410_ts_driver);
}
```

在上面的加载函数中，其调用 `driver_register` 函数来注册驱动程序本身。驱动程序的注册，包括驱动程序本身的注册和设备的注册。驱动程序本身的注册先进行，即由上述的 `driver_register` 函数来完成。而设备注册程序则应当写在驱动程序的 `probe` 代码中，在检测设备的时候进行注册，该触摸屏设备的注册是由变量 `s3c2410_ts_driver` 的 `probe` 方法实现的。关于 `s3c2410_ts_driver` 变量的定义如下：

```
static struct device_driver s3c2410_ts_driver = {
    .name          = DEVICE_NAME,
    .bus           = &platform_bus_type,
    .probe         = s3c2410_ts_probe,
    .remove        = s3c2410_ts_remove,
};
```

从上面的代码可以看出，`s3c2410_ts_driver` 变量定义了两个方法，分别是 `probe` 和 `remove`。其中 `probe` 方法是由 `s3c2410_ts_probe` 函数实现，它的主要功能就是对触摸屏设备的注册，包括了硬件的初始化工作。`remove` 方法是由 `s3c2410_ts_remove` 函数实现，它的功能与 `s3c2410_ts_probe` 函数相反，完成设备的注销功能。首先分析一下 `s3c2410_ts_probe` 函数的具体实现，实现代码如下：

```
1  static int __init s3c2410_ts_probe(struct device *dev)
2  {
3      int ret = 0;
4
5      tsEvent = tsEvent_dummy;
6      adc_clock = clk_get(NULL, "adc");
7      if (!adc_clock) {
8          printk(KERN_ERR "failed to get adc clock source\n");
9          return -ENOENT;
10     }
11     clk_use(adc_clock);
12     clk_enable(adc_clock);
13
14     base_addr=ioremap(S3C2410_PA_ADC,0x20);
15     if (base_addr == NULL) {
16         printk(KERN_ERR "Failed to remap register block\n");
17         return -ENOMEM;
18     }
19     if(alloc_chrdev_region(&chrdev,0,1,"ts")){
```

```

20         printk(KERN_ERR"Couldn't alloc chrdev region\n");
21         return 1;
22     }
23     cdev_init(&ts,&s3c2410_fops);
24     if(cdev_add(&ts, chrdev, 1)){
25         unregister_chrdev_region(chrdev,1);
26         printk(KERN_ERR"Couldn't register ts driver\n");
27         return 1;
28     }
29
30     s3c2410_gpio_cfgpin(S3C2410_GPG12, S3C2410_GPG12_XMON);
31     s3c2410_gpio_cfgpin(S3C2410_GPG13, S3C2410_GPG13_nXPON);
32     s3c2410_gpio_cfgpin(S3C2410_GPG14, S3C2410_GPG14_YMON);
33     s3c2410_gpio_cfgpin(S3C2410_GPG15, S3C2410_GPG15_nYPON);
34
35     if (request_irq(IRQ_ADC, s3c2410_isr_adc, SA_SAMPLE_RANDOM,
36         "s3c2410_action", &chrdev)) {
37         printk(KERN_ERR " Could not allocate ts IRQ_ADC !\n");
38         iounmap(base_addr);
39         return -EIO;
40     }
41     if (request_irq(IRQ_TC, s3c2410_isr_tc, SA_SAMPLE_RANDOM,
42         "s3c2410_action", &chrdev)) {
43         printk(KERN_ERR " Could not allocate ts IRQ_TC !\n");
44         iounmap(base_addr);
45         free_irq(IRQ_ADC,&chrdev);
46         return -EIO;
47     }
48     writel(wait_down_int(), base_addr+S3C2410_ADCTSC);//wait_down_int();
49     ret  =  devfs_mk_cdev(chrdev,S_IFCHR | S_IRUGO | S_IWUSR,
DEVICE_NAME);
50     if(ret)
51         goto out_chrdev;
52
53     writel(0xFFFF, base_addr+S3C2410_ADCDLY);
54     printk(KERN_INFO "Tochu screen successfully loaded\n");
55
56     goto out;
57 out_chrdev:
58     unregister_chrdev(chrdev, DEVICE_NAME);
59 out:
60     return ret;
61 }

```

现在对上面的代码进行分析，第6行，用内核提供的时钟API函数clk\_get获得ADC时钟

源。第11-12行，应用和使能ADC时钟源。第14行，其中*ioremap()*的作用是把一个物理内存地址映射为一个内核指针，被映射数据的长度由size参数设定。该函数的实质是把一块物理区域二次映射到一个可以从驱动程序里访问的虚拟地址上去。第19行，用*alloc\_chrdev\_region*函数来获得设备的主设备号和将设备的名称记录到内核的字符设备链表中，该方法将会动态的分配设备号，该函数的功能在前面章节中已经讲过。第23行，用*cdev\_init*函数来初始化cdev（字符设备）结构。第24-28行，用*cdev\_add*函数将字符设备加入到内核的字符设备数组中，如果没有成功加入到内核的字符设备数组中，则需要调用*unregister\_chrdev\_region*函数来释放占用的设备号。第30-33行，用来配置S3C2410触摸屏控制端口功能，即配置GPIO\*（注2）（General-Purpose I/O，通用输入输出端口）。第35-47行，分别注册IRQ\_ADC（ADC中断）和IRQ\_TC（触摸屏中断）。第48行，设置触摸屏接口为等待中断模式。第49行，用*devfs\_mk\_cdev*函数自动创建字符设备文件在/dev目录下。对于2.4内核，利用*devfs\_register*函数注册设备文件。第50-51行，如果没有成功注册设备文件，那么需要注销字符设备同时释放占用的设备号，否则返回0表示触摸屏字符设备正确初始化。总之，这段程序完成了触摸屏设备的初始化工作，包括触摸屏字符设备的注册，GPIO的设置，注册中断IRQ\_ADC和IRQ\_TC，初始化触摸屏为等待中断模式，以及建立触摸屏设备目录等。

\*注2：GPIO是General-Purpose I/O的缩写，它映射到CPU的memory map中，可以把一组GPIO当作一个寄存器，该寄存器的每一位对应一个GPIO引脚，因此你可以用内存访问指令来设置和读取GPIO引脚上的信号以驱动外设。

接下来顺便介绍一下s3c2410\_ts\_remove函数的具体实现，它的作用与上述初始化函数功能相反，它的实现代码如下：

```

1 static int s3c2410_ts_remove(struct device *dev)
2 {
3     unregister_chrdev(chrdev, DEVICE_NAME);
4
5     disable_irq(IRQ_ADC);
6     disable_irq(IRQ_TC);
7     free_irq(IRQ_TC,&chrdev);
8     free_irq(IRQ_ADC,&chrdev);
9
10    if (adc_clock) {
11        clk_disable(adc_clock);
12        clk_unuse(adc_clock);
13        clk_put(adc_clock);
14        adc_clock = NULL;
15    }
16
17    iounmap(base_addr);
18    return 0;
19 }
```

对上述s3c2410\_ts\_remove函数的实现代码进行分析，第3行，用来注销字符设备同时释放占用的设备号。第5-8行，先是禁止IRQ\_ADC中断和IRQ\_TC中断，然后释放IRQ\_ADC中断和IRQ\_TC中断。第10-15行，用来释放ADC时钟源。第17行，iounmap()函数用于取消ioremap()所做的地址映射。第18行，返回0表示注销字符设备和其它退出操作完成。

其中s3c2410\_ts\_init函数指定为启动触摸屏设备的入口函数，而s3c2410\_ts\_exit指定为退

出触摸屏设备的卸载函数，s3c2410\_ts\_exit函数的实现代码如下，它只调用了driver\_unregister函数来注销驱动程序。

```
void __exit s3c2410_ts_exit(void)
{
    driver_unregister(&s3c2410_ts_driver);
}
```

### 6.2.5.2 触摸屏设备文件操作

关于设备文件操作（file\_operations）在前面的章节中已经介绍，它是一个非常重要的数据结构。通过对 file\_operations 结构体对象的定义，从而知道访问驱动函数都提供哪些操作。该触摸屏字符设备文件操作的对象具体定义如下：

```
static struct file_operations s3c2410_fops = {
    owner:    THIS_MODULE,
    open:     s3c2410_ts_open,
    read:     s3c2410_ts_read,
    release:  s3c2410_ts_release,
    fasync:   s3c2410_ts_fasync,
    poll:    s3c2410_ts_poll,
};
```

该文件操作对象定义了 open，read，release，fasync 和 poll 方法，并且定义了 owner 属性为 THIS\_MODULE。其中 open 方法由 s3c2410\_ts\_open 函数实现，read 方法由 s3c2410\_ts\_read 函数实现，release 方法由 s3c2410\_ts\_release 函数实现，fasync 方法由 s3c2410\_ts\_fasync 函数实现和 poll 方法由 s3c2410\_ts\_poll 函数实现。下面将分别介绍这些文件操作对象中定义的方法实现。

### 6.2.5.3 open 和 release 方法

设备驱动中 open 方法是用来为以后的操作完成初始化准备工作的，并且通常字符设备驱动程序都会实现它。在大部分驱动程序中，open 方法完成如下工作：

- ü 检查设备相关错误（诸如设备未就绪或相似的硬件问题）。
- ü 如果是首次打开，初始化设备。
- ü 标识次设备号，如有必要更新 f\_op 指针。
- ü 分配和填写要放在 filp->private\_data 里的数据结构。

该字符设备的 open 方法实现根据文件操作 s3c2410\_fops 的定义，是由 s3c2410\_ts\_open 函数实现的，具体实现代码如下：

```
1 static int s3c2410_ts_open(struct inode *inode, struct file *filp)
2 {
3     tsdev.head = tsdev.tail = 0;
4     tsdev.penStatus = PEN_UP;
5     init_timer(&ts_timer);
6     ts_timer.function = ts_timer_handler;
7     tsEvent = tsEvent_raw;
```



```

8      init_waitqueue_head(&(tsdev.wq));
9
10     return 0;
11 }

```

分析 s3c2410\_ts\_open 函数的实现代码，第 3-4 行，初始化自定义的触摸屏设备结构变量 tsdev，它是 TS\_DEV 结构体的对象，关于 TS\_DEV 结构在后面会有介绍。第 5 行，用 init\_timer 函数初始化定时器 ts\_timer，ts\_timer 是全局的 timer\_list 结构体变量，其中 timer\_list 结构用于在未来某一个特定时刻执行某一系列特定任务的功能。第 6 行，初始化定时器处理函数为 ts\_timer\_handler，ts\_timer\_handler 函数是在定时器时间到时被调用的。第 7 行，用于初始化原始的触摸屏事件。第 8 行，初始化触摸屏设备的等待队列。其中上面提到的全局变量 tsdev 是 TS\_DEV 结构体的对象，TS\_DEV 结构体的定义如下：

```

typedef struct {
    unsigned int penStatus;      /* PEN_UP, PEN_DOWN, PEN_SAMPLE */
    TS_RET buf[MAX_TS_BUF];     /* 缓存最大 8 个关于触摸屏触摸信息*/
    unsigned int head, tail;    /* 队列事件的头和尾*/
    wait_queue_head_t wq; /* 等待队列的头*/
    spinlock_t lock; /* 定义一个自旋锁*/
    struct fasync_struct *aq; /*定义一个异步结构指针*/
} TS_DEV;

```

下面来看一下与 open 方法作用完全相反的 release 方法，这个设备方法有时也称为 close。它一般完成以下任务：

- ü 释放 open 分配在 filp->private\_data 中的内存。
- ü 在最后一次关闭操作时关闭设备。

该触摸屏 release 方法是由 s3c2410\_ts\_release 函数完成，它的具体实现如下：

```

static int s3c2410_ts_release(struct inode *inode, struct file *filp)
{
    del_timer(&ts_timer);
    return 0;
}

```

该函数的内容很简单，只是做了删除定时器然后返回的功能，这是由于在 open 方法中它只实现了初始化定时器的功能。

#### 6.2.5.4 read 和 poll 方法

通常情况下设备驱动程序会实现 read 和 write 方法，read 是从设备读取数据，而 write 是向设备发送数据。通常不需要发送数据给触摸屏设备，所以这里不需要实现 write 方法，而只需要实现 read 方法即可。这里 read 方法是由 s3c2410\_ts\_read 函数来实现，它的具体实现代码如下：

```

1  static ssize_t s3c2410_ts_read(struct file *filp, char *buffer, size_t count, loff_t *ppos)
2  {
3      TS_RET ts_ret;
4  retry:
5      if (tsdev.head != tsdev.tail) {
6          int count;

```

```

7         count = tsRead(&ts_ret);
8         if (count) copy_to_user(buffer, (char *)&ts_ret, count);
9         return count;
10    } else {
11        if (filp->f_flags & O_NONBLOCK)
12            return -EAGAIN;
13        interruptible_sleep_on(&(tsdev.wq));
14        if (signal_pending(current))
15            return -ERESTARTSYS;
16        goto retry;
17    }
18
19    return sizeof(TS_RET);
20 }

```

分析 s3c2410\_ts\_read 函数的代码，第 5-9 行，当触摸屏事件队列中有事件时，读取触摸屏坐标信息，真正读取函数为 tsRead。此外还有一个内核中非常重要的函数，即 copy\_to\_user，它的作用是复制内核空间数据到用户空间，与 copy\_to\_user 内核函数作用类似的还有 copy\_from\_user，该函数的作用是复制用户空间的数据到内核空间，这两个函数的作用类似 memcpy 函数，但还是和 memcpy 是有区别的，不然的话就直接用 memcpy 函数，当内核空间内运行的代码访问用户空间时，被寻址的用户空间的页面可能当前不在内存中，此时虚拟内存子系统会将该进程转入睡眠状态，直到该页面被传送到期望的页面。对于驱动程序编程人员来说，访问用户空间的任何函数都必须是可重入的，并且必需能和其他驱动程序并发执行，同时必须处于能够合法 sleep 的状态。这两个函数除了在内核空间与用户空间之间拷贝数据之外，它们还会检查用户空间的地址是否有效，如果指针无效就不会进行拷贝，如果在拷贝过程中发现无效指针，则仅仅会复制部分正确的数据。第 10-16 行，当触摸屏事件队列为空时，如果定义为非阻塞操作，则直接返回 -EAGAIN 错误信息。打开可中断睡眠队列，此外，检查当前进程是否有信号处理，当 signal\_pending(current) 为非 0 时表示有信号需要处理，则返回 -ERESTARTSYS 表示信号函数处理完毕后重新执行信号函数前的某个系统调用。最后执行 goto 语句调用 retry。第 19 行，返回 TS\_RET 结构体的大小，也就是表示读取字节的个数。

由于触摸屏设备驱动的读取会可能会被阻塞，所以该程序实现了 poll 方法文件操作，该方法分两步处理：一是在一个或多个可指示 poll 状态变化的等待队列上调用 poll\_wait 函数，如果当前没有文件描述符可用来执行 I/O，则内核将使进程在传递到该系统调用的所有文件描述符对应的等待队列上等待；二是返回一个用来描述操作是否可以立即无阻塞执行的位掩码。该触摸屏设备驱动的 poll 方法是由 s3c2410\_ts\_poll 函数实现，具体实现代码如下：

```

static unsigned int s3c2410_ts_poll(struct file *filp, struct poll_table_struct *wait)
{
    poll_wait(filp, &(tsdev.wq), wait);
    return (tsdev.head == tsdev.tail) ? 0 : (POLLIN | POLLRDNORM);
}

```

#### 6.2.5.5 触摸屏中断和 ADC 中断的实现

该触摸屏设备驱动最重要的实现部分就是关于中断的实现了，该驱动分两个中断，一个

是触摸屏中断（IRQ\_TC），另一个是 ADC 中断（IRQ\_ADC）。这两个中断及其相关的实现代码如下：

```
.....
/*开始 ADC，启动单独的 X 位置转化模式 */
static inline void start_ts_adc(void)
{
    adc_state = 0;
    writel(mode_x_axis(), base_addr+S3C2410_ADCTSC); //mode_x_axis();
    writel(start_adc_x(), base_addr+S3C2410_ADCCON); // start_adc_x();
    tmp=readl(base_addr+S3C2410_ADCCON);
    tmp &=~(S3C2410_ADCCON_STDBM);
    writel(tmp, base_addr+S3C2410_ADCCON);
    readl(base_addr+S3C2410_ADCDAT0);
}
/*获得 ADC 转化后的 X, Y 位置 */
static inline void s3c2410_get_XY(void)
{
    if (adc_state == 0) {
        adc_state = 1;
        tmp = readl(base_addr+S3C2410_ADCCON);
        tmp &= ~(S3C2410_ADCCON_READ_START);
        writel(tmp, base_addr+S3C2410_ADCCON);
        y = (readl(base_addr+S3C2410_ADCDAT0) & 0x3ff);
        writel(mode_y_axis(), base_addr+S3C2410_ADCTSC);
        writel(start_adc_y(), base_addr+S3C2410_ADCCON);
        tmp=readl(base_addr+S3C2410_ADCCON);
        tmp &=~(S3C2410_ADCCON_STDBM);
        writel(tmp, base_addr+S3C2410_ADCCON);
        readl(base_addr+S3C2410_ADCDAT1);
    }
    else if (adc_state == 1) {
        adc_state = 0;
        tmp = readl(base_addr+S3C2410_ADCCON);
        tmp &= ~(S3C2410_ADCCON_READ_START);
        writel(tmp, base_addr+S3C2410_ADCCON);
        x = (readl(base_addr+S3C2410_ADCDAT1) & 0x3ff);
        tsdev.penStatus = PEN_DOWN;
        DPRINTK("PEN DOWN: x: %08d, y: %08d\n", x, y);
        writel(wait_up_int(), base_addr+S3C2410_ADCTSC);
        tsEvent();
    }
}
/*ADC 中断服务程序*/
static irqreturn_t s3c2410_isr_adc(int irq, void *dev_id, struct pt_regs *reg)
```

```

    {
        spin_lock_irq(&(tsdev.lock));
        s3c2410_get_XY();
        spin_unlock_irq(&(tsdev.lock));
        return IRQ_HANDLED;
    }
    /*触摸屏中断服务程序*/
    static irqreturn_t s3c2410_isr_tc(int irq, void *dev_id, struct pt_regs *reg)
    {
        spin_lock_irq(&(tsdev.lock));
        if (tsdev.penStatus == PEN_UP) {
            start_ts_adc();
        }
        else {
            tsdev.penStatus = PEN_UP;
            DPRINTK("PEN UP: x: %08d, y: %08d\n", x, y);
            writel(wait_down_int(), base_addr+S3C2410_ADCTSC);
            tsEvent();
        }
        spin_unlock_irq(&(tsdev.lock));
        return IRQ_HANDLED;
    }
}

```

对于初学驱动程序开发的读者来说，对中断服务程序的执行经常会感觉到迷惑，因为笔者曾经为此也苦恼过。所以在这里会详细的介绍上述代码的执行过程：当触摸屏面板感应到有触摸笔触摸屏幕时，这时就会产生触摸屏中断，也就是调用触摸屏的中断服务程序（s3c2410\_isr\_tc），如果是第一次触发触摸屏中断，那么触摸笔的状态一定是 PEN\_UP，所以将调用 start\_ts\_adc 函数，start\_ts\_adc 函数用来转化系统为单独的 X 位置转化模式，同时设置 ADC 控制寄存器，触发 ADC 中断，也就是调用 ADC 中断服务程序（s3c2410\_isr\_adc），s3c2410\_isr\_adc 函数通过调用 s3c2410\_get\_XY 函数来获得 ADC 转化后的 X 和 Y 位置，在 s3c2410\_get\_XY 函数中有一个全局变量 adc\_state，通过判断该变量来确定是否已经获得 Y 位置，如果 adc\_state 等于 0 表明还没有获得 Y 位置，所以先获得 Y 位置，并且设置 adc\_state 为 1，同时转化系统为单独的 Y 位置转化模式，从而又触发 ADC 中断，即调用 ADC 中断服务程序（s3c2410\_isr\_adc），此时 adc\_state 等于 1，从而获得 X 位置，设置触摸笔的状态为 PEN\_DOWN，并设置系统为等待中断模式，此时 X 和 Y 位置都已经获得，通过 tsEvent 函数将获得的坐标数据发送到触摸屏的事件缓冲（tsdev.buf[MAX\_TS\_BUF]）中。上层应用程序通过读取事件缓冲 tsdev.buf[MAX\_TS\_BUF] 中的值来获取坐标。关于触摸屏中断的代码具体分析可以留给读者自己分析，这里已经介绍了它的主要实现功能。

## 6.2.6 配置和编译驱动程序

编译内核驱动程序不同于一般的应用程序，它通常放在内核代码 driver/ 目录下的相应位置。由于我们所开发的触摸屏设备属于字符设备，所以将源程序放在了 driver/char 目录下。同时需要修改该目录下 Kconfig 配置文件，添加以下内容到该文件：

```

config TOUCHSCREEN_S3C2410
    tristate "Samsung S3C2410 touchscreen input driver"
    depends on ARCH_SMDK2410
    select SERIO
    help
        Say Y here if you have the s3c2410 touchscreen.
        If unsure, say N.
        To compile this driver as a module, choose M here: the
        module will be called s3c2410_ts.

config TOUCHSCREEN_S3C2410_DEBUG
    boolean "Samsung S3C2410 touchscreen debug messages"
    depends on TOUCHSCREEN_S3C2410
    help
        Select this if you want debug messages

```

添加以上内容后,在用 make menuconfig 配置内核时会增加关于 S3C2410 触摸屏的选项,此外,还需要修改 driver/char/Makefile 文件,否则,触摸屏驱动源程序将不能被编译,添加下面一行内容:

```
obj-$(CONFIG_TOUCHSCREEN_S3C2410) += s3c2410_ts.o
```

修改完配置文件和 Makefile 文件,最后重新配置内核选项,将触摸屏选项以模块形式添加或直接添加到内核,然后就可以编译该驱动程序了。

## 6.2.7 测试触摸屏驱动程序

现在我们可以通过简单的程序对编写的触摸屏驱动程序进行测试,测试程序相对比较简单,代码如下:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/ioctl.h>

#define TS_DEV "/dev/TS "
static int ts_fd=-1;

typedef struct {
    unsigned short pressure;
    unsigned short x;
    unsigned short y;
    unsigned short pad;
} TS_RET;

```

```

int main()
{
    TS_RET data;

    if((ts_fd=open(TS_DEV,O_RDONLY))<0)
    {
        printf("Error opening %s device!\n",TS_DEV);
        return -1;
    }

    while(1)
    {
        read(ts_fd,&data,sizeof(data));
        printf("x=%d,y=%d,pressure=%d\n",data.x,data.y,data.pressure);
    }

    return 0;
}

```

通过阅读上面的测试程序,不难看出驱动程序提供给应用程序的正是那些实现的文件操作,例如本测试程序类似应用程序,其调用了 `open` 和 `read` 方法,而这两个方法的具体实现是由底层驱动程序实现。通过对触摸屏设备驱动程序的分析,相信读者对字符设备驱动程序的编写有了更进一步的理解,相信读者通过自身的努力一定能够编写出自己的字符设备驱动程序。

## 6.2.8 触摸屏的校准

在实际应用中,触摸屏和显示屏配合作为输入设备,为了能使从触摸屏采样得到的坐标与屏幕的显示坐标对应,需要一个映射过程,该映射的过程就是触摸屏的校准。函数 `TS_Coordinate_Conversion` 完成触摸屏采样值转换成显示坐标,根据不同的硬件有不同的转换方法。理想的触摸屏映射示意图如 6.5 所示,本触摸屏采样坐标及显示坐标与图 6.5 类似。其中 `TS_MAX_X` 和 `TS_MIN_X` 是触摸屏 X 坐标采样值的最大和最小值。这里使用的是 320×240 的 TFT 屏,所以 `TFT_X` 值为 320, `TFT_Y` 值为 240。假使触摸屏的最左上角坐标为  $(X_1, Y_1)$ , 最右下角坐标为  $(X_2, Y_2)$ , 显示屏的最左上角坐标为  $(x_1, y_1)$ , 最右下角坐标为  $(x_2, y_2)$ , 由于这里使用的分辨率是 320×240, 所以,  $x_2$  与  $x_1$  的差值是 320,  $y_2$  与  $y_1$  的差值是 240, 假定以左上角坐标为  $(x_1, y_1)$  为原点  $(0, 0)$ 。那么在触摸屏上任意一点  $(x, y)$  的坐标影射到显示屏上坐标  $(X, Y)$  的转换公式为:

$$X=320-[320*(x- X_2)/( X_1 - X_2)];$$

$$Y=240-[240*(y- Y_2)/( Y_1- Y_2)];$$

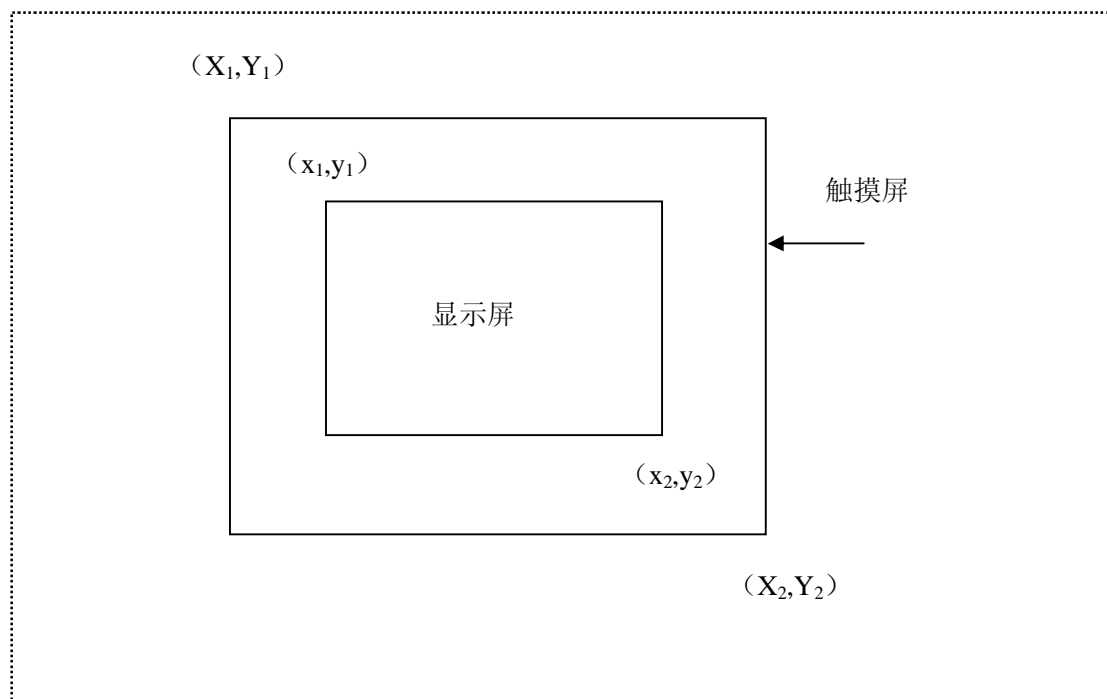


图 6.5 触摸屏与显示屏的坐标映射

下面是触摸屏中任意一点  $x$  坐标映射到显示屏的处理程序：

```
int TS_Coordinate_Conversion(int*x)
{
    int tempX;

    tempX=X2;
    *x=(tempX*TFT_X)/(X1-X2);
    *x=TFT_X-*x;

    return *x;
}
```

利用类似的方法可以计算  $Y$  值坐标的映射，读者可以自己完成。这里使用的触摸屏校准方法是实际应用中最简单的，因为这里使用的是简单的线形算法，而实际中可能是非线形的，所以计算方法上要复杂许多，不过由于我们这里并不是学习数学算法，所以只给出了简单情况下的例子。

## 6.3 本章小节

本章作为嵌入式 Linux 驱动开发中应用最广泛的字符设备驱动程序进行了详细的讲解，首先讲述了字符设备驱动中最重要的三个数据结构（`file_operations`、`file` 和 `inode`），然后讲述主、次设备号的使用，最后重点讲述一个字符设备驱动的典型实例——触摸屏设备驱动，不仅讲述该设备驱动程序的硬件原理，而且重点讲述了其驱动程序的实现过程。总之，通过学习本章知识，使读者真正了解字符设备驱动程序的实现过程，从而使读者熟悉了字符设备驱动程序的实际开发。下一章讲述与字符设备驱动实现方法不同的块设备驱动程序。

## 6.4 常见问题

1. 结构 `file_operations`, `file` 和 `inode` 在字符设备驱动程序中的作用？

参考答案： `file_operations` 结构用来定义驱动程序的文件操作，也就是定义驱动程序中底层实现的方法，如 `open`, `read`, `write` 等。而 `file` 结构代表一个打开的文件，它由内核在 `open` 时创建，并传递给在该文件上进行操作的所有函数，直到最后的 `close` 函数来释放这个数据结构。`inode` 结构表示内部文件，与 `file` 结构不同，`file` 表示打开的文件描述符，对单个文件，可能会有许多打开的文件描述符的 `file` 结构，但它们都指向单个 `inode` 结构。

2. 在设备驱动程序中，`open` 方法通常完成哪些工作？

参考答案：在设备驱动程序中，`open` 方法通常完成以下工作：

- ü 检查设备相关错误（诸如设备未就绪或相似的硬件问题）。
- ü 如果是首次打开，初始化设备。
- ü 标识设备号，如有必要更新 `f_op` 指针。
- ü 分配和填写要放在 `filp->private_data` 里的数据结构。

3. 根据 6.2.6 节中计算 X 坐标的方法，写出计算 Y 坐标的实现代码？

参考答案：Y 坐标的实现代码如下：

```
int TS_Coordinate_Conversion(int*y)
{
    int tempY;
    tempY -= Y2;
    *y = ( tempY *TFT_Y) / ( Y1- Y2);
    *y =TFT_Y-*y;
    return *y;
}
```



## 第 7 章 块设备驱动程序

本章学习目标：

- | 熟悉块设备驱动程序概念
- | 熟悉块设备驱动相关的三个结构：block\_device\_operations, gendisk, request
- | 理解块设备的请求处理
- | 理解 MMC/SD 的硬件实现原理
- | 理解 MMC 块设备驱动程序实现

### 7.1 块设备驱动介绍

上一章介绍了设备驱动程序的第一大类——字符驱动程序，接下来就要介绍第二大类的设备驱动程序——块设备驱动程序。所谓块设备，即面向块的设备，是指数据传输是以块为单位的（例如软盘和硬盘），这里硬件上的块一般被称作“扇区（Sector）”。而名词“块”常用来指软件上的概念，驱动程序常常使用 1024 字节大小为块的大小，而扇区大小为 512 字节。下面首先介绍一下块设备驱动相关的三个重要数据结构。

#### 7.1.1 块设备驱动相关的重要结构

与字符设备中使用的 file\_operations 结构类似，块设备使其专门的数据结构，即在 <include/linux/fs.h> 文件中声明的 block\_device\_operations。与块设备相关的还有一个重要的数据结构 gendisk，它被声明在 <include/linux/genhd.h> 文件中。此外，块设备驱动中还有一个非常重要的 request 结构，它被经常使用在 request 函数中，它在 <include/linux/blkdev.h> 文件中被声明。接下来分别介绍这三个重要的数据结构。

##### 7.1.1.1 block\_device\_operations（块设备操作）结构

块设备操作 block\_device\_operations 结构告诉系统它能提供哪些接口，与 file\_operations 结构的作用类似，只不过它是专门用于块设备驱动而已。首先让我们看一下内核中是如何声明 block\_device\_operations 结构的：

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    struct module *owner;
};
```

block\_device\_operations 结构成员与 file\_operations 结构相比，确实少了很多。接下来分析一下 block\_device\_operations（块设备操作）结构中各个成员的作用。

- n**     `int (*open) (struct inode *, struct file *);`  
 该成员是 `block_device_operations` 结构的一个方法，当打开设备时调用它，功能与字符设备中对应的函数相同。
- n**     `int (*release) (struct inode *, struct file *);`  
 该成员是 `block_device_operations` 结构的一个方法，当关闭设备时调用它，功能与字符设备中对应的函数相同。如果用户将介质放入设备中锁住，那么在调用 `release` 函数时一定要进行解锁。
- n**     `int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);`  
 该成员是 `block_device_operations` 结构的一个方法，实现 `ioctl` 系统调用的函数。块设备会首先截取大量的标准请求，因此大多数块设备的 `ioctl` 函数都比较短小。
- n**     `int (*media_changed) (struct gendisk *);`  
 该成员是 `block_device_operations` 结构的一个方法，内核调用该函数以检查用户是否更换了驱动器内的介质，如果更换了，则返回一个非零值。该函数适用那些支持可移动介质的驱动器，其他情况下，该函数被省略。
- n**     `int (*revalidate_disk) (struct gendisk *);`  
 该成员是 `block_device_operations` 结构的一个方法，当介质被更换时，调用该函数做出响应，它告诉驱动程序完成必要的工作，以便使用新的介质。
- n**     `struct module *owner;`  
 该成员不是 `block_device_operations` 结构的一个方法，它是一个指向拥有该结构的模块指针，通常被初始化为 `THIS_MODULE`，与 `file_operations` 结构中对成员的作用相同。

以上是对 `block_device_operations` 结构成员的一一介绍，细心的读者会发现该结构没有包含 `read/write` 方法。其实在块设备的 I/O 子系统中，这些操作是由 `request` 函数处理的，这也就是字符设备驱动程序与块设备驱动程序实现的不同之处，本章后面会专门介绍 `request` 函数。

### 7.1.1.2 gendisk 结构

内核使用 `gendisk` 结构来表示一个独立的磁盘设备，此外内核还使用 `gendisk` 结构表示分区。`gendisk` 结构的声明如下：

```
struct gendisk {
    int major;           /* major number of driver */
    int first_minor;
    int minors;          /* maximum number of minors, =1 for
                        * disks that can't be partitioned. */
    char disk_name[32];  /* name of major driver */
    struct hd_struct **part; /* [indexed by minor] */
    struct block_device_operations *fops;
    struct request_queue *queue;
    void *private_data;
    sector_t capacity;

    int flags;
    char devfs_name[64]; /* devfs crap */
}
```

```

        int number;                /* more of the same */
        struct device *driverfs_dev;
        struct kobject kobj;

        struct timer_rand_state *random;
        int policy;

        atomic_t sync_io;          /* RAID */
        unsigned long stamp, stamp_idle;
        int in_flight;
#ifdef CONFIG_SMP
        struct disk_stats *dkstats;
#else
        struct disk_stats dkstats;
#endif
    };

```

接下来分析一下 gendisk 结构的各个成员。

- n**     int major;  
该成员表示设备驱动的主设备号。
- n**     int first\_minor;  
该成员表示设备驱动第一个次设备号。
- n**     int minors;  
该成员表示设备驱动有多少个次设备号，可以包含 minors-1 个分区。
- n**     char disk\_name[32];  
该成员表示磁盘设备名字，该名字将显示在 /proc/partitions 和 sysfs 中。
- n**     struct hd\_struct \*\*part;  
该成员表示通过次设备号索引的硬件设备扇区信息。
- n**     struct block\_device\_operations \*fops;  
该成员表示块设备的各种操作。
- n**     struct request\_queue \*queue;  
该成员表示设备管理 I/O 请求。
- n**     void \*private\_data;  
该成员用来保存其内部数据的指针。
- n**     sector\_t capacity;  
该成员表示该驱动器可包含的扇区数，以 512 字节为一个扇区。
- n**     int flags;  
该成员表示驱动器状态的标志。
- n**     char devfs\_name[64];  
该成员表示该驱动器的设备文件系统名字。
- n**     int number;  
该成员表示多少个相同的块设备。
- n**     struct device \*driverfs\_dev;  
该成员表示该驱动器的设备文件系统。
- n**     struct timer\_rand\_state \*random;

该成员表示定时器随机状态。

**n**     int policy;

该成员用来表示该设备驱动的可读/可写属性。

**n**     atomic\_t sync\_io;

该成员用来表示同步 I/O 的原子计数器。

**n**     unsigned long stamp, stamp\_idle;

该成员表示磁盘设备的时间戳。

**n**     int in\_flight;

该成员表示设备驱动的时间信息包数。

**n**     struct disk\_stats \*dkstats;

该成员表示磁盘的状态信息。

虽然 `gendisk` 结构的成员比较繁多，但实际应用中有许多成员是没有用到的。内核为 `gendisk` 结构提供了一些函数，在这里一同介绍一下。

**ü**     struct gendisk \*alloc\_disk(int minors);

该函数用来动态分配 `gendisk` 结构，并进行初始化。参数 `minors` 是磁盘使用次设备号的个数。该函数的实现代码在 `<drivers/block/genhd.c>` 文件。

**ü**     void del\_gendisk(struct gendisk \*disk);

该函数用来卸载磁盘，当不再使用一个磁盘时，调用该函数来卸载它。该函数的实现代码在 `<fs/partitions/check.c>` 文件中。

**ü**     void add\_disk(struct gendisk \*disk);

该函数用来激活磁盘设备，并提供随时调用它提供的方法。由于 `alloc_disk` 函数只是分配了 `gendisk` 结构并不能使磁盘对系统可用，所以需要调用 `add_disk` 函数。该函数的实现代码在 `<drivers/block/genhd.c>` 文件。

### 7.1.1.3 request 结构

`request` 结构用来代表一个块设备的 I/O 执行请求，是块设备驱动中非常重要的一个结构。`request` 结构在内核中的声明如下：

```
struct request {
    struct list_head queuelist;
    unsigned long flags;          /* see REQ_ bits below */
    sector_t sector;              /* next sector to submit */
    unsigned long nr_sectors;     /* no. of sectors left to submit */
    unsigned int current_nr_sectors;
    sector_t hard_sector;         /* next sector to complete */
    unsigned long hard_nr_sectors; /* no. of sectors left to complete */
    unsigned int hard_cur_sectors;
    struct bio *bio;
    struct bio *biotail;
    void *elevator_private;
    int rq_status; /* should split this into a few status bits */
    struct gendisk *rq_disk;
    int errors;
    unsigned long start_time;
```

```

        unsigned short nr_phys_segments;
        unsigned short nr_hw_segments;
        int tag;
        char *buffer;
        int ref_count;
        request_queue_t *q;
        struct request_list *rl;
        struct completion *waiting;
        void *special;
        unsigned int cmd_len;
        unsigned char cmd[BLK_MAX_CDB];
        unsigned int data_len;
        void *data;
        unsigned int sense_len;
        void *sense;
        unsigned int timeout;
        struct request_pm_state *pm;
    };

```

由于 request 结构包含许多成员，而在实际应用中只用到很少一部分，所以这里只介绍一些 request 结构的常用成员：

- n**     struct list\_head queuelist;  
该成员用于把请求链接到请求队列中，一般不能直接访问它，而是通过 blkdev\_dequeue\_request 函数来访问。
- n**     sector\_t sector;  
该成员用来表示下一个要提交的扇区。
- n**     unsigned long nr\_sectors;  
该成员表示剩下多少个扇区没有被提交。
- n**     sector\_t hard\_sector;  
该成员表示还未传输的第一个扇区。
- n**     unsigned long hard\_nr\_sectors;  
该成员表示等待传输扇区的总数。
- n**     struct bio \*bio;  
该成员表示该请求的 bio 结构链表，不能直接访问该成员，而要使用 rq\_for\_each\_bio 函数访问。其中 bio 结构是在底层对部分块设备 I/O 请求的描述。
- n**     unsigned short nr\_phys\_segments;  
该成员表示相邻的页被合并后，在物理内存中被这个请求所占用的段的数目。
- n**     char \*buffer;  
该成员表示查找需要传输的缓冲区。
- n**     request\_queue\_t \*q;  
该成员表示一个请求队列。

每个 request 结构都代表了一个块设备的 I/O 请求，在较高层次，它可能是通过对多个独立请求合并而来。一个 request 结构是作为一个 bio 结构的链表实现的。当然是通过一些管理信息来组合的，这样保证在执行请求时驱动程序能知道执行的位置。关于 request 结构和 bio 结构的结合如图 7.1 所示。从图中可知，cbio 和 buffer 成员指向第一个未被传输的 bio

结构。下一节将介绍请求处理。

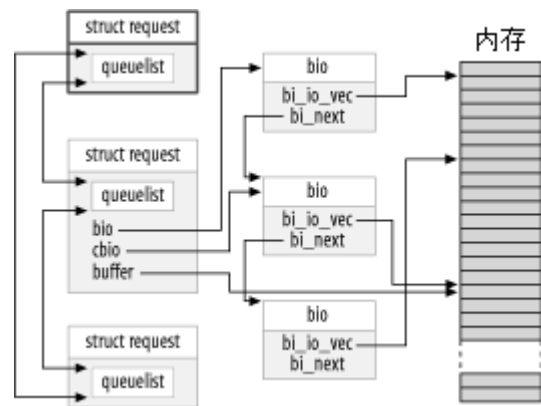


图 7.1 正在处理请求的请求队列

### 7.1.2 请求处理

上一节中说到 `block_device_operations` 结构中没有负责读写的方法，那是因为块设备驱动程序中是用 `request` 函数来实现的。`request` 请求函数是块设备驱动程序的核心，实际的工作中，设备的启动都是在 `request` 函数中实现的。磁盘驱动程序的性能是整个操作系统中重要的组成部分。因此内核的块设备子系统在编写的时候就非常注意性能方面的问题，除了从所控制的设备上获得信息之外，块设备子系统为驱动程序完成了所有可能的工作。下面将介绍 `request` 函数。

#### 7.1.2.1 request 函数

块设备驱动程序中 `request` 函数原型如下：

❶ `void request( request_queue_t *q );`

该函数用来实现驱动程序处理读写以及其他对设备的操作，在其返回前，`request` 函数不必完成所有队列中的请求。

对大多数设备而言，`request` 函数可能没有完成任何请求，但是它必须启动对请求的响应，并且保证所有请求最终被驱动程序所处理。每个设备都有一个请求队列，这是由于磁盘数据实际传出和传入发生的时间与内核请求的时间相差较大，因此内核需要有一定的灵活性以安排在适当时刻进行传输。当请求队列生成的时候，`request` 函数就与该队列绑定在一起了。对 `request` 函数的调用通常是与用户空间进程的动作是异步的，我们不能假设内核正运行在初始化当前请求进程的上下文中。并且我们也无法知道请求所提供的 I/O 缓存是在内核空间中还是在用户空间中。因此直接对用户空间的任何类型的访问都是会导致错误，所以块设备驱动程序所需要知道的任何关于请求的信息都是通过请求队列来实现的。接下来介绍一个 `request` 函数的实例。

#### 7.1.2.2 request 函数实例

选择内核中应用的 Amiga 伪设备访问 16 位 RAM 在 ZorroII 空间作为一个块设备的例子，该 `request` 函数的实现代码如下：

