

嵌入式

Linux

系统开发全程解析

韩超 等著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是一本全面介绍嵌入式 Linux 开发的专著，书中涵盖了程序生成工具、调试工具、引导加载器、Linux 系统结构、Linux 内核、驱动程序、用户空间编程、用户空间中间件等方面的内容。本书内容前后照应、贴近实践，且有较强的延伸型，有利于读者建立嵌入式 Linux 开发系统化的知识结构和技术理念。

本书不仅适用于嵌入式 Linux 的工程师增强能力，也适用于其他领域的技术人员了解嵌入式 Linux。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

嵌入式 Linux 系统开发全程解析 / 韩超等著. —北京：电子工业出版社，2014.5

ISBN 978-7-121-22888-9

I. ①嵌… II. ①韩… III. ①Linux 操作系统—程序设计 IV. ①TP316.89

中国版本图书馆 CIP 数据核字（2014）第 066337 号

策划编辑：李 冰

责任编辑：李云静

印 刷：中国电影出版社印刷厂

装 订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×1092 1/16 印张：27.25 字数：646 千字

印 次：2014 年 5 月第 1 次印刷

定 价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

本书写作目的

嵌入式 Linux 开发已经不算很新的技术。本书作者从 2003 年开始从事 Linux 方面的研究、开发和科普工作。在 10 年多的时间里，Linux 内核已经从 2.4 版本发展到了 3.x 版本，广泛使用的嵌入式处理器也从 ARMv4 的 ARM7 核心发展成为 ARMv6 的 Cortex 核心。这 10 年多嵌入式 Linux 技术逐步成熟，产品已经数不胜数。

作者在几年前出版过一些有关嵌入式系统、Linux 和 C 语言编程的书籍。目前面对技术的发展，感觉有必要出版一本全面介绍嵌入式 Linux 的书籍，以帮助相关行业的学习者和开发者更高效地了解嵌入式 Linux，更好地从事相关工作。

希望将本书打造成简明、高效的工具书，成为快速开发 Linux 软件的指导书、硬件移植的工具书，以及计算机专业从业者理论联系实际的桥梁。

本书特点

本书结合了作者多年的开发经验和知识技术的传播经验，主要有下面一些特点。

- 内容来自工程实践，实用性强。
- 覆盖面更全面、知识系统完整。
- 使用框架图+代码路径+关键代码的方式，一目了然。
- 内容紧凑，读者可以结合手头代码对照学习。
- 将工程技巧蕴含于理论知识的网络之中。
- 包含 Linux 软件编程开发的常用技巧：查找代码、运行时看信息等。
- 结合硬件和操作系统的知识。
- 帮助读者深入理解 Linux 系统的关键结构，具有完备的开发调试能力。
- 重点关注目前的主要应用场景：用户空间开发和驱动开发。

本书以 Linux 尤其是嵌入式 Linux 中最常用的内容为主，这些内容大部分是 Linux 开发不同方面的工程师均需要掌握的。通过对本书的学习，可以让有 C 语言基础的工程师在 Linux 环境中开发用户空间软件；让有硬件基础的工程师可以在嵌入式 Linux 平台中具有适

配硬件的能力；让具有一定经验的嵌入式 Linux 工程师具有更广泛的视野、更强的开发能力。

本书主要内容

本书包含了嵌入式 Linux 系统的主要内容，按照知识结构分成四个方面。

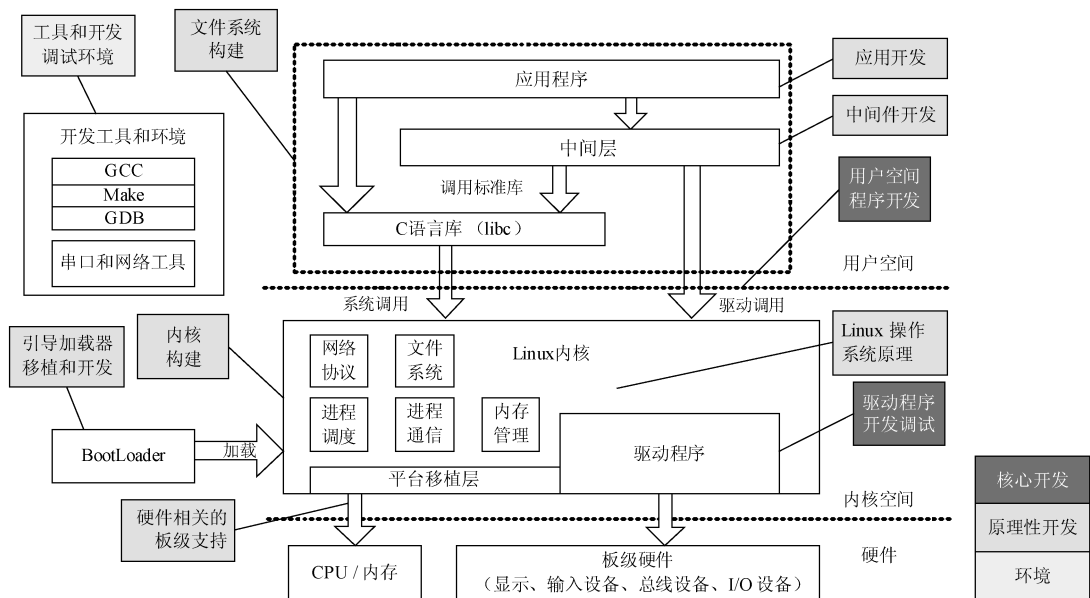
第一个方面：开发环境和编程基础（第 1 章到第 4 章）。

第二个方面：Linux 系统结构（第 5 章到第 8 章）。

第三个方面：嵌入式 Linux 的用户空间（第 9 章到第 13 章）。

第四个方面：嵌入式 Linux 的驱动开发（第 14 章到第 24 章）。

按照内容的侧重点，本书具有环境、原理性开发和核心开发几个方面。嵌入式 Linux 系统和本书知识结构如下图所示。



对读者的话

目前的 IT 技术领域有很多热点，除了嵌入式设备的开发外，还有移动开发、互联网开发等。嵌入式 Linux 是很多产品的技术基础。在实践过程中，很多问题都是嵌入式 Linux 最基础的问题，只是由于工程师可能来自其他领域，不熟悉嵌入式 Linux，从而小问题就成了大问题。因此，对于嵌入式 Linux，是目前从事 IT 技术领域的各类工程师都至少需要了解的。

本书不仅适用于嵌入式 Linux 的工程师增强能力，也适用于其他领域的技术人员了解嵌入式 Linux。本书尤其注重和高校计算机专业的互补关系，基于高校计算机专业知识基础，书中大量补充了在实践中的应用，帮助读者将知识“落地扎根”，引领读者进一步在工作中

让知识和技能“生根发芽”，直到在工作中“结出丰硕的果实”。

本书的几个基础方面是 C 语言编程、嵌入式处理器、操作系统，这也是计算机行业的基础。通过对本书的学习，读者得到的不仅仅是各个方面的知识和技巧，更有它们之间的有机结合。

本书作者

韩超是中国大陆长期工作于一线的知名工程师、架构师，也是嵌入式 Linux 相关技术在大陆发展 10 年的技术领航人之一，同时也是畅销书作者。其主要从事相关技术研发方向，包括嵌入式 Linux 板级平台、GUI 系统和应用、移动多媒体。韩超对嵌入式 Linux 的技术把握以实用技术为主，以操作系统本身为辅，重视在系统使用特定硬件的技术，重视内核与用户空间的交互的要点，适用于嵌入式 Linux 的软件工程等方面。

韩超完成了本书主要部分的编写工作，此外，众多不同规模的企业开发成果也为本书的编写提供了重要的素材。参与本书编写的还有康硕、于仕林、张超等人，以及清华大学计算机系操作系统研究兴趣小组的肖奇学、徐永健、王欢、何嘉权、范文良、茅俊杰等人。