```
static void do_z2_request(request_queue_t *q)
```

```

{
    struct request *req;
    while ((req = elv_next_request(q)) != NULL) {
        unsigned long start = req->sector << 9;
        unsigned long len = req->current_nr_sectors << 9;

        if (start + len > z2ram_size) {
            printk( KERN_ERR DEVICE_NAME ": bad access: block=%lu,
count=%u\n",
                    req->sector, req->current_nr_sectors);
            end_request(req, 0);
            continue;
        }
        while (len) {
            unsigned long addr = start & Z2RAM_CHUNKMASK;
            unsigned long size = Z2RAM_CHUNKSIZE - addr;
            if (len < size)
                size = len;
            addr += z2ram_map[ start >> Z2RAM_CHUNKSHIFT ];
            if (rq_data_dir(req) == READ)
                memcpy(req->buffer, (char *)addr, size);
            else
                memcpy((char *)addr, req->buffer, size);
            start += size;
            len -= size;
        }
        end_request(req, 1);
    }
}

```

该函数中的 `request` 结构在本章前面的章节中已经介绍，它表示一个块设备的 I/O 执行请求。内核提供了函数 `elv_next_request` 来获得队列中第一个未完成的请求，当没有请求处理时，该函数返回 `NULL`。如果两次调用 `elv_next_request` 函数，则两次都返回相同的 `request` 结构。其中还有一个非常重要的函数 `end_request`，该函数的原型如下：

ü `void end_request(struct request *req, int uptodate);`

当传递参数 `uptodate` 为 0 时，表示不能成功的完成请求，而当为非 0 时则表明成功的完成了请求处理。

## 7.2 块设备驱动开发实例

本章将以一个典型的块设备实例为学习对象，讲述块设备驱动的实现过程。本章的研究实例是 MMC/SD 驱动开发，它是嵌入式设备中经常用到的存储设备。首先介绍一下什么是 MMC/SD。

7.2.1 MMC/SD 介绍

MMC 的英文全称是 MultiMedia Card，是一种体积小容量大的快闪存储卡,由西门子公司（现在称为 Infineon）和首推 CF 卡的 SanDisk 于 1997 年推出。由于它的封装技术较为先进，而且目前已经相当成熟，它的外形尺寸大约为 32mm×24mm×1.4mm，重量在 2 克以下，7 针引脚，它的体积甚至比 SmartMedia 还要小，不怕冲击，可反复读写记录 30 万次，驱动电压在 2.7-3.6V，目前容量多为 64M 和 128M 容量。现在这种闪存卡已广泛用于移动电话，数码相机，数码摄像机，MP3 等多种数码产品上。MMC 在设计之初是瞄准手机和寻呼机市场，之后因其小尺寸等独特优势而迅速被引进更多的应用领域，如数码相机、PDA、MP3 播放器、笔记本电脑、便携式游戏机、数码摄像机乃至手持式 GPS 等。Siemens 公司最初之所以将 MMC 主要定位在手机上，是因手机产品的发展需求而致。当前的大多数手机存储容量相当有限，而功能和各种通信增值服务的增加又对手机的存储容量提出越来越高的要求(更多的电话号码、短消息、手机铃声、图片、录制的声音、记事本甚至多媒体短消息等)，但是为越做越小的手机扩充存储容量的难度比其他掌上电子产品更高。而邮票般大小的 MMC 会比 CF 等传统产品更适合手机。JVC 公司早在 1999 年已推出了采用 MMC 的数码摄像机(保存抓拍的静态画面)。为了推动 MMC 的应用，在 1998 年成立了 MMC 协会(MMCA)，成为推广 MMC 标准化的全球性组织，目前其成员除了发起厂商以外还包括 Motorola、Nokia、Ericsson、Intel、TI、HP 和 Toshiba 等数百家业界知名厂商。MMC 卡的外观图如 7.2 左图所示，它有 7 个管脚，所有的 MMC 卡被直接 MMC 总线信号相连，表 7-1 表示这种情况下，MMC 各管脚连接信号。其中 MMC 模式是指利用 MMC 总线方式传输数据，SPI（Serial Peripheral Interface）模式是指利用 SPI 信道传输数据。

表 7-1MMC 总线信号连接说明

管脚号	MMC模式			SPI模式		
	名字	类型 *(注1)	描述	名字	类型	描述
1	RSV	NC	保留	CS	I	芯片选择
2	CMD	I/O/PP/OD	命令/响应	DI	I/PP	数据输入
3	VSS1	S	供电接地	Vss	S	供电接地
4	VDD	S	供电	VDD	S	供电
5	CLK	I	时钟	SCLK	I	时钟
6	VSS2	S	供电接地	Vss2	S	供电接地
7	DAT	I/O/PP	数据	DO	O/PP	数据输出

\*注 1：类型中：S 代表 Power supply；I 代表 input；O 代表 output；PP 代表 push-pull；OD 代表 open-drain；NC 代表 Not connected。

只用 CMD 信道来初始化 MMC 栈，所以兼容所有的卡，每一个卡都有一组设置信息的寄存器，如表 7-2 所示。

名字	宽度 （位）	描述
CID	128	Card identification number，卡的唯一识别号。必须的
RCA	16	Relative card address，卡的本地系统地址，主机在初始化时动态的分配。必须的
DSR	16	Driver stage register，配置卡的输出驱动。可选的
CSD	128	Card specific data，关于卡操作条件的信息。必须的



OCR	32	Operation condition register, 用于那些不支持全电压范围的卡, 用一个特殊的广播命令探测受限制的卡。可选的
-----	----	---

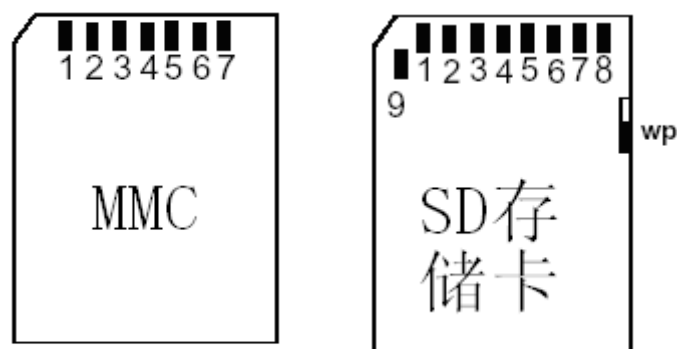


图 7.2 MMC 和 SD 卡

SD卡（Secure Digital Memory Card）是一种基于半导体快闪记忆器的新一代记忆设备。SD卡由日本松下、东芝及美国SanDisk公司于1999年8月共同开发研制。大小犹如一张邮票的SD记忆卡，重量只有2克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。SD卡在24mm×32mm×2.1mm的体积内结合了SanDisk快闪记忆卡控制与MLC（Multilevel Cell）技术和Toshiba（东芝）0.16u及0.13u的NAND技术，通过9针的接口界面与专门的驱动器相连接，不需要额外的电源来保持其上记忆的信息。而且它是一体化固体介质，没有任何移动部分，所以不用担心机械运动的损坏。SD卡数据传送和物理规范由MMC发展而来，大小和MMC差不多，尺寸为32mm x 24mm x 2.1mm。长宽和MMC一样，只是厚了0.7mm，以容纳更大容量的存储单元。SD卡与MMC卡保持着向上兼容，也就是说，MMC可以被新的SD设备存取，兼容性则取决于应用软件，但SD卡却不可以被MMC设备存取。SD卡外型采用了与MMC厚度一样的导轨式设计，以使SD设备可以适合MMC。SD接口除了保留MMC的7针外，还在两边多加了2针，作为数据线。采用了NAND型Flash Memory，基本上和SmartMedia的一样，平均数据传输率能达到2MB/s。SD卡的结构能保证数字文件传送的安全性，也很容易重新格式化，所以有着广泛的应用领域，音乐、电影、新闻等多媒体文件都可以方便地保存到SD卡中。因此不少数码相机也开始支持SD卡。目前市场上SD卡的品牌很多，诸如：SANDISK，Kingmax，松下和Kingston。关于SD卡的示意图如7.2右图所示，关于SD卡各管脚的连接情况如表7-3所示。其中SD模式是指利用SD总线方式传输数据，SPI模式是指利用SPI信道传输数据。

表 7-3SD 卡管脚分配

管脚号	SD模式			SPI模式		
	名字	类型*（注2）	描述	名字	类型	描述
1	CD/DAT3	I/O/PP	卡探测/数据线[位3]	CS	I	芯片选择
2	CMD	PP	命令/响应	DI	I	数据输入
3	VSS1	S	供电接地	VSS	S	供电
4	VDD	S	供电	VDD	S	供电
5	CLK	I	时钟	SCLK	I	时钟
6	VSS2	S	供电接地	VSS2	S	供电接地
7	DAT0	I/O/PP	数据线[位0]	DO	O/PP	数据输出

8	DAT1	I/O/PP	数据线[位1]	RSV	-	-
9	DAT2	I/O/PP	数据线[位2]	RSV	-	-

\*注 2：类型中：S 代表 Power supply；I 代表 input；O 代表 output；PP 代表 push-pull。

与 MMC 类似，SD 卡也有一组设置信息的寄存器，关于 SD 卡的这些寄存器描述在表 7-4 中。

表 7-4SD 卡寄存器

名字	宽度（位）	描述
CID	128	Card identification number, 卡的唯一识别号。必须的
RCA*（注3）	16	Relative card address, 卡的本地系统地址，主机在初始化时动态的分配。必须的
DSR	16	Driver stage register, 配置卡的输出驱动。可选的
CSD	128	Card specific data, 关于卡操作条件的信息。必须的
SCR	64	SD Configuration register, 关于SD卡特殊功能配置的寄存器。必须的
OCR	32	Operation condition register, 用于那些不支持全电压范围的卡，用一个特殊的广播命令探测受限制的卡。可选的

\*注 3：RCA 寄存器是无效的在 SPI 模式。

## 7.2.2 S3C2410 提供的 SDI 接口

S3C2410 芯片内部提供的 SDI（Secure Digital Interface，安全数字接口）支持 SD 存储卡，SDIO 设备和 MMC 接口。S3C2410 芯片的 SDI 有以下特点：

- l 遵守 SD 存储卡 V1.0 规范/兼容 MMC V2.11 规范
- l 兼容 SDIO 卡 V1.0 规范
- l 16 字（64 字节）的 FIFO（先进先出）用于数据的收发
- l 40 位命令寄存器(SDICARG[31:0]+SDICCON[7:0])
- l 136 位的响应寄存器(SDIRSPn[127:0]+ SDICSTA[7:0])
- l 8 位的预定标器逻辑
- l CRC7&CRC16 发生器
- l 正常的模式和 DMA 数据模式（字节或字传输）
- l 1 位/4 位（总线宽度）模式&块/流模式支持选择
- l 支持高达 25MHz 数据传输模式用于 SDI
- l 支持高达 10MHz 数据传输模式用于 MMC

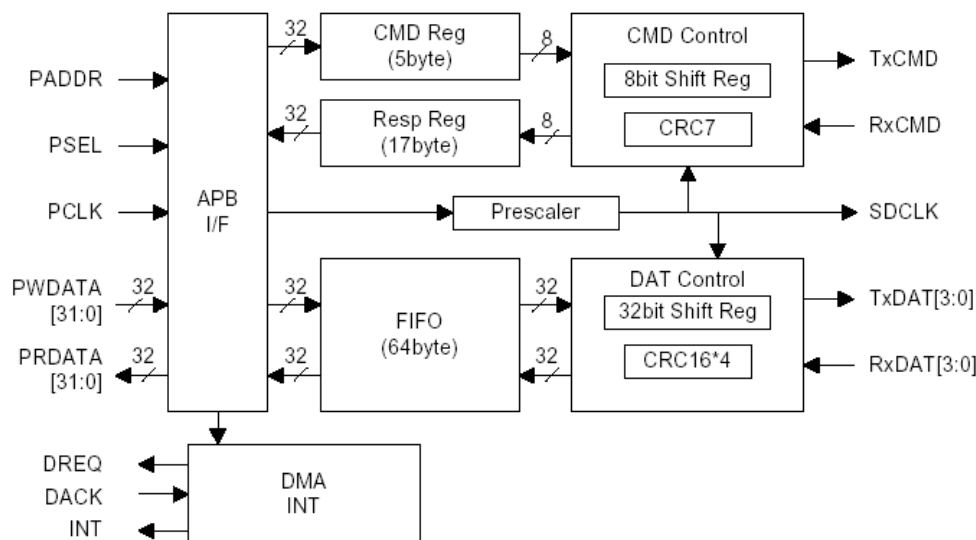


图 7.3 SDI 内部结构图

关于 SDI 内部结构图如 7.3 所示，一个串行时钟线与五条数据线同步用于频移和抽样信息。根据传输频率设置 SDIPRE 寄存器相应的位，可以通过调节波特率数据寄存器的值来修改频率。通常对于 SDI 模块编程基本步骤是：一是设置 SDICON 寄存器用来配置合适的时钟和中断；二是设置 SDIPRE 寄存器配置合适的频率；三是等待 74SDCLK 时钟用来初始化相应的卡。就命令（CMD）通道编程来说：一是写命令参数（32 位）到 SDICARG 寄存器；二是通过设置 SDICCON[8]寄存器确定命令类型和开始命令；三是当 SDICSTA 寄存器某具体的位被设置时确认结束 SDI 命令操作；四是通过写 SDICSTA 寄存器相应的位来清除标识。就数据（DAT）通道编程来说：一是写 SDIDTIMER 寄存器来设置间歇周期时间；二是写 SDIBSIZE 寄存器来设置块的大小；三是通过设置 SDIDCON 寄存器来确定块传输的模式，如宽总线，DMA 等。四是通过检查 SDIFSTA 寄存器确定发送数据的 FIFO 是否可用，当可用时写发送数据；五是通过检查 SDIFSTA 寄存器确定接收数据的 FIFO 是否可用，当可用时读接收数据；六是当数据传输完成位（SDIDSTA[4]）被设置时确认结束 SDI 数据操作；七是清除 SDIDSTA 寄存器相应的位。注意，如果是 MMC，最大的数据传输时钟是 10MHz，在 MMC 写模式，尽管写正确但 CRC（Cyclic Redundancy Check，循环冗余码校验）\*（注 4）错误还是会发生，为了能可靠的传输数据，在写完数据后就读数据并且比较它。如果遇到长时间的响应，在接收来自 SD 设备的响应数据之后，CRC 错误应该被探测。编程人员应该通过软件来检测接收响应的 CRC。

\*注 4：CRC 是一种算法，常用于数据校验，CRC 校验的基本思想是利用线性编码理论，在发送端根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的监督码（CRC 码）r 位，并附在信息后边，构成一个新的二进制码序列数共(k+r)位，最后发送出去。在接收端，则根据信息码和 CRC 码之间所遵循的规则进行检验，以确定传送中是否出错。

### 7.2.3 SDI 相关的寄存器

接下来讲述编程时需要用到的 SDI 寄存器，关于 SDI 寄存器的相关信息需要查阅

S3C2410 芯片的用户手册获得。

7.2.3.1 SDI 控制（SDICON）寄存器

寄存器	地址	R/W	描述	复位值
SDICON	0x5A000000	R/W	SDI控制寄存器	0x0

SDICON	位	描述	初始状态
ByteOrder	[4]	确定字节顺序类型当读（写）数据从（到）SD主机FIFO的字边界时 0=类型A      1=类型B	0
RcvIOInt	[3]	确定是否SD主机接收到来自卡的SDIO中断 0=忽略      1=接收SDIO中断	0
RWaitEn	[2]	当SD主机等待下一个块在多个块读模式时确定读等待请求信号产生。这个位需要延迟下一个被传输的块。 0=禁止（不产生）      1=读等待启动（使用SDIO）	0
FRST	[1]	复位FIFO值，这个位自动被清除。 0=正常模式      1=FIFO复位	
ENCLK	[0]	确定是否SDCLK输出被启动 0=禁止      1=时钟被启动	0

注意： 字节顺序类型  
类型A: D [7:0] -> D [15:8] -> D [23:16] -> D [31:24]  
类型B: D [31:24] -> D [23:16] -> D [15:8] -> D [7:0]

7.2.3.2 SDI 波特率预定标（SDIPRE）寄存器

寄存器	地址	R/W	描述	复位值
SDIPRE	0x5A000004	R/W	SDI波特率预定标寄存器	0x0

SDIPRE	位	描述	初始状态
Prescaler Value	[7:0]	确定SDI时钟频率 波特率=PCLK/2/(预定标值+1)	0

7.2.3.3 SDI 命令参数(SDICARG)寄存器

寄存器	地址	R/W	描述	复位值
SDICARG	0x5A000008	R/W	SDI命令参数寄存器	0x0

SDIPRE	位	描述	初始状态
CmdArg	[31:0]	命令参数	0x00000000

#### 7.2.3.4 SDI 命令控制(SDICCON)寄存器

寄存器	地址	R/W	描述	复位值
SDICCON	0x5A00000C	R/W	SDI命令控制寄存器	0x0

SDICCON	位	描述	初始状态
AbortCmd	[12]	确定命令类型是否是异常中止（用于SDIO） 0=正常命令      1=异常中止（CMD12, CMD52）	0
WithData	[11]	确定命令类型是否包含数据（用于SDIO） 0=不包含数据      1=包含数据	0
LongRsp	[10]	确定主机是否接收一个136位的长响应。 0=短响应      1=长响应	0
WaitRsp	[9]	确定主机是否等待一个响应。 0=没有响应      1=等等响应	
CMST	[8]	确定是否命令操作开始 0=命令就绪      1=命令开始	0
CmdIndex	[7:0]	命令索引从2位开始（8位）	0x00

#### 7.2.3.5 SDI 命令状态(SDICSTA)寄存器

寄存器	地址	R/W	描述	复位值
SDICSTA	0x5A000010	R/W	SDI命令状态寄存器	0x0

SDICSTA	位	描述	初始状态
RspCrc	[12]	当命令响应接收时，CRC检测失败 0=没有检测到      1=CRC失败	0
CmdSent	[11]	发送命令（不关心响应） 0=没有检测到      1=命令结束	0
CmdTout	[10]	命令响应时间到。 0=没有检测到      1=时间到	0
RspFin	[9]	命令响应接收。 0=没有检测到      1=响应结束	
CmdOn	[8]	命令传输过程中 0=没有检测到      1=命令传输中	0
RspIndex	[7:0]	响应索引从2位开始（8位）	0x00

#### 7.2.3.6 SDI 响应(SDIRSP)寄存器

寄存器	地址	R/W	描述	复位值
SDIRSP0	0x5A000014	R	SDI响应寄存器0	0x0

SDIRSP0	位	描述	初始状态
Response0	[31:0]	卡状态[31:0]（短响应），卡状态[127:96]长响应	0x00000000

寄存器	地址	R/W	描述	复位值
SDIRSP1	0x5A000018	R	SDI响应寄存器1	0x0

SDIRSP1	位	描述	初始状态
RCRC7	[31:24]	CRC7（包含结束位，短响应），卡状态[95:88]（长响应）	0x00
Response0	[23:0]	没有使用（短响应），卡状态[87:64]长响应	0x00000000

寄存器	地址	R/W	描述	复位值
SDIRSP2	0x5A00001C	R	SDI响应寄存器2	0xy0

SDIRSP2	位	描述	初始状态
Response2	[31:0]	没有使用（短响应），卡状态[63:32]长响应	0x00000000

寄存器	地址	R/W	描述	复位值
SDIRSP3	0x5A000020	R	SDI响应寄存器3	0x0y

SDIRSP3	位	描述	初始状态
Response3	[31:0]	没有使用（短响应），卡状态[31:0]长响应	0x00000000

### 7.2.3.7 SDI 数据/占用定时器(SDIDTIMER)寄存器

寄存器	地址	R/W	描述	复位值
SDIDTIMER	0x5A000024	R/W	SDI数据/占用定时器寄存器	0x2000

SDIDTIMER	位	描述	初始状态
DataTimer	[15:0]	数据/占用时间中止周期（0~65535周期）	0x2000

### 7.2.3.8 SDI 块大小(SDIBSIZE)寄存器

寄存器	地址	R/W	描述	复位值
SDIBSIZE	0x5A000028	R/W	SDI块大小寄存器	0x0

SDIBSIZE	位	描述	初始状态
BlkSize	[11:0]	块大小值（0~4095字节）	0x000

注意：如果是多个块，BlkSize 必须组成 4 字节大小。

与 SDI 相关的寄存器还有许多，如果想了解更多寄存器的信息请参考 S3C2410 芯片用户手册，这里就不再一一介绍。

## 7.2.4 MMC/SD 驱动概要设计

由于 SD 卡是继承 MMC 而诞生的，并且 SD 卡设备完全兼容 MMC，所以在本实例中主要讲述 MMC 驱动的实现。首先让我们来看一下 MMC/SD 卡的硬件连接图。

### 7.2.4.1 MMC/SD 与主机的接口连接

图 7.4 是 SD/MMC 与 S3C2410 的接口连接图，从这个图我们很清楚的看到 SD/MMC 的管脚是如何与主机设备相连接的。

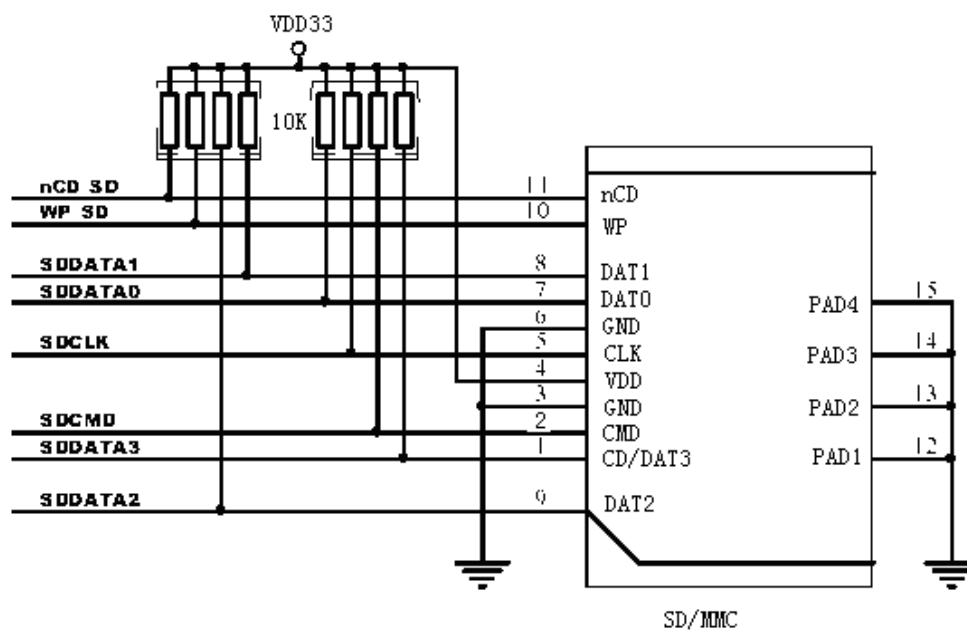


图 7.4 MMC/SD 接口连接

### 7.2.4.2 MMC/SD 驱动框架

关于 MMC/SD 驱动的设计，首先需要遵守 MMC/SD 规范，其次要有 MMC/SD 核心的支持，而 MMC/SD 驱动程序是基于其核心和通信协议之上的，如图 7.5 所示。



图 7.5 MMC/SD 驱动程序的位置

为了不使读者混淆 MMC 和 SD 驱动的实现，这里只介绍 MMC 驱动的实现过程，SD 驱动的实现与 MMC 非常类似。根据 MMC 规范 V2.11，MMC 的内部实现有两种通信协议可以选择，MMC 总线协议和 SPI 总线协议。MMC 总线被设计用于连接电子晶体的大容量内存或在一个卡中的 I/O 设备，经常用于多媒体应用。该总线可用于低成本系统并且拥有快速数据传输特点。MMC 总线系统如图 7.6 所示，它是单个主控总线连接多个从设备。MMC 总线被分成三组：电源，数据传输（命令和数据）和时钟。

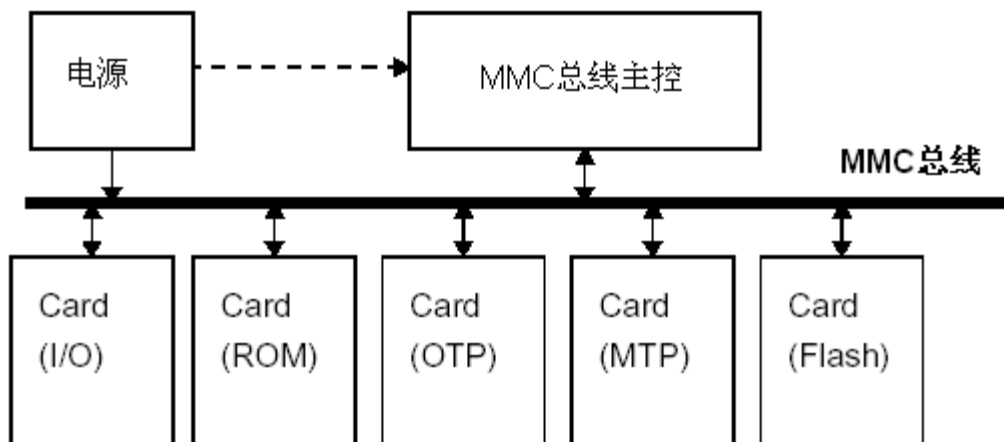


图 7.6 MMC 总线系统

对任何一种存储卡应用而言，它必须具备下列的基本功能：

- Ø 热插拔功能。
- Ø 当存储卡插入或移除时，中断服务程序（ISR）必须能够立即发现和反应。当存储卡插入时，ISR 必须能将新加入的存储卡地址或名称在应用程式中显示出来。移除时，此存储卡地址或名称也能从应用程式中消失。当存储卡插入时，存储卡必须像磁盘一样，被挂载（mount）在磁盘的根目录下。同理，当存储卡被移除时，它就会被卸载（unmount）。
- Ø 如果存储卡内部具有文件系统，它虽然可以被虚拟成磁盘，也就是块设备（block device），但是当系统发出移除存储卡的请求时，发现存储卡早已不存在了，此时，块程式会不知所



措，认为是错误。所以，在系统发出移除存储卡的请求之前，块设备层必须能先侦测出存储卡是否有存在。而且如果它不存在，就立即终止这项请求。因此，在设计存储卡的最底层驱动程序时，必须提供一个函式而且它必须指向上层的块设备层。

### 7.2.4.3 MMC 驱动的核心设计

我们即将要介绍的 MMC 驱动的利用 MMC 总线实现的，关于 MMC 规范希望读者在开发 MMC 驱动之前仔细阅读它，这里就不详细介绍了。设计 MMC 驱动程序最主要的工作就是实现 MMC 数据传输的控制，当开机之后，MMC 的初始化是通过一种特殊的数据流通信协定来完成。此通信协定的信息格式具有下列几种类型：

**命令：**表示开始执行一个作业。它是从主机发出，至单一的 MMC 存储卡（单一位址的命令），或至全部有连接的 MMC 存储卡（广播命令）。此命令信息是在 CMD 线上以串行方式传输。

**回应：**它的传送方向和命令信息相反，它是从一个特定地址的 MMC 存储卡，或全部的 MMC 存储卡同步地传送给主机，是对之前接收到的命令信息的回应。

**数据：**此信息是从主机或 MMC 存储卡发出。

MMC 存储卡的地址是由数据流控制器在初始化时设定的。它们的唯一的 CID (Customer Identity) 号码代表它们自己。MMC 有两种数据传输命令，如下所示：

**序列式命令：**这个命令会启动连续性的数据流。唯有收到停止命令时，传输作业才会结束。这个模式可以减少传送额外的命令。

**块式命令：**此命令在传送一个数据块之后，会传送 CRC 位元。它的读写作业支持单一块或多个块的传输。和序列式命令类似，在收到停止命令时，多块传输作业才会结束。

MMC 的数据读写作业可以包含：单一区块、多区块、串流的传输。这些作业可以通过 DMA 来加速传输，这是由设定模式寄存器（mode register）的位元值来做切换。块的长度也必须在模式寄存器中设定。下面开始分析 MMC 驱动程序的实现。

## 7.2.5 MMC 驱动程序分析

现在要分析 MMC 驱动程序的实现过程，读者将会看到它的实现过程与字符设备驱动的实现有很大的不同，从而让读者感受到块设备驱动实现方法与字符设备驱动的不同之处。首先来分析它的初始化过程。

### 7.2.5.1 MMC 初始化

对于 MMC 的初始化主要完成三个任务：一是注册 MMC 块设备；二是建立 MMC 设备文件系统的目录；三是注册具体的 MMC 介质驱动，实现代码如下所示：

```
static int __init mmc_blk_init(void)
{
    int res = -ENOMEM;

    res = register_blkdev(major, "mmc");
    if (res < 0) {
        printk(KERN_WARNING "Unable to get major %d for MMC media: %d\n",
```

```

        major, res);

    goto out;
}
if (major == 0)
    major = res;

devfs_mk_dir("mmc");
return mmc_register_driver(&mmc_driver);

out:
return res;
}

```

用 `register_blkdev` 函数来实现注册 MMC 块设备，如果没有指定主设备号，则将动态的分配一个，其中注册的块设备名为 `mmc`。其次用 `devfs_mk_dir` 函数来建立一个 MMC 的设备文件目录。最后注册 MMC 介质的驱动，其中参数 `mmc_driver` 是 `mmc_driver` 结构的对象，该结构对象的定义如下：

```

static struct mmc_driver mmc_driver = {
    .drv      = {
        .name   = "mmcblk",
    },
    .probe     = mmc_blk_probe,
    .remove    = mmc_blk_remove,
    .suspend   = mmc_blk_suspend,
    .resume    = mmc_blk_resume,
};

```

该 `mmc_driver` 对象定义了 MMC 驱动的 `probe`、`remove`、`suspend` 和 `resume` 方法，并且程序中相应的实现了这些方法。

顺便让我们看一下卸载 MMC 设备的实现代码，它的作用正好与初始化相反，首先完成 MMC 介质驱动的注销，接着删除 MMC 设备文件系统的目录，最后注销 MMC 块设备，具体实现代码如下：

```

static void __exit mmc_blk_exit(void)
{
    mmc_unregister_driver(&mmc_driver);
    devfs_remove("mmc");
    unregister_blkdev(major, "mmc");
}

```

此外，可以看一下 MMC 块设备操作的定义，如下代码所示，定义了 `open`、`release` 和 `ioctl` 方法，下面将分别介绍这几个方法的具体实现。

```

static struct block_device_operations mmc_bdops = {
    .open      = mmc_blk_open,
    .release   = mmc_blk_release,
    .ioctl     = mmc_blk_ioctl,
    .owner     = THIS_MODULE,
};

```

### 7.2.5.2 open 和 release 方法

块设备的 open 方法作用与字符设备的 open 方法类似，它把相关的 inode 和 file 结构作为参数，当一个 inode 指向一个块设备时，inode->i\_bdev->bd\_disk 成员包含了指向相应 gendisk 结构的指针，该指针可用于获得驱动程序内部的数据结构。MMC 驱动的 open 方法实现代码如下：

```
static int mmc_blk_open(struct inode *inode, struct file *filp)
{
    struct mmc_blk_data *md;
    int ret = -ENXIO;

    md = mmc_blk_get(inode->i_bdev->bd_disk);
    if (md) {
        if (md->usage == 2)
            check_disk_change(inode->i_bdev);
        ret = 0;
    }

    return ret;
}
```

其中 mmc\_blk\_data 结构表示 MMC 设备的一个内部数据结构，它的定义以及对各成员的注释如下：

```
struct mmc_blk_data {
    spinlock_t    lock; /*用于互斥锁*/
    struct gendisk *disk; /*gendisk 结构*/
    struct mmc_queue queue; /*MMC 设备请求队列*/
    unsigned int  usage; /*用户数目*/
    unsigned int  block_bits; /*块的大小，以位为单位*/
};
```

用 mmc\_blk\_get 函数来获得与参数 gendisk 结构对应的 mmc\_blk\_data 结构体信息，如果正确获得则 open 方法返回 0，否则返回-ENXIO 错误信息。当获得的 mmc\_blk\_data 结构对象的用户数是 2 时，调用 check\_disk\_change 函数来检查是否更换了磁盘介质，如果更换了介质将使那些所有的磁盘相关的缓冲无效。

接下来讲述 release 方法，同样它的作用与字符设备中的 release 方法相似，用它来释放 open 方法打开的设备，它的实现代码如下：

```
static int mmc_blk_release(struct inode *inode, struct file *filp)
{
    struct mmc_blk_data *md = inode->i_bdev->bd_disk->private_data;
    mmc_blk_put(md);
    return 0;
}
```

该方法实现的关键是 mmc\_blk\_put 函数，其参数传递的是指向 mmc\_blk\_data 结构的指

针，该函数完成的任务有，首先将 `mmc_blk_data` 结构的对象用户数减 1，然后判断它的用户数目是否为 0，如果为 0，首先释放它的 `gendisk` 结构对象，然后清除 MMC 的请求队列，最后释放 `mmc_blk_data` 结构对象的内存。

### 7.2.5.3 ioctl 方法

块设备驱动程序提供了 `ioctl` 函数执行设备的控制功能，高层的块设备子系统在驱动程序获得 `ioctl` 命令之前已经截取了大量的命令，所以实际中 `ioctl` 函数的实现比较简单，MMC 驱动的 `ioctl` 方法实现代码如下：

```
static int mmc_blk_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct block_device *bdev = inode->i_bdev;
    if (cmd == HDIO_GETGEO) {
        struct hd_geometry geo;
        memset(&geo, 0, sizeof(struct hd_geometry));
        geo.cylinders = get_capacity(bdev->bd_disk) / (4 * 16);
        geo.heads = 4;
        geo.sectors = 16;
        geo.start = get_start_sect(bdev);
        return copy_to_user((void __user *)arg, &geo, sizeof(geo))? -EFAULT : 0;
    }
    return -ENOTTY;
}
```

MMC 驱动的 `ioctl` 方法只实现了一个命令，就是获得块设备的几何物理信息，如获得柱面，磁头和扇区大小等信息，然后将这些信息发送给用户空间的应用程序。

### 7.2.5.4 MMC 驱动的 request 方法

块设备的 `request` 方法应该是与字符设备驱动的最大不同之处，因为块设备操作中没有定义 `read/write` 等方法，所以块设备中这些方法的实现就交给了 `request` 方法来实现。其实 `request` 方法的实现是在 MMC 初始化时就被实现的，只是为了能更详细的介绍它的实现所以专门放在这里介绍。MMC 驱动的 `request` 方法是由 `mmc_blk_prep_rq` 和 `mmc_blk_issue_rq` 这两个函数实现，这两个函数被 `mmc_blk_alloc` 函数调用，而这个函数又被 `mmc_blk_probe` 函数调用。`mmc_blk_probe` 函数是在 MMC 驱动初始化时调用的，所以说 `request` 方法也在初始化时被调用。`mmc_blk_prep_rq` 函数用来检查是否有 MMC 设备和 MMC 主机，它的实现代码如下所示：

```
static int mmc_blk_prep_rq(struct mmc_queue *mq, struct request *req)
{
    struct mmc_blk_data *md = mq->data;
    int stat = BLKPREP_OK;
    if (!md || !mq->card) {
        printk(KERN_ERR "%s: killing request - no device/host\n",
```

```

        req->rq_disk->disk_name);
    stat = BLKPREP_KILL;
}
return stat;
}

```

当检测到没有 MMC 主机或没有 MMC 设备时，该函数返回一个非 0 值。否则，返回一个 0 代表设备都正常存在。

mmc\_blk\_issue\_rq 函数实现的块设备的真正 read/write 等操作，它的实现代码如下所示：

```

static int mmc_blk_issue_rq(struct mmc_queue *mq, struct request *req)
{
    .....
    do {
        struct mmc_blk_request brq;
        struct mmc_command cmd;
        .....
        if (rq_data_dir(req) == READ) {
            brq.cmd.opcode = brq.data.blocks > 1 ? MMC_READ_MULTIPLE_BLOCK :
MMC_READ_SINGLE_BLOCK;
            brq.data.flags |= MMC_DATA_READ;
        } else {
            brq.cmd.opcode = MMC_WRITE_BLOCK;
            brq.cmd.flags = MMC_RSP_R1B;
            brq.data.flags |= MMC_DATA_WRITE;
            brq.data.blocks = 1;
        }
        brq.mrq.stop = brq.data.blocks > 1 ? &brq.stop : NULL;

        brq.data.sg = mq->sg;
        brq.data.sg_len = blk_rq_map_sg(req->q, req, brq.data.sg);

        mmc_wait_for_req(card->host, &brq.mrq);
        .....
        do {
            int err;

            cmd.opcode = MMC_SEND_STATUS;
            cmd.arg = card->rca << 16;
            cmd.flags = MMC_RSP_R1;
            err = mmc_wait_for_cmd(card->host, &cmd, 5);
            if (err) {
                printk(KERN_ERR "%s: error %d requesting status\n",
                    req->rq_disk->disk_name, err);
                goto cmd_err;
            }
        }
    }
}

```

```

        } while (!(cmd.resp[0] & R1_READY_FOR_DATA));
        spin_lock_irq(&md->lock);
        ret = end_that_request_chunk(req, 1, brq.data.bytes_xfered);
        if (!ret) {
            add_disk_randomness(req->rq_disk);
            blkdev_dequeue_request(req);
            end_that_request_last(req);
        }
        spin_unlock_irq(&md->lock);
    } while (ret);
    mmc_card_release_host(card);
    return 1;

cmd_err:
    .....
    return 0;
}

```

该函数主要完成 MMC 的命令请求，以及相应的数据传输。它可以是单个块的操作，也可以是多个块的操作。其中 `mmc_wait_for_req` 函数用来实现 MMC 主机的命令请求，用来启动一个 MMC 主机请求。`mmc_wait_for_cmd` 函数用来启动一个 MMC 命令并且等待这个命令的完成。用 `end_that_request_chunk` 函数来结束一个 I/O 请求，当该函数返回 0 时，证明已经处理了这个请求，返回 1 时，表明这个请求还在悬挂在缓冲中。当结束这个 I/O 请求时，必须调用 `mmc_card_release_host` 函数来释放 MMC 主机，以便其他 MMC 主机声明使用 MMC 驱动操作。

此外，还有一个一般的 MMC 请求处理函数，那就是 `mmc_request` 函数，这个函数被那些 MMC 主机的队列调用，当 MMC 主机空闲时，将利用这个函数查找一个等待的请求，然后把它唤醒处理。`mmc_request` 函数的实现代码如下：

```

static void mmc_request(request_queue_t *q)
{
    struct mmc_queue *mq = q->queuedata;

    if (!mq->req)
        wake_up(&mq->thread_wq);
}

```

到此为止，关于 MMC 驱动程序的主要实现代码已经分析完毕，该程序是基于 Linux 2.6.10 内核中提供的 MMC 代码为基础的，读者可以通过附件的光盘获取 MMC 驱动的实现代码。

## 7.2.6 S3C2410 SDI 接口驱动分析

要实现 MMC 设备驱动的正常工作的，还需要实现它 S3C2410 的 SDI 接口驱动。前面已经介绍过，SDI 接口用于驱动 SD 存储卡、SDIO 设备和 MMC。首先来看一下该接口驱动的初始化。

### 7.2.6.1 SDI 初始化

首先在设计 S3C2410 的 SDI 接口时，将其作为一个设备驱动来实现的，所以该驱动入口函数就是它的加载函数 `s3c2410sdi_init`，它的实现代码比较简单，具体如下：

```
static int __init s3c2410sdi_init(void)
{
    return driver_register(&s3c2410sdi_driver);
}
```

其调用 `driver_register` 函数来注册 SDI 接口驱动，其中传递参数 `s3c2410sdi_driver` 被定义为 SDI 接口驱动，它的定义如下：

```
static struct device_driver s3c2410sdi_driver =
{
    .name          = "s3c2410-sdi",
    .bus           = &platform_bus_type,
    .probe         = s3c2410sdi_probe,
    .remove        = s3c2410sdi_remove,
};
```

通过上面代码可以看出，它定义了 SDI 接口驱动的两个方法，分别是 `probe` 和 `remove`。这两个函数的作用类似于 MMC 设备驱动中的 `probe` 和 `remove` 方法。这里 `probe` 方法是由 `s3c2410sdi_probe` 函数来实现，主要完成 SDI 接口的初始化工作，包括 MMC host 的初始化，中断的初始等。由于篇幅的关系，这里不再列举它的实现代码，读者可以参考附件光盘中的参考代码。此外，`remove` 方法是由 `s3c2410sdi_remove` 函数来完成，用它来实现与 `s3c2410sdi_probe` 函数相反功能的操作。

最后再看一下 SDI 接口驱动的卸载程序，它的实现也非常简单，调用 `driver_unregister` 函数来实现 SDI 接口驱动的卸载功能，具体代码如下：

```
static void __exit s3c2410sdi_exit(void)
{
    driver_unregister(&s3c2410sdi_driver);
}
```

### 7.2.6.2 SDI 接口驱动方法

在分析 SDI 接口驱动方法的实现之前，首先看一下 SDI 接口驱动方法的定义，它是由 `s3c2410sdi_ops` 变量定义的，该变量的定义如下：

```
static struct mmc_host_ops s3c2410sdi_ops = {
    .request = s3c2410sdi_request,
    .set_ios = s3c2410sdi_set_ios,
};
```

其中 `mmc_host_ops` 结构定义在 `<include/linux/mmc/host.h>` 文件中，用于定义 MMC host 驱动的方法，该结构中只定义了 `request` 和 `set_ios` 两个方法。其中 `request` 方法由 `s3c2410sdi_request` 函数实现，主要完成 MMC 所有请求命令相关的处理功能，包括数据的读/写等实现。另外 `set_ios` 方法由 `s3c2410sdi_set_ios` 函数实现，主要用于设置 MMC 电源的

开/关，以及开启 MMC 时钟等功能。由于 SDI 接口驱动方法实现代码较长，并且也不是我们研究的重点，所以这里只是做一个简单的介绍，有兴趣的读者可以分析附件光盘中的提供的参考代码。

### 7.2.7 配置和编译驱动程序

与第 6 章中介绍的触摸屏设备驱动类似，需要相应的配置才能进行正确的编译。由于 MMC 设备驱动内核中已经提供，只不过内核中提供的驱动是应用于 X86 平台的，而我们这里开发的目标平台是 ARM，所以需要一定的修改。本实例将源程序放在了 driver/mmc 目录下。同时需要修改该目录下 Kconfig 配置文件，添加以下内容到该文件：

```
config MMC_S3C2410
    tristate "Samsung S3C2410 Multimedia Card Interface support"
    depends on ARCH_S3C2410 && MMC
    help
        This selects the Samsung S3C2410 Multimedia Card Interface
        support.
        If unsure, say N.
```

添加以上内容后，在用 make menuconfig 配置内核时会增加关于 S3C2410 的 MMC 选项，此外，还需要修改 driver/mmc/Makefile 文件，添加下面一行内容：