目 录

第 1 章 Linux 的开发环境	1	3.2.3 Makefile 中的函数	36
1.1 开发环境概述	1	3.3 Makefile 使用示例	39
1.2 串口终端工具	2	3.3.1 简单的 Makefile	39
1.3 TFTP	6	3.3.2 依赖关系实例	39
1.4 NFS	7	3.3.3 隐含规则的编译实例	41
1.5 SAMBA 共享	8	3.3.4 指定依赖的编译实例	44
1.6 Linux 系统的软件发布协议	9	3.4 自动生成 Makefile	46
第 2 章 程序生成和 GCC	11	3.4.1 autoconf 工具介绍	46
2.1 程序生成工具概述	11	3.4.2 automake 工具介绍	46
2.1.1 GUN 的 GCC 工具	11	3.4.3 其他工具	47
2.1.2 ELF 文件格式	14	3.4.4 自动生成 Makefile 的流程	47
2.2 GCC 工具的使用	16	第 4 章 调试和 GDB	49
2.2.1 示例工程	16	4.1 嵌入式系统的调试技术	49
2.2.2 编译、汇编和连接	18	4.1.1 调试技术	49
2.2.3 预处理和汇编	20	4.1.2 硬件调试	50
2.2.4 归档工具 (ar) 和静态库	20	4.1.3 代码调试	51
2.2.5 动态库	22	4.2 Linux 的基本信息	51
2.2.6 ELF 格式文件信息读取		4.3 GDB 调试和远程调试	52
(readelf)	22	4.4 GDB 的安装与使用	57
2.2.7 符号信息工具 (nm)	25	4.4.1 使用 gdbstub 实现调试用户	
2.2.8 字符串工具 (strings)	26	程序	57
2.2.9 去除符号 (strip)	27	4.4.2 GDB 和 GDB Server 的编译	59
2.2.10 目标文件复制 (objcopy)	28	4.5 使用 gdbserver 调试	61
2.2.11 目标文件信息 (objdump)	28	第 5 章 Linux 系统的结构	65
第 3 章 工程管理和 make 机制	33	5.1 Linux 操作系统基本概念	65
3.1 make 工具	33	5.1.1 Linux 的进程信息	65
3.2 Makefile 的基本原则	34	5.1.2 Linux 的文件系统和文件	
3.2.1 Makefile 的变量	34	信息	70
3.2.2 Makefile 的条件执行	36	5.1.3 文件的另外三位属性	71
		5.2 Linux 系统的组成和构建	72

5.2.1	Linux 系统的组成	72
5.2.2	嵌入式 Linux 的构建	73
5.3	Linux 系统的启动流程	74
第 6 章	BootLoader 及其构建	76
6.1	嵌入式 Linux 的 BootLoader	76
6.1.1	BootLoader 的开发要点	76
6.1.2	BootLoader 的结构	78
6.2	U-Boot 的使用	80
6.2.1	U-Boot 概述	80
6.2.2	U-Boot 的结构	81
6.2.3	U-Boot 的生成	83
6.2.4	U-Boot 的启动流程	84
6.3	U-Boot 的命令	86
6.3.1	U-Boot 命令概述	86
6.3.2	增加命令	88
6.4	U-Boot 的移植	91
6.4.1	U-Boot 的移植概述	92
6.4.2	U-Boot 的扩展	92
6.4.3	板级支持	94
第 7 章	Linux 内核及其构建	97
7.1	Linux 内核概述	97
7.1.1	Linux 内核结构	97
7.1.2	Linux 源文件结构	98
7.2	嵌入式 Linux 的配置和编译	99
7.2.1	Linux 内核配置结构	99
7.2.2	Linux 内核的配置	99
7.2.3	Linux 内核的生成	107
7.3	Linux 内核的启动过程	108
7.4	特定系统的 Linux 的构建	114
7.4.1	Linux 内核的移植	114
7.4.2	ARM 处理器上运行的 Linux 系统	115
7.4.3	S3C6410 Linux 内核的构建	117
7.4.4	S3C6410 Linux 内核的移植内容	118

第 8 章	文件系统及其构建	123
8.1	Linux 文件系统特性	123
8.2	Linux 文件系统的结构	125
8.2.1	文件系统的主要接口	125
8.2.2	文件系统的实现	130
8.2.3	默认的公共实现	134
8.3	几种 Linux 使用的文件系统	136
8.3.1	EXT2/3 (扩展文件系统2/3)	136
8.3.2	NFS (网络文件系统)	136
8.3.3	ROMFS (只读文件系统)	137
8.3.4	CRAMFS (压缩 ROM 文件系统)	137
8.3.5	JFFS2 (日志 Flash 文件系统)	138
8.3.6	YAFFS (另一种 Flash 文件系统)	138
8.3.7	UBIFS (非排序块映像文件系统)	139
8.4	Linux 文件系统的构建	140
8.4.1	根文件系统的结构	140
8.4.2	制作根文件系统映像	141
8.4.3	内核启动中根文件系统的参数	142
第 9 章	Linux 用户空间的核心	143
9.1	嵌入式系统中的操作系统和系统关系	143
9.2	C 语言库	144
9.3	Shell 工具 Busybox	147
9.3.1	Busybox 配置和编译	148
9.3.2	Busybox 的源代码结构	150
第 10 章	Linux 用户空间的编程	152
10.1	Linux 用户空间编程概述	152
10.2	文件的相关内容	154
10.2.1	文件的打开、关闭和读写等	155
10.2.2	文件的控制、映射和查询等	157

10.2.3	文件的其他操作	158
10.3	进程相关的内容	159
10.3.1	fork 和 exec	159
10.3.2	管道	161
10.3.3	System V IPC	162
10.3.4	POSIX IPC	165
10.4	信号相关的内容	166
10.5	pthread 线程	168
10.5.1	线程的基本使用	169
10.5.2	线程的属性	171
10.5.3	线程互斥量	172
10.5.4	线程条件量	173
10.5.5	线程取消	175
10.6	dlopen 机制	176
10.6.1	dlopen 的结构和意义	176
10.6.2	在 C 语言中使用 dlopen	178
10.6.3	在 C++ 中使用 dlopen	180
第 11 章	Linux 用户空间的中间件	185
11.1	基于嵌入式 Linux 的系统与中间件	185
11.2	网络协议相关	186
11.2.1	Linux 套接字编程的基础	186
11.2.2	TCP 和 UDP 协议的流程	189
11.2.3	TCP 编程实例	189
11.2.4	UDP 编程实例	193
11.2.5	深入网络编程	196
11.2.6	用作 IPC 的 UNIX Socket	198
11.3	GUI 应用开发	201
11.3.1	Qt 系统	203
11.3.2	MiniGUI 应用程序	209
11.3.3	MicroWindows (Nano-X Window)	216
11.4	数据库	217
11.4.1	关于嵌入式数据库	217
11.4.2	SQLite	218
第 12 章	Linux 驱动基础	228
12.1	Linux 驱动概述	228

12.1.1	驱动的理念和结构	228
12.1.2	驱动程序对用户空间的接口	230
12.2	设备文件和相关文件系统	230
12.2.1	设备文件	230
12.2.2	sys 文件系统	231
12.2.3	proc 文件系统	233
第 13 章	Linux 的内核编程	237
13.1	Linux 内核编程概述	237
13.2	内核模块的编写	237
13.2.1	Linux 内核中的模块	237
13.2.2	内核模块的编译结构	239
13.3	内核编程接口	241
13.3.1	Linux 编程风格	241
13.3.2	Linux 编程主要接口	242
第 14 章	Linux 的驱动核心架构	248
14.1	用户空间的接口	248
14.1.1	用户空间的驱动调用接口	248
14.1.2	系统调用	248
14.1.3	驱动的主要调用函数	249
14.2	字符设备和块设备的框架	250
14.2.1	文件操作 file_operations	250
14.2.2	字符设备的基本框架	251
14.2.3	块设备的框架	252
14.2.4	字符设备和块设备的默认 file_operations 实现	254
14.3	网络协议和网络设备的框架	258
14.3.1	网络系统的核心	259
14.3.2	网络协议的实现	261
14.3.3	网络设备的框架	263
14.4	proc 文件系统的框架	264
14.4.1	proc 文件系统的编程接口	264
14.4.2	proc 文件系统的实现	266
14.5	sys 文件系统的框架	266
14.5.1	sys 文件系统的编程接口	266
14.5.2	sys 文件系统的实现	267

第 15 章 Linux 驱动的要 点.....269

- 15.1 驱动程序的核心实现269
- 15.2 设备、驱动和资源273
- 15.3 中断的处理276
- 15.4 中断的下半部277
 - 15.4.1 软中断277
 - 15.4.2 软中断之 tasklet278
 - 15.4.3 软中断之定时器279
- 15.5 竞态处理280
 - 15.5.1 自旋锁280
 - 15.5.2 信号量280
- 15.6 阻塞处理281
- 15.7 异步操作282

第 16 章 几个典型的简单驱动.....283

- 16.1 设备驱动概述283
- 16.2 内存设备驱动284
 - 16.2.1 内存设备驱动的公共内容.....284
 - 16.2.2 空设备286
 - 16.2.3 零设备287
 - 16.2.4 满设备288
- 16.3 内存块设备驱动288
- 16.4 回环块设备驱动291
- 16.5 回环网络设备驱动294

第 17 章 几个典型的驱动框架和相应 的驱动296

- 17.1 Misc 驱动框架296
- 17.2 帧缓冲驱动框架和具体驱动.....297
 - 17.2.1 帧缓冲驱动框架297
 - 17.2.2 虚拟帧缓冲驱动300
 - 17.2.3 针对硬件实现的帧缓冲
驱动302
- 17.3 输入-事件驱动框架305
 - 17.3.1 输入-事件驱动框架概述305
 - 17.3.2 针对硬件的事件驱动.....307
- 17.4 GPIO 驱动框架和具体驱动310

- 17.4.1 GPIO 驱动框架310

- 17.4.2 GPIO 具体硬件的驱动312

17.5 Power Supply 驱动框架和具体驱动 ... 312

- 17.5.1 Power Supply 驱动框架312

- 17.5.2 Power Supply 驱动313

17.6 TTY 驱动框架和驱动..... 314

- 17.6.1 TTY 驱动框架314

- 17.6.2 伪 TTY 驱动316

- 17.6.3 串口 TTY 和虚拟 TTY316

第 18 章 MTD 系统和驱动 318

18.1 MTD 概述 318

18.2 MTD 的核心..... 319

- 18.2.1 MTD 的接口部分320

- 18.2.2 MTD 的核心实现部分322

18.3 MTD 的设备层..... 322

- 18.3.1 MTD 字符设备.....322

- 18.3.2 MTD 块设备.....323

18.4 CFI 硬件实现层 324

- 18.4.1 公用部分324

- 18.4.2 ROM 的 MTD 实现.....325

- 18.4.3 RAM 的 MTD 实现.....325

- 18.4.4 Nor Flash 的 MTD 实现326

18.5 Nand Flash 的硬件实现层 326

- 18.5.1 公用部分326

- 18.5.2 GPIO 的 Nand Flash 实现327

- 18.5.3 处理器芯片上的 Nand Flash
实现330

第 19 章 USB 系统和驱动..... 331

19.1 USB 概述..... 331

- 19.1.1 USB 规范.....331

- 19.1.2 USB 的软件系统.....333

19.2 Linux 的 USB 主机端支持 334

- 19.2.1 USB 主机端的软件结构334

- 19.2.2 USB 主机端的核心部分335

- 19.2.3 USB 驱动的实现.....337

- 19.2.4 HCI 的实现.....339

19.3	Linux 的 USB 设备端支持	340	22.1.4	PCI 的总线配置	379
19.3.1	USB 设备端的软件结构	340	22.1.5	PCI 的发展和衍生标准	381
19.3.2	Gadget 的核心部分	340	22.2	PCI 总线的驱动框架	381
19.3.3	Gadget 驱动	342	22.3	PCI 设备的驱动	384
19.3.4	UDC 驱动的实现	345	22.3.1	PCI 的桩实现	384
第 20 章	SPI 总线和驱动	348	22.3.2	网卡的 PCI 实现	385
20.1	SPI 概述	348	第 23 章	音频系统和驱动	389
20.2	SPI 总线驱动的框架	349	23.1	音频系统概述	389
20.3	简单字符设备 spidev	353	23.2	OSS 架构	389
20.4	SPI 主控制器的实现	355	23.2.1	OSS 系统的结构	390
20.4.1	GPIO 实现的 SPI 主控制器	355	23.2.2	OSS 系统的核心	391
20.4.2	S3C64xx 的 SPI 主控制器	356	23.2.3	OSS 系统的实现	392
20.5	SPI 从设备的驱动	358	23.3	ALSA 架构	393
第 21 章	I2C 总线和驱动	361	23.3.1	ALSA 系统的结构	393
21.1	I2C 概述	361	23.3.2	ALSA 系统的核心	395
21.1.1	基本概念	361	23.3.3	ALSA 系统芯片层	395
21.1.2	SMBus	362	23.3.4	ALSA 的用户空间	400
21.2	I2C 总线驱动的框架	362	第 24 章	视频系统和驱动	403
21.2.1	I2C 核心框架	362	24.1	视频系统概述	403
21.2.2	I2C 总线接口	367	24.2	Video for Linux 系统	403
21.2.3	I2C 设备和驱动	368	24.2.1	基本结构	404
21.3	具体的 I2C 主控制器	370	24.2.2	Video for Linux 的核心 结构	405
21.4	I2C 从设备的驱动	372	24.2.3	Video for Linux 的其他 方面	410
第 22 章	PCI 总线和驱动	375	24.2.4	Video for Linux 驱动的 接口	413
22.1	PCI 概述	375	24.2.5	Video for Linux 驱动的实 现层	417
22.1.1	PCI 的基本结构	375			
22.1.2	PCI 的总线信号	377			
22.1.3	PCI 的总线操作	378			

第1章

Linux 的开发环境

1.1 开发环境概述

在 Linux 和嵌入式 Linux 的开发中，一般将生成代码的机器称为“主机”，而将运行代码的机器称为“目标机”。主机和目标机不是同一个机器的开发，则被称为交叉开发。

主机和目标机代码生成的过程如图 1-1 所示。

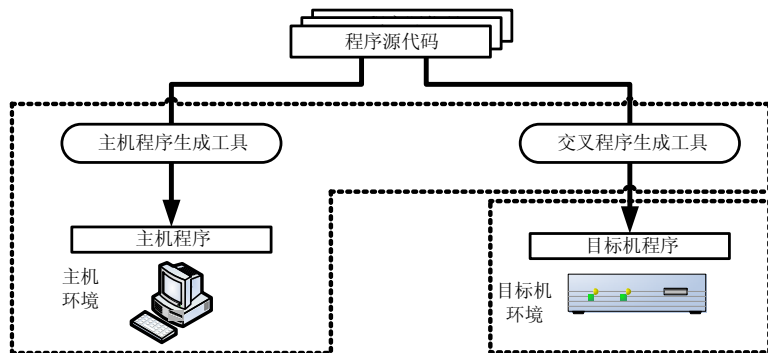


图 1-1 主机和目标机的代码生成

如果是单纯桌面（x86）上的程序开发，主机和目标机的结构类似，生成的程序一般可以在开发机和目标机同时运行。如果在嵌入式系统中开发，通常在主机环境中开发，通过交叉编译工具生成目标机的程序，然后通过某种手段放入目标机系统中运行。

对于嵌入式的开发，一般分成主机和目标机通常是两个不同的设备。在嵌入式 Linux 的开发中，主要是指目标机使用 Linux 作为操作系统，并在操作系统上构造不同应用。嵌入式 Linux 开发的主机环境，主要是指在主机上生成运行于目标机的代码。主机一般是运行 Linux 操作系统的 x86 主机，也可以在 Windows 上开发嵌入式 Linux。

Linux 系统最基本的程序开发是 C 语言的程序开发。C 语言程序的生成成分编译、汇

编、连接等几个步骤。最终的目标文件的主要部分是处理器可执行的机器代码组合。根据系统的不同，生成的目标包括了可执行的二进制机器代码，也包括一定的头信息。

Linux 系统的开发主要涉及了如下内容：

- GCC 和交叉 GCC 均运行于开发机，但程序生成的部分存在区别。
- Make 和 Makefile 运行于开发机。
- GDB 可分为运行于主机的部分和运行于目标机的部分，也适用于本机开发。
- 主机-目标机的连接方式有多种，也适用于本机开发。

在嵌入式系统的开发中，除了程序生成工具之外，还需要使用主机到目标机的通信工具。主要的工具包括目标机的终端模拟工具和文件传输工具，前者一般使用从主机到目标机的串口连接完成，后者一般使用网络协议来实现。