```
obj-$(CONFIG_MMC_S3C2410) += s3c2410mci.o
```

修改完配置文件和 Makefile 文件，最后重新配置内核选项，将触摸屏选项以模块形式添加或直接添加到内核，然后就可以编译该驱动程序了。

## 7.3 本章小结

本章讲述了另一类常见的 Linux 设备驱动程序——块设备驱动，首先讲述了块设备相关的重要数据结构，包括 block\_device\_operations，gendisk，request 结构，并且讲述了块设备中非常重要的请求处理函数；然后重点讲述了一个块设备驱动的典型实例——MMC 驱动开发，读者通过学习该设备驱动程序将对块设备驱动开发有深入的了解，此外对 MMC/SD 的工作原理有一定的了解。下一章将介绍另一类常见的 Linux 设备驱动程序——网络设备。

## 7.4 常见问题

### 1. 字符设备与块设备之间的主要区别？

参考答案：字符设备与块设备的最大不同就是：字符设备传输数据是按字节的大小传输，而块设备是以块为单位传输，通常块的大小是 1024 字节。其次字符设备通常是直接作用于 I/O 端口的，而块设备是间接作用的，因为它与内核之间是有缓冲区的。

### 2. 块设备中请求处理函数的作用？

参考答案：请求函数是块设备驱动程序的核心，实际的工作中，设备的启动，以及对设备的读取都是在 request 函数中实现的。使用请求函数的原因是块设备利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求时就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的



CPU 时间。

### 3. MMC 与 SD 卡的主要区别？

参考答案：MMC 与 SD 卡都是当今应用非常广的小型存储设备，当然它们还是有区别的。首先，在外形上，MMC 外形尺寸大约为  $32\text{mm} \times 24\text{mm} \times 1.4\text{mm}$ ，而 SD 卡外形尺寸是  $32\text{mm} \times 24\text{mm} \times 2.1\text{mm}$ 。其次，MMC 有 7 个针脚。而 SD 卡有 9 个针脚。最后，支持 SD 卡的设备一定支持 MMC，而支持 MMC 的设备不能支持 SD 卡，所以它们之间并不是互相兼容的。

## 第 8 章 网络设备驱动程序

本章学习目标：

- | 熟悉网络设备驱动程序概念
- | 熟悉网络设备驱动相关的两个结构：net\_device, sk\_buff
- | 熟悉常见的网络协议和以太网
- | 理解 CS8900A 网卡的硬件接口连接
- | 理解 CS8900A 网卡驱动程序实现

### 8.1 网络设备驱动介绍

在分析了字符设备驱动和块设备驱动之后，接下来应该分析网络驱动程序了。与之前两种设备驱动不同的是，网络驱动程序不再是对文件操作了，而是由专门的网络接口来实现。应用程序不能直接访问网络驱动程序，只能由网络子系统与它交互。此外，不象字符设备和块设备在/dev目录下有一个特殊文件作为其设备，而网络设备就没有这样的入口点。网络设备与块设备最重要的不同就是：块设备只响应来自内核的请求，而网络驱动程序异步的接受来自外部世界的数据包。首先让我们来看一下网络设备驱动相关的重要数据结构。

#### 8.1.1 网络设备驱动相关的重要结构

与网络设备驱动相关的重要数据有两个，分别是和 net\_device 和 sk\_buff 结构，下面将详细介绍这两个结构体。

##### 8.1.1.1 net\_device 结构

该结构是网络设备驱动的核心，它包含了许多成员。由于 net\_device 结构体庞大，这里就不列举它的详细定义，读者可以参考<include/linux/netdevice.h>文件，阅读它的完整定义。这里将介绍一下该结构的一些常用成员。该结构可分为五个部分：

###### 1. 全局成员

**n**     char   name[IFNAMSIZ];

该成员表示设备的名称。

**n**     unsigned long     state;

该成员表示设备的状态，它包含许多标志，驱动程序无需直接操作这些标志，而是内核提供一组工具函数来实现。

**n**     struct net\_device   \*next;

该成员表示全局链表下一个设备的指针，注意，驱动程序不应该修改这个成员。

**n**     int   (\*init)(struct net\_device \*dev);

该成员是一个初始化函数，如果这个指针被设置了，则 register\_netdev 函数将调用该函数完成对 net\_device 结构的初始化。目前大多数网络驱动程序不再使用这个函数了，通常他

们是在注册接口之前完成初始化工作。

## 2. 硬件相关成员

下面的成员包含相关设备的底层硬件信息。

**n**     **unsigned long**     **mem\_end;**  
该成员表示共享内存的终止地址。

**n**     **unsigned long**     **mem\_start;**  
该成员表示共享内存的起始地址。

**n**     **unsigned long**     **base\_addr;**  
该成员表示网络接口的 I/O 基地址。

**n**     **unsigned int**       **irq;**  
该成员表示设备的中断号。

**n**     **unsigned char**     **if\_port;**  
该成员表示多个端口设备上使用哪个端口。

**n**     **unsigned char**     **dma;**  
该成员表示为设备分配的 DMA 通道。

## 3. 接口相关成员

**n**     **unsigned**         **mtu;**  
该成员表示最大传输单元 (MTU) 值的接口。

**n**     **unsigned short**    **type;**  
该成员表示硬件类型的接口。

**n**     **unsigned short**    **hard\_header\_len;**  
该成员表示数据包的头信息的长度。

**n**     **unsigned char**     **broadcast[MAX\_ADDR\_LEN];**  
该成员表示硬件广播地址。

**n**     **unsigned char**     **dev\_addr[MAX\_ADDR\_LEN];**  
该成员表示硬件地址。

**n**     **unsigned char**     **addr\_len;**  
该成员表示硬件地址长度。

**n**     **unsigned long**     **tx\_queue\_len;**  
该成员表示在设备的传输队列中排队的最大 frame 个数。

**n**     **unsigned short**    **flags;**  
该成员表示接口标志。

## 4. 设备方法成员

**n**     **int**    **(\*open)(struct net\_device \*dev);**  
该方法用于打开接口，该函数应该注册所有的系统资源 (I/O 端口, IRQ, DMA 等等)，打开硬件并对设备执行其所需的设置。

**n**     **int**    **(\*stop)(struct net\_device \*dev);**  
该方法用于终止接口，与 open 执行的操作相反。

**n**     **int**    **(\*hard\_start\_xmit)(struct sk\_buff \*skb, struct net\_device \*dev);**  
该方法用于初始化数据包的传输。

**n**     **int**    **(\*hard\_header)(struct sk\_buff \*skb, struct net\_device \*dev, unsigned short type, void \*daddr, void \*saddr, unsigned len);**  
该方法根据先前检索到的源和目标硬件地址建立硬件头。该函数在调用 hard\_start\_xmit 函数之前调用，它的任务是将作为参数传递进入的信息组织成设备特有的硬件头信息。

- n**     struct net\_device\_stats\* (\*get\_stats)(struct net\_device \*dev);  
该方法用于获得接口的统计信息，例如查看 ip 地址等调用该方法。
  - n**     int (\*rebuild\_header)(struct sk\_buff \*skb);  
该方法用于重新建立硬件头，在传输数据包之前并且在完成 ARP 解析之后调用该方法。
  - n**     int (\*set\_config)(struct net\_device \*dev, struct ifmap \*map);  
该方法用于改变接口配置。
  - n**     void (\*tx\_timeout) (struct net\_device \*dev);  
该方法用于解决数据包传输失败并重新开始数据包的传输。
5. 工具相关成员
- n**     unsigned long       trans\_start;  
该成员用于保存 jiffies 值，在传输开始及接收到数据包时负责更新。
  - n**     void \*priv;  
该成员等价于 filp->private\_data。在现代的驱动程序中，该成员由 alloc\_netdev 设置，不能直接访问，如果要访问需要使用 netdev\_priv 函数。
  - n**     int watchdog\_timeo;  
该成员表示在网络层确定传输已经超时并且调用驱动程序的 tx\_timeout 函数之前的最小时间。
  - n**     spinlock\_t xmit\_lock;  
该成员用来避免同时对驱动程序的 hard\_start\_xmit 函数的多次调用。
  - n**     int xmit\_lock\_owner;  
该成员用于获得 xmit\_lock 的 CPU 编号。
  - n**     struct dev\_mc\_list \*mc\_list;  
该成员表示处理组播传输。
  - n**     int mc\_count;  
该成员表示 mc\_list 所包含的项数目。
- net\_device 结构还包括许多其他成员，不过在网络驱动程序中很少用它们，所以这里只介绍了常用的一些成员。

### 8.1.1.2 sk\_buff 结构

该结构是 Linux 内核网络子系统中一个非常重要的结构，sk\_buff 是 socket（套接字）缓冲区结构，即 socket buffer 的简写。该结构定义在<include/linux/skbuff.h>文件中。下面将介绍 sk\_buff 结构中那些可能被驱动程序中使用的成员。

- n**     struct sk\_buff       \*next;  
该成员指向下一个 sk\_buff 结构对象。
- n**     struct sk\_buff       \*prev;  
该成员指向前一个 sk\_buff 结构对象。
- n**     unsigned char       \*head;  
该成员表示指向已分配空间的开头。
- n**     unsigned char       \*data;  
该成员表示有效 octet\*（注 1）的开头。
- n**     unsigned char       \*tail;  
该成员表示有效 octet 的结尾。
- n**     unsigned char       \*end;

该成员指向 `tail` 可达到的最大地址。

**n**      `unsigned int`          `len;`

该成员表示数据包中全部数据的长度。

**n**      `unsigned int`          `data_len;`

该成员表示分隔存储的数据片断的长度。

**n**      `unsigned char`      `ip_summed;`

该成员表示对数据包的校验策略。

**n**      `unsigned char`      `pkt_type;`

该成员表示在发送过程中使用的数据包类型。

\*注 1: `octet` 在网络中表示的 8 位数据, 和 `Byte` (字节) 类似, 不过在网络术语中经常使用 `octet` 而不使用 `byte`。

## 8.1.2 常见的网络术语

开发网络设备驱动程序需要对网络技术有一定的了解, 尤其是对常见的网络通信协议和以太网有所了解。首先介绍一下常见的网络协议。

### 8.1.2.1 常见的网络协议

TCP/IP 是 Transfer Control Protocol/ Internet Protocol 的缩写, 即传输控制协议/网际协议。它是互联网中最基本的协议, 主要用于定义网络层和网络层之上的协议标准。下面将介绍在嵌入式设备中经常会用到的几种协议。

#### I      TCP

Transfer Control Protocol, 传输控制协议。在发送端, TCP 将要发送的数据拆分为数据段, IP 将数据段组装为包含数据段以及发送方地址和目的地址的分组。然后 IP 将分组发送到路由器以便传递。在接收端, IP 接收分组并将其拆分为数据段。TCP 将数据段组装为原始的数据集。

#### I      IP

Internet Protocol, 网际网协议。将数据从一个 Internet 位置传递到另一个位置的 TCP/IP 的部分。IP 负责通过网络定址和发送 TCP 分组。IP 提供不保证分组到达目的地或以发送时的序列接受的最大程度的无连接传递系统。

#### I      ICMP

Internet Control Messages Protocol, 网间控制报文协议。网际协议(IP)的扩展, ICMP 允许出错消息的生成、检测分组和与 IP 相关的信息邮件。

#### I      UDP

User Datagram Protocol, 用户数据报协议。它是 ISO (International Organization for Standardization, 国际标准化组织) 参考模型中一种无连接的传输层协议, 提供面向事务的简单不可靠信息传送服务。UDP 协议基本上是 IP 协议与上层协议的接口。UDP 协议适用端口分别运行在同一台设备上的多个应用程序。

#### I      ARP/RARP

Address Resolution Protocol, 地址解析协议。一种网络维护协议, 是 TCP/IP 套件的成员 (与数据传输没有直接关系)。它用于动态发现与给定主机的高层 IP 地址对应

的低层物理网络硬件地址。ARP 限于支持广播包的物理网络系统。与其作用相反的是 Reverse Address Resolution Protocol, 逆向地址解析协议。

### 8.1.2.2 以太网介绍

1973 年, 施乐 (Xerox) 公司设计了第一个局域网系统, 被命名为 Ethernet, 带宽为 2.94Mbps。1982 年, DEC、Intel 和 Xerox 联合发表了 Ethernet Version 2 规范, 将带宽提高到了 10Mbps, 并正式投入商业市场。1983 年, IEEE 通过了 802.3 CSMA/CD 规范。IEEE 802 标准是由 IEEE (国际电气和电子工程师学会) 制订的局域网标准, IEEE 802 委员会有 10 多个分委员会。关于 802 标准分类如下:

- | 802.1A, 概述、体系结构和网络互连, 网络管理
- | 802.1B, 寻址、网络管理、网间互连及高层接口
- | 802.2, 逻辑链路控制 LLC
- | 802.3, CSMA/CD 共享总线网, 即 Ethernet
- | 802.5, 令牌环网 (Token-Ring)
- | 802.11, 无线局域网

IEEE 802.3 命名规则是: IEEE 802.3 X TYPE-Y NAME

其中 X 表示传输介质, 5 指粗同轴电缆, 2 指细同轴电缆, T 指双绞线, F 指光纤。TYPE 代表传输方式, Base 指基带传输, Broad 指宽带传输。Y 与 X 一样表示传输介质。NAME 表示局域网的名称, Ethernet 为以太网, FastEthernet 为快速以太网, GigaEthernet 为千兆以太网, F 指光纤。

以太网的数据链路层, 由以下部分组成:

- | 逻辑链路控制子层 (Logical Link Control, 即 LLC 子层), 为网络层定义了各种服务及接口
- | 介质访问控制子层 (Media Access Control, 即 MAC 子层), 定义了对各种物理传输介质的访问及控制技术

IEEE 802 标准为各种局域网技术定义了统一的 LLC 子层, 而各种局域网技术的 MAC 子层不尽相同, 所以以太网的数据链路层主要是以太网的 MAC 子层。

CSMA/CD (Carrier Sense Multiple Access/Collision Detected) 协议的全称为带冲突检测的载波监听多路访问技术, 是以太网中所使用的介质访问控制技术。这种介质访问控制技术是基于共享介质的, 采用共享总线的拓扑结构, 以广播的形式进行数据传送。在某一时刻, 连接在共享总线上的所有站点中, 只能有一个站点可以发送数据。

CSMA 协议是指:

- | 总线有两个状态: “空闲” 和 “忙”
- | 每个站点在使用总线发送数据帧之前首先监听总线, 查看总线是否处于 “空闲” 状态, 如果总线 “忙” 就继续等待, 继续监听, 一直到总线 “空闲”
- | 要发送数据帧的站点在监听到总线 “空闲” 时, 开始发送数据帧

CD 技术是指:

- | 在使用 CSMA 协议时, 有可能会出现两个或两个以上的站点同时监听到总线 “空闲” 的情况, 此时这些站点将同时开始发送数据帧, 出现这种情况时, 总线会发生冲突, 导致所有站点的发送全部失败
- | 每个站点在发送数据后必须检测是否发生了冲突
- | 在发生冲突的情况下, 站点使用二进制指数退避算法重发数据帧

MAC 地址是指每张网卡中包含一个独一无二的物理地址, 由 48 位二进制构成, 前 24 位

代表设备生产商，由 IEEE 管理分配，后 24 位为各生产商内部的编号。由于 CSMA/CD 协议中，帧以广播的形式进行传输，所以总线上的每个站点要根据帧中包含的目的 MAC 地址和自己网卡中的 MAC 地址是否一致来决定是否接收该帧。

## 8.2 网络设备驱动开发实例

本章的以网络设备中非常典型的实例——CS8900A 以太网适配器为开发对象，主要讲述网络设备驱动的开发过程，使读者清楚认识网路驱动开发与前面讲述的字符设备和块设备驱动开发的不同之处。首先介绍一下 CS8900A 设备。

### 8.2.1 CS8900A 介绍

CS8900A 是一个真正的单芯片、全双工的以太网解决方案产品。它的主要功能块包括：一个 ISA (Industry Standard Architecture, 工业标准结构) 总线接口，一个 802.3MAC (Media Access Control, 介质访问控制) 引擎，集成内存，一个串行的 EEPROM (电可擦除只读存储器) 接口和一个拥有 10BASE-T 和 AUI 的完整模拟前后端。

#### 8.2.1.1 CS8900A 的组成部分介绍

下面分别介绍一下 CS8900A 的五个组成部分。

##### ü ISA 总线接口

它的配置选项有四个可选中断和 3 个 DMA 通道 (在初始化时选择)。在存储模式，它支持标准的或读总线循环不用引入附加的等待状态。

##### ü 集成内存

CS8900A 合并一个 4k 字节的页在芯片内存中，不但消除了这方面的成本还节省了连接外部内存芯片空间。不像其他以太网控制器需要复杂且无效的内存管理机制，CS8900A 内存完整的传输和接收 frame 都在芯片内。此外，CS8900A 可以操作在内存空间，I/O 空间或外部的 DMA 控制器，提供了设计的最大灵活性。

##### ü 802.3 以太网 MAC 引擎

CS8900A 以太网 MAC 引擎完全兼容 IEEE (Institute of Electrical and Electronics Engineers, 电气和电子工程师协会) 802.3 以太网标准，并且支持全双工操作。它处理以太网 frame 传输和接收的所有方面，包括：冲突检测，CRC 产生和测试等。

##### ü EEPROM 接口

CS8900A 提供一个简单且有效的串行 EEPROM 接口，允许配置信息保存在一个可选的 EEPROM，并且在上电时自动转载。

##### ü 完整的模拟前后端

CS8900A 模拟前后端结合一个 Manchester 编/解码，一个时钟恢复电路，10BASE-T 收发器和完整的附加单元接口。它提供一个手动或自动选择 10BASE-T 或 AUI，并且提供 3 个片上 LED 驱动用于显示连接状态，总线状态和以太网线的活动情况。

### 8.2.1.2 CS8900A 的系统应用

CS8900A 被设计用于主板应用和适配器应用。下面分别介绍这两种系统应用。

#### ü CS8900A 主板方案应用

CS8900A 要求最少数量的外部组件用于完全的以太网结点。所以它的 PCB 尺寸可以设计的非常小，此外 CS8900A 有良好的省电特点并且他的 CMOS 设计使得它完美的应用移动设备或桌面 PC 机上。主板设计选项包括：一个 EEPROM 用于储存节点相关的信息；20MHz 晶体振荡器（也可用 20MHz 时钟信号代替）。CS8900A 的主板方案应用如图 8.1 所示。

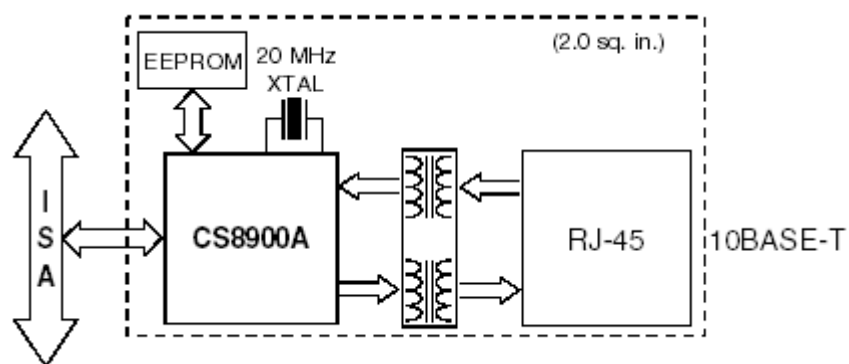


图 8.1 CS8900A 完整的以太网主板应用方案

#### ü CS8900A 适配器方案应用

CS8900A 高效的数据包结构，流传输技术和自动选择 DMA 选项，这些使得它用于高性能、低成本的 ISA 适配卡选择。适配卡包括设计选项包括：一个引导 PROM（Programmable Read-Only Memory，可程序的只读存储器）能被添加用于支持无磁盘应用；10BASE-T 发射机和接收机；一个外部的可上锁的地址总线解码电路；片上 LED 端口。该系统应用如图 8.2 所示。

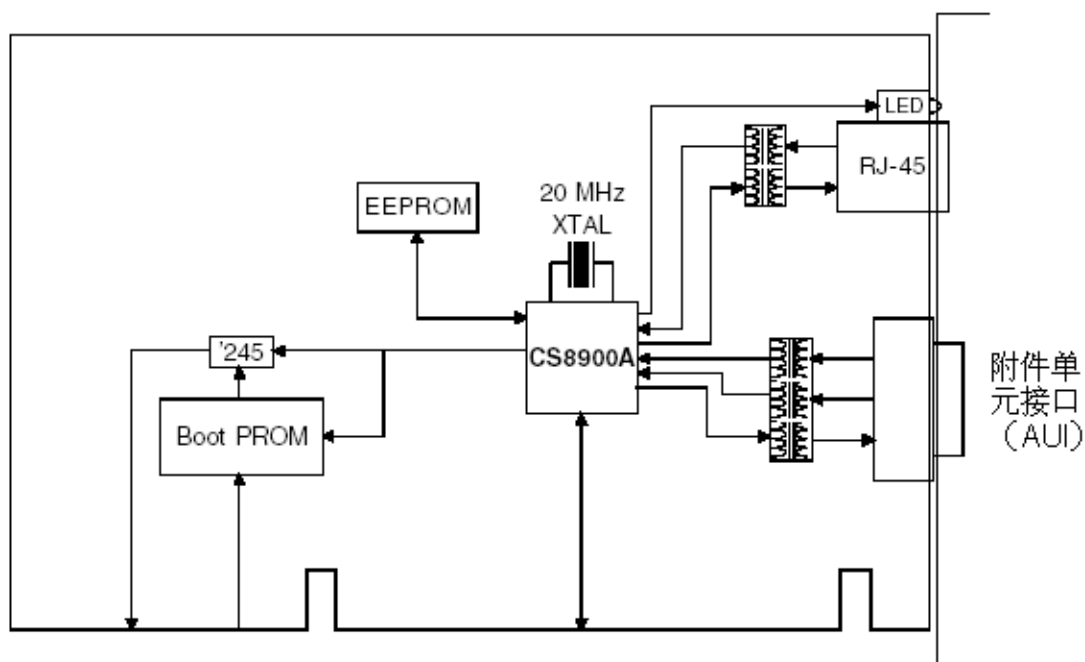


图 8.2 CS8900A 全特征的 ISA 适配器方案



### 8.2.2CS8900A 网卡驱动概要设计

在设计 CS8900A 以太网卡驱动程序之前，首先需要设计其硬件电路接口，不过一般这部分工作是由硬件工程师来完成，但是对于驱动开发人员来说，必须熟悉这些芯片之间的连接接口，进而设计其驱动程序。

#### 8.2.2.1 CS8900A 网卡接口

CS8900A 网卡接口连接如图 8.3，它主要由三部分组成：RJ45，10BASE-T 和 CS8900A。其中 CS8900A 的芯片图没有画出，不过图中给出了 10BASE-T 与 CS8900A 连接的针脚，如 RXD-，RXD+，TXD-和 TXD+。

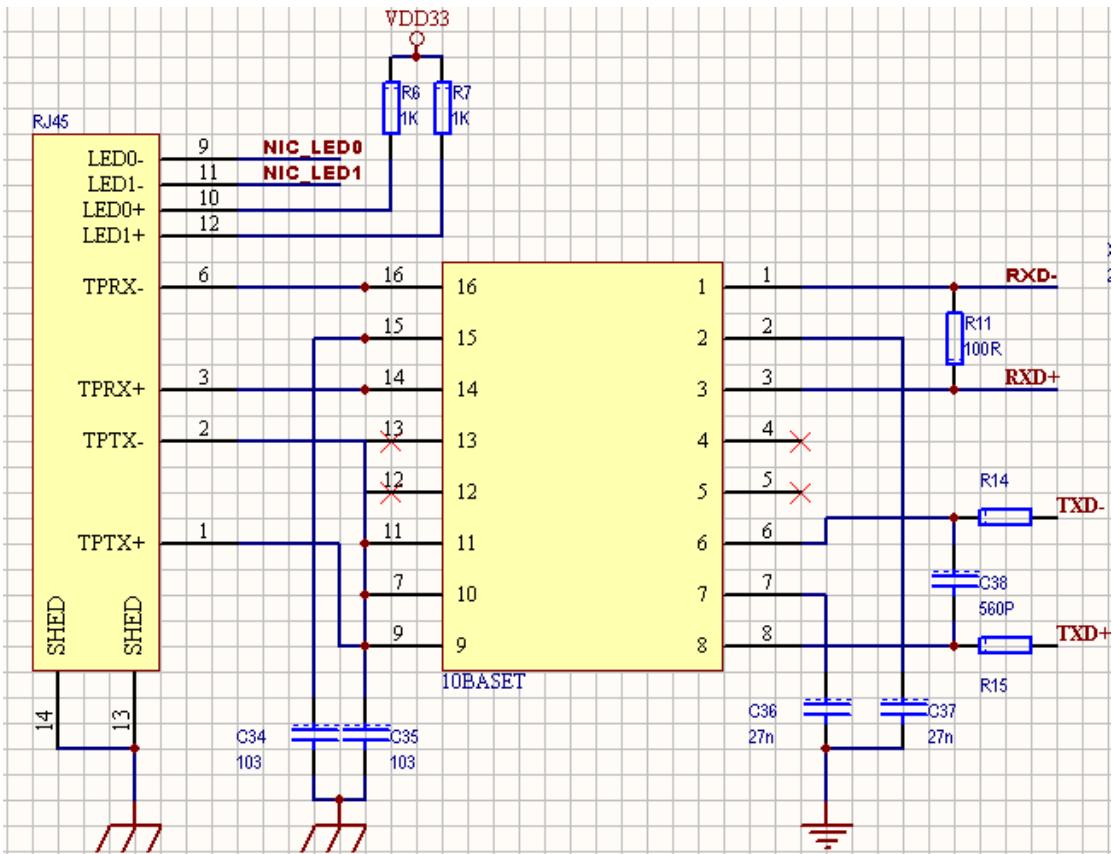


图 8.3 CS8900A 网卡接口连接

#### 8.2.2.2 网络驱动程序的体系结构

所有的 Linux 网络驱动程序遵循通用的接口。设计时采用的是面向对象方法\*（注 2）。一个设备就是一个对象(net\_device 结构)，它内部有自己的数据和方法。每一个设备的方法被调用时的第一个参数都是这个设备对象本身。这样这个方法就可以存取自身的数据(类似面向对象程序设计时的 this 引用)。

\*注 2：面向对象方法（Object-Oriented Method）是一种把面向对象的思想应用于软件开发过程中，指导开

发活动的系统方法，简称 OO（Object-Oriented）方法，是建立在“对象”概念基础上的方法学。

Linux 网络驱动程序的体系结构可以划分为四层如图 8.4 所示，从上到下分别为：网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层以及网络设备媒介层。我们在设计网络驱动程序时，最主要的工作就是完成设备驱动功能层，使其满足我们所需的功能。在 Linux 中所有网络设备都抽象为一个接口，这个接口提供了对所有网络设备的操作集合。由数据结构 `struct net_device` 来表示网络设备在内核中的运行情况，即网络设备接口。它既包括纯软件网络设备接口，如环路（Loopback），也包括硬件网络设备接口，如以太网卡。而由以 `dev_base` 为头指针的设备链表来集体管理所有网络设备，该设备链表中的每个元素代表一个网络设备接口。数据结构 `net_device` 在前面的小结中已经介绍，这里就不再重复。

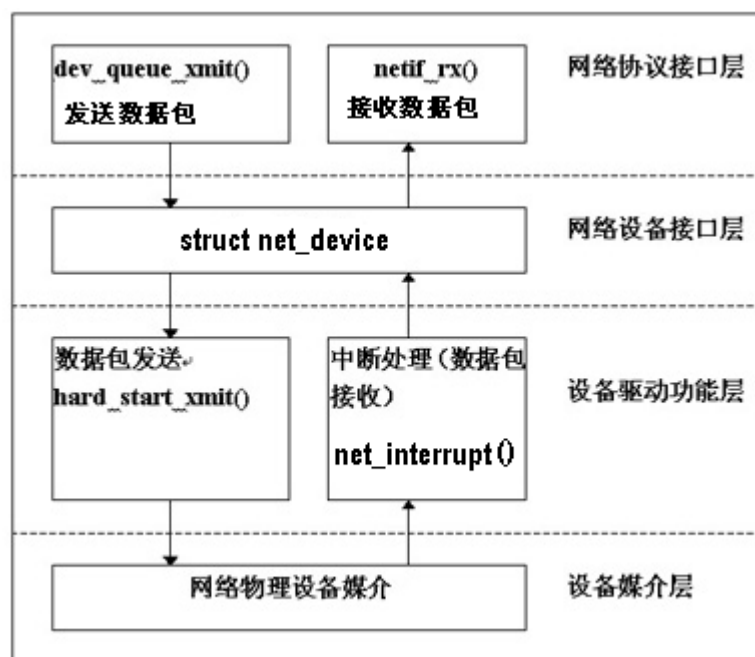


图 8.4 Linux 网络驱动体系结构

### 8.2.2.3 网络驱动程序的主要功能

网络驱动程序主要完成系统的初始化、数据包的发送和接收。网络设备的初始化主要由 `net_device` 数据结构中的 `init` 函数指针所指向的初始化函数来完成，当内核启动或加载网络驱动模块的时候，就会调用这个初始化函数。在初始化函数中通过检测物理设备的硬件特征来侦测网络物理设备是否存在，然后再对设备进行资源配置，接下来构造设备的 `net_device` 数据结构，并用检测到的数据对 `net_device` 中的变量初始化，最后向 Linux 内核注册该设备并申请内存空间。数据包的发送和接收是实现 Linux 网络驱动程序中两个最关键的过程，对这两个过程处理的好坏将直接影响到驱动程序的整体运行质量。在网络设备驱动加载时，通过 `net_device` 域中的 `init` 函数指针调用网络设备的初始化函数对设备进行初始化，如果操作成功再通过 `net_device` 域中的 `open` 函数指针调用网络设备的打开函数打开设备，并通过 `net_device` 域中建立硬件包头函数指针 `hard_header` 来建立硬件包头信息。最后通过协议接口层函数 `dev_queue_xmit`（参见 `<net/core/dev.c>` 文件）来调用 `net_device` 域中的 `hard_start_xmit` 函数指针完成数据包的发送。该函数把存放在套接字缓冲区中的数据发送到物理设备，该缓冲区是由数据结构 `sk_buff` 来表示的。数据包的接收是通过中断机制来完成的。当有数据到

达时产生中断信号，网络设备驱动功能层调用中断处理程序（即数据包接收程序）来处理数据包的接收，随后网络协议接口层调用 `netif_rx`（参见`<net/core/dev.c>`文件）函数，把接收到的数据包传输到网络协议的上层进行处理。

## 8.2.3 CS8900A 适配器驱动程序分析

CS8900A 适配器驱动程序在 Linux 内核中已经提供，我们使用内核 2.6.10 版本中的代码 CS8900A 适配器驱动程序为分析对象，讲述网络驱动程序的开发过程。首先来分析一下 CS8900A 模块的初始化。

### 8.2.3.1 初始化

CS8900A 适配器的初始化主要由 `net_device` 数据结构中的 `init` 函数指针所指向的初始化函数来完成，当内核启动或加载网络驱动模块的时候，就会调用这个初始化函数。在初始化函数中通过检测物理设备的硬件特征来侦测网络物理设备是否存在，然后再对设备进行资源配置，接下来构造设备的 `device` 数据结构，并用检测到的数据对 `net_device` 中的变量初始化，最后向 Linux 内核注册该设备并申请内存空间。

首先，定义一个全局变量 `cs8900_dev`，它为 `net_device` 结构的对象，并且定义了 `cs8900_dev` 对象的初始化函数为 `cs8900_probe`，代码如下：

```
static struct net_device cs8900_dev =
{
    init: cs8900_probe
};
```

接下来，分析 `cs8900_probe` 函数的具体实现，它主要用于对网卡进行检测，并初始化系统中网络设备信息用于后面的网络数据的发送和接收。它的完整代码实现如下：

```
1  int __init cs8900_probe (struct net_device *dev)
2  {
3      static cs8900_t priv;
4      int i,result;
5      u16 value;
6
7      printk (VERSION_STRING"\n");
8      memset (&priv,0,sizeof (cs8900_t));
9      ether_setup (dev);
10     dev->open          = cs8900_open;
11     dev->stop           = cs8900_stop;
12     dev->hard_start_xmit = cs8900_send_start;
13     dev->get_stats      = cs8900_get_stats;
14     dev->set_multicast_list = cs8900_set_receive_mode;
15     dev->tx_timeout     = cs8900_transmit_timeout;
16     dev->watchdog_timeo = HZ;
17     #if defined(CONFIG_ARCH_SMDK2410)
18     dev->dev_addr[0] = 0x08;
```

```

19     dev->dev_addr[1] = 0x00;
20     dev->dev_addr[2] = 0x3e;
21     dev->dev_addr[3] = 0x26;
22     dev->dev_addr[4] = 0x0a;
23     dev->dev_addr[5] = 0x5b;
24 #else
25     dev->dev_addr[0] = 0x00;
26     dev->dev_addr[1] = 0x12;
27     dev->dev_addr[2] = 0x34;
28     dev->dev_addr[3] = 0x56;
29     dev->dev_addr[4] = 0x78;
30     dev->dev_addr[5] = 0x9a;
31 #endif
32     dev->if_port    = IF_PORT_10BASET;
33     dev->priv      = (void *) &priv;
34     spin_lock_init(&priv.lock);
35 #if defined(CONFIG_ARCH_SMDK2410)
36     dev->base_addr = vSMDK2410_ETH_IO + 0x300;
37     dev->irq = SMDK2410_ETH_IRQ;
38 #endif /* #if defined(CONFIG_ARCH_SMDK2410) */
39     if ((result = check_mem_region (dev->base_addr, 16))) {
40         return (result);
41     }
42     request_mem_region (dev->base_addr, 16, dev->name);
43     /* verify EISA registration number for Cirrus Logic */
44     if ((value = cs8900_read (dev,PP_ProductID)) != EISA_REG_CODE) {
45         return (-ENXIO);
46     }
47     /* verify chip version */
48     value = cs8900_read (dev,PP_ProductID + 2);
49     if (VERSION (value) != CS8900A) {
50         return (-ENXIO);
51     }
52     /* setup interrupt number */
53     cs8900_write (dev,PP_IntNum,0);
54     result = cs8900_eeprom (dev);
55     if (result == -ENODEV) {
56         /* no eeprom or invalid config block, configure MAC address by hand */
57         for (i = 0; i < ETH_ALEN; i += 2)
58             cs8900_write (dev,PP_IA + i,dev->dev_addr[i]
59                 | (dev->dev_addr[i + 1] << 8));
60         printk (" , no eeprom ");
61     }
62     else if( result == -EFAULT)

```

```

63     {
64         printk (" , eeprom (invalid config block)");
65     }
66     else
67     {
68         printk (" , eeprom ok");
69     }
70     printk (" , addr:");
71     for (i = 0; i < ETH_ALEN; i += 2)
72     {
73         u16 mac = cs8900_read (dev,PP_IA + i);
74         printk ("%c%02X:%02X", (i==0)?' ':', mac & 0xff, (mac >> 8));
75     }
76     printk ("\n");
77     return (0);
78 }

```

接下来对上述代码进行逐行分析，第 1 行 `cs8900_t` 结构是专门定义的一个用于描述网络设备私有信息的结构，其结构包含的成员有：网络设备的统计信息结构成员（`stats`），传输数据长度成员（`txlen`），该网络设备包含字符设备的个数（`char_devnum`）成员以及自旋锁（`lock`）成员。第 9 行用 `ether_setup` 函数给网络设备填充以太网相关的值，如 MTU 值、地址长度、以太网类型等信息。第 10-15 行，用来初始化网络设备的方法，包括：`open`、`stop`、`hard_start_xmit` 等。由于我们的程序是基于 SMDK2410 开发板的，所以内核中会配置 `CONFIG_ARCH_SMDK2410` 这个选项，在以下关于这个选项的条件编译代码中，都会执行到它包含的代码，第 18-23 行，用来定义网络设备的 MAC 地址，也就是硬件地址（48 位长度）。第 32 行，定义网络设备的接口介质类型为 10BaseT\*（注 3）。第 33 行，将 `cs8900_t` 结构对象的地址赋给网络设备的私有数据成员 `priv`。第 34 行，以 `cs8900_t` 结构对象的自选锁成员 `lock` 为参数，传递给宏 `spin_lock_init`，用于动态初始化自选锁。第 36-37 行，分别定义以太网设备 I/O 地址和中断号。第 39-41 行，用来检测网络设备 I/O 地址空间是否可用。第 42 行，如果该网络设备 I/O 地址可用，则注册这块区域。第 44-46 行，用于检查 EISA\*（注 4）是否正确。第 48-51 行，用于验证 CS8900A 芯片的版本号是否正确。第 53 行，用来设置网路设备的中断号。第 54-69 行，用来确定网络设备是否有 EEPROM\*（注 5）。由于我们的系统中没有 EEPROM 设备和其他的配置块，所以需要手动配置 MAC 地址。第 71-75 行，用来读取 MAC 地址。第 77 行，一切都正确的情况下返回 0，表示网络设备接口初始化完成。

\*注 3：10BaseT-----原始 IEEE802.3 标准的一部分，10BaseT 是 10Mb/s 基带以太网规范，它使用两对双绞电缆(3 类、4 类或 5 类)，一对用于发送数据另一对用于接收数据。10BaseT 每段的距离限制大约为 100 米。

\*注 4：EISA: Extended Industry Standard Architecture，即扩充的工业标准体系结构，一种总线标准，用于连接插入到 PC 主板上的插件，如显示卡、内置调制解调器、声霸卡、驱动控制器以及支持其他外设的插卡。EISA 有 32 位的数据通路，并且使用的连接器可以连接 ISA 插件，但 EISA 卡只兼容 EISA 系统。EISA 操作频率和数据吞吐量都远远高于 ISA 总线。

\*注 5：EEPROM: Electrically Erasable Programmable ROM，即电可擦可编程只读存储器，为一种将资料写入后即使在电源关闭的情况下，也可以保留一段相当长的时间，且写入资料时不需要另外提高电压，只要写入某一些句柄，就可以把资料写入内存中了。

到这里为止，完整讲述了网络设备接口的初始化函数 `cs8900_probe`，那么有人会问谁会调用 `cs8900_probe` 函数呢？其实前面已经说过，当在内核启动或加载网络驱动模块的时候，就会调用这个初始化函数，这里我们是以模块加载形式来调用的，它的加载函数为 `cs8900_init`，具体实现代码如下：

```
static int __init cs8900_init (void)
{
    strcpy(cs8900_dev.name, "eth%d");
    return (register_netdev (&cs8900_dev));
}
```

该加载函数实现的功能是：首先给网络设备取名，然后传递参数 `cs8900_dev` 的地址到网络设备注册函数 `register_netdev`，从而完成注册网络设备的任务。

介绍了网络设备的加载，顺便看一下它的卸载函数，它的作用与加载函数相反，实现代码如下：

```
static void __exit cs8900_cleanup (void)
{
    cs8900_t *priv = (cs8900_t *) cs8900_dev.priv;
    if( priv->char_devnum)
    {
        unregister_chrdev(priv->char_devnum,"cs8900_eeprom");
    }
    release_mem_region (cs8900_dev.base_addr,16);
    unregister_netdev (&cs8900_dev);
}
```

该卸载函数实现的功能是：首先判断网络设备中是否定义了字符设备 `EEPROM`，如果定义了，则首先卸载它。实际在我们的系统中没有使用 `EEPROM`，所以就不用执行注销字符设备。然后，释放网络设备 I/O 地址空间。最后，注销网络设备 `cs8900_dev`。

### 8.2.3.2 open 和 stop 方法

网络设备的 `open` 方法，就是激活网络接口，使它能接收来自网络的数据并且传递到网络协议栈的上层，也可以将数据发送到网络上。`cs8900_dev` 设备的 `open` 方法实现代码如下：

```
1 static int cs8900_open (struct net_device *dev)
2 {
3     int result;
4
5     #if defined(CONFIG_ARCH_SMDK2410)
6         set_irq_type(dev->irq, IRQT_RISING);
7         /* enable the ethernet controller */
8         cs8900_set (dev,PP_RxCFGRxOKiE | BufferCRC | CRCerroriE
9             | RuntiE | ExtradataiE);
10        cs8900_set (dev,PP_RxCTL,RxOKA | IndividualA | BroadcastA);
11        cs8900_set (dev,PP_TxCFGRxOKiE | Out_of_windowiE | JabberiE);
12        cs8900_set (dev,PP_BufCFGRdy4TxIE | RxMissiE | TxUnderruniE
```

```

13     | TxColOvfIE | MissOvfloiE);
14     cs8900_set (dev,PP_LineCTL,SerRxON | SerTxON);
15     cs8900_set (dev,PP_BusCTL,EnableRQ);
16 #ifdef FULL_DUPLEX
17     cs8900_set (dev,PP_TestCTL,FDX);
18 #endif /* #ifdef FULL_DUPLEX */
19     udelay(200);
20     /* install interrupt handler */
21     if ((result = request_irq (dev->irq, &cs8900_interrupt, 0,
22         dev->name, dev)) < 0) {
23         return (result);
24     }
25 #else
26     /* install interrupt handler */
27     if ((result = request_irq (dev->irq, &cs8900_interrupt, 0,
28         dev->name, dev)) < 0) {
29         return (result);
30     }
31     set_irq_type(dev->irq, IRQT_RISING);
32     /* enable the ethernet controller */
33     cs8900_set (dev,PP_RxCFG,RxOKiE | BufferCRC | CRCerroriE
34         | RuntiE | ExtradataiE);
35     cs8900_set (dev,PP_RxCTL,RxOKA | IndividualA | BroadcastA);
36     cs8900_set (dev,PP_TxCFG,TxOKiE | Out_of_windowiE | JabberiE);
37     cs8900_set (dev,PP_BufCFG,Rdy4TxIE | RxMissiE | TxUnderruniE
38         | TxColOvfIE | MissOvfloiE);
39     cs8900_set (dev,PP_LineCTL,SerRxON | SerTxON);
40     cs8900_set (dev,PP_BusCTL,EnableRQ);
41 #ifdef FULL_DUPLEX
42     cs8900_set (dev,PP_TestCTL,FDX);
43 #endif /* #ifdef FULL_DUPLEX */
44 #endif /* #if defined(CONFIG_ARCH_SMDK2410) */
45     /* start the queue */
46     netif_start_queue (dev);
47     return (0);
48 }

```

现在来分析上述代码，第 6 行，设定网络设备的中断触发类型为上升沿触发。第 8-17 行，配置以太网控制寄存器，包括：接收总线配置寄存器 PP\_RxCFG，接收控制寄存器 PP\_RxCTL，发送配置寄存器 PP\_TxCFG，缓冲配置寄存器 PP\_BufCFG，线控制寄存器 PP\_LineCTL，总线控制寄存器 PP\_BusCTL 和测试控制寄存器 PP\_TestCTL，分别介绍一下这几个寄存器的作用：

- PP\_RxCFG：用来确定如何传输帧到主机和哪种类型帧被触发中断。
- PP\_RxCTL：位 8，C，D，和 E 定义接收什么样的帧。位 6，7，9，A 和 B 配置目的地址过滤器。

- l PP\_TxCFG: 用来配置发送数据相关的中断是否可用。
- l PP\_BufCFG: 用来配置总线缓冲相关的中断是否可用。
- l PP\_LineCTL: 用来配置 MAC 引擎和物理接口。
- l PP\_TestCTL: 用来控制 ISA 总线接口操作。

继续分析上述代码，第 21-24 行，注册网络设备的中断服务程序 cs8900\_interrupt，后面会详细讲述这个中断服务程序。如果中断服务程序注册成功将执行第 46 行，用来启动一个网络接口队列，最后返回一个 0 代表网络设备被正确打开。

接下来讲述 stop 方法，即停止网络设备，它的作用与 open 方法相反，具体实现代码如下：

```

1 static int cs8900_stop (struct net_device *dev)
2 {
3     /* disable ethernet controller */
4     cs8900_write (dev,PP_BusCTL,0);
5     cs8900_write (dev,PP_TestCTL,0);
6     cs8900_write (dev,PP_SelfCTL,0);
7     cs8900_write (dev,PP_LineCTL,0);
8     cs8900_write (dev,PP_BufCFG,0);
9     cs8900_write (dev,PP_TxCFG,0);
10    cs8900_write (dev,PP_RxCTL,0);
11    cs8900_write (dev,PP_RxCFG,0);
12    /* uninstall interrupt handler */
13    free_irq (dev->irq,dev);
14    /* stop the queue */
15    netif_stop_queue (dev);
16    return (0);
17 }
```

第 4-11 行，用来禁止以太网控制器各相应的寄存器。第 13 行，释放网络设备的中断服务程序。第 15 行，停止网络设备接口队列。最后返回 0 表示完成停止设备。

### 8.2.3.3 数据发送

网络设备数据发送是由 net\_device 结构中的 hard\_start\_xmit 方法实现的，所有的网络设备驱动程序都必须有这个发送方法。在驱动程序层次中，发送和接收数据都是通过系统低层对硬件的读写来完成的。根据初始化函数 cs8900\_probe 定义网络设备的发送实现函数为 cs8900\_send\_start，首先来看一下 cs8900\_send\_start 函数的实现代码：

```

1 static int cs8900_send_start (struct sk_buff *skb,struct net_device *dev)
2 {
3     cs8900_t *priv = (cs8900_t *) dev->priv;
4     u16 status;
5
6     spin_lock_irq(&priv->lock);
7     netif_stop_queue (dev);
8     cs8900_write (dev,PP_TxCMD,TxStart (After5));
9     cs8900_write (dev,PP_TxLength,skb->len);
```



```

10     status = cs8900_read (dev,PP_BusST);
11     if ((status & TxBidErr)) {
12         spin_unlock_irq(&priv->lock);
13         priv->stats.tx_errors++;
14         priv->stats.tx_aborted_errors++;
15         priv->txlen = 0;
16         return (1);
17     }
18     if (!(status & Rdy4TxNOW)) {
19         spin_unlock_irq(&priv->lock);
20         priv->stats.tx_errors++;
21         priv->txlen = 0;
22         return (1);
23     }
24     cs8900_frame_write (dev,skb);
25     spin_unlock_irq(&priv->lock);
26     dev->trans_start = jiffies;
27     dev_kfree_skb (skb);
28     priv->txlen = skb->len;
29     return (0);
30 }

```

下面来分析上述代码，在系统调用驱动程序的 `hard_start_xmit` 方法时，发送的数据放在一个 `sk_buff` 结构中，也就是前面讲过的套接字缓冲结构。第 6 行，获得一个禁止中断的自旋锁。第 7 行，关闭网络接口队列。第 8-9 行，主机在发送数据前写 `PP_TxCMD` 这个寄存器，利用这个命令告诉 CS8900A 主机有一个数据帧要传输，并且告诉如何传输这个帧。此外，写 `PP_TxLength` 寄存器用来告诉将要传输数据帧的长度。第 10 行，读取 `PP_BusST` 寄存器来获得当前传输操作的状态。第 11-17 行，数据传输失败由于数据帧的大小不符合要求，比如数据帧大于 1518 字节。于是释放自旋锁，设置数据传输长度为 0，返回 1。第 18-23，数据传输失败由于传输缓冲空间没有准备好，此时释放自旋锁，设置数据传输长度为 0，返回 1。第 24-29 行，如果传输数据的状态都正常，此时将 `skb` 指向的数据帧发送给 CS8900A。然后释放自旋锁并且释放 `skb` 指向的内存空间，最后返回 0 表示传输数据正确。

### 8.2.3.4 数据接收

数据包的接收实现方式不同于数据发送利用 `hard_start_xmit` 方法实现，而它是通过中断机制来完成的。当有数据到达时产生中断信号，网络设备驱动功能层调用中断处理程序（即数据包接收程序）来处理数据包的接收，随后网络协议接口层调用 `netif_rx` 函数把接收到的数据包传输到网络协议的上层进行处理。CS8900A 的中断服务程序在讲述 `open` 方法时已经提到，它是利用 `cs8900_interrupt` 函数来实现的，首先来看一下它的具体实现代码：

```

1  static irqreturn_t cs8900_interrupt (int irq,void *id,struct pt_regs *regs)
2  {
3      struct net_device *dev = (struct net_device *) id;
4      cs8900_t *priv;
5      volatile u16 status;

```

```

6      irqreturn_t handled = 0;
7
8      if (dev->priv == NULL) {
9          printk (KERN_WARNING "%s: irq %d for unknown
device.\n",dev->name,irq);
10         return 0;
11     }
12     priv = (cs8900_t *) dev->priv;
13     while ((status = cs8900_read (dev, PP_ISQ))) {
14         handled = 1;
15         switch (RegNum (status)) {
16             case RxEvent:
17                 cs8900_receive (dev);
18                 break;
19             case TxEvent:
20                 priv->stats.collisions += ColCount (cs8900_read (dev,PP_TxCOL));
21                 if (!(RegContent (status) & TxOK)) {
22                     priv->stats.tx_errors++;
23                     if ((RegContent (status) & Out_of_window))
priv->stats.tx_window_errors++;
24                     if ((RegContent (status) & Jabber)) priv->stats.tx_aborted_errors++;
25                     break;
26                 } else if (priv->txlen) {
27                     priv->stats.tx_packets++;
28                     priv->stats.tx_bytes += priv->txlen;
29                 }
30                 priv->txlen = 0;
31                 netif_wake_queue (dev);
32                 break;
33             case BufEvent:
34                 if ((RegContent (status) & RxMiss)) {
35                     u16 missed = MissCount (cs8900_read (dev,PP_RxMISS));
36                     priv->stats.rx_errors += missed;
37                     priv->stats.rx_missed_errors += missed;
38                 }
39                 if ((RegContent (status) & TxUnderrun)) {
40                     priv->stats.tx_errors++;
41                     priv->stats.tx_fifo_errors++;
42                     priv->txlen = 0;
43                     netif_wake_queue (dev);
44                 }
45                 break;
46             case TxCOL:
47                 priv->stats.collisions += ColCount (cs8900_read (dev,PP_TxCOL));

```

```

48         break;
49     case RxMISS:
50         status = MissCount (cs8900_read (dev,PP_RxMISS));
51         priv->stats.rx_errors += status;
52         priv->stats.rx_missed_errors += status;
53         break;
54     }
55 }
56 return IRQ_RETVAL(handled);
57 }

```

接下来分析一下上述代码，当触发一个中断时，首先通过读取 PP\_ISQ 寄存器来判断产生中断的类型，这是在第 13 行代码实现。第 15-54 行，根据获得的中断类型来执行相应的处理，其中有五种中断类型，按优先级顺序分别为：RxEvent，TxEvent，BufEvent，RxMISS，TxCOL。每次主控制器读 ISQ（Interrupt Status Queue）寄存器，相应寄存器中产生中断的位将清零，下一个中断报告将会自动移到中断队列最前面。数据接收的核心功能是在 RxEvent 中断类型的处理程序中由 cs8900\_receive 函数来完成。这里专门讲述一下 cs8900\_receive 函数，它实现了网络数据接收的核心功能。cs8900\_receive 函数的实现代码如下：

```

1  static void cs8900_receive (struct net_device *dev)
2  {
3      cs8900_t *priv = (cs8900_t *) dev->priv;
4      struct sk_buff *skb;
5      u16 status,length;
6
7      status = cs8900_read (dev,PP_RxStatus);
8      length = cs8900_read (dev,PP_RxLength);
9      if (!(status & RxOK)) {
10         priv->stats.rx_errors++;
11         if ((status & (Runt | Extradata))) priv->stats.rx_length_errors++;
12         if ((status & CRCError)) priv->stats.rx_crc_errors++;
13         return;
14     }
15     if ((skb = dev_alloc_skb (length + 4)) == NULL) {
16         priv->stats.rx_dropped++;
17         return;
18     }
19     skb->dev = dev;
20     skb_reserve (skb,2);
21     cs8900_frame_read (dev,skb,length);
22 #ifdef FULL_DUPLEX
23     dump_packet (dev,skb,"recv");
24 #endif /* #ifdef FULL_DUPLEX */
25     skb->protocol = eth_type_trans (skb,dev);
26     netif_rx (skb);
27     dev->last_rx = jiffies;

```

```
28     priv->stats.rx_packets++;
29     priv->stats.rx_bytes += length;
30 }
```

接下来分析上述代码，第 4 行，定义一个 `sk_buff`（套接字）结构的指针 `skb`，用来存放要传递的数据信息。第 7-14 行，首先读取接收数据寄存器的状态是否正常，如果读取的状态不正确，则将这些错误信息赋给相应的统计变量，最后返回。第 15-18 行，给 `skb` 指针变量申请内存空间，如果没有申请成功则返回。第 21 行，将 CS8900A 的数据帧读取到 `skb` 指向的内存中。第 25 行，确定传输数据协议类型，比如 802.3、802.2 协议等。第 26 行，调用 `netif_rx()` 把 `skb` 中存放的数据传送给协议层，它是网络数据接收中必须使用的函数。

## 8.3 本章小结

本章讲述了另一类设备驱动——网络设备驱动。首先介绍了网络设备驱动中的两个重要数据结构，即 `net_device` 和 `sk_buffer` 结构，然后介绍了常见的网络术语，包括基本的网络协议以及以太网。最后重点讲述了 CS8900A 以太网适配器驱动的开发，这里不仅讲述了网络驱动程序的概要设计，并且分析了 CS8900A 以太网适配器驱动程序的重要实现代码。通过本章的学习，使读者学会了另一类 Linux 驱动设备的开发方法，读者可以看出网络设备驱动程序开发与字符设备和块设备驱动开发的不同之处。下一章将开始讲述本书的第三部分，即 Linux 系统下的 GUI 开发——Qt。

## 8.4 常见问题

1. 网络设备驱动实现时经常会用到哪两个数据结构？

参考答案： `net_device` 结构和 `sk_buff` 结构是网络驱动实现时最重要的两个数据结构，`net_device` 结构是网络设备驱动的核心，该结构用来定义网络设备。`sk_buff` 结构是套接字缓冲结构，该结构经常用于定义网络协议之间传输的数据。

2. 网络设备驱动程序中数据接收是如何实现的？

参考答案：数据接收的实现是通过中断机制来完成的。当有数据到达时产生中断信号，网络设备驱动功能层调用中断处理程序（即数据包接收程序）来处理数据包的接收，随后网络协议接口层调用 `netif_rx` 函数把接收到的数据包传输到网络协议的上层进行处理。

## 第三部分 Qt GUI 开发

这部分介绍嵌入式 Linux 下常用的 GUI 编程——Qt 编程。Qt 是跨平台的 C++ GUI 应用标准框架，它不仅能运行在 Linux/Unix 上，而且还可以运行在微软的 Windows 和 Mac OS 等。该部分由四章内容组成，首先第 9 章介绍了 Qt 的概要知识，包括 Linux 桌面 GUI 系统，Qt/X11，Qtopia Core 等，通过对本章的学习使读者对 Qt 及其相关知识有个大概了解。紧接着第 10 章讲述了 Qt/X11 的安装以及介绍了它的应用实例，通过对本章的学习使读者对 Qt/X11 有更多地了解，不但会安装 Qt/X11 开发环境，而且可以开发基本的 Qt/X11 程序。第 11 章讲述 Qt 核心技术，重点是通过剖析 Qt 的源代码来深入的学习 Qt 的对象模型及信号和槽机制的实现，同时也详细介绍了 Qt 窗口系统以及国际化等常用知识，通过对本章的学习使读者对 Qt 的核心技术有更深入的了解，也能为以后实际工作中的 Qt 开发打下良好的基础。最后第 12 章讲述 Qtopia Core，包括 Qtopia Core 的安装，Frame Buffer 和 qvfb，以及轻量级的窗口系统，移植 Qt/X11 程序到 Qtopia Core，进程间通信等，这一章里我们重点讲述 Qtopia Core 和 Qt/X11 的一些不同之处，使读者在熟悉了 Qt/X11 的基础上能够很快过渡到 Qtopia Core 开发。

总之，目前 Qt GUI 开发在嵌入式设备中应用越来越广，所以这方面的人才需求也在每年递增，相信会有更多的人加入到这个行业中来，同时希望读者能从这部分内容中学到 Qt 开发知识的基础知识，并能很快应用于实际开发中。

# 第 9 章 Qt 概述

本章学习目标：

- l 了解 X Window 构架
- l 了解 Qt 在 Linux GUI 系统中的作用
- l 了解 Qt/X11 和 Qtopia Core

## 9.1 Linux 下 GUI 介绍

术语 GUI 是 Graphical User Interface 的简写，即指图形用户接口。现代操作系统一般都提供图形化的操作界面系统，这种 GUI 系统一般由视窗、图标、菜单、对话框及其他一些可视特征组成，它允许终端用户可以方便地利用鼠标和键盘来操作电脑。GUI 的概念是 70 年代由施乐公司帕洛阿尔托研究中心提出的，1984 年苹果的 Macintosh 电脑则是首例成功使用 GUI 并用于商业用途的产品。之后各种 GUI 系统发展迅速，包括后来居上的微软的 Windows 系列，以及广泛应用于类 Unix 系统上的 X Window 系统（常简称为 X11 或者 X）。我们先来了解一下 Linux 下的桌面 GUI 系统和嵌入式 GUI 系统，以及 Qt 等开发包在 Linux GUI 系统中所起到的作用。

### 9.1.1 Linux 桌面 GUI 系统

与多数的 Unix 系统类似，Linux 桌面 GUI 系统同样是基于 X Window 协议构架来实现的。早在 1994 年第一个 Linux kernel 1.0 的版本当中，就已经有了 XFree86(X Window 的一个实现)的支持了。目前各种 Linux 的发行版本基本都采用 X Window 的构架，而在 X Window 的上层则开发了多种高级图形库、开发包及整合的桌面环境可供选择。常见的 Linux GUI 系统架构如图 9.1 所示：

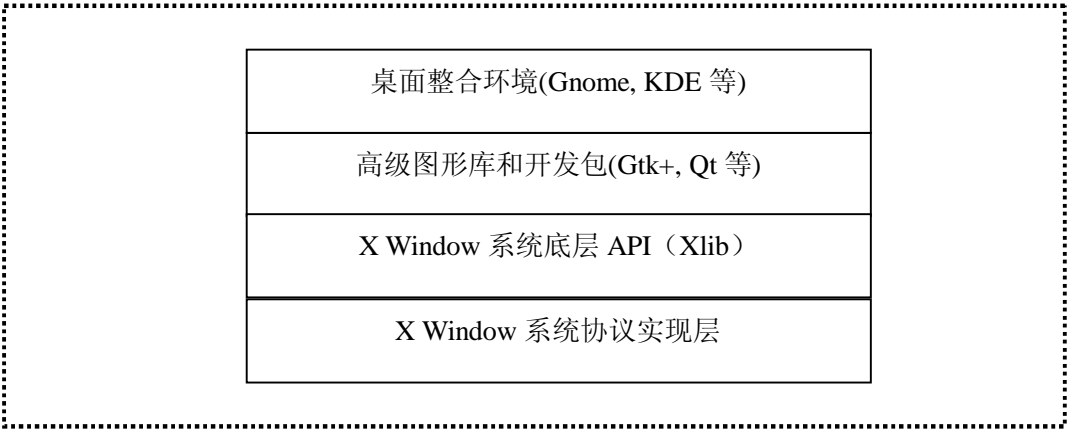


图 9.1 常见的 Linux 桌面 GUI 系统架

从图中我们可以看到，各种流行的桌面环境和开发包实际上都是在 X Window 的基础上开发的，在所有的类 Unix 系统中，X Window 几乎完全占据统治地位。

### 9.1.1.1 X Window 系统

X 起源于 1984 年麻省理工学院(MIT)计算机科学研究所和 Athena 计划的共同研究。当时 MIT 的 Bob Scheifler 正在发展分布式系统，同一时间 DEC 公司的 Jim Gettys 也在麻省理工学院做 Athena 计划的一部分，两个计划都需要一个相同的东西——一套在 Unix 机器上的优良的视窗系统。因此他们开始合作，从斯坦福大学得到了一套叫做 W 的实验性视窗系统，并在基于 W 视窗系统的基础上开始发展，当发展到了足以和原先系统有明显区别时，他们把这个新系统叫做 X。X 是第一个真正的与硬件和制造商无关的窗口系统环境。

X 在之后的几年内迅速发展，1987 年 9 月第 11 版本即 X11 发行，并取得明显成功，成为早期的较大规模开源项目之一。后来 X 发行的版本都被称为 X11 的某个版本，如 X11R2、X11R6 等。这也是 X Window 系统常被简称为 X11 的原因。

X Window 系统架构基于 C/S 模型，即客户机-服务器模型，主要由 X Server 和 X Client 通过 X Protocol 在网络上通信完成应用任务。当然，在很多情况下，X Server 和 X Client 运行在同一主机上：启动 Linux 登录到一个图形界面，这时服务器和客户端同时并发的在同一主机运行，这时它们之间的通信只需利用本地的一些通信方式即可。

X Window 系统架构图如图 9.2 所示：

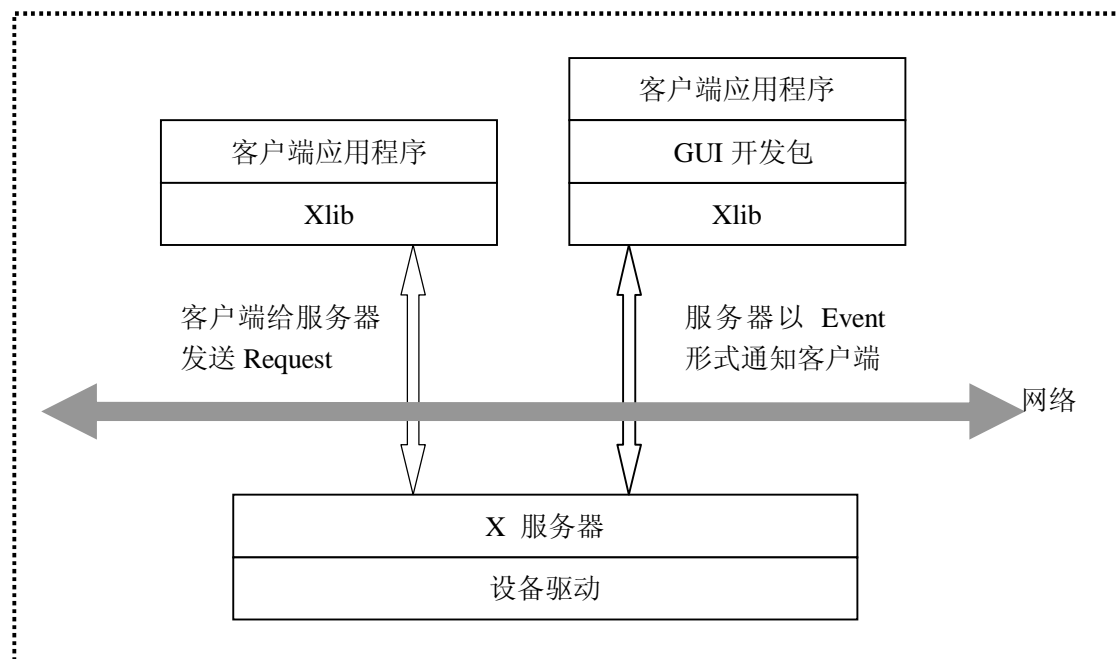


图 9.2 X Window 系统架构图

X Window 为 GUI 环境提供了基本的框架：在屏幕上绘图和移动窗口，以及与鼠标和键盘的互动等。X Server 用来控制显示器和输入设备，它可以建立视窗，在视窗中画图形和文字，响应 Client 程序的请求 (Request)，但它不会自己动作，只有在 Client 程序提出需求后才完成动作。而 X Client 就是 X 中的应用程序，一个 X Client 若想要运行，必须打开一个显示器，连接一个 X Server，然后通过 X Server 之间的通信来完成所有的工作。X Client 主要是完成应用程序计算处理的部分，并不接受用户的输入信息，输入信息都是输入给 X Server，然后由 X Server 以 Event 的形式传递给 X Client。需要注意的是，X 的“客户端”和“服务器”与普通意义上的 C/S 模型中的客户端和服务端有所不同，如果在网络上使用，比如使用 telnet 远程登录某台计算机，并在本地显示所登录的机器的情况，则“服务器”是使用者本地的显示，而“客户端”反而是远程的机器。

X Protocol 是 X Server 与 X Client 之间的通信协议。X Client 一般通过调用底层库 Xlib

的接口来和 X Server 进行通信，Xlib 提供一些基本的函数如连接某个服务器、绘制窗口和响应事件等，它在 X Window System 中的角色，就好比 Windows APIs (或者说 Windows SDK) 在 Microsoft Windows 中的角色一样，是最接近窗口系统的程序设计接口。Xlib 中的函数最终通过底层通信协议来实现，所采用的通信协议被定义为一种异步的双向协议，任何提供字节流通信的方式都可以使用，可以是 IPC，也可以是 TCP/IP 等。

X 并不内置于操作系统，它只是比用户层次稍高一些，在系统中它也是一个相对独立的组件，上层应用稍作移植就可以应用于各种操作系统。X 的另外一个优点在于稳定性，在 X Server 上进行工作时，如果程序异常中断，只会影响到视窗系统，不会造成机器的损坏或操作系统核心的破坏。

### 9.1.1.2 GNOME/Gtk+和 KDE/Qt

尽管 Linux 桌面 GUI 系统一般都基于 X Window 系统，但 X Window 本身并不是一个直接的图形操作环境，它只是作为图形环境与 UNIX 系统内核沟通的中间桥梁，任何厂商都可以在 X Window 基础上开发出不同的 GUI 图形环境。上个世纪九十年代中期，以开源模式推进的 Linux 还没有找到一个可靠并且免费的图形界面——IBM, Sun 等大厂商开发的 Motif/CDE 桌面环境等价格都非常昂贵，而一些小型的免费系统还远远不够完善。

在这时，就读于图宾根大学德国人 Matthias Ettrich 发起了 KDE(K Desktop Environment) 计划，来开发一个易于使用及人性化的桌面系统，并选择了当时新推出的功能强大的 Qt 作为 GUI 开发包。很快地他和其它志愿开发人员于 1997 年初发布了第一个较大规模的 KDE 版本。

由于 KDE 本身采用 GPL(GNU 通用公共许可证)发布，而作为 KDE 底层基础的 Qt 却是不遵循 GPL 的商业软件，于是一大批自由程序员对 KDE 项目的决定深为不满，他们认为利用非自由软件开发违背了 GPL 的精神，于是在墨西哥程序员 Miguel De Icaza 的组织下发起了 GNOME(GNU Network Object Enviroment)计划来替代 KDE,并选择了使用 LGPL 的 Gtk+ 来作为 Qt 开发包的替代，担当 GNOME 桌面的基础。

KDE 和 GNOME 在之后多年的互相竞争中迅速发展，逐渐成为了 Linux 下的桌面环境的两大阵营。目前 GNOME/Gtk+ 吸引的公司比较多，而 KDE/Qt 则在 Office/嵌入式 环境中先走一步。

KDE 和 GNOME 在设计上稍有差别：一般的说，KDE 桌面环境更加倾向于易用性方面，提供了很多图形化的方式来进行各种操作和配置，而 GNOME 桌面环境设计得比较简捷，速度稍快。GNOME 和 KDE 的桌面截图如下所示：



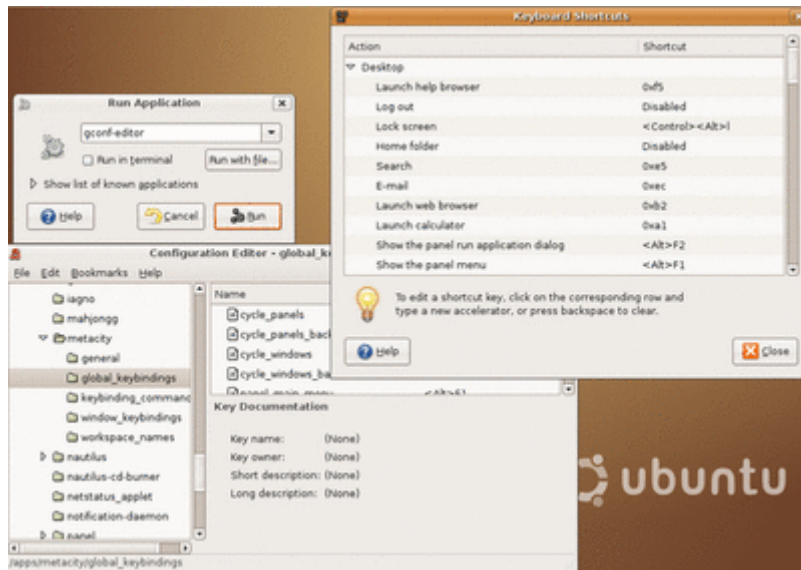


图 9.3 GNOME 桌面示例

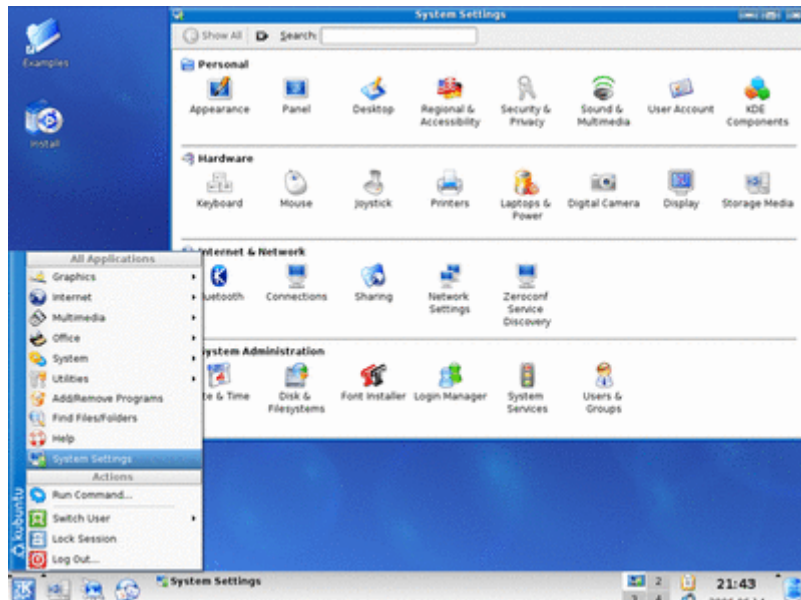


图 9.4 KDE 桌面示例

尽管 KDE 和 GNOME 之间竞争激烈，它们之间却并非水火不相容的关系。我们可以在 Linux 中同时安装这两种桌面系统，并可以方便的互相切换。而且从 2003 年以来，KDE 与 GNOME 阵营开始逐渐相互支持对方的程序——只要你在 KDE 环境中安装 GTK 库，便可以运行 GNOME 的程序，反之亦然。经过几年多的努力，KDE 和 GNOME 都已经实现高度的互操作性，两大平台的程序都是完全共享的。

作为 GNOME 所使用的开发包，Gtk+最初只是 GIMP(GNU Image Manipulation Program, GNU 图像处理程序)的专用开发库，后来随着 GNOME 的发展而逐渐成为 Linux 下开发图形界面的应用程序的主流开发工具之一。Gtk+是自由软件，并且是 GNU 工程的一部分，采用 LGPL 协议发布。

Gtk 是在 GDK (GIMP Drawing Kit) 和 gdk-pixbuf 的基础上建立起来的，GDK 基本上是对访问窗口的底层函数 (在 X 窗口系统中是 Xlib) 的一层封装，而 gdk-pixbuf 则是一个用于客户端图像处理的库。一般的，我们用 Gtk 代表软件包和共享库，用 Gtk+代表 Gtk 的

图形构件集。Gtk+图形库使用一系列称为“构件”的对象来创建应用程序的图形用户接口。它提供了许多常用的构件如窗口、标签、命令按钮、开关按钮、检查按钮、无线按钮、框架、列表框、组合框、树、列表视图、笔记本、状态条等。可以用它们来构造非常丰富的用户界面。另外，Gtk+也提供了一些独具特色的部件，譬如不包含标签而包含子部件的按钮，几乎可以在这样的按钮上放置任何窗口部件，设计者可以根据自己的需求来自己设计或组合。

在用 Gtk+开发 GNOME 的过程中，由于实际需要，在上面的构件基础上，又开发了一些新构件。一般把这些构件称为 GNOME 构件(与 Gtk+构件相对应)。这些构件都是 Gtk+构件库的补充，它们提供了许多 Gtk+构件没有的功能。从本质上来说，Gtk+构件和 GNOME 构件是完全类似的东西。

Gtk+使用 C 语言开发，但是其设计者使用了面向对象技术，基于类和回调函数（指向函数的指针）的思想来实现。如果想使用 C++ 来开发 GTK 应用程序，我们还可以利用一个叫 gtkmm 的绑定。

Qt 是由挪威 TrollTech 公司于 1995 年推出的一个跨平台的 C++ 图形用户界面库。基本上，Qt 同 X Window 上的 Motif、Open Look、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 Qt 具有优良的跨平台特性（支持 Windows、Linux、各种 UNIX、OS390 和 QNX 等）、面向对象机制以及丰富的 API，同时也可支持 2D/3D 渲染和 OpenGL API 等。如前文所述，正是这些强大的功能使得当时 Matthias Ettrich 发起了 KDE 项目时，理所当然地选择了当时新推出 Qt 作为 GUI 开发包。

我们将在 8.2 节详细介绍 Qt 在桌面系统中的应用。

## 9.1.2 嵌入式 Linux 下的 GUI 系统

由于嵌入式系统中硬件条件的限制，在嵌入式 Linux 系统中庞大臃肿的 X Window 不太适合，我们需要一个高性能、轻量级的 GUI 系统。一般的说，适合于嵌入式 Linux 系统的 GUI 应该具有下面的一些特点<sup>[14]</sup>：

- 体积小，占用较少的 Flash 和 RAM。安装 GUI 系统的时候可以根据实际的需求对 GUI 系统进行方便的裁剪和精简，以减少安装所需要的存储空间；在系统运行的时候应占用尽可能少的 RAM。
- 耗用系统资源尤其是 CPU 的资源较少，在硬件性能受限的条件下能达到相对较快的系统响应速度，同时减小 CPU 的功耗，以达到节电的效果。
- 系统独立，能适用于不同的硬件。

目前常见的面向嵌入式 Linux 的 GUI 系统主要有 Qtopia Core(Qt/Embedded), Microwindows(Nano-X Window), Tiny X, 以及国内的 MiniGUI 等。

MicroWindows（2005 年更名为 Nano-X Window, <http://microwindows.censoft.com/>）是一个基于典型客户/服务器体系结构的 GUI 系统，其主要特色在于提供了类似 X 的客户/服务器体系结构并提供了相对完善的图形功能。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 项目的进展一直很慢，目前已基本停滞。另外它的图形引擎中也存在不少低效算法。2005 年 1 月由于其名字与微软的 Windows 商标相冲突，MicroWindows 更名为 Nano-X Window，但之后也不再有新的版本发布。

Tiny X(<http://www.xfree86.org/>)实际上是 XFree86 Project 的一部分，由 SuSE 公司所赞助，XFree86 Project 的核心成员之一 Keith Packard 开发，其目标是可以小内存或几乎无内存的情况下良好运行。目前 Tiny X 是 XFree86 自带的编译模式之一，只要通过修改编译

选项，就能编译生成 Tiny X。Tiny X 在 XFree86 的基础上精简了不少东西，在 x86 CPU 中体积可以减小到 1M 以下，以适用于嵌入式环境之中。Tiny X 的最大优点在于可以方便的移植桌面版本的基于 X 的软件到嵌入式系统中，不过这个优点有时也会变成缺点，因为从桌面版本移植过去的软件相对于嵌入式环境来说，一般体积都过大，需要一定的简化，这种简化有时还不如开发新的程序来得方便。

MiniGUI (<http://www.minigui.org/>) 是原清华大学教师魏永明先生所主持开发的一个自由软件项目，旨在为基于 Linux 的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。MiniGUI 于 1999 年初遵循 GPL 条款发布了第一个版本，目前国内已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、便携式多媒体播放机、查询终端等产品和领域，可在 Linux/uClinux、VxWorks、uC/OS-II、pSOS、ThreadX、Nucleus 等操作系统以及 Win32 平台上运行，并能支持 Intel x86、ARM(ARM7/ARM9/StrongARM/xScale)、PowerPC、MIPS、M68K (DragonBall/ColdFire) 等硬件平台。MiniGUI 的开发建立在比较成熟的图形引擎如 Sglib 和 LibGGI 之上，主要着重于窗口系统、图形接口的开发，面向中低端的嵌入式产品市场。另外由于 MiniGUI 是中国人自己开发的 GUI 系统，它对于中文的支持非常好。

Qtopia Core 是 TrollTech (<http://www.trolltech.com/>) 发布的面向嵌入式系统的 Qt 版本。它的前身是 Qt/Embedded (常简称为 Qt/E)。与桌面版本 Qt/X11 不同的是，Qtopia Core 直接取代了 X Server 及 X Library 等角色，仅采用 Framebuffer 作为底层图形接口，从而大大减少了系统开销。因为 Qt 是 KDE 等项目使用的 GUI 支持库，所以有许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/E 版本上。Qtopia Core 延续了 Qt 在 X 上的强大功能，但相对消耗系统资源也比较多，一般用于手持式高端信息产品。我们将在 8.3 节详细介绍 Qtopia Core。

## 9.2 Qt/X11 介绍

上一节中我们以较大的篇幅介绍了 X Window 系统以及 GNOME/Gtk+ 和 KDE/Qt，目的是希望读者对 Qt 在整个 Linux GUI 系统中的位置和作用有一个全局意义上的了解。下面的两节我们来详细介绍一下 Qt 在 Unix/Linux 系统中的对应版本 Qt/X11 和嵌入式中的版本 Qtopia Core(Qt/E)。

### 9.2.1 Qt 的历史和 Qt/X11 的由来

早在 1991 年，Qt 的两位创始人 Haavard Nord (目前 Trolltech 公司的 CEO) 和 Eirik Chambe-Eng (目前 Trolltech 公司的总裁) 就已经开始着手开发 Qt。据说之所以命名为 Qt 是因为“Q”这个字母在 Haavard 的 Emacs (Linux/Unix 上的一个著名的编辑器) 的字体中看起来非常漂亮，而“t”则是受到当时的“Xt”(X Toolkit) 的启发，代表“toolkit”(工具包) 的意思。到 1994 年的时候 Qt 已基本成型，于是他们成立了 Trolltech 公司 (在中国现在一般成为“奇趣”)，开始用 Qt 来开发应用软件。次年的 5 月 Qt 0.90 版本公开发布，而在这之后由于作为开发 KDE 的基础而随着 KDE 的发展得以广泛应用。

作为一个跨平台的 GUI 开发包，Qt 发行了不同的版本以支持各种流行的操作系统，如微软的 Windows 系列，基于 X Window (简称为 X11 或 X) 的各种类 Unix 系统，苹果的 Mac OS X 及带有 Framebuffer 支持的嵌入式 Linux 系统等。其中支持基于 X11 的类 Unix 系统的版本我们常简称为 Qt/X11。

## 9.2.2 Qt/X11 的版权问题

Qt/X11 作为开发 KDE 的基础，因为 KDE 本身采用 GPL 协议，而 Qt 却在一定程度上是一种商业软件，其版权问题一直倍受关注。如 8.1.1.2 中所述，一些不满 Qt 版权的程序员开发了 GNOME，在后来 GNOME 和 KDE 之间的竞争中， Trolltech 公司被迫数次修改 Qt 的版权，直到 2000 年 10 月遵循 GPL 发布了 Qt 自由版用于 X11，彻底解决了屡被攻击的版权问题。因此目前的 Qt/X11 即 Qt 的自由版，在 Q 公共许可证(Q Public License)和 GPL 下是可以免费使用的，我们可以到 Trolltech 公司的官方网站 <http://www.trolltech.com> 去下载 Qt/X11 的安装包。

## 9.2.3 Qt/X11 及 Qt/Windows 的系统架构图对比

Qt 优良的跨平台特性是其得以广泛应用的重要原因之一。与 Java 图形界面的自成一体不同，Qt 通过调用操作系统底层与绘制图形界面相关的 API 来绘图，因而用 Qt 开发的应用程序在不同的操作系统下看起来会有所区别——它们都非常贴近各自操作系统的图形界面风格。

下面的图 9.5 给出了 Qt/X11 及 Qt/Windows 版本在各自对应的操作系统之上的系统架构图：

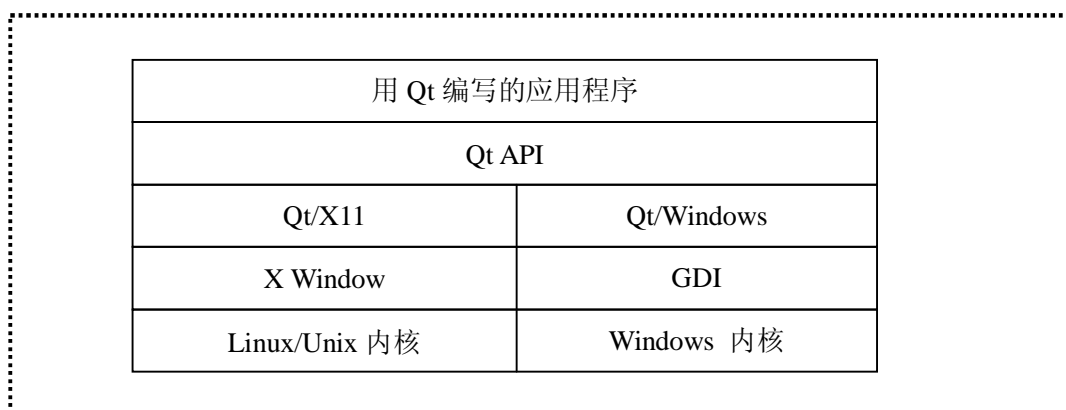


图 9.5 Qt/X11 及 Qt/Windows 在各自对应的操作系统之上的系统架构图对比

从图中可以看出，对于 Qt 针对不同操作系统发布的不同版本，它们所定义的提供给开发应用程序的开发人员的 API 其实是相同的，在应用程序开发人员看来，他们不必关心当前的操作系统是哪一种，只需要调用同一套 API 来实现他们的应用即可。用 Qt/X11 开发的应用只需要在 Windows 下重新用 Qt/Windows 版本编译，即可顺利运行于 Windows 系统中，这种优良的跨平台特性使得跨平台的应用开发显得非常方便。

## 9.2.4 Qt 的特性简介

下面我们简单的介绍一些 Qt 的特性。由于 Qt 的不同版本提供同样的 API，这些特性基本上不仅在 Qt/X11 的版本上存在，在其他版本上也都是存在的，对于在下一节我们要介绍的 Qtopia Core(Qt/E)也同样适用。

Linux/Unix 下的 Qt/X11 通过调用 Xlib 来与 X Server 通信，并在此基础上开发了一整套

窗口部件(Widget)。这些窗口部件组合起来可用于创建用户界面的可视元素，如按钮，菜单，滚动条，消息框和应用程序窗口等都是窗口部件的实例。所有的窗口部件本身也是容器，一个窗口部件可包含任意数量的子部件，子部件在父部件的区域内显示，没有父部件的部件是顶级部件（比如一个窗口）。父部件和子部件之间形成一种树形结构以便于维护——如果父部件被停用，隐藏或删除，同样的动作会递归地应用于它的所有子部件。Qt 提供的这套窗口部件库内容相当丰富，很多时候只需简单的继承已存在的 Qt 部件，稍作修改即可满足开发者的需求。

还在 1992 年的时候，Qt 的两位创始人之一 Eirik 想到了一种后来被称为“信号与槽”(Signals and Slots)的技术来方便对象之间的通信，这种简单而强大的技术在后来还被一些其他的开发包所采用。信号与槽的技术主要用来代替传统的不安全的回调技术。在我们所熟知的很多 GUI 工具包中，窗口部件(Widget)都有一个回调函数用于响应它们能触发的每个动作，这个回调函数通常是一个指向某个函数的指针。采用回调函数的方式不是类型安全的，因为我们无法确定处理函数是否使用了正确的参数来调用回调函数，有些时候这种错误可能导致整个进程的退出。在 Qt 中，信号和槽取代了这些凌乱的函数指针，信号和槽能携带任意数量和任意类型的参数，他们是类型完全安全的，并且信号和槽之间的耦合比较松散，可以使得我们编写这些通信程序更为简洁明了，是一种比较适合面向对象开发的通信机制。另外除了信号和槽之外，Qt 也提供传统的事件模型用来处理诸如鼠标点击、击键等动作，作为信号和槽的补充。

Qt 中还提供了一个叫 Qt 设计器(Qt Designer)的工具来帮助开发人员进行快速的应用程序开发。Qt 设计器提供了方便和强大的图形化布局能力，可使用户界面设计变得十分简单。你可以采用图形化的方式来定制你的程序界面，各种控件可以在 Qt 设计器方便的移动或者缩放。Qt 设计器还包含了一个代码编辑器，并支持信号和槽机制以使部件间能够进行有效的通信，你可以在合成的代码里面嵌入自己定制的槽来响应某种输入或者变化。

Qt 普遍使用 Unicode 并提供一系列的工具和机制来支持国际化。Qt 提供了采用 Unicode 实现的 QString 类来存储用户可见的文本，并提供了 Qt Linguist 等工具用来协助翻译。应用程序可以在代码中用 QObject::tr()方法来处理所有需要多语言翻译的文本，翻译工作者针对不同的语言提供不同的翻译后的资源包，Qt 程序在运行的时候即可装载不同的资源包并自动寻找对应的翻译之后的文本来进行显示，从而支持不同的语言，这种方法既能支持方便的语言切换，也使得开发人员和翻译人员的工作可以很好的区分开来。

## 9.3 Qtopia Core 介绍

Qtopia Core 是 Trolltech 公司在 Qt/Embedded 的基础上，于 2006 年 1 月推出一款基于嵌入式 Linux 的面向单一应用嵌入式产品的开发平台。Qtopia Core 采用与桌面版本同样的一套 API，但在其内部实现上作了很多调整和优化来适用硬件有所限制的嵌入式平台。

### 9.3.1 Qtopia Core 与 Qt/Embedded

Qt/Embedded 是 Trolltech 公司开发的面向嵌入式系统的 Qt 版本，最早在 2000 年 11 月发布了第一个 Qt/E 版本，而 Qtopia 是最初是构建于 Qt/E 之上的类似桌面系统的应用环境，包括了 PDA 和手机等掌上系统常见的功能如电话簿、图像浏览、Media 播放器、日程表等。最初 Qtopia 和 Qt/E 是不同的两套程序——Qt/E 是基础类库，Qtopia 是构建于 Qt/E 之上的一系列应用程序。但后来 Trolltech 调整了其产品策略，从版本 4.1 开始将 Qt/E 并入了 Qtopia，

改称为 Qtopia Core，作为嵌入式版本的核心，并在此基础上开发了面向于 PDA、手机等的版本，称为 Qtopia Phone Edition 和 Qtopia PDA Edition 等。

### 9.3.2 Qtopia Core 的体系结构

Qtopia Core 与 Qt/X11 最大的区别在于 Qtopia Core 不依赖于 X Server 或者 Xlib，而是直接访问帧缓存(Frame Buffer)，只需要一个 Qtopia Core 的动态共享库就足以替代 X server、Xlib 库和其他嵌入式解决方案的图形工具包，这样做最显著的效果是减少了内存消耗。

图 9.6 给出了 Qt/X11 与 Qtopia Core 系统架构图的对比：

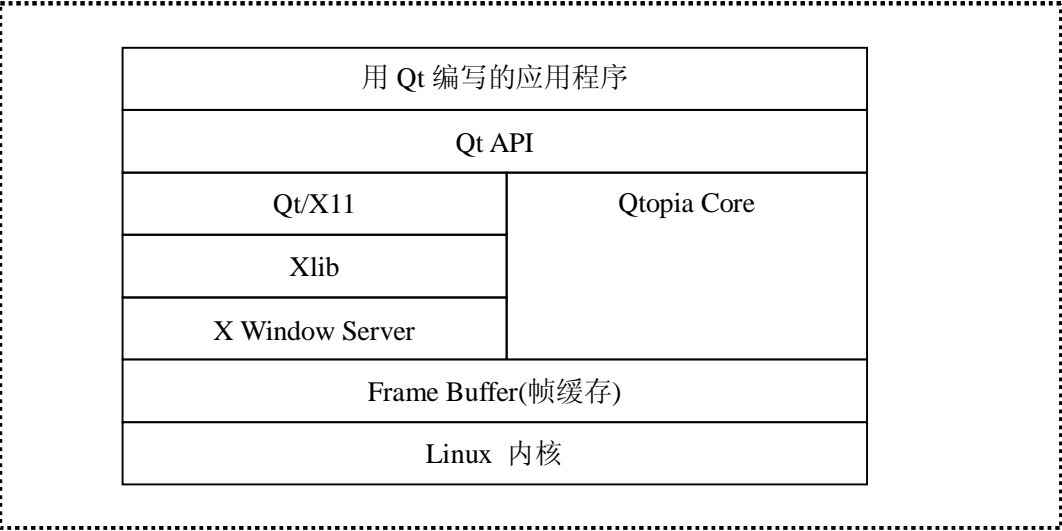


图 9.6 Qt/X11 与 Qtopia Core 系统架构图的对比

可以说，从效率上来看，Qtopia Core 是 Qt/X11 的精简版本，它去掉了一些 X Server 中需要消耗大量系统资源的特性；而从整个体系结构上来看，Qtopia Core 包含的内容其实要更多一些，因为它需要实现直接调用 Frame Buffer 来实现图形的绘制。

由此我们也可以看到嵌入式系统的一个特点，即很多情况下牺牲一些灵活性来换取效率的提高。与 Qt/X11 的结构相比，Qtopia Core 直接操作 Frame Buffer 的方式无疑会 X 架构的灵活性，但在嵌入式系统中，这种牺牲灵活性以换来效率提高的方法往往是必须的，也是值得的。

#### 9.3.2.1 Frame Buffer(帧缓存)简介

图 9.6 中我们提到的 Frame Buffer 实际上是一种对图形硬件设备的抽象<sup>[13]</sup>。Frame Buffer 是在 Linux 内核版本 2.2 以后推出的标准显示设备驱动接口，它将显示设备抽象为帧缓冲区，应用程序可以通过一组定义好的接口来操作显示设备，从而将底层的硬件细节隐藏起来。

从用户的角度看，Frame Buffer 设备与/dev 下的其他设备并无二致，它也是一种的字符设备。按照惯例，Frame Buffer 设备一般采用下面的节点：

0 = /dev/fb0 First frame buffer  
1 = /dev/fb1 Second frame buffer  
...

31 = /dev/fb31 32nd frame buffer

缺省情况下，应用程序（例如 X Server）使用/dev/fb0 作为 Frame Buffer 设备。用户可以通过设定环境变量\$FRAMEBUFFER 来指定使用其他设备作为 Frame Buffer 设备。

从程序员的角度来看，Frame Buffer 还是一种存储设备，可以对它进行读、写等操作，通过 mmap() 条用可以将其映射到进程地址空间，另外还可以通过几条 ioctl 命令，来获得显示设备的一些固定信息（比如显示内存大小）、与显示模式相关的可变信息（比如分辨率、像素结构、每扫描线的字节宽度），以及伪彩色模式下的调色板信息等等。所有这些硬件抽象使得应用程序的开发和移植变得更为简单。

由于 Qtopia Core 需要 Frame Buffer 的支持，我们所使用的嵌入式 Linux 内核要求 2.2 或之后的版本，或者将 Frame Buffer 移植到老的内核版本中。

### 9.3.2.2 Qtopia Core 的窗口系统

Qtopia Core 的窗口系统仍然采用 Client/Server 模型，任何一个 Qtopia Core 的应用程序都需要运行在一个 Server 应用上，而 Qtopia Core 的应用程序本身也可以作为 Server 来运行。一般情况下，我们可以用一个主程序来作为 Server 应用，而其他的应用程序都运行在 Server 之上。

与 X 相比，Qtopia Core 的窗口系统是比较轻量级的——很多 X 中需要由 Server 完成的工作都直接交给 Client 去完成，Server 和 Client 之间的通信开销也大大减少了。Server 进程通常会生成 QWSServer 类的一个对象，主要用来分配 Client 进程的显示区域，并产生鼠标和键盘事件等。而 Client 进程则通常生成 QWSClient 类的一个对象，负责处理与各种应用相关的逻辑。Server 和 Client 之间的通信通过 socket 来实现，这种通信通常被保持在一个较低的水平以减少通信的开销——比如窗口的绘制，并不是象 X 那样完全由 Server 来完成，而是由 Client 进行直接操作 Frame Buffer 来实现，Server 进程所作的仅仅是通知 Client 需要重绘的事件等。

## 9.4 本章小结

我们从 X Window 系统开始介绍了 Linux 下的 GUI 系统，以及 Qt 在其中所起的作用，这有助于我们在 Qt 应用开发中对整个系统有一个全局性的了解。同时我们也比较了作为桌面版本的 Qt/11 和嵌入式版本的 Qtopia Core 之间的一些差别，希望读者能够从中体会到嵌入式开发中常见的平衡哲学——牺牲一些灵活性完整性等来换取效率的提高。

## 9.5 常见问题

1. X Window 系统主要由哪几部分组成，分别有什么作用？

参考答案：

X Window 系统主要由 X Server，X Client 和 X Protocol 三部分组成。X Server 可以响应 X Client 的需求来绘制窗口，以及接受输入设备的输入信息并传递给 X Client；X Client 主要是处理应用程序本身的逻辑，接收 X Server 的事件并通过给 X Server 发送请求来绘制窗口；X Protocol 是 X Server 与 X Client 之间的通信协议。

2. Qt/X11，Qtopia 与 Qtopia Core 三者之间有什么不同？

参考答案：

Qt/X11 是基于 X11 架构的面向 Linux/Unix 操作系统的 Qt 版本。Qtopia Core 是面向嵌入式系统的 Qt 版本，它并不基于 X11 架构，而是直接操作 Frame Buffer。Qtopia 是建立在 Qtopia Core 之上的一个面向嵌入式 Linux 开发的平台，它在 Qtopia Core 的基础上提供了一系列的应用程序集以方便进一步的开发。



## 第 10 章 Qt/X11 初步

本章学习目标:

- I 了解 Qt/X11 的双重授权问题
- I 成功搭建 Qt/X11 开发环境
- I 学习利用 qmake 来协助编译 Qt 程序
- I 了解 Qt 开发的基础知识

### 10.1 Qt/X11 的安装

我们可以在 Trolltech 公司的主页下载或其他镜像站点免费下载 Qt/X11 开发包来进行安装, 不过我们需要准备好 Gcc 等编译工具以及 Xlib 库等, 而且由于 Qt/X11 的安装过程需要编译整个 Qt/X11 的源代码, 可能需要数小时的时间。

#### 10.1.1 Qt/X11 的下载及双重授权问题的说明

Qt/X11 自由版 (Qt/X11 Open Source Edition, 或称为 Qt/X11 开放源代码版本) 可以在 Trolltech 公司的主页下载: <http://www.trolltech.com/developer/downloads/qt/x11>。另外我们也可以选择在國內速度比较快的镜像站点, 比如 <ftp://ftp.qtopia.org.cn/mirror/ftp.trolltech.com/> 或者 <http://www.qtopia.org.cn/ftp/mirror/ftp.trolltech.com/>。

如上一章中所述, Qt 的版权曾经数次修改, 目前采用双重版权的方式来授权, 即分为自由版和商业版。我们所下载的 Qt/X11 自由版遵循 Q 公共许可证 (THE Q PUBLIC LICENSE) 和 GNU 通用公共许可证 (GPL, GNU GENERAL PUBLIC LICENSE) 而发布。在遵循 QPL 和 GPL 的条件下, 我们可以免费使用 Qt/X11 自由版来进行应用程序开发。

GPL 是由自由软件基金会 (Free Software Foundation, FSF) 发行的用于计算机软件的许可证, 最初由 FSF 的发起者 Richard Stallman 为 GNU 计划而撰写。目前大多数的 GNU 程序和超过半数的自由软件使用此许可证。Qt 所遵循的 GPL 版本是 1991 年发布的“版本 2”, 其详细内容请参见 <http://www.gnu.org/licenses/gpl.html> 或者非官方的简体中文翻译版 <http://www.thebigfly.com/gnu/gpl/> (目前 GPL v3 正在讨论中并遭到了不少反对, 有兴趣的读者可参考 <http://gplv3.fsf.org/>)。

QPL 是 Trolltech 公司所发布的许可证, 目前使用的 1.0 版本由 1999 年发布, 用于当时的 Qt2.0 自由版以使其成为开放源代码的软件, 并约束使用 Qt2.0 自由版的开发者不能用它来开发商业版本。QPL 的详细内容可参见 Trolltech 公司的官方网站 <http://www.trolltech.com/products/qt/licenses/licensing/qpl>。

使用 Qt/X11 自由版开发的应用程序需要公开源代码及遵循 GPL 和 QPL 的一些其他约束。因此, 如果用 Qt 开发商业软件的开发者不希望开放源代码, 则可以使用 Trolltech 提供的 Qt 的商业版本, 这种授权方式不再受 GPL 的约束。关于 Qt 商业版的详细情况请参见 <http://www.trolltech.com/products/qt/licenses/pricing>。

## 10.1.2 Qt/X11 的安装详解

安装 Qt/X11 之前需要保证你的 Linux 系统下已经安装了 gcc, make 等编译工具, 以及 xlib 相关的库等。如果在编译 Qt 的过程中出现错误, 一般都是因为缺少工具或库文件, 需要重新安装这些工具或库文件再继续编译 Qt。

假设我们现在已经将 Qt/X11 自由版的压缩包 qt-x11-opensource-src-4.2.2.tar.gz 下载到了自己的 Linux 机器的某个路径下 (不妨假设为 /home/user\_name/), 下面我们就可以解压缩并进行安装了, 依照机器硬件的配置, 安装的时间可能需要数小时:

步骤一: 解压缩 tar 包。

```
# cd /home/user_name/  
# tar xzvf qt-x11-opensource-src-4.2.2.tar.gz
```

这样解压后会生成 /home/user\_name/qt-x11-opensource-src-4.2.2 的目录, 安装 Qt 所需要的文件都在这个目录下。我们还可以在这个目录下找到 Qt 的相关文档: 在目录 doc 下有很多 html 文件, 我们可以参考这份本地文档而不必每次都登录到 Trolltech 的网站上去查看了。

步骤二: 运行配置程序

```
# cd qt-x11-opensource-src-4.2.2  
# ./configure
```

这时会看到一个问你是不是同意 GPL/QPL 的协议的问题, 回答 yes 就可以继续了。

configure 程序可以用来配置很多 Qt 的安装选项, 键入 “./configure -help” 可以看到很多配置选项以及说明。比如默认情况下, Qt 的安装路径是 /usr/local/Trolltech/Qt-4.2.2, 这样安装的时候需要有 root 权限, 如果我们没有 root 权限才可以选择将 Qt 安装到自己的 HOME 目录下, 键入 “./configure -prefix /home/user\_name /qt/” 则可以修改安装路径为 /home/user\_name/qt/。如果没有特殊要求的话, 一般默认的配置就可以满足我们的要求, 所以我们可以不作修改, 直接运行 ./configure 就可以了。

步骤三: 编译 Qt 源代码

```
# make
```

上一个步骤完成后, 我们可以看到在 /home/user\_name/qt-x11-opensource-src-4.2.2 的目录下生成了一个 Makefile, 在步骤三里, 我们利用 make 命令来编译 Qt 需要的共享库、工具以及例子等。这一步骤需要相当长的时间。

步骤四: 安装 Qt

```
# su -c "make install"
```

安装 Qt 到默认的路径 /usr/local/Trolltech/Qt-4.2.2 需要有 root 权限。如果没有的话, 如步骤二中所述, 需要在运行配置程序的时候修改安装路径。

步骤四完成后, 我们可以在目录 /usr/local/Trolltech/Qt-4.2.2 看到 Qt 的头文件、库文件、工具及例子程序等。

步骤五: 设置 PATH

前面的四个步骤实际上已经大致完成了安装 Qt 的工作了。为了我们在平时的 Qt 开发中能更方便的使用 Qt 提供的 qmake, moc 等工具, 我们 /usr/local/Trolltech/Qt-4.2.2/bin 添加到 PATH 变量——修改 \$HOME/.bash\_profile 或者 \$HOME/.profile 并加入 (或修改):

```
PATH=/usr/local/Trolltech/Qt-4.2.2/bin:$PATH
```

```
export PATH
```

修改完成后需要用 source 命令重新运行修改的脚本, 使设置生效, 如:

```
# source .bash_profile
```

这时候再运行

```
# echo $PATH
```

可以看到 PATH 中已经加入了 /usr/local/Trolltech/Qt-4.2.2/bin，我们可以尝试运行 Qt 自带的工具 Qt 助手：

```
# assistant
```

这时应该弹出如下图所示的 Qt 助手应用程序。Qt 助手提供了丰富的帮助文档，并且有索引和查询等功能，在 Qt 程序的开发中可以提供方便的资料查阅功能。

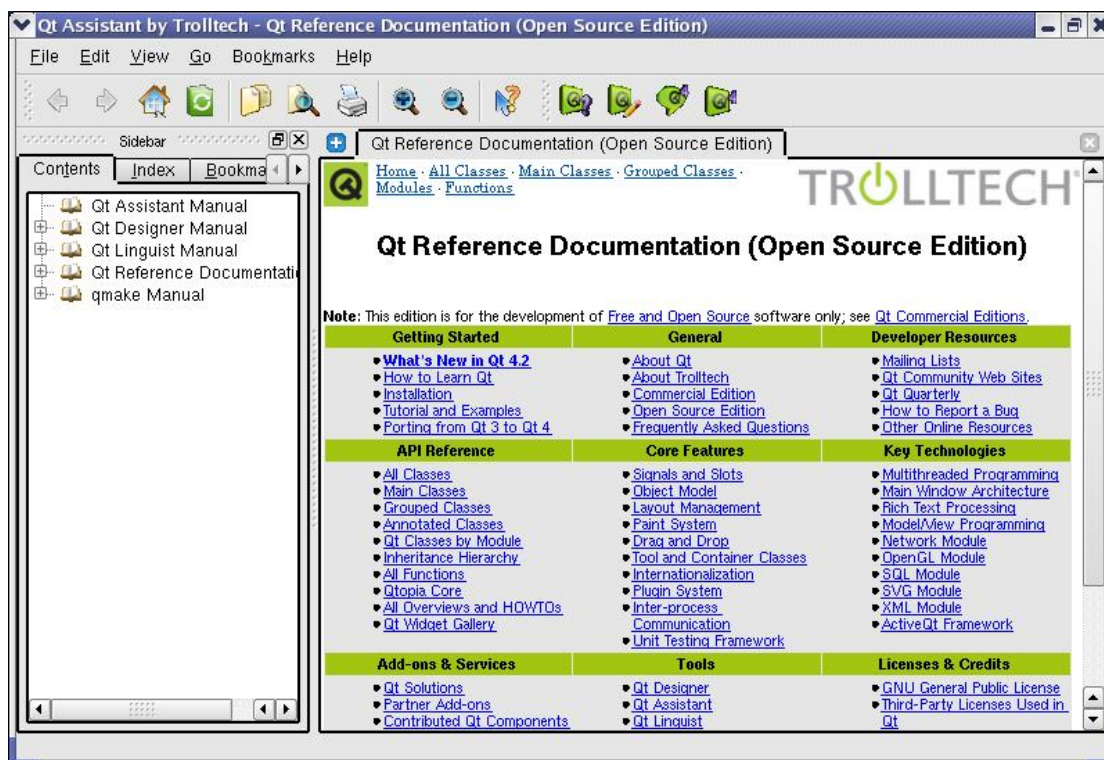


图 10.1 Qt 助手

如果上面的五个步骤都很顺利的话，则 Qt 的安装已经完成，下一节我们就可以尝试用 Qt 来写著名的 Hello World 程序了。

## 10.2 Qt 下的 Hello World

我们从最简单也是最著名的 Hello World 来学习 Qt。首先创建 helloworld.cpp:

```
# mkdir helloworld
# vi helloworld.cpp
```

helloworld.cpp 的内容如下所示：

```
1 #include <QApplication>
2 #include <QLabel>
3
4 int main(int argc, char *argv[])
5 {
```

```
6     QApplication app(argc, argv);
7
8     QLabel hello("Hello world!");
9
10    hello.show();
11    return app.exec();
12 }
```

然后我们来试图编译它。需要注意的是一般我们不直接用 `gcc` 命令或者直接编写 `Makefile` 来编译 Qt 的源代码，因为 Qt 中有一些对 C++ 的扩展，手工写的话会比较麻烦。通常我们利用 Qt 提供的 `qmake` 工具来编译 Qt 的源代码。

首先我们键入：

```
# qmake -project
```

可以看到 `qmake` 工具为我们自动生成了 `helloworld.pro` 文件。

接下来利用 `qmake` 自动生成 `Makefile`，键入：

```
# qmake
```

可以看到在当前目录生成了 `Makefile`。可以稍微浏览一下这个 `Makefile`，它有 100 多行，比较繁杂，这就是为什么我们一般不采用手工编写 `Makefile` 而利用 `qmake` 来生成的原因。

得到了 `Makefile` 就可以回到我们熟悉的方式来编译了，键入：

```
# make
```

就可以生成可执行的程序 `helloworld` 了。来看看我们的第一个 Qt 程序吧，键入：

```
# ./helloworld
```

就可以看到 `helloworld` 窗口了，调整一下大小，如下所示：



图 10.2 Hello World 示意图

请注意上面的编译过程。在后面的章节和实际开发中，我们都大致采用相同的顺序来编译 Qt 程序——从 `.pro` 到 `Makefile` 到生成库或应用。当然很多时候我们需要手工去编写或者修改 `.pro` 文件，出现编译错误的时候还很可能需要检查所生成的 `Makefile`，我们不能完全依赖于 `qmake` 工具。

下面我们回过头来看看 `helloworld.cpp` 中的代码。

整体来看，这就是一个我们非常熟悉的 `main()` 函数。对比控制台编程中用 `printf` 写的 `helloworld`，差别并不是很多。这对于只学习过基础 C/C++ 的读者应该还比较容易过渡，不像当年类似 MFC 之类的复杂框架，一个 `helloworld` 需要几百行代码，还找不到 `main()` 在哪里。

当然 Qt 的 `helloworld` 与 `printf` 写的 `helloworld` 是有些不同的，它是图形界面的程序。通常 GUI 应用程序和控制台程序比较明显的区别是，后者一般都是顺序执行，而前者总是进入一个循环等待，随着用户的动作而有所响应。这在我们的 `helloworld.cpp` 中主要体现在第 6 行和第 11 行。第 6 行生成一个 `QApplication` 对象 `app`，第 11 行将控制权交给了 `app`，进入循环等待。附带一提的是参数 `argc` 和 `argv`，它们并不是 Qt 的特性，而是属于 C 语言的特性，用来接受命令行参数，前者是命令行变量的数量，后者是命令行变量的数组。

我们再来看看其它的代码：前两行是简单的包含所需要的头文件，头文件结尾并没有加

上.h, 注意这是 Qt 版本 4 以来的改动, 以遵循 C++ 标准。如果读者所使用的是以前的 Qt 版本, 是编译不过这段示例代码的。第 8 行和第 10 行用来显示 "Hello world!"。类 QLabel 是 Qt 控件集中的一员, 用来显示一个标签。Qt 中所有控件都是 QWidget 的子类, 它们都能够用来作为程序的主窗口。

这只是一个最简单的 Qt 程序。下一节我们通过一个复杂一点的小例子来初步了解 Qt 的一些特性。

## 10.3 温度转换的小例子

这一节我们通过一步一步的演示, 来实现一个温度转换的小例子, 希望读者朋友可以从这个小例子中初步了解一些 Qt 的特性及用 Qt 进行程序开发的大致方法。

### 10.3.1 背景知识

摄氏温标 (Celsius) 和华氏温标 (Fahrenheit) 是两种不同的温度测量标准, 前者被大多数国家所采用, 后者由于出现较早, 在欧美一些国家一直沿用至今。两者之间的换算关系如下:

$$F = (C * 9 / 5) + 32$$

$$C = (F - 32) * 5 / 9$$

其中 F 表示华氏温度, C 表示摄氏温度。

### 10.3.2 Quit 按钮

下面我们开始真正的工作了。在这一小节中我们来搭建温度转换程序的主要结构, 并创建一个 Quit 按钮, 用来退出程序。

与前面的 Helloworld 程序相似, 首先我们需要一个 main() 入口函数, 并且这个函数内部也需要定义一个 QApplication 对象来将控制权交给 Qt。剩下的工作就是如何安排我们自己的窗口, 这比 Helloworld 要复杂一点, 如果还是把所有的代码都放到 main() 函数中就不太好了, 所以我们定义一个类 ConversionScreen 来绘制窗口。这样我们需要创建三个文件:

main.cpp

ConversionScreen.h

ConversionScreen.cpp

这样一来 main.cpp 变得非常简单了, 只有下面短短的数行代码:

```
1 #include <QApplication>
2
3 #include "ConversionScreen.h"
4
5 int main(int argc, char *argv[])
6 {
7     QApplication app(argc, argv);
8     ConversionScreen screen;
9 }
```

```
10     screen.show();
11     return app.exec();
12 }
```

这段代码与前面的 `HelloWorld` 非常类似，在这里不作重复解释了。注意唯一不同的是我们生成了一个 `ConversionScreen` 的对象，而不是一个 `QLabel` 对象了。

我们打算一步一步来完善 `ConversionScreen` 类。在这一小节里我们仅仅计划添加一个按钮用来关闭窗口，这需要用到 Qt 的核心特性之一——信号和槽(Signals and Slots)。我们将在下一章详细讲述信号和槽，这里先给大家一个直观的印象。

`ConversionScreen.h` 的代码如下所示：

```
1  #ifndef CONVERSION_SCREEN_H
2  #define CONVERSION_SCREEN_H
3
4  #include <QWidget>
5
6  class ConversionScreen : public QWidget
7  {
8  public:
9      ConversionScreen();
10     ~ConversionScreen() {};
11
12 private:
13     void createScreen();
14 };
15
16 #endif //CONVERSION_SCREEN_H
```

一般地当我们创建新的窗口或者窗口部件(Widget)时，都可以从 `QWidget` 或者 `QWidget` 的某个子类继承，这样可以充分的利用 Qt 窗口部件集提供的许多功能。关于 Qt 的窗口系统我们也安排在下一章详细讲述。

`ConversionScreen` 暂时只对外提供了一个构造函数。析构函数没有什么特殊要求，所以我们将它定义为空。私有函数 `createScreen()` 用来真正的创建窗口中的内容。

附带一提的是第 1、2 行和第 16 行。这是 C/C++ 语言里的一种惯用法，用来防止头文件被重复包含(include)。如果不采取任何措施，在稍具规模的软件开发中很容易出现头文件被重复包含，导致变量被重复定义的编译错误。而第 1、2 行和 16 行的做法则利用 C/C++ 的预编译功能防止了这种现象——一旦头文件被包含了一次，则 `CONVERSION_SCREEN_H` 已经被定义，下次预编译器就会略过第 1 行至第 16 行之间的内容了。

我们在 `ConversionScreen.cpp` 中来实现 `createScreen()` 函数——生成一个“Quit”按钮，并实现关闭程序的功能，代码如下所示：