嵌入式系统主机和目标机的连接如图 1-2 所示。

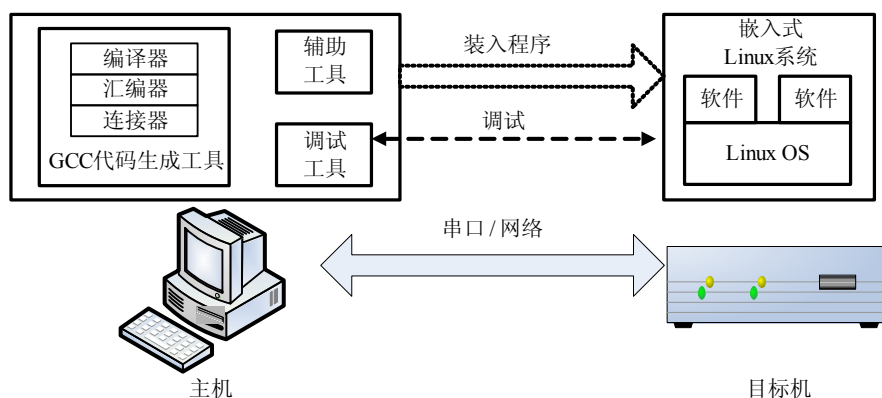


图 1-2 主机-目标机的开发结构

在嵌入式系统的开发中，需要使用主机到目标机的通信工具。主要的工具包括目标机的终端模拟工具和文件传输工具，前者一般使用从主机到目标机的串口连接完成，后者一般使用网络协议来实现。

主机到目标机通常有两条通道：一条是目标机 UART 到主机串口的连接，可以提供打印信息和控制台支持；另一条是网络连接，可以建立于网络协议 TCP/IP 之上，这样可以屏蔽数据链路层以下的细节，并且有多种基于上层协议的软件可以使用。基于网络协议的通信不仅限于在网络芯片硬件上，也可以通过 USB 或其他通信端口作为网络协议的硬件基础。

1.2 串口终端工具

串口是嵌入式开发过程中，主机和目标机最基本的连接方式，可以用于字符形式的输出和输入。例如：在 C 语言的程序开发中，使用 `printf()` 作为调试语句，主机的终端常使用显示器作为屏幕输出。在嵌入式系统的目标机调试过程中，由于一般不具有屏幕等输出设

备，因此通常使用目标机处理器的串口作为输出，在主机端使用串口工具接收这些信息。同样，在目标机需要交互调试的时候，串口的连接方式也可以提供目标机的输入，例如：`getchar()`、`scanf()`等功能。在开发过程中，终端虽然硬件基础不一样，但是其表现形式是类似的。

使用 UART 连接的几个主要参数。

- 波特率 (Speed, Baud rate)，串口数据的速率，一般可能有 1200 b/s、38 400 b/s、115 200 b/s 等。
- 数据位 (Data)：通常有 5、6、7、8 几种。
- 奇偶校验方式 (Parity)：一般有无校验 (None)、偶校验 (Even)、奇校验 (Odd)、标记 (Mark)、空格 (Space) 等。
- 停止位 (Stopbits)：一般有 1 位、2 位两种方式。
- 流控制 (Flow Control)：使用硬件流控制或者软件流控制。

提示：如果主机的串口终端的输出为乱码，则通常是波特率设置的问题；如果主机串口的控制台不能输入，通常是流控制的设置问题。

对于主机上使用的串口终端工具，在 Windows 系统中可使用超级终端，在 Linux 系统中可以使用 minicom 工具。

1. Windows 的超级终端

超级终端是 Windows 自带的工具，在使用的过程中很简单。在超级终端中，选择“文件”→“新建连接”，在建立的过程中，需要设置连接方式为 com (串口)，根据实际情况选择使用 com1 或 com2，设置的对话框如图 1-3 所示。



图 1-3 主机-目标机的连接

以上设置的参数，需要与目标机处理器的 UART 端口的设置一致，目标机的设置一般在 UART 的初始化过程中通过配置寄存器完成。

2. Linux 的 minicom

minicom 是 Linux 下的串口终端工具,使用 minicom 可以在 Linux 下实现目标机和主机的连接。minicom 可以建立于 Linux 的串口设备 (ttyS0) 或者调制解调器设备 (modem) 上,其使用起来没有 Windows 下的工具友好,需要从命令行启动。

minicom 命令行启动的参数如下所示:

```
minicom [OPTION]... [configuration]
```

minicom 命令主要参数的含义如表 1-1 所示。

表 1-1 minicom 的参数

参数	含义
-s, --setup	输入建立模式, 仅在 root 上
-o, --noinit	不要初始化调制解调器
-m, --metakey	使用 Alt 或 meta 键作为命令键
-M, --metakey8	使用 8 位的 meta 键作为命令键
-l, --ansi	逐字翻译, 假设屏幕使用 IBM-PC 字符集
-L, --iso	同 L, 但是假设屏幕使用 SO8859
-w, --wrap	换行
-z, --statline	使用终端状态行
-8, --8bit	8 位干净模式, 例如为日本字符所使用的
-c, --color=on/off	ANSI 样式的颜色使用开关
-a, --attrib=on/off	用户反转高亮的开关
-t, --term=TERM	重载 TERM 环境变量
-S, --script=SCRIPT	使用启动的脚本文件
-d, --dial=ENTRY	从拨号目录拨入 ENTRY
-p, --ptty=TTY	连接伪终端
-C, --capturefile=FILE	启动捕获文件 FILE
-v, --version	输出版本信息并退出
configuration	配置文件

一种较为方便的方式是使用下列命令启动交互式配置界面:

```
# minicom -s
```

在 minicom 主配置界面中, Exit from Minicom 表示退出 minicom 应用, Exit 表示退出配置程序, 进入 minicom 主程序。Save setup as df1 和 Save setup as 可以选择保存当前的配置。minicom 选择 Save setup as df1 时, 配置将保存在/etc/minirc.df1 文件中。

配置的主要工作是针对串口的参数设置, 因此选择 Serial port setup 选项。

minicom 主配置界面和串口配置界面如图 1-4 所示。

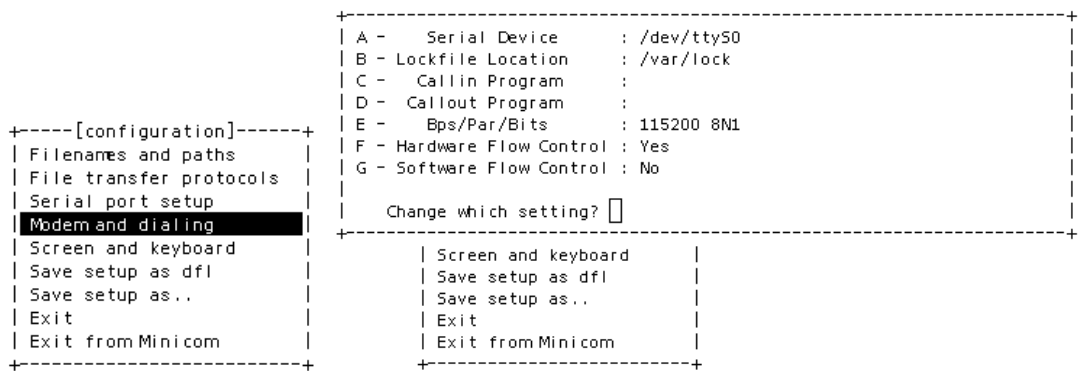


图 1-4 minicom 主配置界面（左）和串口配置界面（右）

在 minicom 串口界面中，可以使用 A~G 等字母选择相应的配置内容，A 用于选择串行的设备，使用 Linux 下的设备名称，如/dev/modem、/dev/ttyS0 等。