```
1  #include <QPushButton>
2  #include <QApplication>
3
4  #include "ConversionScreen.h"
```

```

5
6  ConversionScreen::ConversionScreen() : QWidget()
7  {
8      createScreen();
9  }
10
11 void ConversionScreen::createScreen()
12 {
13     QPushButton* quit = new QPushButton("Quit", this);
14
15     connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
16 }

```

构造函数 `ConversionScreen()` 暂时只是简单的调用了 `createScreen()` 来创建窗口，并在初始化过程中调用了父类的构造函数 `QWidget()`。`QWidget` 的构造函数的声明如下：

```
QWidget ( QWidget * parent = 0, Qt::WindowFlags f = 0 )
```

我们在调用的时候两个参数都用默认值。第一个参数用来指定父部件——Qt 的窗口部件之间通过一种树型结构来组合，默认值 0 表示因为我们希望 `ConversionScreen` 类成为顶级部件；第二个参数指定部件的风格，默认值 0 表示普通风格。

`createScreen()` 函数则做了两件事：创建了一个 `QPushButton` 对象，并将它的 `clicked()` 信号连接到 `qApp` 的槽 `quit()`。

第 13 行中创建 `QPushButton` 对象时我们调用的构造函数原型如下：

```
QPushButton ( const QString & text, QWidget * parent = 0 )
```

第一个参数指定按钮上显示的字符，而第二个参数则指定这个按钮的父部件，在这里我们将 `Quit` 按钮的父部件设为 `this`，即 `ConversionScreen` 本身。

第 15 行我们将 `Quit` 按钮发出的 `clicked()` 信号连接到 `qApp` 的槽 `quit()`，当用户点击这个按钮时，`clicked()` 信号将被发射，从而槽 `quit()` 被执行，程序被关闭。这是一个简单的信号和槽如何工作的过程。以前没有接触过信号和槽的读者可能还不是很理解这种过程，没有关系，在下一章我们将详细讲述信号和槽的来龙去脉。

来看一下 `connect()` 函数。`connect()` 函数是 `QObject` 类所提供的成员函数，而 `QObject` 类是所有 Qt 类的基类，所以它可以方便的在各处应用，其大致用法如下：

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

其中 `sender` 和 `receiver` 是两个指向 `QObject` 对象的指针，`sender` 发射信号 `signal`，`receiver` 接收到信号后则执行 `slot`。`SIGNAL` 和 `SLOT` 类似两个关键字，来标志信号和槽。

在我们的代码中，前两个参数比较简单，就是我们刚刚定义的 `quit` 按钮和它被点击时发出的信号 `clicked()`；后两个参数中的 `qApp` 则可能让人疑惑，它并没有在我们的类中定义，那么它是父类中定义的一个公共成员吗？我们来查查 Qt 的文档。

首先我们来看一下 `html` 格式的文档，在 Qt 安装路径或者解压缩路径下的目录 `doc/html` 中我们可以找到很多 `html` 文件，打开其中的 `index.html` 并把它加到浏览器的书签中，以后我们就可以方便的找到它了。不过在 `html` 格式的文档中，我们很难找到关于 `qApp` 的内容，因为我们不是很清楚它在哪个类中，事实上在 `QWidget` 类的文档中找不到 `qApp`。这种时候我们可以借助一个更方便的工具——Qt 提供的助手工具 `assistant`（还记得吗？我们在 Qt 安装完成后还试过启动它呢），它的索引功能可以帮助我们快速找到 `qApp` 的相关说明，如图 10.3 所示：



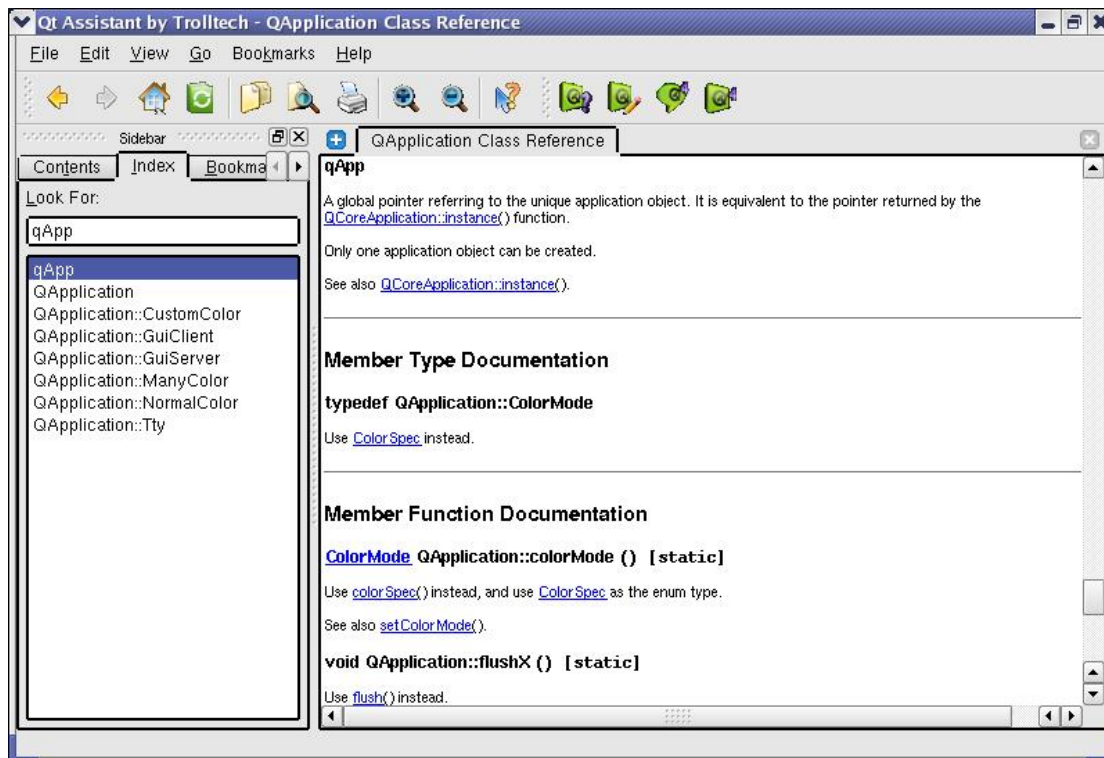


图 10.3 利用 Qt 助手查找 qApp 的相关说明

从 Qt 助手的说明中我们可以看到，qApp 是在 QApplication 中的一个宏，它被定义成指向进程中唯一的 QApplication 对象的指针，这个指针由 QCoreApplication::instance() 所返回。

喜欢寻根问底的朋友可以继续追查下去：对于 Qt 这样的开源工具包，一个很大的好处是我们可以查看它的源代码。在我们所下载的 Qt 安装包中，解压缩后可以在 src 目录中看到它所有的源代码，来看看 qApp 到底是什么：

```
# cd /home/user_name/qt-x11-opensource-src-4.2.2/src
# find . -name qapplication.h
./gui/kernel/qapplication.h
```

在 qapplication.h 的第 62 行可以看到 qApp 的定义如下：

```
#define qApp (static_cast<QApplication*>(QCoreApplication::instance()))
```

可以看到 qApp 是一个指向 QApplication 对象的指针，作为 SLOT 的 quit() 函数是 QApplication 的成员函数，调用它将退出 Qt 的循环等待，在我们的程序中相当于 main.cpp 中的第 11 行 app.exec() 返回，于是整个应用都退出了。

在这里我们主要是希望能够通过简单的例子来引导大家如何借助 Qt 助手和源代码来学习 Qt，在后面的章节中我们还会讲解一些 Qt 的源代码来帮助大家更加深刻的了解 Qt 的特性。

细心的读者可能还会对第 13 行有疑问：我们用 new 创建了一个堆上的对象，却没有用 delete 删除它，会不会造成内存的泄漏呢？不用担心，Qt 中有一种内在机制，任何部件在自身销毁的同时，会自动销毁它所有的子部件，Qt 部件之间的树型结构保证了只要顶层的部件被销毁，它下面所有的子部件即整棵树都会被自动销毁，在一定程度上避免了内存泄漏的发生。在 13 行我们已经将 Quit 按钮的父部件设为 this，这样在 main 函数中 screen 对象自动析构后，Quit 按钮对象即会被销毁，并不会造成内存泄漏。在后面的示例代码中我们会看



到还有很多类似的情况，并且我们将在下一章详细讲述这种特性。

我们采用于 Helloworld 同样的方法来编译：

```
# qmake -project
# qmake
# make
```

运行一下编译后得到的程序，如下所示：



图 10.4 Quit 按钮

点击 Quit 按钮，程序退出了。回顾一下，事实上我们只是搭了一个框架，并没有真正写几行代码，对比 Helloworld，我们只是增加了一个连接——希望大家慢慢熟悉这种信号和槽的连接，在下一节中我们马上又会要用到。

### 10.3.3 摄氏温度的显示

这一小节中我们将完成用来显示摄氏温度的滑动条，并进行简单的窗体布局。

我们来选择一个看上去与那种水银温度计尽可能接近的部件来显示摄氏温度——QSlider 提供的滑动条看起来还可以（至少比按钮、文本框、菜单等要接近一点吧^\_^）。同时我们还需要标志一下这个温度计的读数，简单起见我们就用一个 QLabel 对象来显示温度吧（还记得我们在 Helloworld 程序里用过的 QLabel 类吗）。这个标签上的读数应该随着滑动条的滑动而变化，这需要一个类似上一小节中用到的信号和槽的连接来完成。

现在我们将会有三个部件了：Quit 按钮、摄氏温度滑动条以及温度标签，而且滑动条和标签需要离得比较近以方便标记读数。这样我们需要安排一下它们彼此的位置。

我们只需要改动 ConversionScreen.cpp 中的部分代码就可以完成这些工作，如下所示：

```
1  #include <QPushButton>
2  #include <QSlider>
3  #include <QLabel>
4  #include <QVBoxLayout>
5  #include <QHBoxLayout>
6  #include <QApplication>
7
8  #include "ConversionScreen.h"
9
10 ConversionScreen::ConversionScreen() : QWidget()
11 {
12     createScreen();
13 }
14
15 void ConversionScreen::createScreen()
16 {
17     QPushButton* quit = new QPushButton("Quit");
```

```

18
19     QSlider* slider = new QSlider(Qt::Vertical);
20     slider->setRange(0, 100);
21     slider->setValue(0);
22     slider->setTickPosition(QSlider::TicksLeft);
23
24     QLabel* label = new QLabel("0");
25
26     QHBoxLayout* cLayout = new QHBoxLayout;
27     cLayout->addWidget(label, 0, Qt::AlignRight);
28     cLayout->addWidget(slider, 0, Qt::AlignLeft);
29
30     QVBoxLayout* mainLayout = new QVBoxLayout;
31     mainLayout->addWidget(quit);
32     mainLayout->addLayout(cLayout);
33     setLayout(mainLayout);
34
35     slider->setFocus();
36
37     connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
38     connect(slider, SIGNAL(valueChanged(int)), label, SLOT(setNum(int)));
39 }

```

从 19 到 21 行我们创建了一个竖向显示的滑动条，简单起见我们假设这个温度转换器的转换范围为 0-100 摄氏度，其默认值我们暂时设为 0 度。第 22 行设置刻度的显示在滑动条的左边。

24 行我们创建一个标签来显示滚动条的读数，它的默认值也被设为 0。

我们需要把滑动条和标签绑定在一起，使得这个标签看起来真的象是温度计的读数，同时我们还要安排一下原来的 Quit 按钮，这时我们需要借助 Qt 的布局管理器相关的类，主要有三种基本的布局管理器可以用来安排部件的位置：

- QHBoxLayout 在水平方向上从左到右（默认情况）或者从右到左布置部件。
- QVBoxLayout 在竖直方向从上往下（默认情况）或者从下往上布置部件。
- QGridLayout 以网格形式布局部件。

从 26 行到 33 行我们都在安排这些部件的位置。我们首先将滚动条和标签装到一个 QHBoxLayout 对象中，并且设置它们的对齐方式（第 27 行和 28）使它们看起来总是比较靠近。成员函数 addWidget 的定义如下：

**U** void addWidget(QWidget \* widget, int stretch = 0, Qt::Alignment alignment = 0)

其中第一个参数是我们要加入布局管理器的部件指针；第二个参数用来定义部件的伸展，我们采用默认值 0，即不作任何伸展变化；第三个参数设置部件的对齐方式，默认情况下部件将充满整个区域，我们设定标签和滚动条分别向右和向左对齐，这样它们看起来总是在一起。

然后我们把 Quit 按钮和这个 QHBoxLayout 对象装到一个 QVBoxLayout 对象中，按钮在上，而我们绑定后的滚动条和标签放在下方。在 32 行我们采用的是 addLayout() 成员函数来将一个布局对象放到另一个布局对象中。在 34 行我们设置焦点为滚动条，这样当程序运行

时键盘的响应焦点在滚动条上，用向上或向下的方向键即可调节温度。

类似上一小节，我们来看一下用 `new` 生成的堆对象，我们的代码中并没有显式的调用 `delete` 来销毁它们，因为 Qt 提供了自动删除子部件的内在机制。不过我们在 17 行、19 行及 24 行等处所创建的对象呢并没有指定父部件，而是借助 `addWidget` 和 `addLayout` 两个函数，将自动删除对象的任务暂时的交给了这些布局对象——这并不是说布局对象成为了它们的父部件，而是当布局对象被 `setLayout()` 函数设置为某个部件的布局风格时（如第 33 行所示），这些对象的父部件将被自动设置为这个部件。这样一来，情况就与上一小节中提到的自动销毁类似了。

我们还需要完成的一件事是使标签上的读数随着滑动条的滑动而变化，这样我们做温度转换的时候才能自由选择度数。这个功能在第 38 行，同样通过一个类似上一小节中用到的信号和槽的连接来完成——滑动条发生变化时，将发射 `valueChanged(int)` 信号，其变化的数值通过 `int` 型参数被传递到标签对象的 `setNum(int)` 槽，这个槽的作用就是改变标签所显示的数值。注意这个信号和槽的连接稍有不同，它向我们演示了如何在信号和槽之间传递参数。

采用与上一小节相同的方法编译，如果我们只是改动了 `ConversionScreen.cpp` 而 `Makefile` 还在的话，只需要重新运行 `make` 就可以了：

```
# make
```

`make` 工具会自动为我们找到修改了的源文件并重新编译它。

重新编译之后得到的程序看起来如下图所示：

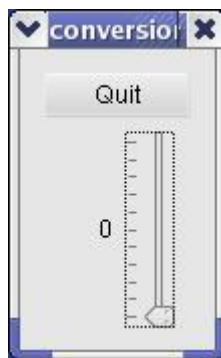


图 10.5 摄氏温度计

### 10.3.4 华氏温度的显示

我们用与上面类似的方法，用一个部件来模拟温度计的形状，用另外一个部件来显示读数，不过这一次我们不用滚动条了——我们来画一个转盘型的温度计，这可以利用 Qt 提供的 `QDial` 类。`QDial` 类与 `QSlider` 虽然形状不同，其实在功能上非常类似。同时我们采用 `QLCDNumber` 类来显示温度计读数，这种模拟液晶显示的数字看起来更像仪表盘上读数。转盘和读数之间同样需要一个信号和槽的连接来使它们保持一致。

加上这个转盘温度计之后我们需要重新摆放一下这些部件了。上一小节中我们简单介绍了窗口的布局管理，这里我们需要进一步的应用。我们将采用 `QGridLayout` 来学习如何进行网格布局，将这两个温度计以及 `Quit` 按钮摆放到网格中。

为了不使 `createScreen()` 函数过于庞大，我们把它的内容分解一下，增添两个私有成员函数来创建两个温度计，同时也增加几个成员变量，这样源文件 `ConversionScreen.h` 的代码如下所示：

```

1  #ifndef CONVERSION_SCREEN_H
2  #define CONVERSION_SCREEN_H
3
4  #include <QWidget>
5
6  class QSlider;
7  class QDial;
8  class QHBoxLayout;
9  class QVBoxLayout;
10
11 class ConversionScreen : public QWidget
12 {
13 public:
14     ConversionScreen();
15     ~ConversionScreen() {};
16
17 private:
18     void createScreen();
19     void createCel();
20     void createFah();
21
22     QSlider* slider;
23     QDial* dial;
24
25     QHBoxLayout* celLayout;
26     QVBoxLayout* fahLayout;
27
28 };
29
30 #endif //CONVERSION_SCREEN_H

```

我们在 19 行和 20 行增加了两个函数 `createCel()` 和 `createFah()`，分别用来创建摄氏温度计和华氏温度计。23—26 行增加了四个成员变量，分别对应上文中提到的滚动条、转盘及摄氏温度计的布局和华氏温度计的布局。

增加了成员变量之后我们在 6—9 行增加了所用到的 Qt 类的声明。稍微解释一下：如果没有这些类的声明显然是无法通过编译的，那么是否可以采用象第 4 行那样的包含相应的头文件的方式呢？这样做是可行的，编译或链接都不会有任何问题，得到的二进制文件也不会有任何区别，它主要的不利之处在于增加了预编译的时间及头文件之间的依赖关系。在大型项目的开发中，如果头文件之间的包含非常复杂，预编译处理所耗费的时间会相当的长，头文件之间的依赖关系也很可能导致一些潜在的风险，因此我们一般都采用 6—9 行的方式，如果用声明的方式足够的话，就不用直接在头文件中包含其他头文件的方式。当然，第 4 行的包含是不可避免的，因为 `ConversionScreen` 类是从 `QWidget` 继承来的。

来看看增加新函数之后它们的实现：

```
1  #include <QPushButton>
2  #include <QSlider>
3  #include <QLabel>
4  #include <QDial>
5  #include <QLCDNumber>
6  #include <QVBoxLayout>
7  #include <QHBoxLayout>
8  #include <QGridLayout>
9  #include <QApplication>
10
11 #include "ConversionScreen.h"
12
13 ConversionScreen::ConversionScreen() : QWidget()
14 {
15     createScreen();
16 }
17
18 void ConversionScreen::createScreen()
19 {
20     QPushButton* quit = new QPushButton("Quit");
21
22     createCel();
23     createFah();
24
25     QGridLayout *mainLayout = new QGridLayout;
26     mainLayout->addWidget(quit, 0, 0);
27     mainLayout->addLayout(cellLayout, 1, 0);
28     mainLayout->addLayout(fahLayout, 1, 1);
29     mainLayout->setSpacing(40);
30     mainLayout->setMargin(40);
31     setLayout(mainLayout);
32
33     slider->setFocus();
34
35     connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
36
37     setWindowTitle("Temperature Conversion");
38 }
39
40 void ConversionScreen::createCel()
41 {
42     slider = new QSlider(Qt::Vertical);
43     slider->setRange(0, 100);
44     slider->setValue(0);
```

```

45     slider->setTickPosition(QSlider::TicksLeft);
46
47     QLabel* celLabel = new QLabel("0");
48
49     cellayout = new QHBoxLayout;
50     cellayout->addWidget(celLabel, 0, Qt::AlignRight);
51     cellayout->addWidget(slider, 0, Qt::AlignLeft);
52     cellayout->setSpacing(10);
53
54     connect(slider, SIGNAL(valueChanged(int)), celLabel, SLOT(setNum(int)));
55 }
56
57 void ConversionScreen::createFah()
58 {
59     QLCDNumber* lcdNum = new QLCDNumber(3);
60     lcdNum->setSegmentStyle(QLCDNumber::Filled);
61
62     dial = new QDial;
63     dial->setRange(32, 212);
64     dial->setValue(32);
65     dial->setNotchesVisible(true);
66
67     fahLayout = new QVBoxLayout;
68     fahLayout->addWidget(lcdNum, 0, Qt::AlignBottom | Qt::AlignHCenter);
69     fahLayout->addWidget(dial);
70     fahLayout->setSpacing(10);
71
72     connect(dial, SIGNAL(valueChanged(int)), lcdNum, SLOT(display(int)));
73 }

```

函数 `createCel()` 其实就是把上一小节中 `createScreen()` 函数里面的一段代码移过来并稍作修改，并增加了第 52 行来设置滚动条和标签之间的距离为 10 个像素。

函数 `createFah()` 用来创建转盘形状的华氏温度计。59 行和 60 行我们创建了一个模拟液晶显示的数字，并设置其显示风格为向外凸出并用前景色填充。62—64 行与创建滑动条的温度计时非常类似，设置范围和初始值。第 58 行我们将刻度显示设为真。这样我们创建并设置了所需要的两个部件对象：`lcdNum` 和 `dial`。在 67 行至 70 行我们将这两个部件装到一个竖向的布局容器中，同样设置布局容器内的部件之间的距离为 10 个像素。注意第 68 行我们设置对齐方式时采用了 `"Qt::AlignBottom | Qt::AlignHCenter"` 来作为参数，这意味着在竖直方向上采用向底部对齐，而水平方向上则设置为居中。函数最后的第 72 行采用类似的连接来保证读数随转盘的指针转动而变化。

再来看看 `createScreen()` 函数。除了调用私有成员函数 `createCel()` 和 `createFah()` 来创建两个温度计之外，这个函数的主要工作还有进行整体的布局。我们采用网格布局的方式，利用 `QGridLayout` 来进行布局。网格布局时需要指定一个二维的坐标来设置布局的位置，如 26

—28 行所示, `addWidget` 和 `addLayout` 两个函数的前两个参数就是指定二维坐标, 即第几行和第几列, 行和列都从 0 开始计数。我们把 `Quit` 按钮安排在第一行第一列, 第一行的二列先空着 (留给后面的另外一个按钮), 第二行的第一列和第二列分别摆放两个温度计。29 行和 30 行设置网格内的部件之间的距离以及网格外的边界大小。注意联系第 52 行和第 70 行, 如果采用默认设置而不作 52 行和 70 行的设置的话, 两个温度计内部的部件之间的距离也依照第 29 行的设置而变为 40, 这样就显得太松散了, 读者可以试一试注释掉 52 行和 70 行的效果。`createScreen()` 函数的最后, 我们在 37 行设置了窗口的标题为 "Temperature Conversion"。

编译后得到的窗口如图 10.6 所示:

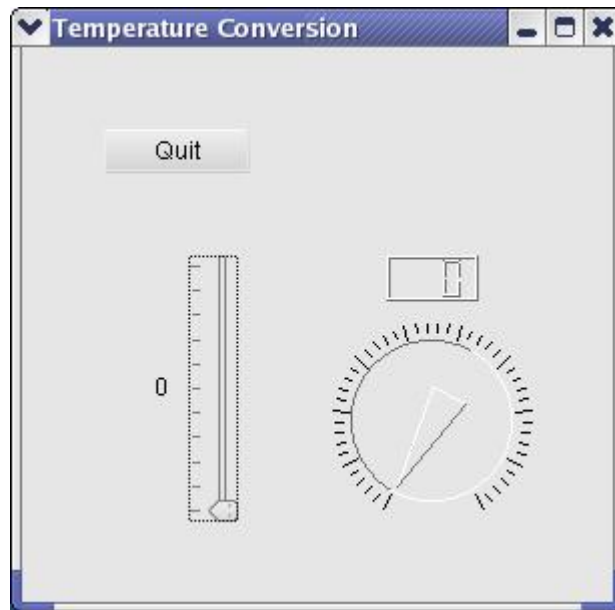


图 10.6 摄氏温度计和华氏温度计

这时候程序看起来已经很接近我们要完成的目标了, 可以试着用鼠标来改变两个温度计的读数, 还可以试一试利用键盘的上下键来移动滑块或者转动指针, 这时旁边的读数会相应的发生变化。当然, 这些都还是准备工作, 真正的温度转换工作还没有做呢。

### 10.3.5 华氏温度和摄氏温度之间的转换

这一小节我们来真正的完成温度转换的工作。经过前面的一系列准备工作之后, 真正的温度转换只需要一些简单的换算工作了。当然, 可能有的读者已经猜到了, 要使两个温度计之间的读数互相随着对方的变化而变化, 我们还需要用到信号和槽的连接, 而且由于两个温度计的读数之间需要换算, 这次我们不能只依赖于 Qt 提供的槽了, 而需要自己定义槽来完成两种温度之间的换算。

在类中定义自己的槽的方法与定义成员函数类似, 事实上槽就是一种特殊的成员函数, 它有一些限制, 我们将在下一章中详细讨论, 在这里我们先来看看 `ConversionScreen.h` 中如何来声明所需要的槽:

```
1  #ifndef CONVERSION_SCREEN_H
2  #define CONVERSION_SCREEN_H
3
4  #include <QWidget>
5
6  class QSlider;
```

```

7  class QDial;
8  class QHBoxLayout;
9  class QVBoxLayout;
10
11 class ConversionScreen : public QWidget
12 {
13     Q_OBJECT
14
15 public:
16     ConversionScreen();
17     ~ConversionScreen() {};
18
19 private slots:
20     void celToFah(int celNum);
21     void fahToCel(int fahNum);
22
23 private:
24     void createScreen();
25     void createCel();
26     void createFah();
27
28     QSlider* slider;
29     QDial* dial;
30
31     QHBoxLayout* celLayout;
32     QVBoxLayout* fahLayout;
33
34 };
35
36 #endif //CONVERSION_SCREEN_H

```

可以看到，在 19—21 行我们定义了两个槽用来做两种温度之间的转换，Qt 自定义的关键字 `slots` 用来表示下面的函数是作为槽而不是普通的成员函数，`slots` 前面的 `private` 仍然保持 C++ 关键字的意义，被定义为 `private` 的槽只能被这个类内部被调用。

另外需要注意的是第 13 行的宏 `Q_OBJECT`，当类中需要自定义的信号或者槽时，必须在这个位置使用宏 `Q_OBJECT`，这是 Qt 的元对象系统所要求的，只有借助这个宏才能完整的实现元对象系统，才能完成信号和槽的连接，在下一章中我们会详细讨论 Qt 的元对象系统，这里我们只需要有一个直观的印象就可以了。另外需要说明一下的是，在前面的代码中我们为了简单起见，没有过早的引入这个宏，只有在需要自定义槽时才提到，但一般我们建议在从 `QObject` 继承来的类中都添加这个宏，以避免将来无意的改动所带来的危险。

为了节省篇幅在这里我们不列出 `ConversionScreen.cpp` 的源代码了，完整的源代码可以在随书所附的光盘上找到。我们仅仅来分析一下相对上一小节所作的改动。

首先我们需要实现两个自定义的槽，在 `ConversionScreen.cpp` 的第 77 行到第 87 行可以



找到:

```
77 void ConversionScreen::celToFah(int celNum)
78 {
79     int fahNum = (celNum * 9 / 5) + 32;
80     dial->setValue(fahNum);
81 }
82
83 void ConversionScreen::fahToCel(int fahNum)
84 {
85     int celNum = (fahNum - 32) * 5 / 9;
86     slider->setValue(celNum);
87 }
```

这两个槽的实现非常类似: 首先计算两种温度之间的换算后的结果, 然后用 `setValue()` 函数来改变另外一个温度计的读数。

我们还需要在信号和槽之间建立连接, 如下所示(我们略去了与上一小节中类似的部分, 用省略号表示):

```
40 void ConversionScreen::createCel()
41 {
...
54     connect(slider, SIGNAL(valueChanged(int)), celLabel, SLOT(setNum(int)));
55     connect(slider, SIGNAL(valueChanged(int)), this, SLOT(celToFah(int)));
56 }
57
58 void ConversionScreen::createFah()
59 {
...
73     connect(dial, SIGNAL(valueChanged(int)), lcdNum, SLOT(display(int)));
74     connect(dial, SIGNAL(valueChanged(int)), this, SLOT(fahToCel(int)));
75 }
```

我们在 57 行和 74 行增加了两个类似的连接: 55 行将摄氏温度计的变化连接到槽 `celToFah()`, 74 行则将华氏温度计的变化连接到槽 `fahToCel()`, 由于这两个槽都是 `ConversionScreen` 中所定义的, 所以连接时第三个参数设置为 `this` 指针。可以看到这里的连接有一些复杂, `slider` 对象的同一个信号 `valueChanged` 被连接到了两个不同的槽, 类似的 `dial` 对象的同一个信号 `valueChanged` 被也连接到了两个不同的槽, 这种连接是允许的——在 Qt 中可以将单个的信号与很多的槽进行连接, 也可以将很多信号与单个的槽进行连接。

来编译一下看看结果如何。如果我们还是利用上一小节中的 `Makefile` 来编译, 输入:

```
# make
```

则很可能会得到类似下面的链接错误:

```
ConversionScreen.o(.text+0x20): In function
`ConversionScreen::ConversionScreen[not-in-charge]()':
: undefined reference to `vtable for ConversionScreen'
```

```

ConversionScreen.o(.text+0x27): In function
`ConversionScreen::ConversionScreen[not-in-charge]()':
: undefined reference to `vtable for ConversionScreen'
ConversionScreen.o(.text+0x74): In function
`ConversionScreen::ConversionScreen[in-charge]()':
: undefined reference to `vtable for ConversionScreen'
ConversionScreen.o(.text+0x7b): In function
`ConversionScreen::ConversionScreen[in-charge]()':
: undefined reference to `vtable for ConversionScreen'
main.o(.text+0x48): In function `main':
: undefined reference to `vtable for ConversionScreen'
main.o(.text+0x4f): more undefined references to `vtable for ConversionScreen' follow
collect2: ld returned 1 exit status

```

链接错误有时候会比较难找出来，不过对这种情况不用慌张，在 Qt 的开发中可能会常遇到类似的错误，原因在于 Qt 复杂的元对象系统。在这个例子中由于我们增加了两个自定义的槽，并且在 `ConversionScreen.h` 的第 13 行使用了宏 `Q_OBJECT`，这些必须借助 Qt 的 `moc`（Meta Object Compiler）工具来生成一个额外的源文件，加入这个源文件到编译和链接过程中，才能正确的得到最后的二进制可执行程序。

我们重新来生成一下 `Makefile` 就可以利用 `moc` 工具了：

```

# qmake
# make

```

这时可以看到由 `moc` 工具自动生成了一个名为 `moc_ConversionScreen.cpp` 的源文件，这个源文件也参与到编译和链接过程中，这样我们就可以得到可运行的二进制程序了，如下图所示，我们可以拖动左边的滑块或者拉动右边的指针，另外一个的读数会发生相应的变化：

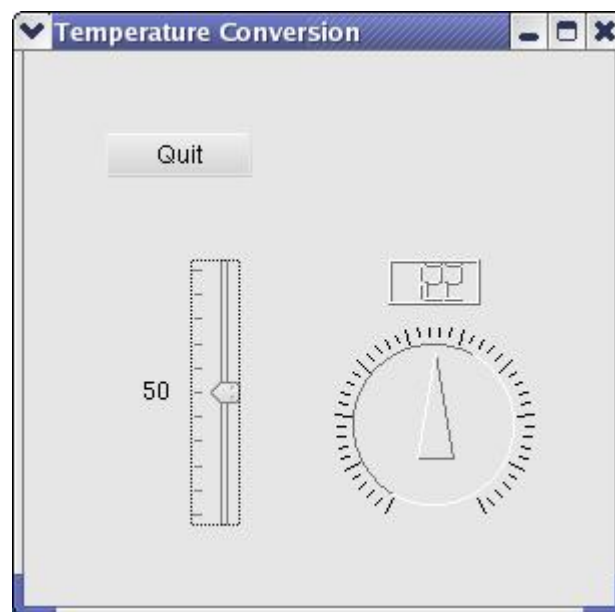


图 10.7 温度的转换示意图

如上图所示，让我们来回顾一下这些信号和槽之间的连接：假设我们拖动左边的滑动条到了中间位置，这时滑动条将发射信号 `valueChanged`，一共有两个槽与这个信号相连接：一

个是滑动条左边的标签对象的槽 `setNum`，它将标签显示的数字设置为相应的值即 50；另一个是我们自定义的槽 `celToFah`，它将温度值转换之后再设置右边的 `dial` 对象的值——这次设置将转盘的指针转到了中间位置，并且 `dial` 对象的改变也会发射 `valueChanged` 信号，这个信号也被连接到两个槽：一个设置液晶显示数字为相应的华氏温度值即 122，这样我们完全已经得到了所需要的温度转换的功能了；另一个是设置左边 `slider` 对象的值，这样会不会形成死循环呢？设置左边 `slider` 对象的值之后，`slider` 对象又发射信号 `valueChanged`，然后前面的又重来一遍，这样一直循环下去？不会的，因为 `slider` 对象的值现在已经是 50，Qt 内部自动做了这种检查，当没有改变的时候就不会发射信号了。当然即使 Qt 不提供这种自动检查，我们也可以自己来作检查以避免死循环，在实现槽的时候加一个判断就可以了。

### 10.3.6 保存当前的数值

这是我们这个小例子的最后一部分了。前面我们已经完成了完整的温度转换的工作，这一小节我们来增加一个附加功能——保存这次转换后的数值，程序退出后下一次再运行时直接显示上次所保存的数值。对这个例子来说，这样做并没有太多实际的意义，我们主要是想给大家演示一下在 Qt 中如何实现这种保存设置的功能，以作为实际 Qt 开发中的参考，因为这毕竟是一种很常见的需求。

Qt 提供了 `QSettings` 类来将一些数值保存到文件系统中（对 Windows 系统来说可能是注册表），这样当程序退出时这些数值不致于丢失，在下次需要的时候还可以从文件系统（或注册表）中读取。`QSettings` 类提供了数个构造函数以方便使用，常用的构造函数如下所示：

```
QSettings(const QString & organization, const QString & application = QString(), QObject *
parent = 0);
QSettings ( QObject * parent = 0 );
QSettings ( Format format, Scope scope, const QString & organization, const QString &
application = QString(), QObject * parent = 0 )
```

我们使用第一个构造函数时常常需要设定好前两个参数，如下所示：

```
QSettings settings("MySoft", "MyApp");
```

对 Qt/X11 来说，这两个参数指定了所保存的文件存储时的目录和文件名。

而使用第二个构造函数时则需要事先调用 `QCoreApplication::setOrganizationName()` 和 `QCoreApplication::setApplicationName()` 来设定类似上面的两个参数，即：

```
QCoreApplication::setOrganizationName("MySoft");
QCoreApplication::setApplicationName("MyApp");
QSettings settings;
```

这样设定后与上面的构造函数作用是相同的，它的好处在于，当应用程序中多处需要这种 `QSettings` 对象时，只需要在这里设定一次，以后就都可以简单的使用第二个构造函数了。

第三个构造函数比较复杂，增加了前两个参数，分别用来设置文件的存储格式和作用范围。对第一个参数我们可以选择两种文件格式来存储设置，分别是 `QSettings::NativeFormat` 和 `QSettings::IniFormat`。对于 Windows 操作系统来说，`NativeFormat` 使用注册表来存储设置，而 `IniFormat` 则顾名思义是采用 INI 格式的文件来保存设置。对于 Linux/Unix 来说这两个格式区别不大，其实都是采用 INI 格式的文件来保存，唯一不同的是文件的扩展名，前者是 `.conf`，后者则是 `.ini`。注意默认情况下存储格式是 `NativeFormat`。另外 Qt 还支持自定义的文件格式，不过我们的小例子中暂时不需要这种功能，有兴趣的读者可自行参考 Qt 的官方

文档。

对第二个参数我们可以选择 `QSettings::UserScope` 和 `QSettings::SystemScope`，这两种作用范围主要关系到文件的存储路径，其中 `UserScope` 是默认的设置。对于 Qt/X11 来说，如果采用默认的 `UserScope`，则文件将存储到 `$HOME/.config` 路径下；如果采用 `SystemScope`，则路径为 `/etc/xdg`。我们可以看看完整的存储路径：如果采用 `NativeFormat` 和 `UserScope`，并且构造 `QSettings` 对象时采用上面的示例代码的方法，则文件的路径应该是 `$HOME/.config/MySoft/MyApp.conf`。

了解了这些基本用法之后，我们结合温度转换的小例子来看看如何使用 `QSettings` 类来保存当前的数值。我们首先需要在 `ConversionScreen.h` 文件中增加槽、成员函数以及成员变量，相关的代码如下所示：

```
...
19 private slots:
20     void celToFah(int celNum);
21     void fahToCel(int fahNum);
22     void saveSettings();
23
24 private:
...
29     void initSettings();
30     void readSettings();
...
38     int currentCelNum;
39     int currentFahNum;
...
```

其中第 22 行我们增加了一个槽，用来响应用户点击“Save”按钮，保存当前的两个温度计数值；第 29 行和第 30 行增加了两个私有成员函数，分别用来初始化设置和从文件中读取设置；第 38 行和 39 行则是两个成员变量，代表当前两个温度计的读数。

来看看这些新增加的函数和变量在 `ConversionScreen.cpp` 中的实现以及应用。

```
1  #include <QPushButton>
2  #include <QSlider>
3  #include <QLabel>
4  #include <QDial>
5  #include <QLCDNumber>
6  #include <QVBoxLayout>
7  #include <QHBoxLayout>
8  #include <QGridLayout>
9  #include <QSettings>
10 #include <QApplication>
11 #include <QCoreApplication>
12
13 #include "ConversionScreen.h"
14
15 ConversionScreen::ConversionScreen() : QWidget()
```

```
16 {
17     initSettings();
18     readSettings();
19     createScreen();
20 }
21
22 void ConversionScreen::createScreen()
23 {
24     QPushButton* quitBtn = new QPushButton("Quit");
25     QPushButton* saveBtn = new QPushButton("Save");
26
27     createCel();
28     createFah();
29
30     QGridLayout *mainLayout = new QGridLayout;
31     mainLayout->addWidget(quitBtn, 0, 0);
32     mainLayout->addWidget(saveBtn, 0, 1);
33     mainLayout->addLayout(cellLayout, 1, 0);
34     mainLayout->addLayout(fahLayout, 1, 1);
35     mainLayout->setSpacing(40);
36     mainLayout->setMargin(40);
37     setLayout(mainLayout);
38
39     slider->setFocus();
40
41     connect(quitBtn, SIGNAL(clicked()), qApp, SLOT(quit()));
42     connect(saveBtn, SIGNAL(clicked()), this, SLOT(saveSettings()));
43
44     setWindowTitle("Temperature Conversion");
45 }
46
47 void ConversionScreen::createCel()
48 {
49     slider = new QSlider(Qt::Vertical);
50     slider->setRange(0, 100);
51     slider->setValue(currentCelNum);
52     slider->setTickPosition(QSlider::TicksLeft);
53
54     QLabel* celLabel = new QLabel(QString::number(currentCelNum));
55
56     cellLayout = new QHBoxLayout;
57     cellLayout->addWidget(celLabel, 0, Qt::AlignRight);
58     cellLayout->addWidget(slider, 0, Qt::AlignLeft);
59     cellLayout->setSpacing(10);
```

```
60
61     connect(slider, SIGNAL(valueChanged(int)), celLabel, SLOT(setNum(int)));
62     connect(slider, SIGNAL(valueChanged(int)), this, SLOT(celToFah(int)));
63 }
64
65 void ConversionScreen::createFah()
66 {
67     QLCDNumber* lcdNum = new QLCDNumber(3);
68     lcdNum->setSegmentStyle(QLCDNumber::Filled);
69     lcdNum->display(currentFahNum);
70
71     dial = new QDial;
72     dial->setRange(32, 212);
73     dial->setValue(currentFahNum);
74     dial->setNotchesVisible(true);
75
76     fahLayout = new QVBoxLayout;
77     fahLayout->addWidget(lcdNum, 0, Qt::AlignBottom | Qt::AlignHCenter);
78     fahLayout->addWidget(dial);
79     fahLayout->setSpacing(10);
80
81     connect(dial, SIGNAL(valueChanged(int)), lcdNum, SLOT(display(int)));
82     connect(dial, SIGNAL(valueChanged(int)), this, SLOT(fahToCel(int)));
83 }
84
85 void ConversionScreen::celToFah(int celNum)
86 {
87     int fahNum = (celNum * 9 / 5) + 32;
88     dial->setValue(fahNum);
89 }
90
91 void ConversionScreen::fahToCel(int fahNum)
92 {
93     int celNum = (fahNum - 32) * 5 / 9;
94     slider->setValue(celNum);
95 }
96
97 void ConversionScreen::initSettings()
98 {
99     QApplication::setOrganizationName("MySoft");
100    QApplication::setApplicationName("Conversion");
101 }
102
103 void ConversionScreen::saveSettings()
```

```

104 {
105     currentCelNum = slider->value();
106     currentFahNum = dial->value();
107
108     QSettings settings;
109     settings.setValue("Temperature/CelNumber", currentCelNum);
110     settings.setValue("Temperature/FahNumber", currentFahNum);
111 }
112
113 void ConversionScreen::readSettings()
114 {
115     QSettings settings;
116     currentCelNum = settings.value("Temperature/CelNumber", 0).toInt();
117     currentFahNum = settings.value("Temperature/FahNumber", 32).toInt();
118 }

```

这一次我们先来编译运行一下这个例子，再来详细讲解它。编译的过程还和以前一样，运行后我们可以看到在 **Quit** 按钮旁边增加了一个 **Save** 按钮。试一试下面的步骤：

1. 拖动滑块或指针来改变当前的温度计数值；
2. 点击 **Save** 按钮；
3. 点击 **Quit** 按钮退出程序；
4. 再重新运行该程序，可以看到刚刚保存的数值被读取出来了；
5. 在目录 `$HOME/.config/MySoft/` 下可以找到所保存的配置文件 `Conversion.conf`，它所采用的也是 `INI` 文件的格式，内容如下所示：

```

[Temperature]
CelNumber=20
FahNumber=68

```

程序运行的示意图如下：

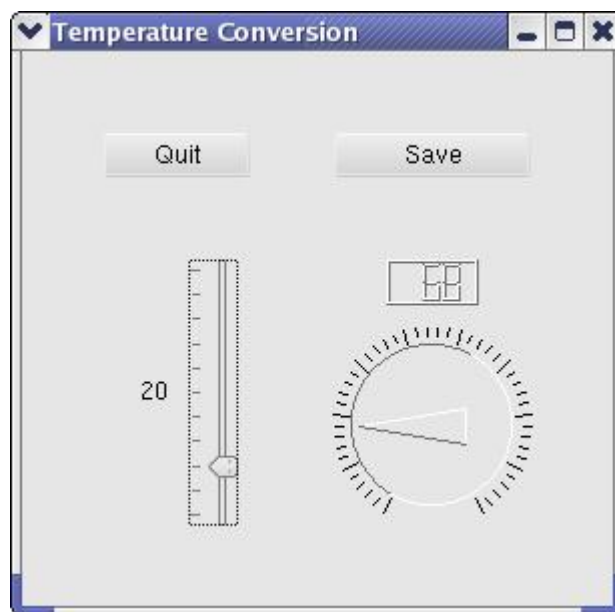


图 10.8 保存当前的数值

来看看我们在上一小节的基础上增加了些什么：首先在 25 行我们增加了一个 Save 按钮，并且在 32 行把它安排在网格布局容器的第 1 行第 2 列，即(0, 1)坐标，后面的第 42 行为它增添了一个连接，当用户点击这个按钮时，就会发射 clicked()信号，于是连接到这个信号的槽 saveSettings()被执行，以真正保存当前数值。

顺着这个思路我们来看看槽 saveSettings()，它在 103 行—111 行，首先得到当前的读数，然后构造一个 QSettings 对象，并用 setValue()方法将读数存储起来。注意我们所用的 QSettings 的构造函数是最简单的一个，因为我们在 initSettings()函数（97 行—101 行）中调用了 QApplication::setOrganizationName() 和 QApplication::setApplicationName() 来设定 organization 和 application。存储时用到的 setValue()方法比较简单而且方便，函数原型如下：

❶ void setValue ( const QString & key, const QVariant & value );

第一个参数为 key，第二个参数则是对应 key 所保存的值。在 Qt/X11 中，由于我们所用的存储格式总是 INI 文件格式，我们可以简单的用一个字符串来表示 key，也可以采用“group/key”的格式。如果省略前面的 group 的话，Qt 会将这个 key 自动加到名为“General”的 group 中。在 109 和 110 行我们使用“Temperature/FahNumber”作为第一个参数，正如我们在配置文件 Conversion.conf 中所看到的，这样生成了一个“Temperature”的 group，两个 key CelNumber 和 FahNumber 的值也都被保存在这个 group 中。第二个参数的类型为 QVariant，QVariant 可以用来表示 Qt 中的很多数据类型并提供方便的相互转换功能，因此大多数的设置都可以用 QSettings 类来直接保存。

读取所保存数值的函数 readSettings()在 113-118 行，与保存时非常类似，我们同样用 QSettings 的构造函数中最简单的一个来创建一个 QSettings 对象，用 QSettings::value()来读取数值。由于这个函数返回的是一个 QVariant 对象，我们需要用 toInt()将它转换为 int 型数值。QSettings::value()的第一个参数是 key，第二个参数是默认值，即当读不到这个 key 时返回一个默认值，我们将默认值分别设置为 0 和 32，即摄氏 0 度。

我们在 ConversionScreen 类的构造函数中的第 17、18 行调用了 initSettings()和 readSettings()，在主窗口显示之前先设置好 organization 和 application，并读取上一次保存的值。在第 51、54、69 和 73 行将读到的数值分别设置到相关的部件上，注意如果上一次有保存的话我们读到所保存的数值并作相关设置，如果以前并没有保存，或者程序还是第一次运行，则根本不会有 Conversion.conf 整个文件，这时我们读取时返回的则是调用 QSettings::value()所给定的第二个参数，即分别为 0 和 32。

保存程序当前的一些设置是应用开发中经常遇到的需求，希望读者从我们的这个小例子基本了解了应用 QSettings 类来保存设置的方法。

## 10.4 本章小结

这一章我们主要来帮助大家了解 Qt/X11 的初步知识，包括它的授权问题，详细的安装过程，以及通过一个温度转换的小例子向大家介绍了利用 Qt 进行应用开发的一些基础知识。希望大家对 Qt 进行应用开发的一整套步骤都有所了解，比如如何利用 qmake 工具协助编译，如何连接信号和槽以及定义自己的槽等，同时到最后我们还简单介绍了应用 QSettings 类来保存设置的方法。

通过本章的学习，读者应该已经搭建了 Qt/X11 的开发环境，了解了 Qt 的编译过程，并可以尝试简单的 Qt 编程了。



## 10.5 常见问题

1. Qt/X11 自由版可以用于商业开发吗？

参考答案：

可以，但是你需要遵循 GPL 和 QPL。由于 GPL 要求你所开发的产品也开发源代码，以及还有其他一些限制，所以多数公司不使用 Qt/X11 自由版来进行商业开发。一般来说商业开发购买 Qt 的商业版本比较合适一些。

2. 使用 QSettings 类时，可以将配置文件保存到我所希望的任意路径，而不是默认的 \$HOME/.config 或者/etc/xdg 吗？

参考答案：

当然可以。Qt 提供的构造函数中有：

```
QSettings ( const QString & fileName, Format format, QObject * parent = 0 )
```

第一个参数就是配置文件的全名（包括路径）。另外你还可以使用它的静态成员函数

```
void setPath ( Format format, Scope scope, const QString & path )
```

来进行设置，不过需要注意这个函数并不会修改已经存在的 QSettings 对象。

## 第 11 章 Qt 核心技术

本章学习目标：

- l 掌握信号和槽的各种用法
- l 了解 Qt 的元对象系统
- l 了解信号和槽的实现机制及其局限性
- l 理解 Qt 的对象树及其方便内存管理的特性
- l 掌握常用的布局管理方法
- l 理解 Qt 国际化的方法并熟悉翻译的过程

### 11.1 信号(Signals)和槽(Slots)

在上一章中我们应该对信号和槽的机制有了一个直观的认识，这一章我们来详细了解一下，因为它是 Qt 的核心特性，也是 Qt 区别于其它工具包的重要地方。

我们可以把信号和槽机制看成是 Qt 自行定义的一种应用于对象之间的通信的高级接口。要注意信号和槽并不是标准 C/C++ 语言的特性，我们可以认为它是 Qt 对 C++ 特性的一种扩展。采用信号和槽机制编写的代码不能被标准 C++ 编译器编译，而需要借助一个被称为 moc (Meta Object Compiler) 的 Qt 工具对信号和槽在编译之前进行预处理。

我们先来了解一下信号和槽机制所解决的问题以及其他的一些解决方案。

#### 11.1.1 常见的 GUI 组件通信方式

在图形用户界面(GUI)编程中，我们经常希望一个可视对象发生某种变化时通知另一个或几个对象，或者采用某种逻辑或调用某个接口来响应这个变化。例如，用户在点击某个按钮时，当前窗口需要发生某种变化，或者需要打开新的窗口，我们希望当前窗口能够收到按钮被点击的信号，并及时响应用户的需求。进一步的，即使没有用户的输入，当底层的数据发生变化时，也需要通知用户界面进行及时地更新。

##### 11.1.1.1 回调函数

对于类似以上的问题，早期的工具包使用一种被称作“回调”(callback)的通信方式来实现。回调是通过函数指针来实现的，即将一个函数指针（我们称作回调函数）传递给处理函数，处理函数在适当的时候调用回调函数。

常见的回调比如写快速排序函数的时候，即在参数中预先声明一个回调函数的指针，调用者需要自己准备一个函数来比较大小。C 语言的标准库函数中快速排序函数的原型如下：

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```

参数中的 base 是需要排序的数组，n 是数组的长度，size 是数组中所存储的对象的大小，而最后面的 int (\*cmp)(const void \*, const void \*) 则正是我们需要提供的用来比较数组成员

大小的回调函数。在 `qsort` 这个库函数的实现中，即通过调用 `cmp` 这个函数来比较数组 `base` 中成员的大小，成员

下面的代码示例说明了如何编写这个回调函数，以调用 `qsort` 来进行排序：

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

int cmpFunc(const void *a, const void *b);
int myArray[6] = { 5, 12, 4, 33, 1, 8 };

int cmpFunc(const void *a, const void *b)
{
    return *(int*)a - *(int*)b;
}

int main()
{
    qsort((void *) myArray, 6,
        sizeof(myArray[0]), cmpFunc);

    for (int i = 0; i < 6; i++)
    {
        printf("%d\n", myArray [i]);
    }

    return 0;
}
```