在串口终端的配置中，主要需要设置的内容是波特率、数据位等内容，使用键 E 启动串口的配置界面，如图 1-5 所示。

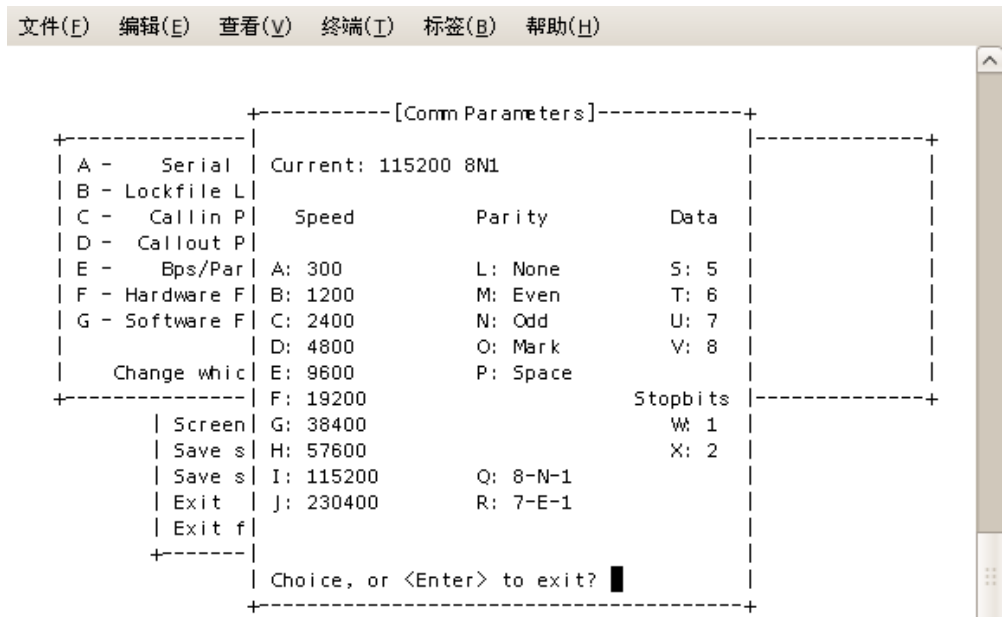


图 1-5 串口波特率、数据位、奇偶校验、停止位界面

本界面用于配置，串口波特率、数据位、奇偶校验、停止位使用按键选择实现。在配置完成后，可以使用没有命令行参数的方式启动 minicom。

在 minicom 启动主界面中如果有来自串口的数据，将在屏幕上显示。在主界面下可以继续使用 Ctrl+A 组合键，然后再按 Z 键实现进行配置。minicom 启动主界面的帮助界面，如图 1-6 所示。

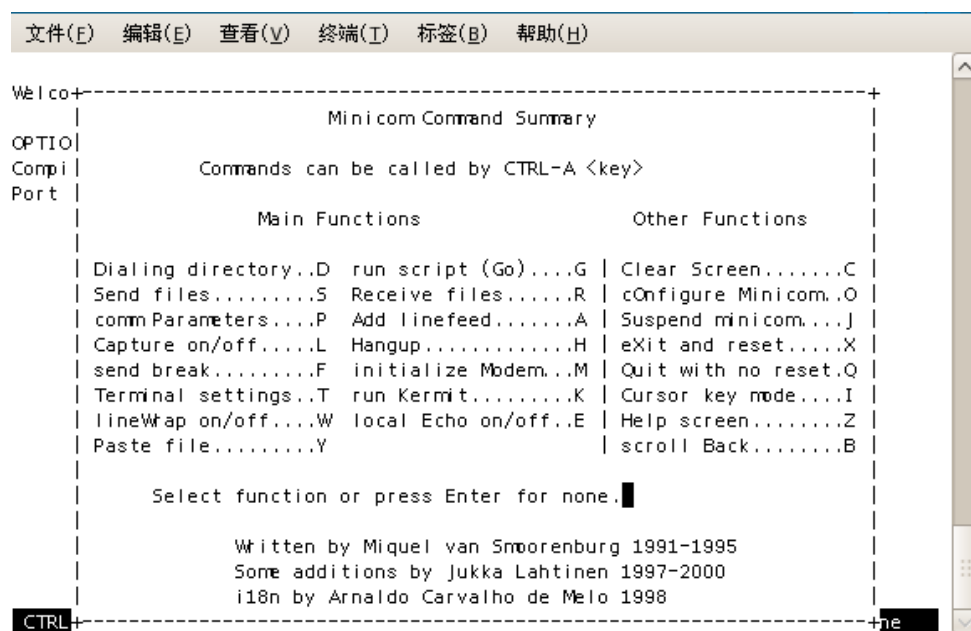


图 1-6 minicom 启动主界面的帮助界面

在主界面的帮助界面中，可以完成一些使用过程中的配置和辅助功能。

1.3 TFTP

TFTP 的全称为 Trivial File Transfer Protocol（简单文件传输协议）。此协议设计时的目标是进行小文件传输的。与基于 TCP 的 FTP（File Transfer Protocol，文件传输协议）不同，TFTP 使用 UDP 的端口 69。TFTP 只能从文件服务器上获得或写入文件，不能列出目录，不进行认证，按照 8 位来传输数据。

TFTP 使用以下 3 种传输模式。

- Netascii：这是 8 位的 ASCII 码形式。
- octet：这是 8 位源数据类型。
- mail：它将返回的数据直接返回给用户而不是保存为文件。

在嵌入式系统的开发过程中，TFTP 常常被使用作为主机-目标机文件传输的协议。一个典型的用法是在主机端开启 TFTP 服务器，在目标机端由是支持 TFTP 的 Bootloader 或者运行于嵌入式 Linux 操作系统的 TFTP 客户端。

例如：在 Ubuntu 环境中设置 TFTP 服务器，需要下载以下的包：

```
$ sudo apt-get install tftpd-hpa
```

进一步，修改 etc 目录中的配置文件/etc/default/tftpd-hpa：

```
#Defaults for tftpd-hpa
RUN_DAEMON="no"
OPTIONS="-l -s /var/lib/tftpboot"
```

修改设置如下：

```
#Defaults for tftpd-hpa
RUN_DAEMON="yes"
OPTIONS="-l -s /home/hanchao/tftp"
```

其中，`OPTIONS="-l -s"`后面的内容，表示 tftp 服务器的目录。

重新启动 TFTP 服务：

```
$ sudo /etc/init.d/tftpd-hpa restart
```

在目标机端需要使用支持 tftp 的程序，可以完成到主机的传输。在运行 Linux 系统的情况下，使用 TFTP 客户端的方式如下：

```
$ tftp 127.0.0.1
tftp> get abc.txt
tftp> quit
```

或者按照如下的方式直接运行命令：

```
$ tftp 127.0.0.1 -c get abc.txt
```

1.4 NFS

NFS（Network File System，网络文件系统）是 FreeBSD 支持的文件系统中的一种。它允许一个系统在网络上与他人共享目录和文件。通过使用 NFS，用户和程序可以像访问本地文件一样访问远端系统上的文件。

在嵌入式系统调试中，常用的使用方式是由主机作为 NFS 服务器，目标机通过 NFS 挂载（mount）主机的文件系统，这样在目标机的调试过程中，就可以使用主机上的文件，而不需要反复将程序烧写或者下载到目标机上。

在主机端，需要配置 NFS 服务器，在 Linux 和 Window 等操作系统中均可以使用，配置的方式有所不同。在目标机端，使用 NFS 服务共享主机的内容。

例如：在 Ubuntu 的主机环境中设置 NFS 服务器，需要下载以下的包：

```
$ sudo apt-get install nfs-kernel-server
```

进一步，修改配置文件 `/etc/exports`，增加 NFS 文件的路径：

```
# /etc/exports: the access control list for filesystems which may be exported
# to NFS clients. See exports(5).
#
# Example for NFSv2 and NFSv3:
#/srv/homes hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4 gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes gss/krb5i(rw,sync,no_subtree_check)
#
/home/hanchao/nfs *(rw,no_root_squash)
```

最后一行为增加的有效内容，其表示了 NFS 服务器在主机上的路径和所使用的权限。

重新启动 NFS 服务的命令如下所示：

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

在命令执行的参数中，start 表示启动，stop 表示停止，一般使用 restart 即可。

在目标机端，如果是直接运行的 Linux 操作系统，则可以直接挂载 NFS 文件系统，一般使用如下格式的命令：

```
$ mount -t nfs -o nolock $(server_IP) :$(server_dir) $(mount_dir)
```

例如：主机 NFS 服务器的地址配置为 192.168.1.1，将其目录/nfs/rootfs 挂接到/mnt 下，在目标机命令行中需使用的命令如下所示：

```
$ mount -t nfs -o nolock 192.168.1.1:/home/hanchao/nfs /mnt
```

目标机需要支持 mount 命令，一般嵌入式 Linux 命令行中的 busybox 就可以支持。

1.5 SAMBA 共享

Linux 主机可以使用 Samba 与 Windows 实现文件的共享。Samba 是基于网络的共享，此协议在 Linux 主机上为 Server Message Block，在 Windows 主机上为 NetBIOS 和 LanManager 协议。

在 Windows 和 Linux 中均可以通过 ip 和路径访问 Samba 服务器，方法如下所示。

- （Windows）在“运行”界面（通过 Ctrl+R 组合键启动）中输入：\\<ip>\<path>。
- （Linux）在文件管理器的地址栏（有时需用 Ctrl+L 组合键显示）输入：smb://<ip>/<path>。

Ubuntu 中安装 Samba 的方法为：

```
sudo apt-get install samba smbfs
sudo apt-get install samba-common-bin smbclient samba-common
```

Samba 服务器的配置文件为/etc/samba/smb.conf，典型的配置项如下所示：

```
security = share
[Share]
comment = shared
path = /home/hanchao/shared
public = yes
writable = yes
valid users = nobody
create mask = 0777
directory mask = 0777
force user = nobody
force group = nogroup
available = yes
browseable = yes
```

主要通过 path 表示主机共享的目录，其他参数表示权限。以上内容可以支持多组，表示共享多个目录。

启动、停止、重新启动 Samba 服务器的方法如下所示：

```
$ sudo /etc/init.d/samba start
$ sudo /etc/init.d/samba stop
$ sudo /etc/init.d/samba restart
```

提示：在一台 Linux 主机中可以将 TFTP、NFS 和 Samba 设置为同一个路径，这样在任何另外的机器上就都可以进行调试。

1.6 Linux 系统的软件发布协议

Linux 本身以自由软件开发为重要理念，其中的大部分软件开发也基于开源软件的原则。无论 Linux 内核还是用户空间的程序开发，很多都是基于开源软件构件的。

自由软件基金会（Free Software Foundation, FSF）是一个致力于推广自由软件的美国民间非营利性组织。它于 1985 年 10 月由理查德·斯托曼创建。自由软件基金会的主要工作是执行 GNU 计划，开发更多的自由软件。FSF 的网站为：<http://www.fsf.org>。

GNU 是 Gnu's Not UNIX 的缩写，这是一个类 UNIX 的操作系统，由 GNU 计划推动，目标在于建立一个完全相容于 UNIX 的自由软件环境。发展 GNU 系统是自由软件基金会最早致力的目标。

GPL 也称为 GNU GPL（GNU General Public License，GNU 通用公共许可协议），是一个广泛被使用的自由软件许可协议条款，最初由理查德·斯托曼于 1989 年为 GNU 计划而撰写。GPL 目前已经有 GPLv1（1989）、GPLv2（1990）和 GPLv3（2006）几个版本。

GPL 授予程序接受人以下权利，或称“自由”：

- 以任何目的运行此程序的自由。
- 再发行复制件的自由。
- 改进此程序，并公开发布改进的自由（前提是能得到源代码）。

copyleft 是一个与 copyright 相对的概念，利用版权法来达到与其相反的目的：copyleft 给人不可剥夺的权利，而不是版权法所规定的诸多限制。

GPL 不会授予许可证接受人无限的权利。再发行权的授予需要许可证接受人开放软件的源代码及所有修改，且复制件、修改版本，都必须以 GPL 为许可证。这些要求就是 copyleft，它的基础就是作品在法律上版权所有。由于它的版权所有，许可证接受人就无权进行修改和再发行，除非它有一个 copyleft 条款。如果某人想行使通常被法律所禁止的权利，只需同意 GPL 的条款。相反地，如果某人发行软件违反了 GPL（例如：不开放源代码），他就有可能被原作者起诉。

GPL 和 LGPL 的标志如图 1-7 所示。



图 1-7 GPL 和 LGPL 的标志

LGPL (GNU Lesser General Public License, GNU 宽通用公共许可证, 或 GNU 轻量级通用公共许可证), 是一个 GPL 协议的衍生版本。在 GPLv2 发布的时候同时发布, 称为 LGPLv2.0, 后又发展成了 LGPLv2.1。

LGPL 比 GPL 对自由化的要求要少。LGPL 与 GPL 的最大不同是, 可以私有使用 LGPL 授权的自由软件, 开发出来的新软件可以是私有的而不需要是自由软件。所以, 任何公司在使用自由软件之前应该保证在 LGPL 或其他 GPL 变种的授权下。

GPL 更好地保证了软件开发的自由性, LGPL 则是为了在自由基础上更好地支持商用软件的开发。如果软件按照 GPL 协议发布, 使用者的程序也必须公布其源代码, 同时允许别人发布、修改, 这就让自由软件产生蔓延。而按照 LGPL 发布的软件被使用后, 使用者的软件不一定被要求是自由软件。

BSD 许可 (Berkeley Software Distribution license) 也是自由软件中广泛使用的许可, 源自加州大学伯克利分校。与 GPL 相比, BSD 对著作权的限制比较宽松, 因此也被认为是 copyleft (就是介于 copyright 与 GPL 的 copyleft 之间)。BSD 可以允许自由软件的后续版本转为非自由软件。

BSD 协议原本有与 GPL 相冲突的地方, 此冲突已经在 1999 年的版本中被去除。在一个软件的开发中, 软件作者可以将 BSD 和 GPL 相结合。

Apache 许可是著名的非盈利开源组织 Apache 采用的协议, 最初为 Apache 的 Web 服务器所写。Apache 协议和 BSD 类似, 同样鼓励代码共享和尊重原作者的著作权, 同样允许代码修改, 并且可以作为开源软件或者商业软件再发布。Apache 许可有 Apache v1.0、Apache v1.1 和 Apache v2.0 几个版本。

Apache 的主要内容如下所示:

- 需要给代码的用户一份 Apache 许可。
- 如果你修改了代码, 需要在被修改的文件中说明。
- 在延伸的代码中 (修改和由源代码衍生的代码中) 需要带有原来代码中的协议、商标、专利声明及其他原来作者规定需要包含的说明。
- 如果再发布的产品中包含一个 Notice 文件, 则在 Notice 文件中需要带有 Apache 许可。可以在 Notice 中增加自己的许可, 但不可以表现为对 Apache 许可进行的更改。

Apache 许可也是对商业应用友好的许可。使用者也可以在需要的时候修改代码来满足需要并作为开源或商业产品发布/销售。

MIT (名称来自 Massachusetts Institute of Technology, 麻省理工学院), 与其他常见的 LGPL 和 BSD 软件许可协议相比, 其赋予软件被授权人更大的权利与更少的限制。

根据 MIT 协议, 被授权人有权利使用、复制、修改、合并、出版发布、散布、再授权及贩售软件及软件的副本。被授权人可根据程序的需要修改许可协议为适当的内容。在软件和软件的所有副本中都必须包含版权声明和许可声明。MIT 的最大特点就是可依照程序著作权者的需求更改内容。

第 2 章

程序生成和 GCC

2.1 程序生成工具概述

2.1.1 GUN 的 GCC 工具

1. GCC 工具概述

GCC 是 GNU Compiler Collection 的缩写，是一个非常优秀的跨平台编译器集合，支持 x86、ARM、MIPS 和 PowerPC 等多种目标平台，支持 C、C++、Java、ADA、FORTRAN 和 Pascal 等多种高级语言。

GCC 完成从 C、C++、Objective-C 等源文件向运行在特定 CPU 硬件上的目标代码的转换。对于通用计算机，一般使用 GCC 生成 x86 的可执行代码；对于嵌入式开发系统，使用交叉编译的 GCC，生成目标机可以执行的程序。GCC 默认处理的文件类型如表 2-1 所示。

表 2-1 GCC 默认处理的文件类型

文件类型	扩展名	文件说明
文本文件	*.c	C 语言源文件
	.C 、.cxx、*.cc	C++语言源文件
	*.i	预处理后的 C 语言源文件
	*.ii	预处理后的 C++语言源文件
	*.s *.S	汇编语言
	*.h	头文件
二进制文件	.o	目标文件
	.so	动态库
	.a	静态库

GCC 是一组工具的集合，包含了预处理器、编译器、汇编器、连接器等部分。当使用 GCC 的时候，将根据需要调用所需要的工具。对于默认的文件名，GCC 可以自动选择工具自动完成文件处理过程。从 C 语言的源程序到可执行文件，实质上是依靠 GCC 调用一系列的工具完成。

2. 编译工具和 Binutils

GCC 的核心是编译工具 `gcc`，用于编译 C 程序，另外还有一些二进制工具。

Binutils 是一个二进制工具的集合，包含了汇编、连接以及一系列的辅助工具。Binutils 是辅助 `gcc` 的重要工具。Binutils 工具如下所示。