上面示例代码的调用关系我们可以用下图来表示：

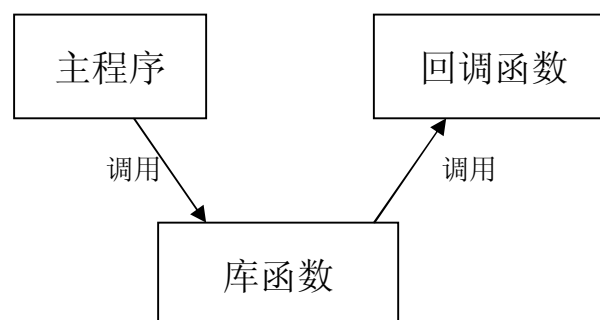


图 11.1 调用关系示意图

如果主程序与库函数之间采用层次结构的话，可以看出，在调用过程中既有上层对下层的调用（主程序调用库函数），也有下层对上层的间接调用（库函数调用回调函数），这种调用关系既没有增加下层对上层的依赖关系，又实现了灵活性，这正是很多时候我们需要回调函数的原因。

早期的一些开发包如 **Win32 SDK** 中即有不少的回调函数，比如 **Win32** 程序中用来处理

窗口过程的 WndProc 函数:

```
LPRESULT CALLBACK WndProc(HWND hWnd, UINT message,  
                           WPARAM wParam, LPARAM lParam)
```

WndProc 函数在 Win32 程序中需要开发者自己定义,来处理窗口收到的各种消息,Windows 系统的消息处理机制会自动调用 WndProc 函数来进行处理。可以看出,对比上面的快速排序的例子,这里的回调稍有不同——主程序的一部分实际上已经由系统自动完成了,开发者只需要准备好回调函数就可以了。

由于回调是通过函数指针来实现的,而根据 C 语言的特性,不同类型的指针之间是可能被转换的,这不是一种类型安全的方式——我们无法确定处理函数是否使用了正确的参数来调用回调函数——有些时候这种错误是致命的,比如参数在压栈的时候发生失误,这时便很可能导致整个进程的退出。另外,回调函数和处理函数间的联系非常紧密,而且这种这无疑会增加系统的耦合程度,对于大型系统的开发是非常不利的。

### 11.1.1.2 面向对象的回调

在面向对象的开发中,我们可以用虚拟和继承以及模版来实现简单的回调机制,比如在父函数中定义一个虚函数来当作回调函数,在主程序中可以调用这个虚函数,而真正的实现则由子函数重写这个虚函数来完成。这种方法可以在一定程度上避免函数指针带来的危险。

比如上面的 qsort() 函数,如果采用虚拟和继承来实现,我们可以作如下的设计:定义基类 CmpObj 及其成员函数 int CmpObj::cmpFunc(const CmpObj & a, const CmpObj& b) 来比较大小,在对不同的一组对象进行快速排序时,通过继承基类 CmpObj 并在子类中重写 cmpFunc 方法即可。当然在这个问题上这样做带来的好处是有限的,我们仅仅是以此为例说明一下如何重写虚函数来代替回调,以避免函数指针带来的类型安全问题。

我们还可以对比一下在 STL 中提供的 sort() 函数与上面的 C 标准库中的 qsort()。STL 中提供的 sort() 函数充分利用了模版特性,其原型如下:

```
template<class _RandomAccessIter, class _Compare>  
void sort(  
    _RandomAccessIter __first, // 需排序数据的第一个元素位置  
    _RandomAccessIter __last,  // 需排序数据的最后一个元素位置(不参与  
    排序)  
    _Compare __comp           // 排序使用的比较算法(可以是函数指针、函数对象  
    等)  
);
```

这种采用模版的方法要求使用迭代器的对象来作为前两个参数以便遍历需要排序的数据,而第三个参数的作用则类似于回调函数。事实上与采用虚函数的方法相比,STL 中的方法可以看作是用模版来代替继承——这正是模版的作用之一。

尽管我们可以上面的一些方法来实现回调机制,但在开发复杂的 GUI 系统时,基于开发效率和运行效率的考虑,上面的这些方法并不是非常适用。现代的 GUI 系统中纷纷采用了更加广义上的回调机制,比如消息机制、事件驱动等,来简化应用程序的开发并提高软件复用的程度。这些机制不再需要处理复杂的函数指针,其结构体系也更为完整,但它们都是采用类似回调的这种思路,只是在它们的框架中已经添加了一些功能来隐藏真正的回调过

程，比如 MFC 的框架中就定义了很多宏来支持消息映射机制，Java GUI 中则用 Observer 模式实现了 Listener 机制，这些都可以看作一种广义上的更加便于使用的回调。

另外，著名的“四人帮”(GOF, Gang of Four)所著的《设计模式》一书中描述了用 Command 模式来实现回调机制的方法[12]，并且能够支持“撤销”功能，有兴趣的读者可以作为参考。

## 11.1.2 Qt 中的信号和槽(Signals and Slots)

### 11.1.2.1 信号和槽历史和所带来的优点

大约在 1992 年的时候，Qt 的两位创始人之一 Eirik 在开发 Qt 的过程中，萌生了 Signal-Slot 的想法来解决 GUI 开发中复杂的通信问题，接着 Haavard 通过扩展 C++ 的特性，实现了这种机制。这种灵活的机制被后来的不少开发包所采用，包括 Gtk+，boost 库\*（注 1）等都提供了类似的概念和实现。

\*注 1: boost 是一个准标准库，相当于 STL 的延续和扩充，它的设计理念和 STL 比较接近，都是利用泛型让复用达到最大化。不过对比 STL，boost 更加实用。STL 集中在算法部分，而 boost 包含了不少工具类，可以完成比较具体的工作。参见 [www.boost.org](http://www.boost.org)

Qt 为信号和槽的机制设计了自己的语法来扩展 C++ 的特性，并利用一个称为 moc (Meta Object Compiler) 的 Qt 工具来自动将这种 Qt 自定义的语法转换为 C++ 代码。信号和槽能携带任意数量和任意类型的参数，它们是类型完全安全的，而且信号和槽之间的联系非常灵活，而耦合却比较松散。定义各种类的开发者互相之间并不需要知道别人有哪些信号或者槽，这些类只需要考虑要提供哪些信号和槽，而且一个发射信号的对象不用考虑哪个槽会接收这个信号，接收信号的槽的所在对象也不知道要连接的信号是哪个对象发射的；而连接信号和槽的人则往往不需要关心信号或者槽的实现细节，他们只需要清楚将哪些信号连接到哪些槽就可以了，这样可以为大型开发中的分工和合作提供很大的方便，可以说信号和槽是一种比较适合面向对象开发的通信机制。

信号和槽的机制可以简单地描述为，当一个特定的事件发生时，一个或几个被指定的信号就被发射，槽则是响应信号的函数，如果存在一个或几个槽和该信号相连接，那在该信号被发射后，这个（些）槽就会立刻被执行。

### 11.1.2.2 信号

信号在 Qt 中由关键字 `signals` 来声明，它的形式类似于一个函数，但是返回值只能是 `void` 类型，而且对初学者非常重要的一点是：信号只需要声明即可，千万不要在 `.cpp` 文件中去实现在头文件中声明的信号。

信号往往在对象的内部状态发生改变时被发射出去，以通知它所连接的槽，并且发射信号只能由定义了一个信号的类和它的子类才能来完成。发射信号的关键字是 `emit`，注意这些关于信号和槽的关键字（包括 `signals`, `emit` 及后面的 `slots` 等）只是 Qt 所定义的语法，并非标准 C++ 中的关键字，它们都会在由 C++ 编译器编译之前被 moc 工具转换成标准 C++ 的语法。我们将在下一节讲述 Qt 的元对象模型时，来仔细探讨这些关键字的真正定义，以及为什么信号只能返回 `void`，不能由开发人员去实现等问题。

### 11.1.2.3 槽

槽是一种可以用来连接到信号的成员函数，信号发射时，它所连接的相应的槽将被执行，不过对于槽本身来说，它并不知道是否被其它信号连接。信号和槽的连接可以非常灵活——我们可以把一个信号和一个槽进行单独连接，这时槽会因为该信号被发射而被执行；也可以把几个信号连接在同一个槽上，这样任何一个信号被发射都会使得该槽被执行；也可以把一个信号和多个槽连接在一起，这样该信号一旦被发射，与之相连接的槽都会被马上执行（但需要特别注意的是，这些槽执行的顺序是不确定的，并且也不可以指定）；我们甚至还可以把一个信号和另一个信号进行连接，这样，只要第一个信号被发射，第二个信号立刻就被发射。

我们可以来回顾一下上一章中温度转换的小例子中用到的连接：

```
connect(slider, SIGNAL(valueChanged(int)), celLabel, SLOT(setNum(int)));  
connect(slider, SIGNAL(valueChanged(int)), this, SLOT(celToFah(int)));
```

这里我们将 `slider` 对象发射的同一个信号 `valueChanged()` 连接到了两个不同的槽，即 `setNum()` 和 `celToFah()`。这两个槽在 `valueChanged()` 信号发射后都将被执行，不过它们的执行顺序是不确定的，并且也不可以指定。

除了与信号连接之外，槽实际上也可以作为普通的成员函数来使用，它可以被直接调用，这时候我们需要考虑它的访问权限，与普通的成员函数类似，我们可以将槽声明为 `public`、`protected` 或者 `private`，声明为 `public slots` 的槽才可以被其他类所直接调用，声明为 `protected slots` 的槽则只能被这个类和它的子类直接调用，而声明为 `private slots` 的槽则只能被这个类本身所直接调用。但是需要注意当它们真正作为槽来使用，即连接它到某个信号时，这时我们定义的访问权限并没有任何效果，即使声明为 `private slots` 的槽也能被任意类的信号所连接。

这里注意比较一下前面提到的信号：信号并不能作为普通的函数来使用，定义它的访问权限也是没有意义的，无论任何情况，都只有定义这个信号的类和它的子类才能发射这个信号，这比较类似于访问权限是 `protected` 的情况。

我们还可以把槽定义为虚函数，Qt 的元对象系统能够识别这种多态特性，在连接信号的时候执行正确的槽。这在实践开发中可以提供很多便利，可能会经常用到。

信号和槽可以携带参数，但是要求它们的参数类型和个数都是一样的，如果信号携带的参数比槽的多，则多余的参数将会被忽略。在下一节中我们可以看到，由于 Qt 的元对象系统的特性，信号和槽携带参数的参数还有诸多限制，比如不能用模版，不能用函数指针，不能有默认值等。

### 11.1.2.4 信号和槽的效率

信号和槽机制的效率当然不如真正的回调那么快，因为 Qt 在背后需要作很多工作来支持它，需要牺牲一些效率来满足它们所提供的灵活性。尽管如此，在实际应用中这种稍微有点慢的情况经常可以被忽略。根据 Qt 的官方文档介绍，在不使用虚函数作为槽的情况下，采用信号和槽机制从发射信号到相应的槽执行时的时间，大约比直接调用这个函数的时间要慢十倍，从下一节中我们可以知道，这主要是定位连接对象所需的开销。这个时间开销实际上并不是很大，举个例子来说，它比任何一个“new”或者“delete”操作还要稍微少一些。当你执行一个字符串、向量或者列表操作时，如果需要“new”或者“delete”操作，信号和槽仅仅对一个完整函数调用的时间开销中的一个非常小的部分负责。在一台 i585-500 机

器上，你每秒钟可以发射 2, 000, 000 个左右连接到一个接收器上的信号，或者发射 1, 200, 000 个左右连接到两个接收器的信号。信号和槽机制的简单性和灵活性对于这点时间开销来说是非常值得的，你的用户甚至察觉不出来。

### 11.1.3 自定义信号和槽的小例子

在 Qt 提供的类库中已经有了很多定义好的信号和槽，在我们上一章的小例子中用到了一些 Qt 已经定义好的信号和槽，同时也自己定义了几个槽来处理一些程序逻辑，这里我们进一步向大家来演示如何定义和发射自己的信号，并且将多个信号连接到一个槽中，以及将一个信号连接到另一个信号。

我们总共定义三个类 **Sender**、**Mediator** 和 **Receiver** 来分别发送、转发和接收信号，因为这三个类都比较简单，我们把它们放在同一个头文件 `signals_slots.h` 中，如下所示：

```
1  #ifndef SIGNALS_SLOTS_H
2  #define SIGNALS_SLOTS_H
3
4  #include <QObject>
5
6  class Receiver : public QObject
7  {
8      Q_OBJECT
9
10 public:
11     Receiver(QObject *parent=0) : QObject(parent)
12     {
13     }
14
15 public slots:
16     void printNumber(int n);
17 };
18
19 class Mediator : public QObject
20 {
21     Q_OBJECT
22
23 public:
24     Mediator(QObject *parent=0) : QObject(parent)
25     {
26     }
27
28     void doSend();
29
30 signals:
31     void send(int);
32 };
```

```

33
34 class Sender : public QObject
35 {
36     Q_OBJECT
37
38 public:
39     Sender(QObject *parent=0) : QObject(parent)
40     {
41     }
42
43     void doTransmit();
44
45 signals:
46     void transmit(int);
47 };
48
49 #endif // SIGNALS_SLOTS_H

```

先从我们已经比较熟悉的自定义的槽开始，我们在类 `Receiver` 中声明了一个 `public` 的槽，即第 16 行的 `void printNumber(int n)`，用来打印出所接收到的整型参数。类 `Mediator` 和 `Sender` 非常类似，它们都定义了各自的信号（分别在 31 行和 46 行），并且定义了公有成员函数用来发射信号。注意所有包含信号或者槽的类必须在它们的声明中用到宏 `Q_OBJECT`，因此这三个类的定义中分别在第 8 行、第 21 行和第 36 行都加上了宏 `Q_OBJECT`。

这三个类的实现也非常简单，我们把它们的实现以及 `main()` 函数都放在 `signals_slots.cpp` 中。如下所示：

```

1  #include <QApplication>
2  #include <iostream>
3  #include "signals_slots.h"
4
5  void Receiver::printNumber(int n)
6  {
7      std::cout << "Recieved: " << n << std::endl;
8  }
9
10
11 void Mediator::doSend()
12 {
13     emit send(5);
14 }
15
16 void Sender::doTransmit()
17 {
18     emit transmit(10);
19 }

```



```

20
21 int main(int argc, char **argv)
22 {
23     QApplication a(argc, argv);
24
25     Receiver r;
26     Mediator m;
27     Sender s;
28
29     QObject::connect(&s, SIGNAL(transmit(int)), &r, SLOT(printNumber(int)));
30     QObject::connect(&s, SIGNAL(transmit(int)), &m, SIGNAL(send(int)));
31     QObject::connect(&m, SIGNAL(send(int)), &r, SLOT(printNumber(int)));
32
33     s.doTransmit();
34     m.doSend();
35
36     return 0;
37 }

```

我们自定义的槽 `printNumber` 非常简单，仅仅调用标准 C++ 的 `iostream` 来打印传入的整型参数。`doSend()` 和 `doTransmit()` 仅仅是发射各自的信号。注意我们在 `signals_slots.h` 中声明的信号 `send()` 和 `transmit()` 都不需要真正的实现，仅仅一个声明就足够了（下一节中我们可以看到自己实现自定义的信号所带来的问题）。另外我们要注意，只有定义信号的类才能发射这个信号，比如想要在类 `Mediator` 发射信号 `transmit()` 是行不通的。

在 `main` 函数中的第 29-31 行我们将 `transmit()` 信号分别连接到槽 `printNumber()` 和信号 `send()`，同时将 `send()` 信号也连接到了槽 `printNumber()`。分析一下这样做的结果：当 `transmit()` 信号发射时，由于槽 `printNumber()` 被连接到这个信号，它会被执行一次；同时 `transmit()` 信号的发射也会引起 `send()` 信号的发射，由于槽 `printNumber()` 也被连接到 `send()` 信号，这样它被执行第二次。

第 33 行和第 34 行分别发射了信号 `transmit()` 和 `send()`。注意发射信号所用的关键字是 `emit`。

同样的方法编译并运行：

```

# qmake -project
# qmake
# make
#
# ls
Makefile  moc_signals_slots.cpp  moc_signals_slots.o  signals_slots  signals_slots.cpp
signals_slots.h  signals_slots.o  signals_slots.pro
#
# ./signals_slots
Recieved: 10
Recieved: 10
Recieved: 5

```

注意 moc 工具为我们生成了 moc\_signals\_slots.cpp 来参与编译和链接。最后的运行结果证实了我们前面的分析：当 transmit()信号发射时，槽 printNumber()被执行了两次，这时参数的值是 doTransmit()函数中所传入的，为 10；而第三次执行则是 doSend()函数中发射了信号 send()并传入参数 5 所得到的结果。

希望读者朋友从这个简单的小例子了解到了如何定义和发射信号，以及如何将一个信号连接到另一个信号。这一节中我们详细的介绍了信号和槽的用法，下一节我们将在介绍 Qt 元对象系统（Meta-Object System）的基础上来仔细研究一下信号和槽到底是怎么回事。

## 11.2 Qt 对象模型

Qt 所提供的灵活利用的特性是建立在它的对象模型的基础上的，而 Qt 的对象模型所提供的最重要的机制便是上一节中所介绍的信号和槽，同时也提供了其他一些便利比如相对简单的内存管理，可查询和可设计的属性等。

我们先从 Qt 对象模型的基础——元对象系统（Meta-Object System）说起。

### 11.2.1 元对象系统（Meta-Object System）

元对象系统（Meta-Object System）在 Qt 中主要用来实现信号和槽的机制，以及运行时的类型信息和动态属性系统。

在前面的一些例子中我们实际上也提到了元对象系统。当一个类从 QObject 继承而来，并在它的声明中用到了宏 Q\_OBJECT 时便具备了应用元对象系统的条件。在使用 qmake 生成 Makefile 的过程中，qmake 工具会识别宏 Q\_OBJECT，因而在生成的 Makefile 中将调用 moc（Meta Object Compiler）工具来自动生成一个 cpp 文件，并将生成的 cpp 文件加入到编译和链接过程中。示意图如下：

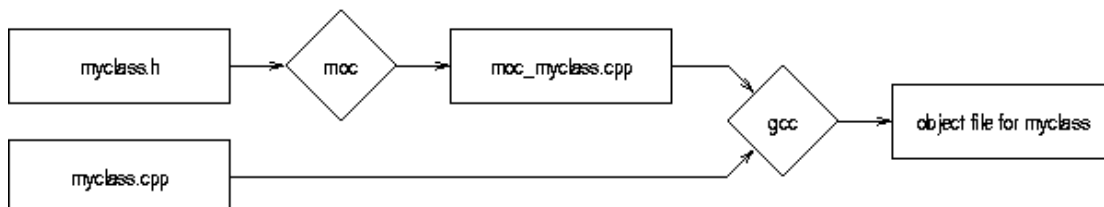


图 11.2 moc 及编译的过程

图中的 myclass.h 文件中包含有宏 Q\_OBJECT，于是在 Makefile 中将有步骤调用 moc 工具自动生成 moc\_myclass.cpp 文件，这个 cpp 文件与类的实现文件 myclass.cpp 一起参与到后面的编译和链接过程，直到生成最后的可执行文件或者库。这些过程都可以由 qmake 和 make 自动完成，所以在前面的例子中我们都没有详细介绍。

那么宏 Q\_OBJECT 除了作为一个标志以便 moc 工具可以识别之外，它本身到底被定义成什么了呢？我们来看看 Qt 的源代码。在解压缩我们所下载的 Qt 安装包 qt-x11-opensource-src-4.2.2.tar.gz 之后，在其中的 src/corelib/kernel/ 目录下可以找到 qobjectdefs.h 文件，这个文件中包含了 Q\_OBJECT 宏的定义：

```
97 #define Q_OBJECT \
98 public: \
99     static const QMetaObject staticMetaObject; \
100     virtual const QMetaObject *metaObject() const; \
```

```

101     virtual void *qt_metacast(const char *); \
102     QT_TR_FUNCTIONS \
103     virtual int qt_metacall(QMetaObject::Call, int, void **); \
104 private:

```

这里的内容有些复杂，对于初学者来说，知道 `Q_OBJECT` 宏为我们自动添加了一些成员变量和成员函数就足够了，再加上了解了 `moc` 工具会为我们自动生成 `moc_myclass.cpp` 文件，当遇到问题时，根据这些线索就可能找到问题的答案。这些隐藏在背后的机制并不一定需要开发者完全了解，毕竟 Qt 已经为我们自动实现了这些功能。

对于希望深入了解 Qt，或者甚至可能需要去修改 Qt 源代码以适用自己开发的读者朋友，我们在这些为大家提供一些较为深入的介绍——Trolltech 公司并没有相关的文档来介绍这些，我们只有通过自己阅读 Qt 源代码来理解，因此下面的内容可能较为晦涩难懂，初学 Qt 的读者如果看不懂的话完全可以先略过，等有了更多的使用经验之后回头再来看看，也许会有所收获。

我们首先来详细分析一下 `Q_OBJECT` 宏的内容。第 99 行声明了一个 `QMetaObject` 类的静态成员对象，下面的第 100 行、101 行和 103 行则重写（override）了三个虚函数，注意这三个虚函数最早是在 `QObject` 中声明的，在 `QObject` 类的定义中，同样使用了宏 `OBJECT`，我们可以在 `qobject.h` 文件中找到下面的代码，在第 99 行用到了宏 `OBJECT`。

```

97  class Q_CORE_EXPORT QObject
98  {
99      Q_OBJECT
100      Q_PROPERTY(QString objectName READ objectName WRITE setObjectName)
101      Q_DECLARE_PRIVATE(QObject)
    ...
}

```

我们甚至可以在 Qt 的文档中都找到 `QObject` 类的成员函数：

```
virtual const QMetaObject *metaObject() const;
```

而它完全是由宏 `OBJECT` 和 `moc` 工具来实现的，`qobject.h` 中并没有显式的定义它，而是将它隐藏在宏 `OBJECT` 中。上面提到的宏 `OBJECT` 展开时的几个重写的虚函数都是这种情况，102 行的 `QT_TR_FUNCTIONS` 如果展开后同样是几个虚函数，它们与 Qt 国际化的实现机制相关，我们留到国际化的相关章节再来详细说明。

读者可能已经注意到宏 `OBJECT` 展开后最重要的是 99 行的 `QMetaObject` 静态对象 `staticMetaObject`，它包含了这个类的很多重要信息。我们来看看 `QMetaObject` 类的定义：

```

211 struct Q_CORE_EXPORT QMetaObject
212 {
    ...
354     struct { // private data
355         const QMetaObject *superdata;
356         const char *stringdata;
357         const uint *data;
358         const QMetaObject **extradata;
359     } d;
360 };

```

`QMetaObject` 类的定义比较长，我们先来看看它所包含的数据结构——从 355 行到 358

行，各个结构体成员的作用如下：

`const QMetaObject * superdata` —— 指向父类的 `QMetaObject` 对象的指针；  
`const char * stringdata` —— 包含该类本身的相关信息(包含类名，信号名，槽名，属性名，枚举名，变量名等)的字符串；  
`const uint * data` —— 数据信息，结合 `stringdata` 来取得类名，信号名，槽名等  
`const QMetaObject ** extradata` —— 附加信息，目前一般为 0，可能是保留给将来扩展时所用

实际上第一个和第四个都是递归定义的指针，最主要的是中间的两个 `stringdata` 和 `data` 指针，理解了它们怎么结合运用就很比较理解 `QMetaObject` 类了。`stringdata` 指针实际上就是由数个字符串连接而成的长字符串，而 `data` 指针则看成是一个整型数组。这两个指针实际上还与一个结构体 `QMetaObjectPrivate` 相关，它定义在 `qmetaobject.cpp` 中：

```
150 struct QMetaObjectPrivate
151 {
152     int revision;
153     int className;
154     int classInfoCount, classInfoData;
155     int methodCount, methodData;
156     int propertyCount, propertyData;
157     int enumeratorCount, enumeratorData;
158 };
159
160 static inline const QMetaObjectPrivate *priv(const uint* data)
161 { return reinterpret_cast<const QMetaObjectPrivate*>(data); }
```

这个结构体用来解析 `data` 数组的前 10 个数，154 行—157 行非常类似，前一个成员表示个数，后一个则表示偏移量，我们在后面结合实际例子来详细说明。在紧接着结构体的定义之后，第 160 行的函数即用来作从 `data` 数组到结构体 `QMetaObjectPrivate` 的转换。事实上每个 `data` 数组都是由 `moc` 工具所自动生成，前 10 个数都是按照 `QMetaObjectPrivate` 的成员的含义而生成。我们来看看实际的例子，在上一节中 `moc` 工具为我们生成了 `moc_signals_slots.cpp`：

```
19 static const uint qt_meta_data_Receiver[] = {
20
21     // content:
22     1,          // revision
23     0,          // classname
24     0,    0, // classinfo
25     1,    10, // methods
26     0,    0, // properties
27     0,    0, // enums/sets
28
29     // slots: signature, parameters, type, tag, flags
30     12,    10,    9,    9, 0x0a,
31
32     0          // eod
```

```

33 };
34
35 static const char qt_meta_stringdata_Receiver[] = {
36     "Receiver\0\0n\0printNumber(int)\0"
37 };
38
39 const QMetaObject Receiver::staticMetaObject = {
40     { &QObject::staticMetaObject, qt_meta_stringdata_Receiver,
41       qt_meta_data_Receiver, 0 }
42 };
43
44 const QMetaObject *Receiver::metaObject() const
45 {
46     return &staticMetaObject;
47 }

```

静态成员变量 `staticMetaObject` 在 39 行被定义，在必要时可由 44 行的 `metaObject()` 函数获得。第 39 行定义的时候分别给定了指向父类的 `QMetaObject` 对象、`stringdata` 字符串和 `data` 数组。`stringdata` 字符串即 `qt_meta_stringdata_Receiver` 变量在 35 行，由 `moc` 工具扫描 `signals_slots.h` 之后自动生成，它真正包含了类中的许多信息，多个字符串之间用“\0”隔开，这些字符串需要一定的规则来进行解析，这就要用到 `data` 数组了。

从第 19 行到第 33 行，`data` 数组被初始化，注意 22-27 行分别对应了结构体 `QMetaObjectPrivate` 的各个成员，这有些类似于文件系统中的文件头或者网络传输中的包的头信息。在 `signals_slots.h` 文件中我们只是声明了槽 `printNumber`，因此头信息中有很多 0，其中的 `revision` 基本不用可以先略过；`classname` 是类的名字，值为 0，表示在 `stringdata` 字符串即 `qt_meta_stringdata_Receiver` 中从一开始保存的就是类名，即 `Receiver`。我们略过这里没有用到的 `classinfo`、`properties` 和 `enums/sets`，来看看 `methods`：两个值 1 和 10 分别对应结构体 `QMetaObjectPrivate` 中的 `methodCount` 和 `methodData`，即 `method` 个数和后面的 `method` 数据区的偏移量。在 `Receiver` 类中我们仅仅定义了一个槽 `printNumber`，因此 `methodCount` 的值为 1，而偏移量我们可以数一数到第 30 行即 `method` 数据区，是不是前面正好有 10 个数字呢？

请注意这里的 **method** 并不是指所有的成员函数，而是仅仅包括元对象系统所需要的信号和槽及与属性等相关的方法。在本节的上下文中所提到的 **method**，其含义都在此范畴。

接着来看 `method` 数据区，在 29 行 `moc` 工具为我们提供了注释，第 30 行的 5 个数分别是 `signature`、`parameters`、`type`、`tag` 和 `flags`，我们主要关心 `signature` 和 `flags`，`signature` 的值为 12，意味着我们从 `qt_meta_stringdata_Receiver` 中的第 12 位去取槽的名字，从字符串 `"Receiver\0\0n\0printNumber(int)\0"` 中第 12 位开始，直到遇到“\0”为止，可以取得“`printNumber(int)`”，这便是槽的名字。`flags` 的值决定了 `method` 的类型和访问权限，在 `qmetaobject.cpp` 中有相关的定义：

```

134 enum MethodFlags {
135     AccessPrivate = 0x00,
136     AccessProtected = 0x01,
137     AccessPublic = 0x02,
138     AccessMask = 0x03, //mask
139

```

```

140     MethodMethod = 0x00,
141     MethodSignal = 0x04,
142     MethodSlot = 0x08,
143     MethodTypeMask = 0x0c,
144
145     MethodCompatibility = 0x10,
146     MethodCloned = 0x20,
147     MethodScriptable = 0x40
148 };

```

在 `moc_signals_slots.cpp` 中槽 `printNumber(int)` 的 `flag` 值为 `0x0a`，这实际上等同于 `0x02+0x08`，即 `AccessPublic + MethodSlot`，表示这是定义为公有的槽。通过与 `AccessMask` 或者 `MethodTypeMask` 的“按位与”操作（即“&”操作），这个 `flag` 值可以还原用来是公有还是私有，是信号、槽还是 `MethodMethod` 类型。比如：

`0x0a & 0x03 (AccessMask) = 1010 & 0011 = 0x02`，可以判断它是公有的；同样

`0x0a & 0x0c (MethodTypeMask) = 1010 & 1100 = 0x08`，可以判断它是槽。

我们还可以查看一下 `moc_signals_slots.cpp` 中为 `Mediator` 类生成的 `stringdata` 和 `data`：

```

70 static const uint qt_meta_data_Mediator[] = {
71
72     // content:
73     1,          // revision
74     0,          // classname
75     0,    0, // classinfo
76     1,    10, // methods
77     0,    0, // properties
78     0,    0, // enums/sets
79
80     // signals: signature, parameters, type, tag, flags
81     10,    9,    9,    9, 0x05,
82
83     0          // eod
84 };
85
86 static const char qt_meta_stringdata_Mediator[] = {
87     "Mediator\0\0send(int)\0"
88 };

```

内容基本是类似的，只是数值稍有变化，`flag` 值变成了 `0x05`，这同样可以被解析成 `0x01 + 0x04`，即 `AccessProtected + MethodSignal`。

我们可以稍微总结一下这种数据结构：

1. `stringdata` 是一个字符串，其中存放所谓的元对象信息，并通过“\0”将各个部分分隔开来，以便于读取；
2. `data` 是一个无符号整型数组，这个数组的前 10 个数保存所谓的头信息，与结构体 `QMetaObjectPrivate` 中的各个成员相对应，并可以互相转换；之后则分别是 `classinfo`、`methods`、`properties` 和 `enums/sets` 的数据区（如果存在的话），数据区的偏移量和大小由头信息中的数值所决定（比如我们可以认为 `methodCount` 的

值就决定了 `method` 数据区的大小，因为每一个 `method` 在数据区中占用了 5 个数字，则总共占用 `methodCount * 5` 个数字，余者可类推）。

了解这些数据结构之后，可能我们已经能够自己编写代码来解析 `stringdata` 中所包含的信息了。可能有读者会产生疑问，为什么我们要用这么复杂的规则和数据结构来保存和解析这些所谓的元对象信息呢？在 `QMetaObject` 中多定义一些成员变量不是同样可以实现吗？事实上这种采用 `data` 和 `stringdata` 结合来保存和读取类的相关信息的方法是在 Qt4 中新引入的，在之前的 Qt 版本中我们可以看到采用更多的成员变量来实现的方法，比如下面的代码片段即来自于 Qt3.3 版的 `qmetaobject.h` 文件中的 `QMetaObject` 类的定义：

```
private:
    QMemberDict *init( const QMetaData *, int );

    const char    *classname;      // class name
    const char    *superclassname; // super class name
    QMetaObject   *superclass;     // super class meta object
    QMetaObjectPrivate *d;         // private data for...
    void *reserved;                // ...binary compatibility
    const QMetaData *slotData;     // slot meta data
    QMemberDict *slotDict;         // slot dictionary
    const QMetaData *signalData;   // signal meta data
    QMemberDict *signalDict;       // signal dictionary
    int signaloffset;
    int slotoffset;
```

这种实现方法应该说是一种比较自然的方法，看起来要简单易懂一些，读者如有兴趣下载 Qt3.3 版的源代码来作为参考。当然，Qt4 中的实现更加紧凑，在理解了之后其实运用起来也非常灵活，作为一种机制的内部实现，牺牲一些可读性是值得的，毕竟大多数的开发者并不一定需要对这种实现机制非常清楚。

我们接着来看在 `qmetaobject.cpp` 中是如何编写代码来解析 `stringdata` 的，我们挑选一个典型的函数 `QMetaObject::indexOfSlot(const char *slot)`：

```
451 int QMetaObject::indexOfSlot(const char *slot) const
452 {
453     int i = -1;
454     const QMetaObject *m = this;
455     while (m && i < 0) {
456         for (i = priv(m->d.data)->methodCount-1; i >= 0; --i)
457             if ((m->d.data[priv(m->d.data)->methodData + 5*i + 4] & MethodTypeMask)
458                 == MethodSlot
459                 && strcmp(slot, m->d.stringdata
460                     + m->d.data[priv(m->d.data)->methodData + 5*i]) == 0) {
461                 i += m->methodOffset();
462                 break;
463             }
464         m = m->d.superdata;
```

```

465     return i;
466 }

```

这个函数的目的是通过槽的名字找到并返回槽的索引，如果找不到则返回-1。首先我们进入一个大的 while 循环，455 行和 463 行表示如果在该类中找不到，则继续从该类的父类中查找，还找不到则继续从父类的父类中查找，直到顶层的类。

while 循环之下是 456 行的 for 循环，i 被初始化为 `priv(m->d.data)->methodCount-1`，即 method 的数目减 1，并且每循环一轮，i 自减 1，一直到 i 变成 0 为止。在循环体中每一轮都检测 method 数据区的一行是不是所寻找的槽，并且从最下面的一个 method 开始比对。

第 457 行用来检查这个 method 的 flag 值是不是槽，这里 `priv(m->d.data)->methodData` 的值是 method 数据区的偏移量，加上 `5*i` 之后则这个偏移值到了第 i+1 个 method 的开头，在加上 4 则可以取得该 method 的 flag 值。然后再与 `MethodTypeMask` 相与，这是我们前面介绍过的解析的方法，与 `MethodSlot` 即可以判断这个 method 是不是一个槽。

查询的另外一个条件，即槽的名字是不是匹配，在 458 行和 459 行用 `strcmp` 函数来判断，它的第一个参数是所要查询的槽的名字，第二个参数则需要我们从 `stringdata` 中去解析，这需要结合 `data` 数组中的信息，来确定从 `stringdata` 中哪一个位置开始提取字符串。与前面的 457 行类似，只是我们这次找的不是 flag，而是 signature，因而从 `m->d.data[priv(m->d.data)->methodData + 5*i]` 可直接获得数值。这里我们应该明白为什么要用“\0”来分隔 `stringdata` 了吧，类似 `strcmp`，很多 c 语言中与字符串处理相关的函数在比较或识别字符串的时候遇到“\0”即认为结束，这样我们可以方便的提取某一个串。

第 460 行是找到了之后的动作，用 `i + m->methodOffset()` 来作为最后的返回值，`m->methodOffset()` 实际上是所有父类的 method 的个数，在 Qt 的元对象信息中子类的 method 放在所有父类的 method 之后，在计算索引（包括计算信号或者槽以及所有的 method 的索引）的时候需要加上这个值。在找到了之后我们通过 461 行的 `break` 语句程序退出 for 循环，并且由于 i 的值将  $\geq 0$ ，也会退出 while 循环，程序将返回索引值 i。

`QMetaObject::indexOfSlot()` 是 `QMetaObject` 类中一个很常用的函数，在下一小节中我们将看到，它在信号和槽的实现中被用来定位槽的位置。类似的在 `QMetaObject` 中还有一些重要的成员函数，在了解了这种结合 `stringdata` 和 `data` 的实现机制之后有兴趣的读者可自行阅读相关代码。

## 11.2.2 信号和槽机制的实现

在充分理解了 `QMetaObject` 类之后，我们来看看如何利用 `QMetaObject` 类提供的信息来实现信号和槽。我们可以简单的猜想实现信号和槽机制所需要的一些步骤：

1. 用 `connection()` 建立连接的时候需要保存这一个连接的相关信息
2. 发射信号之后，需要通过前面保存的连接来查找信号所对应的槽，并且执行它

同样这种内部实现机制是不会有官方文档详细讲述的，我们还是通过 Qt 的源代码来探讨一下吧。

### 11.2.2.1 用 `connection()` 建立连接

`connection()` 是 `QObject` 的成员函数，在 `qobject.h` 中可以找到它的原型：

```

174 static bool connect(const QObject *sender, const char *signal,

```



```

175         const QObject *receiver, const char *member, Qt::ConnectionType =
176 #ifdef QT3_SUPPORT
177         Qt::AutoCompatConnection
178 #else
179         Qt::AutoConnection
180 #endif
181     );

```

回顾一下我们在上一节中的连接实例，比如 `signals_slots.cpp` 中的代码：

```

29   QObject::connect(&s, SIGNAL(transmit(int)), &r, SLOT(printNumber(int)));
30   QObject::connect(&s, SIGNAL(transmit(int)), &m, SIGNAL(send(int)));

```

对照上面的 `connect()` 函数的原型，第一个和第三个参数都是指向 `QObject` 对象的指针，最后一个参数 `ConnectionType` 一般都采用默认值，这些比较好理解；第二个和第四个参数要求是两个指向字符的指针即字符串，那么在我们的实例中如何将 `SIGNAL(transmit(int))` 和 `SLOT(printNumber(int))` 变成字符串呢？这就是宏 `SIGNAL` 和 `SLOT` 所起的作用了，在 `qobjectdefs.h` 中我们可以找到如下的定义：

```

147 #define METHOD(a)      "0"#a
148 #define SLOT(a)       "1"#a
149 #define SIGNAL(a)     "2"#a

```

注意在 `#define` 表达式中，参数左边的符号 `"#"` 表示宏展开时，这个参数将加上一对双引号，比如常见的 C 语言编程中用到的例子：

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

这个宏 `dprint` 可以更加方便的输出打印信息。当使用 `dprint(abc)` 时，宏被展开为：

```
printf("abc" " = &g\n", abc);
```

由于两个字符串会自动合并起来，最后的结果就是我们所需要的：

```
printf("abc = &g\n", abc);
```

宏 `SIGNAL` 和 `SLOT` 的用法与上面类似，前面的数字 0,1,2 用来标记是哪一种，而后面就将宏所包含的参数展开成带引号的字符，所以比如 `SIGNAL(transmit(int))` 展开后就会变成 `"1 transmit(int)"`。这样我们就可以理解第二个和第四个参数的含义了。

那么 `connect()` 函数究竟是怎么实现的呢？这个函数比较大，共有 200 多行，为节省篇幅我们仅仅来分析其中的一些关键部分：

```

2368 bool QObject::connect(const QObject *sender, const char *signal,
2369                        const QObject *receiver, const char *method,
2370                        Qt::ConnectionType type)
2371 {
2372     ...
2396     QByteArray tmp_signal_name;
2397
2398     if (!check_signal_macro(sender, signal, "connect", "bind"))
2399         return false;
2400     const QMetaObject *smeta = sender->metaObject();
2401     ++signal; //skip code
2402     int signal_index = smeta->indexOfSignal(signal);

```

```

2403     if (signal_index < 0) {
2404         // check for normalized signatures
2405         tmp_signal_name =
                QMetaObject::normalizedSignature(signal).prepend(*(signal - 1));
2406         signal = tmp_signal_name.constData() + 1;
2407         signal_index = smeta->indexOfSignal(signal);
2408         if (signal_index < 0) {
2409             err_method_notfound(QSIGNAL_CODE, sender, signal, "connect");
2410             err_info_about_objects("connect", sender, receiver);
2411             return false;
2412         }
2413     }
    ...

```

我们省略了前面检查输入的指针是否为空等代码，所列出的这一段代码的主要目的是通过调用 `indexOfSignal()` 来查询 `signal` 的索引。在 2398 行调用了 `check_signal_macro()` 函数来检查输入的参数是不是一个合法的 `signal` 格式的字符串，主要是检查第一个字符是否为“2”，即前面我们定义 `SIGNAL` 宏时所加上去的数字标记。所以在这里如果我们传入的参数不用宏 `SIGNAL` 包含起来的话很可能是通不过的。在 2401 行 `++signal` 的含义可能稍有些费解，其实了解了前面宏 `SIGNAL` 的定义的话就非常简单了，就是要略过标记的数字，直接跳到表示信号名的字符，这样 `signal` 变量就可以作为 `indexOfSignal()` 函数的参数了。`indexOfSignal()` 与我们前面分析过的 `indexOfSlot()` 的作用和实现都非常类似，它返回我们所查找的信号在元对象系统中的索引值。2403 行至 2413 行处理稍微复杂一些的情况——如果所传入的信号名比较复杂，可能带有数个参数，字符串之间可能有空格什么的，这时先用 `QMetaObject::normalizedSignature()` 函数来处理一下变成标准的格式，再调用 `indexOfSignal()` 查找，如果还找不到就返回 `false` 了。

下面的一段代码与上面类似，主要通过 `indexOfSlot()` 来查询 `slot` 的索引。

```

2415     QByteArray tmp_method_name;
2416     int membcode = method[0] - '0';
2417
2418     if (!check_method_code(membcode, receiver, method, "connect"))
2419         return false;
2420     ++method; // skip code
2421
2422     const QMetaObject *rmeta = receiver->metaObject();
2423     int method_index = -1;
2424     switch (membcode) {
2425     case QSLOT_CODE:
2426         method_index = rmeta->indexOfSlot(method);
2427         break;
2428     case QSIGNAL_CODE:
2429         method_index = rmeta->indexOfSignal(method);
2430         break;
2431     }

```

```

2432     if (method_index < 0) {
2433         // check for normalized methods
2434         tmp_method_name = QMetaObject::normalizedSignature(method);
2435         method = tmp_method_name.constData();
2436         switch (membcode) {
2437             case Q_SLOT_CODE:
2438                 method_index = rmeta->indexOfSlot(method);
2439                 break;
2440             case Q_SIGNAL_CODE:
2441                 method_index = rmeta->indexOfSignal(method);
2442                 break;
2443         }
2444     }
2445
2446     if (method_index < 0) {
2447         err_method_notfound(membcode, receiver, method, "connect");
2448         err_info_about_objects("connect", sender, receiver);
2449         return false;
2450     }
2451     ...
2479     QMetaObject::connect(sender, signal_index, receiver, method_index, type, types);
2480     const_cast<QObject*>(sender)->connectNotify(signal - 1);
2481     return true;
2482 }

```

前面的 `check_method_code()` 函数也检查 `slot` 字符串的第一个字符，这里比较有意思的一行是 2416 行，它将 `slot` 字符串的第一个字符转变成数字，对比前面的 `check_signal_macro()` 函数，在调用前是不需要这么做的，类似的转换在 `check_signal_macro()` 函数中完成，这种不一致的原因大概是后面的 2424 行还需要用到 2416 行得到的 `membcode` 吧。由于信号也可以被连接到信号，所以 2424 行到 2431 行对连接到的对象是信号还是槽分别作了处理，如果找不到的话同样用 `QMetaObject::normalizedSignature()` 处理之后再找一遍，还找不到则返回 `false`。2450 和 2479 行之间还有一些检查信号和槽的参数的一致性等相关代码我们省略了，跳到 2479 行可以看到我们前面的主要工作只不过是得到了 `signal_index` 和 `method_index`，然后再调用 `QMetaObject::connect()` 函数。

继续看 `QMetaObject::connect()` 函数，它也被定义在 `qobject.cpp` 中：

```

2731 bool QMetaObject::connect(const QObject *sender, int signal_index,
2732                          const QObject *receiver, int method_index, int type, int
2733                          *types)
2734 {
2735     QConnectionList *list = ::connectionList();
2736     if (!list)
2737         return false;
2738     QWriteLocker locker(&list->lock);
2739     list->addConnection(const_cast<QObject*>(sender), signal_index,
2740                       const_cast<QObject*>(receiver), method_index, type,

```

```
types);
2740     return true;
2741 }
```

这个函数的主要内容就是 2738 行的 `addConnection()`，即增加一个连接到 `list` 对象。那么 `list` 具体是什么呢？它在 2734 行用一个全局函数 `connectList()` 取得，相关的代码如下：

在 `qobject.cpp` 中实际上用宏 `Q_GLOBAL_STATIC` 定义了函数 `connectList()`：

```
105 Q_GLOBAL_STATIC(QConnectionList, connectionList)
```

在 `qglobal.h` 中与宏 `Q_GLOBAL_STATIC` 相关的代码：

```
1294 template <typename T>
1295 class QGlobalStatic
1296 {
1297 public:
1298     T *pointer;
1299     inline QGlobalStatic(T *p) : pointer(p) { }
1300     inline ~QGlobalStatic() { pointer = 0; }
1301 };
1302
1303 #define Q_GLOBAL_STATIC(TYPE, NAME) \
1304     static TYPE *NAME() \
1305     { \
1306         static TYPE this_##NAME; \
1307         static QGlobalStatic<TYPE > global_##NAME(&this_##NAME); \
1308         return global_##NAME.pointer; \
1309     }
1310
```

这些代码结合起来实际上就是定义了一个静态的 `QConnectionList` 对象，只不过将它封装到了类 `QGlobalStatic` 中，然后可以调用一个全局函数来返回，这种设计非常巧妙，比起单纯的使用全局变量要好多了，我们在平时的开发中也可以借鉴。需要说明的是上面的这段代码只是 `QT_NO_THREAD` 被定义时才使用的，需要支持线程时它的实现稍微复杂一点，不过其作用是同样的，我们不再赘述。

取得全局的 `QConnectionList` 对象之后，如何来保存这个连接的相关信息呢？这主要涉及到 `QConnection` 和 `QConnectionList` 两个类：

```
74 struct QConnection {
75     QObject *sender;
76     int signal;
77     QObject *receiver;
78     int method;
79     uint refCount:30;
80     uint type:2; // 0 == auto, 1 == direct, 2 == queued
81     int *types;
82 };
83 Q_DECLARE_TYPEINFO(QConnection, Q_MOVABLE_TYPE);
```

```

84
85 class QConnectionList
86 {
87 public:
88     QReadWriteLock lock;
89
90     typedef QMultiHash<const QObject *, int> Hash;
91     Hash sendersHash, receiversHash;
92     QList<int> unusedConnections;
93     typedef QList<QConnection> List;
94     List connections;
95
96     void remove(QObject *object);
97
98     void addConnection(QObject *sender, int signal,
99                       QObject *receiver, int method,
100                       int type = 0, int *types = 0);
101     bool removeConnection(QObject *sender, int signal,
102                           QObject *receiver, int method);
103 };

```

结构体 `QConnection` 用来保存一个连接的相关信息，而 `QConnectionList` 则用来存储所有的 `QConnection` 对象。如果我们在代码中加入一个 `connect()` 函数，则全局的 `QConnectionList` 对象调用 `addConnection()` 添加一个连接。注意上面的 91 行还定义了两个哈希表对象作为成员变量，在 `addConnection()` 函数中用它们来记录索引值。`addConnection()` 函数的实现也在 `qobject.cpp` 中，如下：

```

157 void QConnectionList::addConnection(QObject *sender, int signal,
158                                     QObject *receiver, int method,
159                                     int type, int *types)
160 {
161     QConnection c = { sender, signal, receiver, method, 0, 0, types };
162     c.type = type; // don't warn on VC++6
163     int at = -1;
164     for (int i = 0; i < unusedConnections.size(); ++i) {
165         if (!connections.at(unusedConnections.at(i)).refCount) {
166             // reuse an unused connection
167             at = unusedConnections.takeAt(i);
168             connections[at] = c;
169             break;
170         }
171     }
172     if (at == -1) {
173         // append new connection
174         at = connections.size();
175         connections << c;

```

```
176     }
177     sendersHash.insert(sender, at);
178     receiversHash.insert(receiver, at);
179 }
```