- `as`: GNU 汇编器，用以将处理器的汇编代码转换成可执行代码，并存储到目标文件.o 中。
- `ld`: GNU 连接器，用于将一个或多个目标文件.o、库组合成一个可执行程序；或者生成静态库和动态库。
- `ar`: 归档工具，可以将多个文件组合成一个大文件，并且可以读取原始文件的内容。
- `stripe`: 去除文件中的符号。
- `nm`: 用以显示目标文件中的符号。
- `objectcopy`: 转换二进制代码的工具。
- `objdump`: 显示目标文件的反汇编工具。
- `readelf`: 显示 ELF 文件中的各种信息。
- `string`: 显示文件中的可打印字符。
- `ranlib`: 产生归档文件的索引，并将其保存到归档文件中，索引同时列出归档文件各成员所定义的可重分配目标文件。
- `addr2line`: 可以将一个可执行程序的地址映射到源文件的对应行。
- `gprof`: 显示程序调用段的各种数据。

GCC 工具的前缀表示编译工具的类型，依照系统不同，应该使用不同的工具，例如在 x86 Linux 的环境下，编译本机 (x86) 的程序，相应的工具为 `gcc`、`as`、`nm`、`objdump`、`stripe` 等。在 x86 Linux 的环境下，编译 ARM 体系标准 Linux 的程序，工具为 `<prefix>-gcc` 等。其中的 `<prefix>` 根据版本不同，例如 `arm-linux`-等。

提示：Binutils 工具中，除了反汇编的 `objdump` 外，其他的在各个体系结构中基本通用。

3. 基本的程序生成流程

`gcc` 和 `g++` 分别是 GCC 中的 C 语言编译器和 C++ 语言编译器。在命令行执行 `gcc` 的时候，也可能调用 Binutils 当中的二进制工具。

使用 `gcc` (或者 `g++`) 对 C 语言程序的处理如图 2-1 所示。

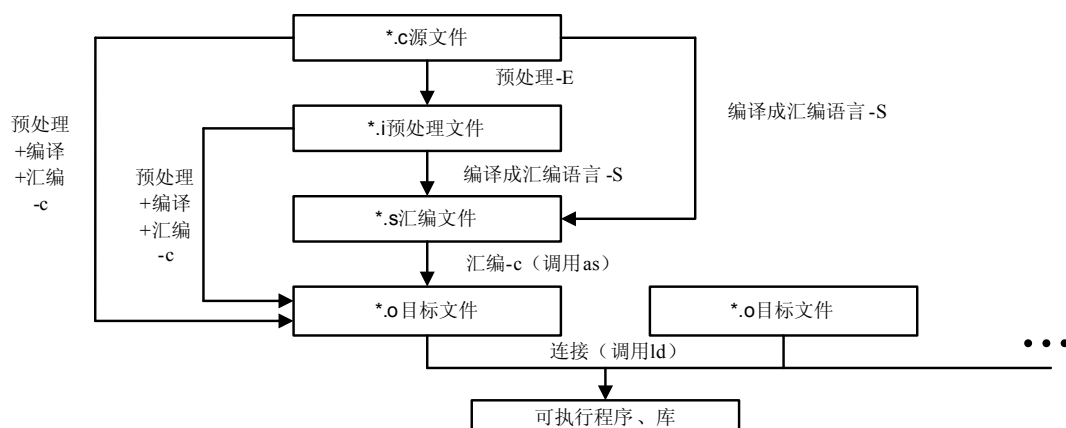


图 2-1 使用 gcc 的程序生成过程

gcc 的开关共分为 11 类，这些开关从 11 个方面控制着它的运行，以达到特定的编译目的。其中主要的开关是：全局开关、语言相关开关、预处理开关、汇编开关、连接开关。

一般来说，gcc 或 g++ 在执行编译工作时，可以分成以下 4 步。

- 预处理：生成后缀名为.i 的预处理后的文件（由预处理器 cpp 完成）。
- 编译：将预处理后的文件转换成汇编语言，生成后缀名为.s 的汇编文件（由编译器 gcc 完成）。
- 汇编：由汇编代码生成目标代码，即机器代码，生成后缀名为.o 的目标文件（由汇编器 as 完成）；
- 连接：由各个文件的目标代码，生成可执行程序（连接器 ld）。

运行编译工具的时候，会根据每一个 C 语言或者 C++ 的源文件生成一个以.o 为后缀名的目标文件，然后将各个目标文件组合成一个可执行程序。对于生成目标文件的过程，实际包括了预处理、编译和汇编 3 个步骤；而组合目标文件的过程即连接。因此，对于用户来说，前 3 个步骤很可能看上去是一步完成的。

提示：从习惯上，有时统一将预处理-编译-汇编 3 个步骤的组合称为“编译”。

gcc 需要使用各种开关，开关控制着 gcc 将调用何种工具。直接使用 gcc 生成可执行程序的例子，如下所示。

```
$ gcc main
```

语句的含义是用 main.c 生成可执行程序，生成的可执行程序采用默认的名称。

gcc 更多的时候需要加参数运行，例如：

```
$ gcc main.c -o Main
```

-o 的含义是指定生成的可执行程序的名称为 Main。

```
$ gcc -c main.c
```

该命令的含义是将 main.c 进行编译和汇编生成目标文件，但不将目标文件连接成可执

行程序，目标文件的名称采用默认名称。

全局开关用来控制 gcc 的 4 个步骤的运行。在默认的情况下，这 4 个步骤都是要执行的，但是当给定一些全局开关后，这些步骤就会在某一步停止执行，产生中间结果，例如可能只需要中间生成的预处理的结果或者是汇编文件。

gcc 的全局开关还决定了 gcc 将调用何种工具完成动作，也决定了后面的编译选项。例如使用如下语句：

```
$ gcc -S main.s -o main.o
```

本语句为单纯的汇编过程，将调用 as 工具，其后的选项和 as 工具一致；

```
$ gcc main.o -o main
```

本语句为单纯的连接过程，将调用 ld 工具，其后的选项和 ld 工具一致。

2.1.2 ELF 文件格式

ELF (Executable and Linking Format, 可执行连接格式) 是 UNIX 系统实验室 (USL) 作为应用程序二进制接口 (Application Binary Interface, ABI) 而开发和发布的。工具接口标准委员会 (TIS) 选择了正在发展中的 ELF 标准作为工作在 32 位 Intel 体系上不同操作系统之间可移植的二进制文件格式。如果开发者定义了一个二进制接口集合，ELF 标准用它来支持流线型的软件发展。应该减少不同执行接口的数量，因此就可以减少由于重新编程所需要重新编译的代码。

在 Linux 下开发程序可以使用 ELF 格式作为可执行程序、库以及程序生成过程的中间文件。这与 ELF 文件格式包括 3 种主要的类型对应，如下所示。

- 可执行文件 (应用程序)：可执行文件包含了代码和数据，是可以执行的程序。
- 可重定向文件 (*.o)：可重定向文件包含了代码和数据（它们是和其他重定位文件和共享的 object 文件一起连接时使用的）。
- 共享 object 文件 (*.so)：包含了代码和数据，它们是在连接时被连接器 ld 和运行时动态连接器使用的。动态连接器可能称为 ld.so.1、libc.so.1 或者 ld-linux.so.1。

object 文件参与程序的连接 (创建一个程序) 和程序的执行 (运行一个程序)。object 文件格式提供了一个方便有效的方法，以并行的视角看待文件的内容，在它们的活动中，反映出不同的需要。

一个 ELF 头在文件的开始，保存了路线图 (road map)，描述了该文件的组织情况。sections 保存着 object 文件的信息，从连接角度看：包括指令、数据、符号表、重定位信息等。

程序头部表 (program header table) 告诉系统如何建立一个进程映像。它是从加载执行的角度来看待 ELF 文件的。从它的角度看，ELF 文件被分成许多段，ELF 文件中的代码、连接信息和注释都以段的形式存放。如果一个程序头部表存在，那么它告诉系统如何来创建一个进程的内存映像。被用来建立进程映像 (执行一个程序) 的文件必须要有一个程序头部表；可重定位文件不需要这个头部表。

节头部表（section header table）包含了描述文件 sections 的信息。每个 section 在这个表中有一个入口；每个入口给出了该 section 的名字、大小等信息。在连接过程中的文件必须有一个节头部表；其他 object 文件可要可不要这个节头部表。