需要增加连接时，首先在 164 行—171 行检查这个连接是不是在 `unusedConnections` 中（当我们 `disconnect()` 一个连接时，它会保存在 `unusedConnections` 中），如果是则直接重用，如果不是则在 175 行添加新的连接，最后在 177 行和 178 行用哈希表保存与 `sender` 和 `receiver` 对象相关的索引 `at`，这样以后可以根据 `sender` 或 `receiver` 比较快的找到它们对应的连接。

总结一下，`connect()` 函数所作的工作，其实就是将这个连接相关的信息存储起来，主要步骤和相关的重要函数大概有：

- (1). 调用 `indexOfSignal()` 和 `indexOfSlot()` 分别得到所传入的信号和槽的索引值 `signal_index` 和 `method_index`；
- (2). 将 `signal_index` 和 `method_index` 传入 `QMetaObject::connect()` 函数中，取得全局的 `QConnectionList` 对象并为其增加连接；
- (3). `QConnectionList::addConnection()` 真正的保存连接的相关信息到全局的 `QConnectionList` 对象中，并且将 `sender` 和 `receiver` 对象相关的索引值保存在两个哈希表中，以便于以后的快速查找。

`connect()` 函数大概完成了一半的工作，剩余的另一半的疑问是用 `emit` 发射信号之后，槽如何被执行的呢？

### 11.2.2.2 信号的发射和槽的执行

在利用 `connect()` 函数保存了连接的相关信息的基础上，信号的发射和槽的执行就显得比较简单一些了，我们的主要工作是要找到与信号对应的槽并执行它。

先来看看信号是怎么发射的。在我们前面的例子中，发射信号就是用 `emit` 后面接信号名，而信号则用“signals”来声明，这些 Qt 的关键字是如何被定义的呢？我们可以在 `qobjectdefs.h` 中找到下面的代码：

```
41 // The following macros are our "extensions" to C++
42 // They are used, strictly speaking, only by the moc.
43
44 #ifndef Q_MOC_RUN
45 # if defined(QT_NO_KEYWORDS)
46 #   define QT_NO_EMIT
47 # else
48 #   define slots
49 #   define signals protected
50 # endif
51 # define Q_SLOTS
52 # define Q_SIGNALS protected
53 # define Q_PRIVATE_SLOT(d, signature)
54 #ifndef QT_NO_EMIT
55 # define emit
56 #endif
```

在 55 行定义了 emit, 48 行和 49 行定义了 slots 和 signals, 从 41 行和 42 的注释我们可以看到, 这些 Qt 的关键字都只是被 moc 工具所使用, 如果被 gcc 预编译时, emit 和 slots 都是空的, 只有 signals 被定义成 protected, 这也符合我们前面介绍过的信号的访问权限特性, 即发射信号只能由定义了一个信号的类和它的子类才能来完成。当然 moc 工具是可以识别这些关键字的, 所以它才能为我们生成类似 moc\_myclass.cpp 的文件。

既然 emit 被定义成空, 那么发射信号时如何才会调用对应的槽呢? 这不仅需要 Qt 源代码, 还需要到 moc 工具为我们生成的文件中去寻找答案。沿用前面的小例子, 我们来看看 moc\_signals\_slots.cpp:

```
181 void Sender::transmit(int _t1)
182 {
183     void *_a[] = { 0, const_cast<void*>(reinterpret_cast<const void*>(&_t1)) };
184     QMetaObject::activate(this, &staticMetaObject, 0, _a);
185 }
```

可以发现在 moc\_signals\_slots.cpp 中 moc 工具将信号 transmit 实现为一个函数, 这也是为什么我们在前面谈到信号的用法时强调信号只需要声明, 而不能真正实现它, 因为它的实现是由 moc 工具自动来完成的。结合前面 emit 和 signals 的定义我们可以看出, 其实信号的发射就是调用了对应的函数: 比如 signals\_slots.cpp 文件中的第 18 行 emit transmit(10), 由于 emit 实际上被展开后为空, 这行代码被 gcc 识别时就变成了 transmit(10), 而信号或者说函数 transmit(int) 在 moc\_signals\_slots.cpp 被实现, 于是 transmit 信号的发射相当于执行了上面的 183 和 184 行。183 行将参数\_t1 转换成了 void\* 格式, 以便之后的统一处理, 而 184 行则通过 activate() 函数来真正调用相关的槽。

来看看 QMetaObject::activate() 函数具体怎样调用相关的槽。184 行所调用的 activate() 函数的定义如下:

```
2979 void QMetaObject::activate(QObject *sender, const QMetaObject *m, int
local_signal_index,
2980                                void **argv)
2981 {
2982     int offset = m->methodOffset();
2983     activate(sender, offset + local_signal_index, offset + local_signal_index, argv);
2984 }
```

2982 行的 methodOffset() 我们在前面讲解 QMetaObject::indexOfSlot() 时有过介绍, 它返回 m 的所有父类的 method 的个数, 在计算索引值的时候我们总是需要加上这个值。然后 2983 行调用的重载的 activate() 函数 QMetaObject::activate(QObject \*sender, int from\_signal\_index, int to\_signal\_index, void \*\*argv), 输入参数 from\_signal\_index 和 to\_signal\_index 的值都是 offset + local\_signal\_index, 即只激活索引值为 offset + local\_signal\_index 的信号。这个重载的 activate() 函数也比较长, 我们挑一些重要代码来说明一下:

```
2863 void QMetaObject::activate(QObject *sender, int from_signal_index, int
to_signal_index, void **argv)
2864 {
...
2868     QConnectionList * const list = ::connectionList();
2869     if (!list)
```

```

2870         return;
...
2880         QConnectionList::Hash::const_iterator it = list->sendersHash.constFind(sender);
2881         const QConnectionList::Hash::const_iterator start = it;
2882         const QConnectionList::Hash::const_iterator end = list->sendersHash.constEnd();
...
2895         int i = 0;
2896         for (it = start; it != end && it.key() == sender; ++it) {
2897             ++i;
2898         }
2899         QVarLengthArray<int> connections(i);
2900         for (i = 0, it = start; it != end && it.key() == sender; ++i, ++it) {
2901             connections.data()[i] = it.value();
2902             ++list->connections[it.value()].refCount;
2903         }
2905         for (i = 0; i < connections.size(); ++i) {
2906             const int at = connections.constData()[connections.size() - (i + 1)];
2907             QConnectionList * const list = ::connectionList();
2908             QConnection *c = &list->connections[at];
2909             --c->refCount;
2910             if (!c->receiver || ((c->signal < from_signal_index || c->signal > to_signal_index) &&
2911                 c->signal != -1))
                continue;
...
2924             const int method = c->method;
...
2936 #if defined(QT_NO_EXCEPTIONS)
2937         c->receiver->qt_metacall(QMetaObject::InvokeMetaMethod, method, argv ?
argv : empty_argv);
2938 #else
...
2949 #endif
...
2961     }
...
2967 }

```

首先我们还是在 2868 行利用全局函数 `connectionList()` 得到全局的 `QConnectionList` 对象，在上一节中我们介绍过的 `connect()` 函数就是将连接保存在这个全局对象中，这里我们来读取它所保存的值。接下来从 2880 行开始我们从哈希表中根据键 `sender` 指针来寻找之前保存的索引值，注意这个索引值对应的是 `list` 对象中所保存的连接的位置。由于一个信号可能被连接到多个槽或者信号，我们需要把它们都找出来。2895 行—2903 行用来找到与 `sender` 相连所有信号或者槽，并将索引值保存到相当于数组的 `connections` 中。接下来从 2905 行对这些连接一一比对，2910 行根据几个条件来判断连接是否有效，即如果 `receiver` 指针为空，或者信号的索引值不在参数 `from_signal_index` 和 `to_signal_index` 所指定的范围内，则略过这



个连接；如果连接有效的话，在 2937 行将通过 `qt_metacall()` 来执行相应的信号或者槽。

那么 `qt_metacall()` 如何来通过索引值找到相应的信号或者槽呢？还记得我们在前面介绍过的 `Q_OBJECT` 宏的定义吗？对的，函数 `qt_metacall()` 就是通过展开 `Q_OBJECT` 宏来声明的，它的实现则被放在 `moc` 工具所生成的 `cpp` 文件中，比如 `moc_signals_slots.cpp` 中我们可以看到如下的代码：

```
57 int Receiver::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
58 {
59     _id = QObject::qt_metacall(_c, _id, _a);
60     if (_id < 0)
61         return _id;
62     if (_c == QMetaObject::InvokeMetaMethod) {
63         switch (_id) {
64             case 0: printNumber((*reinterpret_cast< int(*)>(_a[1]))); break;
65         }
66         _id -= 1;
67     }
68     return _id;
69 }
```

在第 59 行我们先调用父类的 `qt_metacall()` 函数。由于每个类的 `qt_metacall()` 函数都会首先去调用父类的 `qt_metacall()` 函数，其结果是首先调用顶层父类的 `qt_metacall()` 函数，然后一级一级往下。我们调用信号或者槽时，第一个参数输入的是 `QMetaObject::InvokeMetaMethod`，这样进入 63 行开始的 `switch-case` 条件判断，输入的 `_id` 值即为索引值，而且调用完成之后在 66 行索引值会减去这个类中的信号和槽的个数的值，这样每一级中调用 `qt_metacall()` 函数都会将索引值 `_id` 减去相应的个数的值。

我们用这个例子的具体数值来说明一下，`Receiver` 的父类是 `QObject`，`QObject` 就是顶层的类，只有两层，层次结构比较简单。`QObject` 中 `method` 的个数为 4 个（我们可以从源代码包中找到 `moc_qobject.cpp`，然后查看它的 `data` 数组的定义，采用前面所讲述过的 `indexOfSlot()` 中的计算方法，即可得到 `method` 的个数），而 `Receiver` 类中仅有 1 个，即槽 `printNumber()`，它在 `Receiver` 类的本地索引值就是 0，则通过 `indexOfSlot()` 计算出的索引值为 `4+0=4`。这样在调用 `qt_metacall()` 时所传入的参数 `_id` 的值也为 4，而进入 `qt_metacall()` 后，先在 59 行调用 `QObject::qt_metacall()` 并返回 `_id`，这时得到的 `_id` 在 `QObject::qt_metacall()` 被减去 4，所以 59 行返回的 `_id` 的值为 0，然后进入 `switch-case` 在第 64 行调用 `printNumber()` 函数。

简单的回顾一下信号发射之后如何找到并执行所连接的槽（或信号）的这种机制：

- (1). 我们声明的信号在 `moc` 所自动生成的 `moc_myclass.cpp` 中实现为 `protected` 的成员函数，而 `emit` 比定义为空，发射信号实际上就是调用 `moc` 自动生成的成员函数；
- (2). `moc` 自动生成的成员函数中调用了 `QMetaObject::activate()` 函数来寻找并调用相应的信号或槽；
- (3). `activate()` 函数通过 `connect()` 函数中所保存的全局 `QConnectionList` 对象来查找对应的信号或槽，并且利用的哈希表所保存的位置的索引值来快速定位
- (4). `activate()` 函数中找到所保存的 `receiver` 指针和信号或槽的索引值之后，通过虚函数 `qt_metacall()` 来真正调用相应的信号或槽。

### 11.2.3 元对象编译器 moc

我们已经在前面多次谈到并用到了 moc(Meta-Object Compiler, 元对象编译器)。moc 工具可以看成是一个 C++预处理程序, 用来扩展 C++的特性, 比如前面我们详细讲述的信号和槽的机制, 它并不是标准 C++的特性, 但经过 moc 的预处理之后, 所有信号和槽相关的代码都被“翻译”成了标准的 C++, 从而能够被 gcc 等编译。

#### 11.2.3.1 在 Makefile 中使用 moc

我们通常使用 Qt 提供的 qmake 工具来自动生成 Makefile, 这在我们前面的例子中多次用到, 这样做的好处是不用自己手动去添加很多 Qt 需要的规则, 比如调用 moc 工具等。

如果在某些情况下需要手动编写 Makefile 时, 我们可以用下面的规则来调用 moc:

```
moc_%.cpp: %.h
    moc $< -o $@
```

我们已经在前面的章节中介绍过 Makefile 的规则, 这里的含义是对所有头文件\*.h, 利用 moc 进行处理并得到 moc\_\*.cpp。生成 moc\_\*.cpp 之后不要忘记你还同样需要对它进行编译和链接, 所以你需要将它加到 SOURCES 和 OBJECTS (可能是其他类似含义的变量) 中。

如果你对这种 Makefile 的写法还不太熟悉, 可以参考 qmake 自动生成的 Makefile, 依葫芦画瓢就可以了。

#### 11.2.3.2 moc 用法详解

moc 支持的如下所示的一些命令行选项:

**-o file**

将输出写到参数 *file* (不指定的话将写到标准输出)。

**-f [<file>]**

强制在输出文件中生成 `#include` 声明。这个选项在你的头文件没有遵循标准命名法则的时候才有用——当头文件的扩展名以 `H` 或 `h` 开始时, `#include` 声明会自动生成, 不需要使用 `-f` 选项。文件名 `<file>` 为可选项。

**-i**

不在输出文件中生成 `#include` 声明, 与 `-f` 正好相反。当一个 C++ 文件包含一个或多个类声明的时候可能用到。

**-nw**

不产生任何警告。不建议使用。

**-p <path>**

在元对象编译器生成的 `#include` 声明的文件名称中添加路径 `<path>/`。

**-I <dir>**

在头文件的包含路径中添加 `<dir>` 目录。

**-E**

不生成元对象相关代码 (仅用于预编译)

**-D<macro>[=<def>]**

定义宏 `<macro>`, 或者定义宏 `<macro>=<def>`, 后者为可选项

-U<macro>  
    取消宏<macro>的定义

-h  
    显示 moc 的用法和选项

-v  
    显示 moc 的版本

我们还可以使用”#ifndef Q\_MOC\_RUN”来告诉 moc 工具不要处理某些代码。比如：

```
#ifndef Q_MOC_RUN
...
#endif
```

则 moc 会忽略定义在省略号部分的代码。实际上我们在前面分析过的源代码中见识过这个宏，定义 signals/slots/emit 等 Qt 关键字的时候，在 qobjectdefs.h 的第 44 行即用到了它。

### 11.2.3.3 moc 及信号和槽机制的局限性

moc 并不能处理所有的 C++特性。我们可以回想上一节的元对象系统及信号和槽的实现机制，这些实现中很少处理 C++的重要特性之一——模板，是的 moc 对模板的支持非常有限，对预编译宏的处理也不够完善。由于信号和槽是基于元对象系统来实现的，moc 的局限性也可以看作是信号和槽的局限性，这是我们在使用信号和槽的机制时需要特别注意的。

具体说来，moc 或者说信号和槽有如下一些限制：

1. 模板类不能含有信号和槽。比如下面的例子是不支持的：

```
// 错误的用法
class SomeTemplate<int> : public QFrame {
    Q_OBJECT
    ...
signals:
    void bugInMocDetected( int );
};
```

2. QObject（或其子类）作为多重继承的父类之一时，需要把它放在第一个。

如果你使用多重继承，moc 在处理时假设首先继承的类是 QObject 的一个子类，也就是说，我们需要确保首先继承的类是 QObject 或其子类。示例如下：

```
class SomeClass : public QObject, public OtherClass {
    ...
};
```

3. 函数指针不能作为信号和槽的参数。下面是一个不符合语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    // 不合法的
```

```
void apply( void (*apply)(List *, void *), char * );
};
```

对于这种情况，即需要使用函数指针作为信号/槽的参数的情形，一般可以考虑使用继承和虚函数等替代，通常来说使用继承和虚函数是更好的面向对象的实现方法。如果一定需要函数指针的话，也可以使用 typedef 来满足 moc 的处理过程，比如下面的用法是能够被 moc 接受的：

```
typedef void (*ApplyFunctionType)( List *, void * );

class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    void apply( ApplyFunctionType, char * );
};
```

#### 4. enum 和 typedef 变量在作为信号和槽的参数时必须用全名。

由于 moc 在检查信号和槽的参数时，是逐个字母来比较的，比如 Alignment 和 Qt::Alignment 被 moc 视为不同的参数类型。这个限制主要是 Qt 4.0 引入命名空间导致修改 moc 带来的，我们应对的方法是任何时候都在信号和槽的参数中使用全名，比如：

```
class MyClass : public QObject
{
    Q_OBJECT

    enum Error {
        ConnectionRefused,
        RemoteHostClosed,
        UnknownError
    };

signals:
    void stateChanged(MyClass::Error error); //Error 类型使用全名 MyClass::Error
};
```

#### 5. 带参数的宏不能被用于信号和槽的参数。

这主要是因为 moc 并不展开#define，使用带参数的宏在 moc 处理的过程中无法被有效的识别。下面是一个不合法的例子：

```
#ifndef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif

class Whatever : public QObject {
```

```

...
signals:
    void someSignal(SIGNEDNESS(int)); //不合法的参数 SIGNEDNESS(int)
...
};

```

不过不含有参数的宏是可以正常工作的，moc 处理的时候把它当作一个普通变量一样就可以了。

#### 6. 嵌套类不能含有信号和槽

moc 无法处理嵌套类中的信号和槽，它的实现机制中没有完整的考虑嵌套类的情况。下面是一个错误的例子：

```

class A {
    Q_OBJECT
public:
    class B {
        public slots:    // 错误的用法
            void b();
        ...
    };
signals:
    class B {            // 错误的用法
        void b();
        ...
    };
};

```

## 11.3 Qt 的窗口系统

这一节我们主要来介绍 Qt 的窗口系统中为我们管理内存提供了很多便利的部件之间的树型结构，以及灵活的布局管理机制。

### 11.3.1 窗口部件之间的树型结构

Qt 的窗口系统的重要特性之一就是我们在第 9 章提到过的树型结构，即所有的窗口部件本身也是容器，一个窗口部件可包含任意数量的子部件，子部件在父部件的区域内显示，父部件和子部件之间形成一种树型结构以便于维护——当父部件被删除的时候，它所包含的所有子部件都会被自动删除，这大大的方便了部件之间的内存管理。

这种部件之间的树型结构主要得益于我们前面提到的 Qt 元对象系统，元对象系统除了实现 Qt 中一些重要的机制比如信号和槽以及属性等等，也附带的利用了 QObject 作为顶级父类的这一前提，来实现子部件的自动删除机制。事实上不仅是窗口部件之间有这种树型结构，对所有从 QObject 继承而来的

要注意区分这种树型结构与父类子类所形成的类似结构：窗口部件之间的树型结构是一

种对象之间的关系（而不是类之间的关系），我们说父部件与子部件并不意味着它们分别是父类与子类的对象。

我们通过一个实际的例子来详细了解一下。为简单起见我们不采用图形部件，仅使用控制台输出来说明子部件是如何自动销毁的：

```
1  #include <QApplication>
2  #include <QtDebug>
3
4  class TreeMem : public QObject
5  {
6  public:
7      TreeMem(QObject *parent = 0, const QString& name = "")
8          : QObject(parent)
9      {
10         setObjectName(name);
11         qDebug() << "Created: " << objectName();
12     }
13
14     ~TreeMem()
15     {
16         qDebug() << "Deleted: " << objectName();
17     }
18
19     void printName()
20     {
21         qDebug() << "Print Name: " << objectName();
22     }
23 };
24
25 int main( int argc, char **argv )
26 {
27     QApplication a( argc, argv );
28
29     TreeMem top(0, "top");
30     TreeMem *x = new TreeMem(&top, "x");
31     TreeMem *y = new TreeMem(&top, "y");
32     TreeMem *z = new TreeMem(x, "z");
33
34     top.printName();
35     x->printName();
36     y->printName();
37     z->printName();
38
39     return 0;
40 }
```

我们定义了类 `TreeMem` 来追踪内存释放的过程，在构造函数中增加参数 `name` 用来标记不同的对象，在析构函数和函数 `printName()` 用 `QDebug` 来打印出对象的名字。

下面的 `main()` 函数中，第 29 行我们定义了顶层对象，然后定义 `x, y, z` 三个对象。这里我们所说的父对象就是指的参数 `QObject *parent` 的设定：`x, y` 以 `top` 为父对象，而 `z` 以 `x` 为父对象，这样它们之间的树型关系如图 11.3：

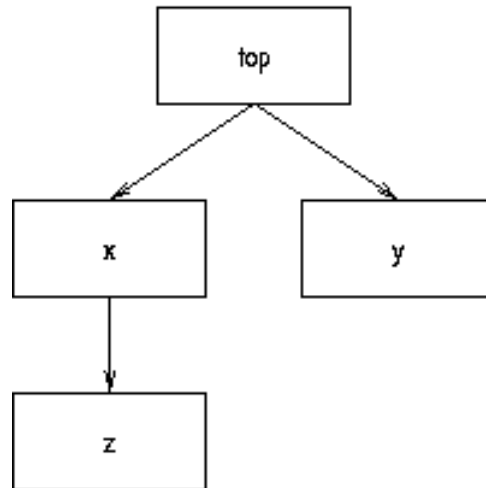


图 11.3 对象之间的树型关系

由于 `top` 定义在栈上，当程序运行到它的作用范围之外时，这个对象就会被自动销毁，这是 C++ 的基本特性之一。当 `top` 被自动析构时，它的所有子对象也都会被自动删除，我们来对照一下运行的结果：

```
$ ./treemem
Created: "top"
Created: "x"
Created: "y"
Created: "z"
Print Name: "top"
Print Name: "x"
Print Name: "y"
Print Name: "z"
Deleted: "top"
Deleted: "x"
Deleted: "z"
Deleted: "y"
```

可以看到正如我们所期望的，`x, y, z` 也都被自动析构了。注意 `z` 的析构在 `y` 之前，这说明它的析构被当作 `x` 析构的一部分来执行，因而比 `y` 的析构要早。

这里我们定义的 `x, y, z` 几乎没有其他实际意义，其实通常我们创建的是以 `QWidget` 为父类的部件对象，读者朋友很快会在 `Qt` 的实际开发应用中发现，以这种树型结构为基础的各部件之间的内存管理非常方便，可以经常用到。

### 11.3.2 窗口部件的布局管理（Layout）

尽管 Qt 提供了设计器(Designer)来帮助可视化的开发，但在嵌入式开发中我们往往需要更加精确也更加具有效率的方式来安排各个窗口部件的位置及大小，这些要求可以利用 Qt 所提供的布局管理器来很好的完成。在上一章的小例子中我们已经学习了一些基本的布局管理的知识和用法，这一小节里我们更全面的来介绍 Qt 的布局管理。

Qt 主要提供了下面的四个类来安排部件的位置：

- QHBoxLayout 在水平方向上从左到右（默认情况）或者从右到左布置部件。
- QVBoxLayout 在竖直方向从上往下（默认情况）或者从下往上布置部件。
- QGridLayout 以网格形式布局部件。
- QStackedLayout 以栈的形式来安排一组部件，这组部件每次只有一个显示。

其中比较常用的是前三个，示意图如下面的图 11.2，我们分别用了五个按钮部件来作为用法示例，对于其他类型的部件也是类似的：

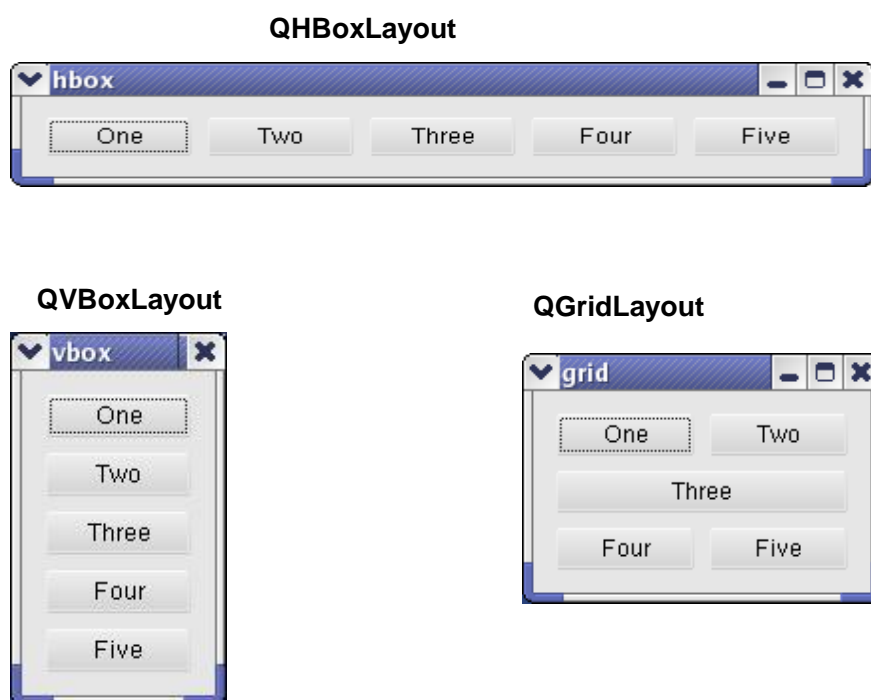


图 11.4 QHBoxLayout，QVBoxLayout 和 QGridLayout 三种布局管理

QHBoxLayout 和 QVBoxLayout 的用法类似，比如上面的 QHBoxLayout 的图形及水平方向摆放的五个按钮可以由下面的代码生成：

```
1 #include <QApplication>
2 #include <QPushButton>
3 #include <QWidget>
4 #include <QHBoxLayout>
5
6 int main(int argc, char *argv[])
7 {
```



```

8     QApplication app(argc, argv);
9
10    QWidget window;
11    QPushButton *button1 = new QPushButton("One");
12    QPushButton *button2 = new QPushButton("Two");
13    QPushButton *button3 = new QPushButton("Three");
14    QPushButton *button4 = new QPushButton("Four");
15    QPushButton *button5 = new QPushButton("Five");
16
17    QHBoxLayout *layout = new QHBoxLayout;
18    layout->addWidget(button1);
19    layout->addWidget(button2);
20    layout->addWidget(button3);
21    layout->addWidget(button4);
22    layout->addWidget(button5);
23
24    window.setLayout(layout);
25    window.show();
26
27    return app.exec();
28 }

```

而需要生成垂直方面摆放的五个按钮时，只需要将上面代码中的 `QHBoxLayout` 替换为 `QVBoxLayout` 就可以了。

`QGridLayout` 稍微有一些不同，因为它在安排部件时需要指定一个二维的坐标来设置布局的位置，而且某些部件可能需要占据两个或多个网格，比如图 11.4 的“Three”按钮。我们可以用下面的代码来生成图 11.4 中关于 `QGridLayout` 的示意图部分：

```

1  #include <QApplication>
2  #include <QPushButton>
3  #include <QWidget>
4  #include <QGridLayout>
5
6  int main(int argc, char *argv[])
7  {
8      QApplication app(argc, argv);
9
10     QWidget window;
11     QPushButton *button1 = new QPushButton("One");
12     QPushButton *button2 = new QPushButton("Two");
13     QPushButton *button3 = new QPushButton("Three");
14     QPushButton *button4 = new QPushButton("Four");
15     QPushButton *button5 = new QPushButton("Five");
16
17     QGridLayout *layout = new QGridLayout;
18     layout->addWidget(button1, 0, 0);

```

```

19     layout->addWidget(button2, 0, 1);
20     layout->addWidget(button3, 1, 0, 1, 2);
21     layout->addWidget(button4, 2, 0);
22     layout->addWidget(button5, 2, 1);
23
24     window.setLayout(layout);
25     window.show();
26
27     return app.exec();
28 }

```

注意 `addWidget` 的前两个参数用来指定二维坐标即第几行和第几列，行和列的计数都从 0 开始，而类似 `layout->addWidget(button3, 1, 0, 1, 2)` 的代码，后面的两个参数用来指定部件在纵向和横向分别占据几个网格。

`QStackedLayout` 提供的功能与常见的 Tab 部件有些类似，在某些情况下如果需要控制多组部件的可见性时可以提供方便。

对于这几个布局管理的类，需要注意 `addWidget` 方法以及父部件的设定情况，以确定部件是否会被自动销毁。上面的示例代码中，我们初始化几个按钮对象的时候都没有设定它们的父部件，也没有显式的删除它们，那么它们是如何被管理的呢？依靠 `addWidget()` 函数和后来的 `setLayout()` 函数——`addWidget()` 函数将自动删除对象的任务暂时的交给了这些布局对象，当然这并不是说它们的父部件是布局对象本身，实际情况是当布局对象被 `setLayout()` 函数设置为某个部件的布局风格时，比如上面代码中的 `window->setLayout(layout)`，这时这些按钮对象的父部件将被自动设置为 `window` 对象，当 `window` 对象被析构时，正如上一节所介绍的，这些按钮对象也被自动销毁。

我们还可以利用 `addWidget()` 函数来设置部件的伸展性，比如 `QBoxLayout` 类（注意它是 `QHBoxLayout` 和 `QVBoxLayout` 的父类）的 `addWidget()` 函数的原型如下：

```
void addWidget(QWidget * widget, int stretch = 0, Qt::Alignment alignment = 0)
```

其中第二个参数 `stretch` 可以用来设置部件的伸展性，如果布局管理对象中的部件都使用默认值 0 的话，部件通过 `QWidget::sizePolicy()` 来设置大小，如果设置为其他值，则当部件可伸展时布局管理对象中的各个部件将按照 `stretch` 的比例来调整大小。比如我们将上面的代码稍作修改来调整 `QHBoxLayout` 的水平布局：

```

17     QHBoxLayout *layout = new QHBoxLayout;
18     layout->addWidget(button1, 1);
19     layout->addWidget(button2, 2);
20     layout->addWidget(button3, 3);
21     layout->addWidget(button4, 4);
22     layout->addWidget(button5, 5);
23
24     window.setLayout(layout);
25     window.setGeometry(20, 50, 700, 100);
26     window.show();

```

这样当调整按钮的大小时，按钮 5 应该是最长的，注意我们在 25 行手动设置了一下部件 `window` 的大小，这样各个按钮不再是原来默认的大小，我们可以观察它们的伸展程度了。编译运行之后的结果如图 11.5：

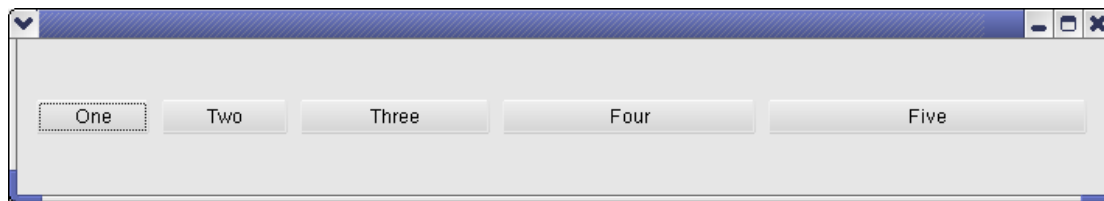


图 11.5 布局管理时的伸缩性示例

另外我们还可以通过 `QBoxLayout::setStretchFactor()` 在部件加入到布局管理对象之后来设置其伸缩性。对于 `QGridLayout` 设置伸展性的方法也是类似的，不再重复。

## 11.4 国际化

一般来说，程序或者软件系统的国际化，指的是如何使所开发的软件在不重写源代码的情况下能够支持多种语言，在英语中我们用 `internationalization` 或者它的简写形式 `i18n`（数字 18 表示 `i` 和 `n` 之间有 18 个字母）来表示。

在 Linux/Unix 系统中，现在流行的国际化方法是使用 GNU 的 `gettext` 套件。利用 `gettext` 套件，程序员将需要翻译的字符串用 `gettext()` 函数来取得，而由翻译人员协助提供翻译后的资源文件 `*.po`，在一次编译之后就可以支持多种语言。在这里我们不打算介绍 `gettext` 套件的详细内容，有兴趣的读者可参考 <http://www.gnu.org/software/gettext/>。

Qt 中的国际化方法与 GNU `gettext` 类似，它提供了 `tr()` 函数与 `gettext()` 函数对应，而翻译后的资源文件则以 `.qm` 来命名。如果对 GNU `gettext` 熟悉的读者，相信很快就可以了解 Qt 的国际化机制。不过值得注意的是 Qt 的国际化的机制与它的元对象系统密切相关，这是与 GNU `gettext` 稍微不同的地方。

下面我们简单介绍利用 Qt 开发国际化程序的基本步骤，并编写一个小例子来了解如何在程序运行的过程中动态改变语言。

### 11.4.1 Qt 国际化的基本步骤

#### 11.4.1.1 程序员的工作

在支持国际化的过程中，程序员的工作一般是在所编写的代码中加入国际化的支持，这在 Qt 中利用 `QString`、`QTranslator` 等类和 `tr()` 函数能够很方便的完成。程序员所需要做的工作有如下一些：

1. 使用 `QString` 对象来表示所有用户可见的文本。由于 `QString` 内部使用 Unicode 编码来实现，它可以用来表示所有需要向用户呈现的文本。当然对于仅仅程序员可见的文本并不需要都变成 `QString` 对象，可利用 Qt 提供的 `QCString` 或者原始的 `char *`。

2. 使用 `tr()` 函数来取得所有需要翻译的文本。在 Qt 的翻译机制下，`QObject::tr()` 函数可以帮我们取得翻译之后的文本。对于从 `QObject` 继承而来的类，`QObject::tr()` 函数最终由 `QMetaObject::tr()` 来实现，我们可以参考 11.2.1 中提到的 `Q_OBJECT` 宏的定义以及在 `qmetaobject.cpp` 中 `QMetaObject::tr()` 的实现。在某些时候如果无法使用 `QObject::tr()` 函数，我们还可以直接调用 `QCoreApplication::translate()` 来取得翻译之后的字符串。

3. 使用 `QString::arg()` 来组织动态文本。有些时候一段文本需要由一些静态文本和动态变量组合起来，比如常见的情况：`printf("The value of i is: %d", i)`。对于这种动态文本的翻译，

由于语言习惯的问题，如果简单的采用这种连接的方法，可能会带来一些问题。比如下面的字符串用来表示任务的完成情况：

```
QString m = tr("Mission status: “) + x + tr("“of “) + y + tr("”are completed”);
```

其中 *x* 和 *y* 是动态的变量，三个字符串被 *x* 和 *y* 分隔开，它们无法被很好的翻译——“*x* of *y*”是英语中分数的表示方法，比如 4 of 5 是分数 4/5，在不同的语言中分子和分母的位置可能是颠倒的，这种情况下数字 4 和 5 的位置在翻译的时候无法被正确的放置，可见孤立地翻译被分隔开的字符串是不行的，改进的办法是使用 `QString::arg()` 方法：

```
QString m = tr("Mission status: %1 of %2 are completed").arg(x).arg(y);
```

这样翻译工作者可以将整个字符串进行翻译，并将参数 %1 和 %2 放到正确的位置。

**4. 利用 `QTranslator::load()` 和 `QCoreApplication::installTranslator()` 来读取对应的翻译之后的资源文件。**翻译工作者将提供包含有翻译之后的字符串的资源文件\*.qm，程序员还需要做的是定义 `QTranslator` 对象，并用 `load()` 函数读取相应的.qm 文件，然后利用 `QCoreApplication::installTranslator()` 函数来安装 `QTranslator` 对象。我们在后面的小例子中将介绍具体的用法。

#### 11.4.1.2 语言资源管理者和翻译工作者的工作

通常来说组织翻译和管理翻译后的资源文件的工作并不是由编写代码的程序员来完成的，毕竟程序员不可能精通所有语言，翻译的工作通常由专门的工作组来协调管理，并聘请专门的翻译人员来翻译。这方面的工作主要是利用 Qt 提供的工具 `lupdate`、`linguist` 和 `lrelease`（它们都可以在 Qt 安装目录的 bin 文件夹下找到）来协助翻译工作并生成最后需要的.qm 文件，包括以下内容：

1. 利用 `lupdate` 工具从源代码中扫描并提取需要翻译的字符串，生成.ts 文件。类似编译时用到的 `qmake`，运行 `lupdate` 时我们也需要指定一个.pro 的文件，这个.pro 文件可以单独创建，也可以利用编译时用到的.pro 文件，只需要定义好变量 `TRANSLATIONS` 就可以了，具体用法可以参见后面的小例子。

2. 利用 `linguist` 工具来协助完成翻译工作，即打开前面用 `lupdate` 生成的.ts 文件，对其中的字符串逐条进行翻译并保存。由于.ts 文件采用了 xml 格式，我们也可以用其它编辑器来打开.ts 文件并翻译。

3. 利用 `lrelease` 工具处理翻译好的.ts 文件，生成格式更为紧凑的.qm 文件。这便是翻译工作者最终需要提供给程序员的资源文件，它所占的空间比.ts 文件小，但基本不具有可读性，只有 `QTranslator` 能正确的识别它。

#### 11.4.2 动态改变语言的小例子

下面我们通过一个小例子来具体说明上面提到的利用 Qt 实现国际化方法。我们设计的界面非常简单，用一个下拉菜单来选择语言，然后下面有一个标签，上面的文字是著名的“Hello World”（或者它的翻译），如图 11.6 所示：

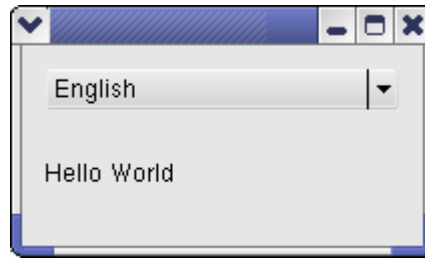


图 11.6 动态改变语言的小例子——英文界面

我们在 `LangSwitch.h` 文件中定义了类 `LangSwitch`，用来创建出上面的用户界面。其中 24 行和 25 行分别是我们在上面界面中可以看见的下拉菜单和标签，第 17 行定义的槽 `changeLang()` 用来响应下拉菜单中语言选项的改变，20 行—22 行定义的几个私有函数则用来协助创建界面和改变语言。

```
1  #ifndef LANGSWITCH_H
2  #define LANGSWITCH_H
3
4  #include <QWidget>
5
6  class QComboBox;
7  class QLabel;
8
9  class LangSwitch : public QWidget
10 {
11     Q_OBJECT
12 public:
13     LangSwitch();
14     ~LangSwitch() {};
15
16 private slots:
17     void changeLang(int index);
18
19 private:
20     void createScreen();
21     void changeTr(const QString& langCode);
22     void refreshLabel();
23
24     QComboBox* combo;
25     QLabel* label;
26 };
27
28 #endif //LANGSWITCH_H
```

来看看这些函数在 `LangSwitch.cpp` 中的具体实现：

```
15 void LangSwitch::createScreen()
16 {
```

```

17     combo = new QComboBox;
18     combo->addItem("English", "en");
19     combo->addItem("Chinese", "zh");
20     combo->addItem("Latin", "la");
21
22     label = new QLabel;
23     refreshLabel();
24
25     QVBoxLayout* layout = new QVBoxLayout;
26     layout->addWidget(combo, 1);
27     layout->addWidget(label, 5);
28
29     setLayout(layout);
30
31     connect(combo, SIGNAL(currentIndexChanged(int)),
32             this, SLOT(changeLang(int)));
33 }

```

createScreen()用来创建基本的界面。首先我们生成了 QComboBox 和 QLabel 对象，并设置其内容。18 行—20 行我们将三个语言选项英语、中文和拉丁语加到了下拉菜单中，并设置三个选项的值分别为”en”、”zh”和”la”（这是 ISO 标准中语言的简写形式）。23 行调用私有函数 refreshLabel()设置标签的内容，之所有使用函数 refreshLabel()是因为我们在后面改变语言的时候还需要重用它。25 行—29 行用到了我们熟悉的布局管理类 QVBoxLayout，31 行则将下拉菜单选项变化时发射的信号与我们自定义的槽 changeLang()连接起来。

来看看 23 行用到的函数 refreshLabel()的实现：

```

61 void LangSwitch::refreshLabel()
62 {
63     label->setText(tr("TXT_HELLO_WORLD", "Hello World"));
64 }

```

这是一个非常简单的实现，不过是用到了我们前面提到过的 tr()函数，以获取翻译之后的字符串。前一个参数是提取翻译串时用到的 ID，后一个则是提供注释的作用，并且在取不到翻译串时注释串会被采用。比如语言设置为中文时，以 TXT\_HELLO\_WORLD 为 ID 的串在对应的.qm 文件中找不到翻译后的字符串的话，就会采用后一个参数，即显示为英文。

最后我们来看看改变语言的具体过程：

```

35 void LangSwitch::changeLang(int index)
36 {
37     QString langCode = combo->itemData(index).toString();
38     changeTr(langCode);
39     refreshLabel();
40 }
41
42 void LangSwitch::changeTr(const QString& langCode)
43 {
44     static QTranslator* translator;

```

```

45
46     if (translator != NULL)
47     {
48         qApp->removeTranslator(translator);
49         delete translator;
50         translator = NULL;
51     }
52
53     translator = new QTranslator;
54     QString qmFilename = "lang_" + langCode;
55     if (translator->load(qmFilename))
56     {
57         qApp->installTranslator(translator);
58     }
59 }

```

槽 `changeLang()` 中我们先从所选的菜单项中取得对应语言的值（即我们在 18—20 行中的第二个参数所加入 “en”、“zh” 和 “la”），传给函数 `changeTr()` 去读取相应的 .qm 文件，然后刷新标签上的文字即可。而函数 `changeTr()` 则负责去读取对应的 .qm 文件，并调用 `installTranslator()` 方法来安装 `QTranslator` 对象。由于我们需要动态改变语言，所以如果已经安装了 `QTranslator` 对象的话，首先需要调用 `removeTranslator()` 移除原来的 `QTranslator` 对象，再安装新的，因此在 44 行我们定义了一个 static 的 `QTranslator` 对象以方便移除和重新安装。另外注意为简单起见我们将 .qm 文件的路径直接设定在当前路径下，命名分别为 `lang_en.qm`、`lang_zh.qm`、`lang_la.qm`。

对于程序员来说，上面的这些工作基本已经足够，剩下的提取需要翻译的字符串并翻译然后生成 .qm 文件的工作一般有专门的工作组去负责，不过为了完整的了解整个过程，我们也来介绍一下这些工作大致是如何完成的。

首先我们需要对 `qmake` 自动生成的 `langswitch.pro` 文件稍作修改，在后面加上 `TRANSLATIONS` 的定义，如下面的 14—16 行所示：

```

1  #####
2  # Automatically generated by qmake (2.01a) Tue Mar 6 16:04:46 2007
3  #####
4
5  TEMPLATE = app
6  TARGET =
7  DEPENDPATH += .
8  INCLUDEPATH += .
9
10 # Input
11 HEADERS += LangSwitch.h
12 SOURCES += LangSwitch.cpp main.cpp
13
14 TRANSLATIONS = lang_en.ts \
15                lang_zh.ts \
16                lang_la.ts

```

然后我们可以利用 `lupdate` 工具来提取需要翻译的字符串，运行命令及结果如下所示：

```
$ lupdate langswitch.pro
Updating 'lang_en.ts'...
    Found 1 source text (1 new and 0 already existing)
Updating 'lang_la.ts'...
    Found 1 source text (1 new and 0 already existing)
Updating 'lang_zh.ts'...
    Found 1 source text (1 new and 0 already existing)
```

这时候我们得到了 `lang_en.ts`，`lang_la.ts` 和 `lang_zh.ts` 三个文件。由于它们都是 xml 格式的，我们可以直接用 `cat` 命令来查看它们的内容，比如：

```
$ cat lang_la.ts
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS><TS version="1.1">
<context>
    <name>LangSwitch</name>
    <message>
        <location filename="LangSwitch.cpp" line="63"/>
        <source>TXT_HELLO_WORLD</source>
        <comment>Hello World</comment>
        <translation type="unfinished"></translation>
    </message>
</context>
</TS>
```

可以看到 ID 为 `TXT_HELLO_WORLD` 的串此时尚未被翻译。那么接下来的工作就是利用 `linguist` 来翻译这几个 .ts 文件了。如果你的 `PATH` 设置正确的话，直接敲入 `linguist` 命令就可以启动它，然后打开需要翻译的 .ts 文件，就可以进行字符串的翻译了，如图 11.7 所示：



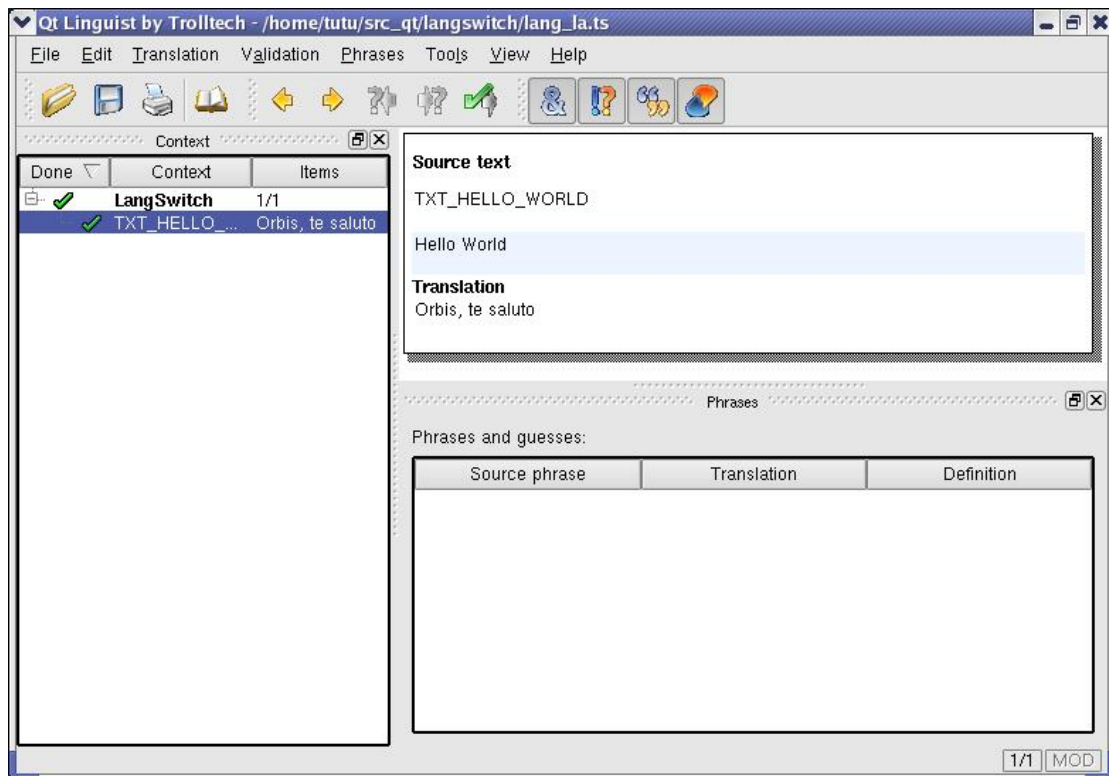


图 11.7 使用 linguist 来翻译字符串

翻译完一个串之后可以选择菜单”Translation->Done and Next”或者 Ctrl 加回车键将这个串设置为翻译完成，然后继续下一个（当然这个小例子中我们只有一个串需要翻译）。当所有的串都翻译完成后可以保存退出，这时我们可以来看一下.ts 文件的变化：

```
$ cat lang_la.ts
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS><TS version="1.1" language="en_US">
<context>
  <name>LangSwitch</name>
  <message>
    <location filename="LangSwitch.cpp" line="20"/>
    <source>TXT_HELLO_WORLD</source>
    <comment>Hello World</comment>
    <translation>Orbis, te saluto</translation>
  </message>
</context>
</TS>
```

可以看到行<translation>Orbis, te saluto</translation>，表示这是翻译之后的结果。对于翻译成中文的情况是类似的。

在得到翻译之后的.ts 文件后，最后的工作就是使将它们变为更紧凑的格式，即生成相应的.qm 文件。这可以利用 lrelease 来完成：

```
$ lrelease langswitch.pro
Updating 'lang_en.qm'...
```

```
Generated 1 translation (1 finished and 0 unfinished)
Updating 'lang_la.qm'...
Generated 1 translation (1 finished and 0 unfinished)
Updating 'lang_zh.qm'...
Generated 1 translation (1 finished and 0 unfinished)
```

现在所有的准备工作都已经做好了，要编译运行一下我们的程序吧。默认情况下的语言是英语，界面正如图 11.4 所示，我们可以尝试改变语言为中文或者拉丁文，结果如图 11.8：

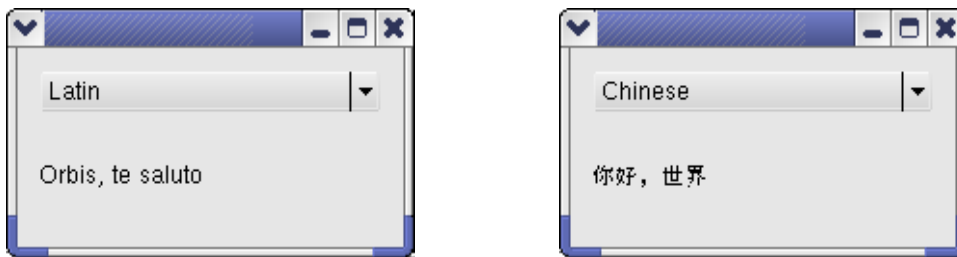


图 11.8 中文和拉丁文的界面显示

尽管这个小例子的界面非常简单，不过它完整的实现了动态改变语言的功能，希望对读者朋友的实际开发起到一定的参考作用。当然稍微复杂一点的软件系统对语言的切换都应该有更加完整和详细的定义，一般应该采用广播事件而不是简单的采用发射信号的方式来处理语言的变化。

### 11.4.3 一些注意事项

作为程序员我们往往容易忽略.qm 文件生成的过程，因为一般来说有一些专门的人员来组织和负责这方面的工作，但事实上由于 `tr()` 函数本身以及 `lupdate` 等工具的一些局限，很多时候我们会发现语言改变时得不到正确的翻译串，这种问题在大型系统中调试起来也可能比较麻烦，如果程序员和翻译人员互相之间不清楚彼此的工作的话，可能不容易找到问题的关键，这也是为什么我们在前面不仅介绍了源代码的编写，也详细介绍了生成.qm 文件的过程的原因，这样有利于我们调试 bug 时更容易发现问题所在。

程序员常犯的错误是拿到字符串对应的 ID 之后，便按照他们的要求来随意使用，比如这些 ID 被存放到一个统一的 txt 文件中，这样的结果很可能是这些串根本就没被 `lupdate` 所提取出来，因为 `lupdate` 在扫描的时候只会注意 `tr()` 等一些关键字，这样在.qm 文件中当然也没有对应的翻译串，最后的结果就是这些串不能被正确的显示翻译出来。

另一个问题是将翻译串放入数组中。简单的将 ID 放入数组中并不能引起 `lupdate` 的注意，这样的结果同上。这时候我们可以使用 Qt 提供的宏 `QT_TR_NOOP()` 或者 `QT_TRANSLATE_NOOP()` 以便于 `lupdate` 识别。比如：

```
static const char *greeting_strings[] = {
    QT_TR_NOOP("Hello"),
    QT_TR_NOOP("Goodbye")
};
```

```
// 或者
static const char *greeting_strings[] = {
    QT_TRANSLATE_NOOP("FriendlyConversation", "Hello"),
    QT_TRANSLATE_NOOP("FriendlyConversation", "Goodbye")
};
```

注意 QT\_TRANSLATE\_NOOP 宏的第一个参数是 `const char * context`，即上下文，这个概念的引入大概是为了改善同一个串在不同的上下文中可能需要不同的翻译的情况，但也带来了不少问题。我们可以看一看 `qmetaobject.cpp` 中 `QMetaObject::tr()` 的实现：

```
QString QMetaObject::tr(const char *s, const char *c) const
{
    return QCoreApplication::translate(d.stringdata, s,
                                        c, QCoreApplication::CodecForTr);
}
```

这里 `context` 的值被设置为 `d.stringdata`，回顾我们前面介绍过的元对象系统可以知道 `d.stringdata` 实际上就是类的名字。但是如果你将某个串放在别的类中，比如说串 `TXT_1` 在 `Class A` 中被加入某个数组中，并用 `QT_TR_NOOP` 宏包起来，但我们在 `Class B` 中才真正用 `tr()` 函数去使用它，那么 `lupdate` 扫描后得到的 `context` 值为 `A`，而在 `Class B` 中用 `tr()` 调用时 `context` 值为 `B`，这种不一致也会导致最后翻译的不成功。如果我们不了解整个过程，又凑巧这么使用了的话，找出这种错误是非常费尽的。

以上的问题可以说是我们在实际工作中的一些经验总结，在各类 Qt 文档或书籍中都鲜有介绍，希望对我们解决类似问题的时候能有一些启发的作用。

## 11.5 本章小结

本章的核心部分是 Qt 的元对象系统(Meta Object System)。Qt 的元对象系统最初是为实现信号和槽的机制而引入的，在后来的发展中又逐步为 Qt 提供了一些其他的方便的特性，比如我们谈到过的对象树，`tr()` 函数等，另外还有限于篇幅我们没有详细说明的属性支持等。

当然，作为 C++ 特性的一种扩展，元对象系统还有很多局限性，Trolltech 公司也在对它进行不断的改进，有兴趣的读者可以对比 Qt3 和 Qt4 的中的实现，变化是相当大的。

## 11.6 常见问题

1. 假设信号 `S` 定义在类 `A` 中，类 `B` 是 `A` 的子类，类 `C` 中包含有类 `A` 的对象 `a` 作为成员变量，`A,B,C` 中哪些可以发射信号 `S`？

参考答案：`A,B` 可以发射信号 `S`，但 `C` 不能。这是因为信号在元对象系统中被定义为 `protected`，仅仅定义信号的类和它的子类可以发射信号。参见 11.2.2.2 节。

2. 在 11.1.3 节的小例子中，如果将 `signals_slots.cpp` 的第 29 行改成用下面的方式来连接信号和槽，即在 `int` 后加上变量 `n`，这种方法正确吗？

```
QObject::connect(&s, SIGNAL(transmit(int n)), &r, SLOT(printNumber(int n)));
```

参考答案：不正确。连接带有参数的信号和槽的时候，只需要加入参数的类型（形参）就足够了，如果加入变量（实参）反而会导致不能通过参数一致性的检查，`connect()`函数会返回 `false`。

3. 在 11.3.2 节中，图 11.3 是将 5 个按钮的伸展性参数分别设置为 1, 2, 3, 4, 5 所得到的。如果将按钮的伸展性参数都设置为 5，结果又会如何呢？

参考答案：其结果是五个按钮的大小长度相等。伸展性参数的意义是当部件可伸展时，布局管理对象中的各个部件将按照参数的比例来调整大小。如果五个参数都设为 5，则五个按钮的伸展程度是相当的。

4. 如果发现通过 `tr()`函数没有得到正确的翻译串，应该如何来查找问题的所在？

参考答案：一般我们先去查该串有没有被 `lupdate` 提取到 `.ts` 文件中，如果没有的话应该检查运行 `lupdate` 时所指定的 `.pro` 文件。如果 `.ts` 文件中能找到该串的话，则进一步检查 `.ts` 文件中的 `context` 设置与调用 `tr()`函数时的 `context` 是否一致，以及经过 `linguist` 处理后的 `.ts` 文件中，该串是否已经被真正的翻译。另外还可能有问题地方是 `QTranslator` 是否读取到了正确的 `.qm` 文件。

## 第 12 章 Qtopia Core

本章学习目标:

- l 成功搭建 Qtopia Core 开发环境, 包括 qvfb 的使用
- l 了解 Qtopia Core 和 Qt/X11 程序的互相之间如何移植
- l 理解 Qtopia Core 的轻量级窗口系统
- l 掌握 QCOP 进程间通信机制

### 12.1 Qtopia Core 的安装

我们在第 9 章曾经简单介绍了 Qtopia Core, 它是 Trolltech 公司开发的面向嵌入式系统的 Qt 版本。Qtopia Core 的前身是 Qt/Embedded, 早在 2000 年 11 月就发布了第一个 Qt/Embedded, Trolltech 从 Qt 版本 4.1 开始将 Qt/Embedded 改称为 Qtopia Core, 作为嵌入式版本的核心, 并在 Qtopia Core 的基础上发行面向于 PDA、手机等的版本, 称为 Qtopia Phone Edition 和 Qtopia PDA Edition 等。

Qtopia Core 与 Qt/X11 最大的区别在于 Qtopia Core 直接访问帧缓存(Frame Buffer), 而不依赖于 X Server 或者 Xlib, 以减少了内存消耗及提高运行时的效率。

Qtopia Core 的安装过程与 Qt/X11 非常类似。它同样有商业版和自由版两种授权方式, 我们可以在 Trolltech 公司的主页下载 Qtopia Core 的自有版本: <http://www.trolltech.com/developer/downloads/qtopia/coregpl>, 同样也可以选择一些速度可能比较快的国内的镜像站点比如 <ftp://ftp.qtopia.org.cn/mirror/ftp.trolltech.com/> 或者 <http://www.qtopia.org.cn/ftp/mirror/ftp.trolltech.com/> 来下载。

如果你在前面已经安装了 Qt/X11, 基本上按照同样的步骤去安装 Qtopia Core 就可以了。安装 Qt/X11 时应该已经确保了你的 Linux 系统中所需要的 gcc, make 等编译工具, 以及 xlib 相关的库等都已经没有问题 (当然如果你准备要将你的 Qtopia Core 程序运行在一些特性的开发板上的, 你需要按照本书前面章节中所讲述的, 配置好你的交叉编译环境), 你可以放心的解压缩 Qtopia Core 自由版的压缩包 qtopia-core-opensource-src-4.2.2.tar.gz 来进行安装了。

步骤一: 解压缩安装包。

```
# tar xzvf qtopia-core-opensource-src-4.2.2.tar.gz
```

步骤二: 运行配置程序

```
# cd qtopia-core-opensource-src-4.2.2
# ./configure --embedded [architecture] -qvfb
```

与 Qt/X11 的安装稍有不同, 这里的 configure 程序需要使用 -embedded 选项并指定 CPU 的体系结构, 根据你所用的机器的不同, 可将 architecture 设置为 arm, mips, x86, 或者 generic。同时由于我们后面要用到 虚拟帧缓存工具 qvfb (Qt Virtual Frame Buffer), 在这里需要添加 -qvfb 选项。configure 程序还有很多其他安装选项, 比如 -prefix 可以设定安装路径等, 我们可以键入 "./configure -help" 来查看。运行 configure 程序后马上会看到与 Qt/X11 的安装时的 License 问题, 回答 yes 就可以继续了。

步骤三: 编译 Qtopia Core 源代码

```
# make
```

configure 程序的主要作用是生成了 qmake 以及相关的 Makefile 和.pro 文件。接下来我们就可以用 make 工具来编译这些 Makefile 了。根据机器配置的情况不同,这一步骤需要几十分钟到几个小时。

#### 步骤四: 安装 Qtopia Core

```
# su -c "make install"
```

如果前面 configure 程序配置时使用的是默认安装路径的话, Qtopia Core 被安装在 /usr/local/Trolltech/QtopiaCore-4.2.2, 这需要你有 root 权限。如果没有 root 权限你也可以在运行配置程序的时候修改安装路径。

#### 步骤五: 设置 PATH

为了更方便的使用 Qtopia Core 提供的各种工具, 我们把/usr/local/Trolltech/QtopiaCore-4.2.2/bin 添加到 PATH 变量——修改\$HOME/.bash\_profile 或者\$HOME/.profile 并加入(或修改):

```
PATH=/usr/local/Trolltech/QtopiaCore-4.2.2/bin:$PATH
```

```
export PATH
```

如果我们前面已经把 Qt/X11 的路径加入到了 PATH, 最好保证将 Qtopia Core 的路径加在前面, 因为后面我们要用到的都是 Qtopia Core 中的工具。比如我们可以直接将 PATH 设置为:

```
PATH=/usr/local/Trolltech/QtopiaCore-4.2.2/bin:/usr/local/Trolltech/Qt-4.2.2/bin:$PATH
```

修改完脚本后需要用 source 命令(或命令".")重新运行修改的脚本, 使设置生效, 如:

```
# source .bash_profile
```

我们可以来测试一下 PATH 设置的情况:

```
# which qmake
```

```
/usr/local/Trolltech/QtopiaCore-4.2.2/bin/qmake
```

这说明我们已经将 Qtopia Core 的路径加在 Qt/X11 的路径之前了。注意这时候如果我们需要再回头去编译 Qt/X11 程序, 则需要修改 PATH 或者指定 qmake 等的路径了。

## 12.2 Frame Buffer 和 qvfb

由于 Qtopia Core 绕过了 Xlib 来直接读写 Frame Buffer, 我们来了解一些 Frame Buffer 的知识, 以及 Qt 所提供的模拟 Frame Buffer 以方便调试的工具——qvfb。

### 12.2.1 Frame Buffer

我们在 9.3.2.1 中简单介绍了一些 Frame Buffer 的知识, 它为我们提供的硬件抽象对程序员来说提供了一种统一的接口, 免去了适应各种硬件的麻烦。注意版本 2.2 以后的 Linux 内核可以支持 Frame Buffer, 而老版本的内核可能需要许多调整并重新编译。关于各种架构下的 Frame Buffer 的支持不在本书讨论的范畴, 读者可参考著名的 Frame Buffer HOWTO 文档: <http://en.tldp.org/HOWTO/Framebuffer-HOWTO.html>, 里面有非常详细的介绍。

Qtopia Core 所带的文档中为我们提供了一段代码来测试 Frame Buffer 是否支持, 我们可以编译运行它来进行测试。为节省篇幅我们不在这里列出相关代码了, 读者可以在本书所附的光盘中找到本章目录下的 testFb 子目录, 即包含有相关的测试代码。这段代码仅仅使用了 C 的库函数, 而没有任何 Qt 的内容, 我们可以直接用 gcc 来编译它:

```
# gcc main.cpp -o testFb
```

```
# ./testFb
The framebuffer device was opened successfully.
800x600, 16bpp
The framebuffer device was mapped to memory successfully.
```

如果你看到的结果如上，并且能够看到屏幕上方所画出的矩形图案，说明 Frame Buffer 已经配置好了。如果你遇到类似”Error: cannot open framebuffer device”的错误，可以尝试使用”ls /dev/fb0”，如果能看到这个设备，再尝试一下”cat /dev/fb0”，cat 的结果很可能是无法打开设备，这说明你的 Linux 内核版本已经支持 Frame Buffer，但是没有打开，你需要修改你的 lilo.conf 或者 grub.conf，加入”vga=[显示模式数值]”，其中显示模式数值的设置包括分辨率及颜色深度，可参考表 12.1：

	640x480	800x600	1024x768	1280x1024
256 色	0x301	0x303	0x305	0x307
32k 色	0x310	0x313	0x316	0x319
64k 色	0x311	0x314	0x317	0x31A
16M 色	0x312	0x315	0x318	0x31B

表 12.1 vga 的显示模式设置表

在笔者的机器上，grub.conf 的设置如下：

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#           all kernel and initrd paths are relative to /boot/, eg.
#           root (hd0,0)
#           kernel /vmlinuz-version ro root=/dev/hda2
#           initrd /initrd-version.img
#boot=/dev/hda
default=0
timeout=2
splashimage=(hd0,0)/grub/splash.xpm.gz
title Red Hat Linux (2.4.20-8)
    root (hd0,0)
    kernel /vmlinuz-2.4.20-8 ro root=LABEL=/ vga=0x314
    initrd /initrd-2.4.20-8.img
```

其中 vga 被设置为 0x314，对照表 12.1，其设置为 64k 色，800×600 分辨率。

修改完成后重启系统，如果启动过程中看到屏幕左上角的企鹅 logo，恭喜你，Frame Buffer 被启动了。修改你的/etc/inittab 的 run level 为 3，使 linux 以控制台启动而 X11 并不起来，来感受一下 Qtopia Core 程序通过直接读写 Frame Buffer 而显示的图形界面效果吧：

```
# cd ~/qtopia-core-opensource-src-4.2.2/examples/widgets/movie
# ./movie -qws
```

这里我们运行的是安装 Qtopia Core 后所带的示例程序，-qws 表示将当前的程序作为 Server 来运行。我们可以看到并操作这个 movie 程序，如图 12.1 所示：

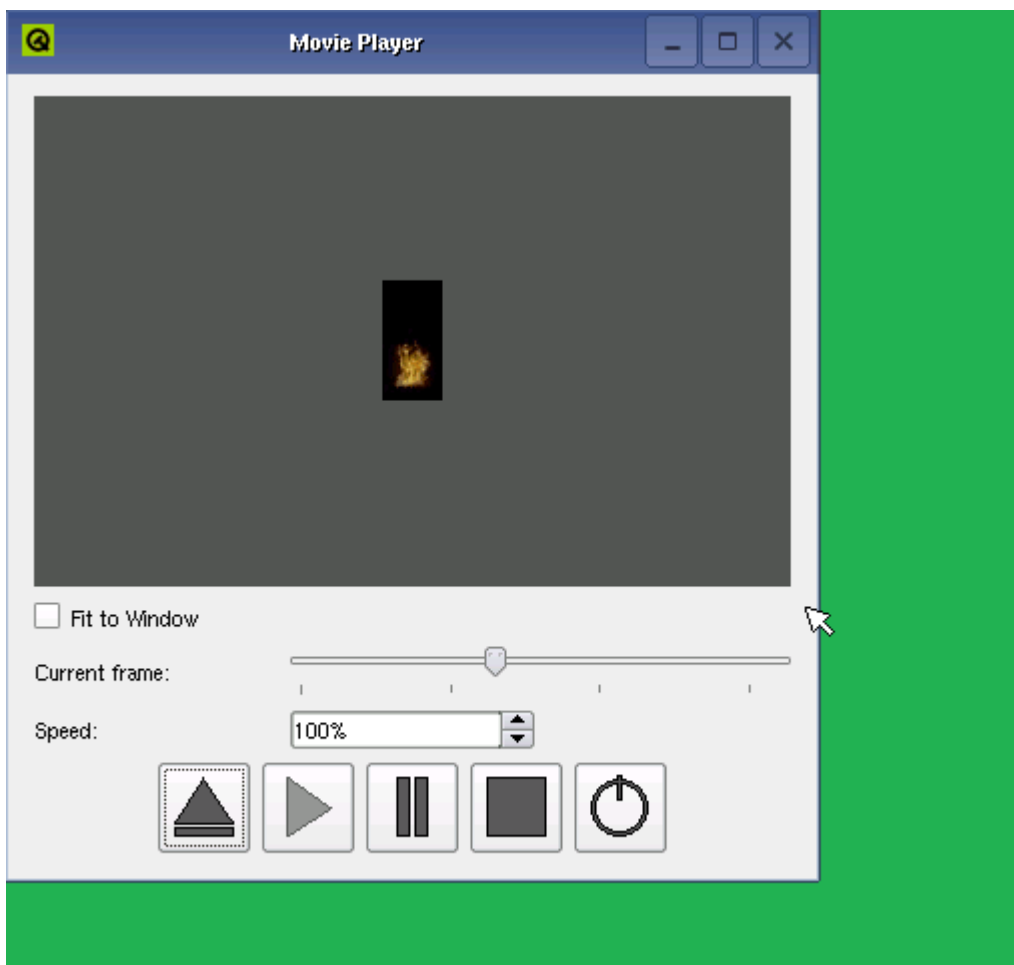


图 12.1 直接运行在 Frame Buffer 上的 movie 程序

### 12.2.2 编译 qvfb

由于 Qtopia Core 程序直接读写帧缓存 Frame Buffer，而开发工作一般在桌面系统中进行，调试程序会比较不方便，因而 Qt 为我们提供了虚拟的帧缓存，即 qvfb 工具来方便开发和调试工作。

我们在上一节的步骤二中运行配置程序时，已经添加了 qvfb 选项，但是这还只是配置好了 Qtopia Core 程序编译和运行时的一些支持，真正的 qvfb 工具并不是一个 Qtopia Core 程序，而是一个 Qt/X11 程序，它被附带在 Qt/X11 的安装程序中，我们需要到 Qt/X11 解压后的路径下来编译它：

```
# cd /home/user_name/qt-x11-opensource-src-4.2.2/tools/qvfb
# make
```

编译成功之后我们可以在目录/home/user\_name/qt-x11-opensource-src-4.2.2/bin/下找到它。为以后方便使用我们可以把它拷贝到/usr/local/Trolltech/Qt-4.2.2/bin 下去。

我们需要在 X11 启动的情况下运行 qvfb，如果你还在控制台下，则需要先用 startx 命令来启动 X Window。之后可以来试试 qvfb 的运行效果：

```
# qvfb
```

结果如图 12.2:



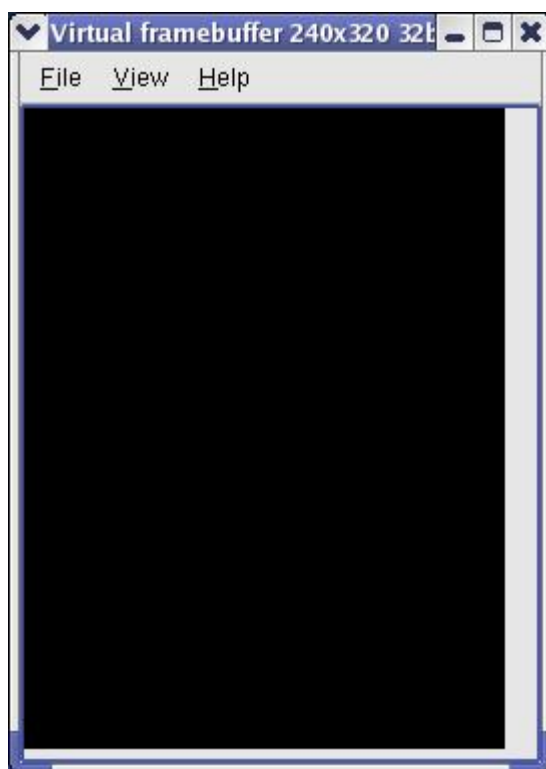


图 12.2 默认情况下的 qvfb

我们并没有在 qvfb 上运行 Qtopia Core 的程序，所以它的中间还是黑色的。

### 12.2.3 在 qvfb 上运行 Qtopia Core 程序

在正确配置的情况下，运行 Qtopia Core 程序时，它会自动找到已经在运行的 qvfb 程序并在这个虚拟的帧缓存上显示图形。我们可以先在后台运行 qvfb，接着就可以运行 Qtopia Core 程序了。

```
# qvfb &  
# cd ~/qtopia-core-opensource-src-4.2.2/examples/widgets/digitalclock  
# ./digitalclock -qws
```

这个示例程序的在 qvfb 上运行的结果如图 12.3 所示：



图 12.3 在 qvfb 上运行 Qtopia Core 示例程序

如果 Qtopia Core 程序没有能够自动检查到你所运行的 qvfb，你也可以加入参数-display 来指定它：

```
# ./digitalclock -qws -display QVfb:0
```

需要小心的是，如果你在没有运行 qvfb 的情况下，直接用 `./digitalclock -qws` 来运行这个 Qtopia Core 示例程序，它会直接改写真正的 Frame Buffer（而不是虚拟的 qvfb），这样你的 X Window 桌面可能会受到破坏。

你可以在 qvfb 的菜单 File->Configure 中修改它的默认配置，Qt 4.2 中所带 qvfb 功能已经相当强大，你可以用改变它的 skin，使它看起来像个手机或者 PDA 等设备，而且这些 skin 上按键也被匹配到键盘上，你可以用鼠标去点击这种模拟的手机或者 PDA 按键。你也可以自己定义 skin，或者修改已经存在的 skin，到 `/home/user_name/qt-x11-opensource-src-4.2.2/tools/qvfb/` 下去看一看你就能依葫芦画瓢的发现该如何做。图 12.4 说明了如何改变 qvfb 的配置，以及将 skin 修改为 Smartphone 之后再运行 Qtopia Core 示例程序 digitalclock 的效果。



图 12.4 修改 qvfb 的配置及修改之后的效果

我们还可以利用 VNC 协议来运行 Qtopia Core 程序，不过在本书中 qvfb 已经足以满足我们的要求，对将 Qtopia Core 程序运行在 VNC Server 上的方法有兴趣的读者可以自行参考 Qtopia Core 的文档。

## 12.3 移植 Qt/X11 程序到 Qtopia Core 中

我们先来回顾一下第 9 章谈到过的 Qt/X11 与 Qtopia Core 系统架构图的对比：开发 Qt/X11 与 Qtopia Core 应用程序时，我们实际上使用的是同一套 API，这使得 Qt/X11 到 Qtopia Core 的移植变得非常简单。但是不是我们前面编译好的一些 Qt/X11 程序就可以运行在 qvfb 或者直接运行在 Frame Buffer 上呢？当然不是，因为我们在编译 Qt/X11 程序时，所链接的 .so 库是由 Qt/X11 提供的路径为 /usr/local/Trolltech/Qt-4.2.2/lib 下的 libQtCore.so.4.2.2，同时还需要 libX11.so 等，而编译 Qtopia Core 程序时，所链接的 .so 库是 /usr/local/Trolltech/QtopiaCore-4.2.2/lib 路径下可以直接读写 Frame Buffer 的 libQtCore.so.4.2.2 等，我们可以用 diff 比较一下 /usr/local/Trolltech/Qt-4.2.2/lib 和 /usr/local/Trolltech/QtopiaCore-4.2.2/lib 这两个目录下的 .so 文件，即使它们的文件名相同，真正的内容也是不一样的。

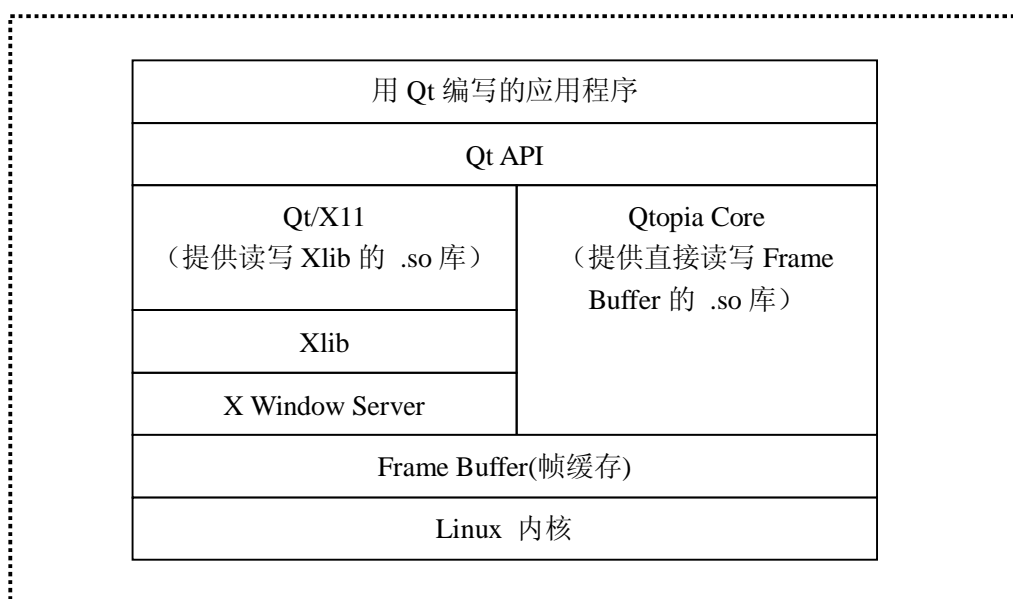


图 12.5 Qt/X11 和 Qtopia Core 提供不同的.so 库

理解了 Qt/X11 与 Qtopia Core 的这些相同和不同之处，相信读者朋友已经知道进行从 Qt/X11 到 Qtopia Core 的移植了：

1. 设置好 PATH 以保证你调用的 qmake 是 Qtopia Core 的 qmake 而不是 Qt/X11 的 qmake。正如 12.1 中提到的，你可以用”which qmake”来查看你的 PATH 设置是否正确。或者你可以直接显式的用全路径来调用你需要的 qmake，即 /usr/local/Trolltech/QtopiaCore-4.2.2/bin/qmake。
2. 去掉以前编译 Qt/X11 时所生成的可执行文件和 makefile 等。
3. 按照与编译 Qt/X11 类似的步骤来重新编译链接。
4. 在后台启动 qvfb，运行你编译后的 Qtopia Core 程序。

我们可以尝试将第 10 章中的温度转换的小例子移植到 Qtopia Core 上来，按照上面的方法，唯一的一点改变在 main.cpp 的第 9 行，用 setGeometry()函数调整设置一下窗口的大小，以适应 qvfb 的窗口。

```

1  #include <QApplication>
2
3  #include "ConversionScreen.h"
4
5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);
8      ConversionScreen screen;
9      screen.setGeometry(0, 0, 240, 320);
10     screen.show();
11     return app.exec();
12 }

```

重新编译运行（注意加入”-qws”参数）之后的温度转换程序如图 12.6 所示：

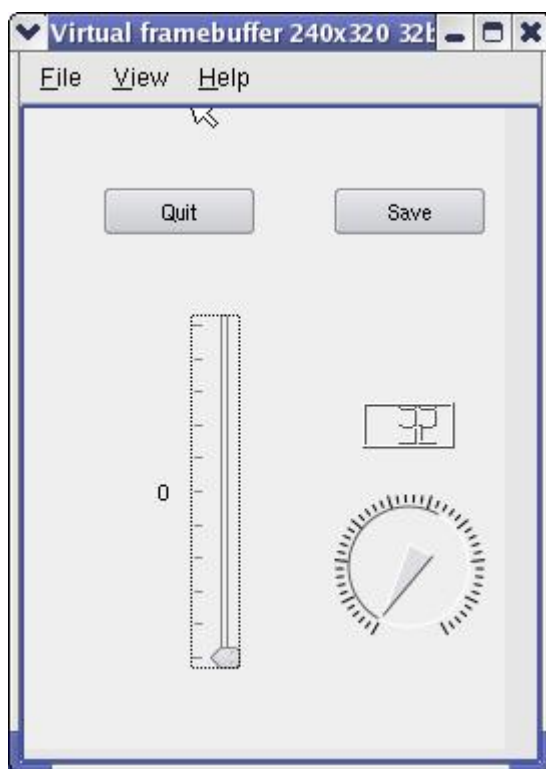


图 12.6 在 qvfb 下运行温度转换程序

前几章中所用的一些小例子都可以作类似的移植，几乎不用修改任何源代码（或者作一些象上面这样的稍作调整），这里不再重复。

## 12.4 轻量级的窗口系统

与 Qt/X11 类似，Qtopia Core 仍然采用 Client/Server 模型来显示窗口，但整体构架与 Qt/X11 有所差别——请读者回顾我们在第 9 章介绍过的 X Window 的系统架构(参见图 9.2)以及上一节中的图 12.5，对 X Window 架构，我们有一个专门的 X Server 用来响应 Client 程序的请求并绘制图形，一个 X Client 若想要运行，必须连接 X Server 并提出需求；对于 Qtopia Core 程序，它们也被要求必须运行在某个 Server 之上，但是 Qtopia Core 中没有这样专门的 Server，正如本章前面所介绍的，一个 Qtopia Core 的应用程序在运行时加上参数“-qws”的话，它本身就被作为 Server 运行。

这种差别主要是为了打造一个轻量级的 Client/Server 窗口系统，因为在嵌入式应用开发中由于硬件条件的限制，我们往往无法负担起庞大的 X Server 所需要的系统开销。相对 X 的架构，Qtopia Core 中的主要差别在于，很多本来需要由 Server 完成的工作都直接交给 Client 去完成。比如窗口的绘制，并不是象 X 系统中那样完全由 X Server 来完成，而是由 Client 进行直接操作 Frame Buffer 来实现，这样一来就可以大大减少 Server 和 Client 之间的通信开销，提高运行时的效率。

当 QApplication 为某个进程生成了一个 QWSServer 对象时，这个进程就成为了 Qtopia Core 中的 Server，它主要负责分配 Client 进程的显示区域，并产生鼠标和键盘事件等。而 Client 进程则通常生成一个 QWSClient 对象，负责处理与各种应用相关的逻辑并绘制它本身。注意这些逻辑都包含在 QApplication 之中，我们不需要自己来构造一个 QWSServer 对象或

QWSSClient 对象。

有两种方法可以让 QApplication 来构造 QWSServer 对象：一种就是我们前面用到的在运行加入“-qws”参数，另一种是构造 QApplication 对象时指定第三个参数为 QApplication::GuiServer。而访问 QApplication 所生成的这个 QWSServer 对象则可以通过全局变量 qwsServer 来得到。QWSServer 类提供了很多函数比如 clientWindows()、windowAt() 等来管理所有的窗口，并提供了 setDefaultMouse()、mouseHandler()、setDefaultKeyboard()、keyboardHandler() 等来处理鼠标和键盘事件等。

尽管对于单独的 Qtopia Core 的应用程序来说，加入“-qws”来运行即已经足够，但如果我们应用 Qtopia Core 来开发一个嵌入式系统时，最好专门运行一个程序作为 Server 应用，来负责处理各种事件消息等，而其他的应用程序都运行在这个 Server 之上，以便于整个系统的统一管理。

下面我们用一个简单的例子来说明如何运行一个 Server，并且通过 Server 来启动其他 Client 进程。为简单起见，我们直接利用 12.3 中编译好的 Conversion 程序来作为 Client 进程，来看看我们在 Server 进程中如何启动它。

首先是 main.cpp:

```
1  #include <QApplication>
2  #include "Launcher.h"
3
4  int main(int argc, char *argv[])
5  {
6      QApplication app(argc, argv, QApplication::GuiServer);
7      Launcher screen;
8      screen.show();
9      return app.exec();
10 }
```

注意第 6 行的改变：我们指定了参数 QApplication::GuiServer，使这个程序作为 Server 来运行。

接下来看看 Launch.cpp 文件中 Conversion 是如何被作为 Client 启动的：

```
1  #include <QPushButton>
2  #include <QVBoxLayout>
3  #include <QApplication>
4  #include <unistd.h>
5  #include <sys/types.h>
6
7  #include "Launcher.h"
8
9  Launcher::Launcher() : QWidget()
10 {
11     createScreen();
12 }
13
14 void Launcher::createScreen()
15 {
```

```

16     QPushButton* launcherBtn = new QPushButton("Launch Conversion");
17     QPushButton* quitBtn = new QPushButton("Quit");
18
19     QVBoxLayout *mainLayout = new QVBoxLayout;
20     mainLayout->addWidget(launcherBtn);
21     mainLayout->addWidget(quitBtn);
22     setLayout(mainLayout);
23
24     connect(launcherBtn, SIGNAL(clicked()), this, SLOT(launchConversion()));
25     connect(quitBtn, SIGNAL(clicked()), qApp, SLOT(quit()));
26
27     setGeometry(0, 25, 240, 320);
28     setWindowTitle("Launcher");
29 }
30
31 void Launcher::launchConversion()
32 {
33     pid_t pid = fork();
34
35     if (pid == 0)
36     {
37         qDebug("new process forked. PID is:%d\n", getpid());
38         int result = execl("../conversion/conversion", "conversion", 0);
39
40         if (result < 0)
41         {
42             qCritical("failed to launch application!\n");
43         }
44         _exit(-1);
45     }
46     else if (pid > 0)
47     {
48         qDebug("A child process with PID %d is just launched by me.", pid);
49     }
50 }

```

createScreen()函数中都是我们已经非常熟悉的代码了，我们添加了两个按钮并稍作布局，按钮 launcherBtn 被连接到了槽 launchConversion()，用来启动 conversion 程序。

槽 launchConversion()中在第 33 行用 fork()函数创建了一个新进程，35 行用来检查刚刚得到的 fork()函数的返回值，这是 fork()的基本用法，它运行之后的返回值对父进程是子进程的进程号，而对于子进程则返回 0，我们可以根据这个特点来确定哪个是父进程，哪个是子进程，并对它们分别进行处理。

35 行—44 行我们在子进程中通过 execl 来调用 12.3 中已经编译好的 conversion 程序，完成之后调用 \_exit()来退出。我们可以在/usr/include/unistd.h 中找到 execl 的函数原型：

```
int execl (__const char * __path, __const char * __arg, ...)
```

其中第一个参数设定路径，后面是运行程序时的参数值，比如 `arg[0]` 为需要运行的程序本身即“conversion”，`arg[1]` 及之后的都是运行 `conversion` 时的参数。注意这里我们并没有给定“-qws”。如果在 Linux 终端中我们直接运行 `../conversion/conversion` 是行不通的，因为它找不到 `Server`。

第 37 行用来在子进程中打印出它自己的进程号，用来帮助读者理解 `fork()` 的这种做法，可以对照第 48 行，在父进程中，我们得到的 `pid` 值就是子进程的进程号，这样我们打印出来的两个 `PID` 应该相等，读者可以在后面运行这个例子时验证这一点。

如果对 `fork()`, `getpid()`, `execl()`, `_exit()` 等函数还有疑问的读者，请参考相关的 Unix/Linux 书籍或文档，这些 Unix/Linux 的基础知识在常见的书籍或文档中都能找到，在这里我们也不给出某个具体的文档了。

这个 `Launcher` 程序的运行结果如图 12.7:



图 12.7 `Launcher` 程序

注意我们运行时不需要“`./launcher -qws`”而只需要“`./launcher`”就可以了。点击“`Launch Conversion`”按钮之后我们就可以看到如图 12.6 所示的 `conversion` 程序，控制台输出中有类似下面的信息：

```
new process forked. PID is:2555
```

```
A chile process with PID 2555 is just launched by me.
```

当然 `PID` 的值不一定是 2555，但这两个 `PID` 的值应该是一样的。

## 12.5 进程间通信

Linux 下的进程间通信的方法基本上是从 Unix 平台上继承而来的，比如管道、信号量、消息队列、共享内存及套接字（`socket`）等。而在桌面环境中，如 KDE 就在传统进程间通



信方式的基础上发展了更为方便的通信方式 DCOP，利用 DCOP 可以很方便地将强大的脚本功能添加到应用程序中，并且 KDE 桌面还附带了 dcop 和 kdcop 工具，使用和开发都非常方便。

Qtopia Core 中也定义了一种自己的进程间通信机制 QCOP，注意 QCOP 只在 Qt 的嵌入式版本（如 Qtopia Core 和以前的 Qt/Embedded）中提供，对于 Qt/X11 还是沿用 KDE 所提供的 DCOP。相对已经是轻量级的 DCOP，QCOP 作了进一步的简化，以提高它在嵌入式系统中的效率。

在 Qtopia Core 中 QCOP 机制用 QCopChannel 类来实现。QCopChannel 从 QObject 类继承而来，提供了静态函数 send() 来发送需要传递的消息和数据，以及 isRegistered() 来查询某个 Channel 是否已经被注册。当在 channel 中接收消息和数据时，我们需要构造一个 QCopChannel 的子类并重写 receive() 函数，或者提供一个槽并利用 connect() 函数将 received() 信号连接起来。

下面我们通过改写 12.3 中移植的温度转换程序（实际上就是第 10 章中所详细介绍的小例子），来说明一下如何在 Qtopia Core 中利用 QCopChannel 来实现进程间的通信。为简单起见我们去掉了保存数值的功能，将两个温度计分别用两个进程来表示，并利用类似上一节中的 Launcher 程序来当作 Server，并启动这两个温度计的进程。

我们在界面上作了一些小小的改动，因为毕竟分成了两个进程，不过这里假设读者朋友已经比较熟悉我们在界面相关代码上的改动了，我们可以在随书所附的光盘上看到完整的代码示例，这里我们重点来消息是怎么用 send() 来传送的。下面是摄氏温度计程序中 Cel.cpp 的相关代码片断：

```
81 void Cel::sendMsg(int celNum)
82 {
83     QByteArray data;
84     QDataStream out(&data, QIODevice::WriteOnly);
85     out << celNum;
86     QCopChannel::send("/System/Temperature", "ConvertCelToFah(int)", data);
87 }
```

第 86 行我们用 QCopChannel::send() 函数来发送消息，它的原型如下：

```
QCopChannel::send(const QString& channel,
                  const QString& message,
                  const QByteArray& data)
```

我们在发送时需要和接收方协商并定义好发送消息的 channel 和消息的内容，比如在这个例子中我们定义 channel 为 "/System/Temperature"，消息内容为 "ConvertCelToFah(int)"，注意这里加入了参数的类型，这只是一种惯例，而并不是必须的，你完全可以在发送和接收时直接用 "ConvertCelToFah"，不过这种写法可以提供参数的信息，便于更准确的辨认消息。第三个参数是需要发送的数据，我们用 QByteArray 来方便数据的传送，83 行到 85 行实际上就是构造了一个 QByteArray 对象，并给它赋值。

这个消息的接收在 Fah.cpp 中，即当摄氏温度计的数值发生变化时，这个进程所发送的消息由华氏温度计的进程所接收，并进行相应的动作：

```
63 void Fah::listenChannel()
64 {
65     QCopChannel *channel = new QCopChannel("/System/Temperature", this);
66     connect(channel, SIGNAL(received(const QString &, const QByteArray &)),
```

```

67         this, SLOT(handleMsg(const QString &, const QByteArray &)));
68     }
69
70     void Fah::handleMsg(const QString &message, const QByteArray &data)
71     {
72         QDataStream in(data);
73
74         if (message == "ConvertCelToFah(int)")
75         {
76             int celNum;
77             in >> celNum;
78             celToFah(celNum);
79         }
80     }

```

首先我们在 65 行生成一个 `QCopChannel` 对象以监听从 `channel "/System/Temperature"` 发送过来的消息，然后我们将 `received()` 信号连接到槽 `handleMsg()`。当然，这个 `listenChannel()` 最后是需要再构造函数中被调用，以保证能及时监听消息。

槽 `handleMsg()` 我们需要比对所传送过来的各种消息，仅仅当消息是我们之前所定义好的 `"ConvertCelToFah(int)"` 时，才从数据包中取出来摄氏温度计的当前数值，并调用私有函数 `celToFah()` 来真正改变摄氏华氏温度计的读数。

我们还需要将上一节的 `Launcher` 程序修改一下来启动这两个进程。修改后的代码在 `CopServ.cpp` 中，主要改动如下：

```

42     void CopServ::launchApp(const char* path)
43     {
44         if (path == NULL)
45         {
46             qDebug("Launch path is NULL!\n");
47         }
48
49         pid_t pid = fork();
50
51         if (pid == 0)
52         {
53             qDebug("new process forked. PID is:%d\n", getpid());
54             int result = execl(path, path, 0);
55
56             if (result < 0)
57             {
58                 qDebug("failed to launch application!\n");
59             }
60             _exit(-1);
61         }
62     }

```

```

63
64 void CopServ::launchCel()
65 {
66     launchApp("./cel/cel");
67 }
68 void CopServ::launchFah()
69 {
70     launchApp("./fah/fah");
71 }

```

其实主要就是增加了 `launchApp()` 函数——现在它变得更加容易被普遍使用了，没准你以后的 Qtopia Core 开发中就可以用到它呢。在这里我们采用相对路径来启动两个可执行文件 `cel` 和 `fah`，注意你需要把它们都编译好。

`CopServ` 类运行起来的结果如图 12.8 的左图所示，而右图则是分别将两个温度计启动之后的示意图。它们看起来还不如第 10 章的例子呢，我们在这里只是用这些简单的代码来说明如何利用 `QCopChannel` 类来实现进程间通信，实际应用中这种情况当然还是直接用 `Signal/Slot` 来得方便，只是有些复杂的情况需要不同进程间进行通信的话，我们就可以用到这种方便的 QCop 机制了。

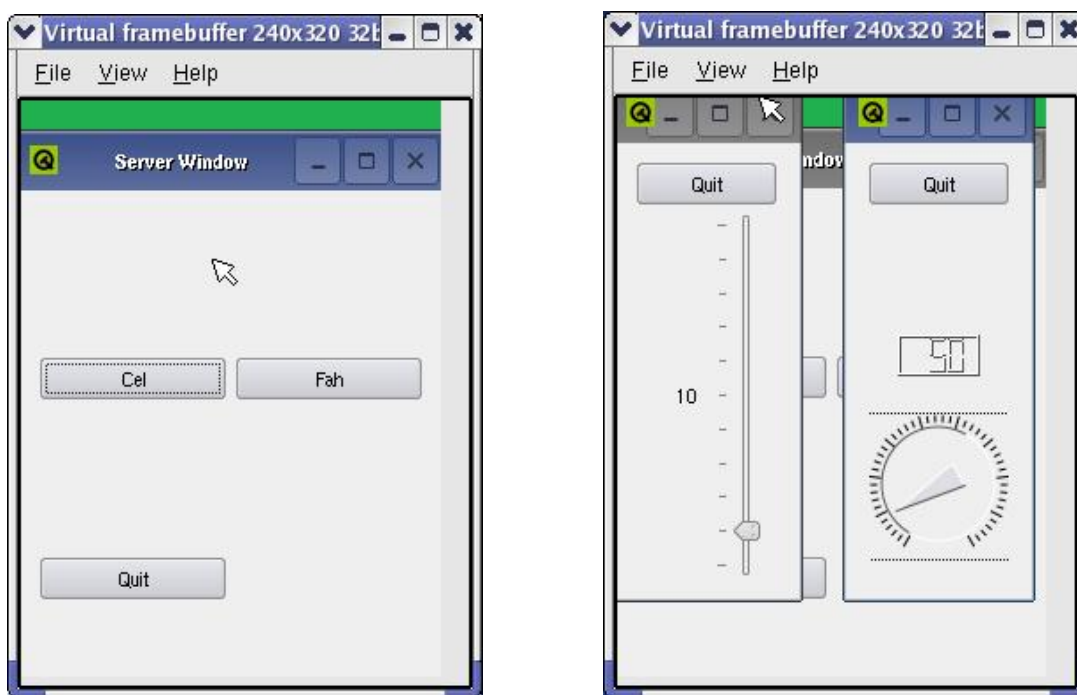


图 12.8 进程间的通信示例

## 12.6 本章小结

这一章我们主要通过比较 Qt/X11 和 Qtopia Core 来帮助大家从 Qt/X11 过渡到 Qtopia Core。在前几章已经能够熟练运用 Qt/X11 的基础上，我们只需要重点关注 Qtopia Core 的一些区别于 Qt/X11 的特点就可以很快熟悉 Qtopia Core 中的开发了。

在 Qtopia Core 的安装过程与环境搭建时，我们需要注意 `qvfb` 的编译和使用，它是一个

很方便的调试工具。

另外我们还分别举例介绍了 Qtopia Core 的轻量级窗口系统和 QCOP 进程间通信机制，这两者都是 Qtopia Core 中为提高效率而设计的，也需要我们熟练掌握。

## 12.7 常见问题

1. 将 Qt/X11 程序移植到 Qtopia Core 中需要注意些什么？反过来呢？

参考答案：将 Qt/X11 程序移植到 Qtopia Core 中，通常只需要重新编译就可以了，当然要注意用 Qtopia Core 的 qmake 而不是 Qt/X11 的 qmake。反过来从 Qtopia Core 向 Qt/X11 移植时则需要注意一些 Qtopia Core 中特有的类比如 等都不能再使用，实际上这种反过来的移植通常费力不讨好，很少有人这么做。

2. Qtopia Core 是 Qt/X11 的一个子集吗？为什么？

参考答案：尽管 Qtopia Core 对 Qt/X11 做了一些精简以减少系统开销、提高运行效率，但这并不意味着 Qtopia Core 是 Qt/X11 的子集，事实上由于 Qtopia Core 需要直接读写 Frame Buffer，它需要在 Frame Buffer 的基础上（而不是 Xlib 的基础上）来实现图形的绘制等，并且 Qtopia Core 中还提供了一些 Qt/X11 所没有的内容，比如 QWSServer 和 QCopChannel 等。从某种意义上来说，Qtopia Core 甚至可以看作是 Qt/X11 的一个超集。

3. qvfb 工具是一个 Qt/X11 程序还是一个 Qtopia Core 程序？

参考答案：qvfb 在 Qtopia Core 的开发中用于模拟 Frame Buffer，在 Qt/X11 开发中它确实没有多少用处，但它本身并不是一个 Qtopia Core 程序，而是一个 Qt/X11 程序。实际上它并不直接读写 Frame Buffer，而是运行在 X11 上，用来提供一种虚拟的 Frame Buffer，以方便 Qtopia Core 的开发和调试。

4. 如果想让一个 Qtopia Core 程序以 Server 模式运行，通常有哪些方法？

参考答案：通常有两种方法：一种是在运行 Qtopia Core 程序时在后面加上“-qws”参数，另一种是构造 QApplication 对象时指定第三个参数为 QApplication::GuiServer。

## 附录 A: 光盘内容

光盘中包含书中出现的所有代码，并且按章节存放，关于章节中的源代码都提供了相应的 README 说明文件，便于读者学习。

## 附录 B: 参考文献

1. [译者]魏永明、耿岳、钟书毅. Linux 设备驱动程序（第三版）[M]. 北京：中国电力出版社，2006
2. 刘淼. 嵌入式系统接口设计与 Linux 驱动程序开发[M].北京：北京航空航天大学出版社，2006
3. 李驹光. ARM 应用系统开发详解——基于 S3C4510B 的系统设计[M]. 北京：清华大学出版社，2005
4. Samsung Electronics Co. S3C2410X 32 bit Microprocessor User Manuel, 2003
5. 赵炯. LINUX 内核完全注释[M].北京：机械工业出版社，2004
6. Wookey. The GNU Toolchain for ARM Target HOWTO.
7. P. Raghavan.Amol Lad. SriramNeelakandan. Embedded Linux System Design and Development. Auerbach Publications,2006
8. Karim Yaghmour. Building Embedded Linux Systems.O'Reilly, April 2003
9. Linux 内核调试器内幕. IBM 软件工程师 Hariprasad Nellitheertha. 参考站点：  
<http://www.ibm.com/developerworks/cn/linux/l-kdebug/index.html>
10. Qt4.2 官方文档. Trolltech Corp. 参见 <http://doc.trolltech.com/4.2/>
11. C++ GUI Programming with Qt 4, Jasmin Blanchette, Mark Summerfield, 2006, Prentice Hall
12. Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995, 机械工业出版社（影印本）
13. The Linux/m6k Frame Buffer Device. Geert Uytterhoeven, 1998, 参见 <http://www.xfree86.org/3.3.6/fbdev.html>
14. The Embedded Linux GUI System. Chen Hanyi, 2001, 参见 [http://people.debian.org.tw/~moto/embedded/Embedded\\_Linux\\_GUI/Embedded\\_Linux\\_GUI.html](http://people.debian.org.tw/~moto/embedded/Embedded_Linux_GUI/Embedded_Linux_GUI.html)
15. Thinking in C++, 2<sup>nd</sup> edition, Bruce Eckel, 2000, MindView Inc.