ELF 可执行程序的头如图 2-2 所示。

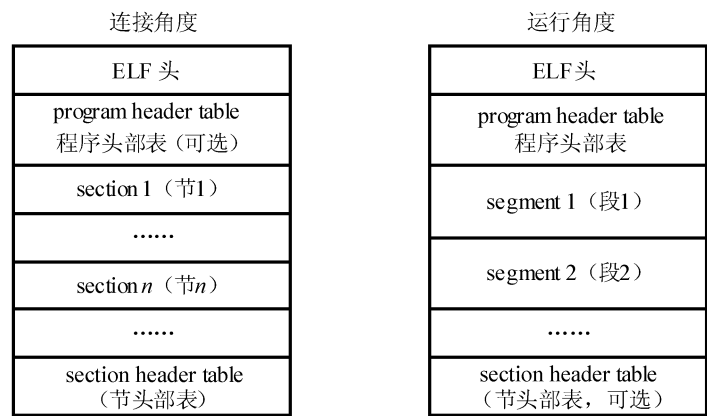


图 2-2 ELF 可执行程序结构

一个 ELF 文件从连接器（linker）的角度看，是一些节（sections）的集合；从程序装载器（loader）的角度看，它是一些段（segments）的集合。ELF 格式的程序和共享库具有相同的结构，只是段的集合和节的集合上有些不同。

ELF 格式文件中的各个节区如下所示。

- **.text:** 包含程序的可执行指令。
- **.rodata 和 .rodata1:** 只读数据，这些数据通常参与进程映像的不可写段。
- **.data 和 .data1:** 初始化了的数据（可写），将出现在程序的内存映像中。
- **.bss:** 包含将出现在程序的内存映像中的未初始化数据。根据定义，当程序开始执行时，系统将把这些数据初始化为 0。此节区不占用文件空间。
- **.init:** 可执行指令，是进程初始化代码的一部分。当程序开始执行时，系统要在开始调用主程序入口之前执行这些代码。
- **.fini:** 此节区包含了可执行的指令，是进程终止代码的一部分。程序正常退出时执行这里的代码。
- **.relaname:** 重定位的信息。如果文件中包含可加载的段，段中有重定位内容，则需要执行过程中占用内存。
- **.got 和 .plt:** 全局偏移表和过程连接表。
- **.dynamic:** 动态连接信息。
- **.comment 和 .debug:** 版本控制信息和调试的信息。
- **.dynstr:** 动态连接的字符串，大多数情况下这些字符串代表了与符号表项相关的名称。
- **.hash:** 一个符号哈希表。

- `.interp`: 程序解释器的路径名。
- `.line` 和 `.note`: 分别表示符号调试的行号信息和注释信息。
- `.shstrtab`: 节区名称。
- `.strtab`: 字符串, 通常代表与符号表项相关的名称。
- `.symtab`: 符号表, 如果文件中包含一个可加载的段, 则包含此节。

提示: 凡是以“.”开头的节区是系统保留的, 目标文件中还可以使用自定义的节区, 各个体系结构和编译器也可以自定义节区。

ELF 格式的文件还使用进程环境中名为 `LD_LIBRARY_PATH` 的变量, 表示连接所使用的连接库的各个路径。其中包含若干用“:”分隔的路径名, 以“;”为结尾。

PIC 表示“与位置无关的代码”(position-independent code), 是 ELF 支持的特殊格式的代码。在 gcc 编译的时候, 可以使用 `-fPIC` 指示 GNU 编译系统生成 PIC 代码。PIC 是实现共享库或共享可执行代码的基础。这种代码的特殊性在于它可以加载到内存地址空间的任何地址执行。这也让加载器可以很方便地在进程中动态连接共享库。

GOT 表示“全局偏移量表”(global offset table), 表示代码段中任何指令和数据段中的任何变量之间的距离, 都是一个与代码段和数据段的绝对存储器位置无关的常量。因此, 编译器在数据段开始的地方创建了一个 GOT 表。GOT 包含每个被这个目标模块引用的全局数据目标的表目。编译器还为 GOT 中每个表目生成一个重定位记录。在加载时, 动态连接器会重定位 GOT 中的每个表目, 使得它包含正确的绝对地址。PIC 在代码中实现通过 GOT 间接地引用每个全局变量。

2.2 GCC 工具的使用

目前的 GCC (GNU Compiler Collection) 已经有了二进制成熟的工具。对于程序的开发, 可以直接下载 GCC 二进制工具包。这个工具包是包含 `gcc`、`binutils`、`libc` 等完整系列的开发工具, 使用很方便。对于主机程序, 一般的 Linux 发行版中都包含了 GCC。如果需要更新版本的 GCC, 可以从网上下载。对于嵌入式系统的开发, 需要交叉版本的 GCC。

在 GCC 工具的使用过程中, 交叉编译系统和主机上的编译系统 (x86 的 `gcc`) 基本没有区别。本部分以一个小工程的交叉编译过程为例, 介绍 GCC 工具的使用。

2.2.1 示例工程

以下工程中包括两个头文件、3 个 C 语言源程序文件, 如下所示。

- `main.c` (源文件): 主程序入口。
- `hello.h` (头文件) 和 `hello.c` (源文件): 打印信息的功能模块。
- `init.h` (头文件) 和 `init.c` (源文件): 初始化执行并访问数据的功能模块。

`main.c` 的内容如下所示:

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

```
# include "hello.h"
# include "init.h"
void aftermain(void)
{
    printf("\n");
    printf("<<<< aftermain >>>>\n");
    printf(".....\n");
    return;
}
int main (int argc, char* argv[])
{
    printf("==== main ==== \n");
    init(1234);
    hello(argc, argv);
    atexit(aftermain);
    printf("..... exit main ..... \n");
    return 0;
}
```

main.c 程序中，调用了两个外部的函数 init()和 hello()，并且使用库函数 atexit()注册 aftermain 为在程序退出时调用的函数。

hello.h 的内容如下所示：

```
# ifndef __HELLO_H__
# define __HELLO_H__
int hello(int argc, char* argv[]);
# endif
```

hello.h 是 hello.c 的头文件，使用宏 __HELLO_H__ 确保不被重复包含，声明了 int hello(int argc, char* argv[]) 函数。

hello.c 的内容如下所示：

```
# include <stdio.h>
# include "hello.h"
int hello(int argc, char* argv[])
{
    int i;
    printf ("Hello world!\n");
    for(i=0; i<argc; i++)
    {
        printf("argv[%d]=%s\n", i, argv[i]);
    }
    return 0;
}
```

hello.c 中实现了 hello() 函数，它的参数类型和 main() 函数是一样的，在函数的执行中，打印传入的参数。

init.h 头文件的内容如下所示：

```
# ifndef __INIT_H__
# define __INIT_H__
int init(int number);
# endif
```

hello.h 是 hello.c 的头文件，使用宏 __INIT_H__ 确保不被重复包含，头文件中声明了 init()

函数，需要在源文件中实现。

init.c 源文件的内容如下所示：

```
# include <stdio.h>
# include "init.h"
const char ro_data[1024]="This is readonly data";
static char rw_data[1024]="This is readwrite data";
static char bss_data[1024];
int init(int number){
    printf("input number:%d\n",number);
    printf("ro_data:%x,%s\n", (unsigned int)ro_data,ro_data);
    printf("rw_data:%x,%s\n", (unsigned int)rw_data,rw_data);
    printf("bss_data:%x,%s\n", (unsigned int)bss_data,bss_data);
    return number;
}
```

在 init.c 中，定义 ro_data 为全局的只读数据段，rw_data 为局部的读写数据段，bss_data 为局部的未初始化数据段。定义函数 init()，打印这几个数据段的地址和内容。

本程序执行的功能很简单，仅仅是显示输入的参数，以及几个数据段的内容。将以上的文件编译成本机（x86）可执行程序 test，执行的过程如下所示：

```
$ ./test abc 123
===== main =====
input number:1234
ro_data:8048640,This is readonly data
rw_data:8049020,This is readwrite data
bss_data:8049540,
Hello world!
argv[0]=./test
argv[1]=abc
argv[2]=123
..... exit main .....

<<<<< aftermain >>>>>
.....
```

在本程序中，打印了几个数据段的地址（只读数据段、读写数据段、BSS 数据段），这在不同的系统上可能是不同的。函数调用和显示输入参数等功能都是相同的。程序中注册了 aftermain 作为退出时执行的函数，在 main() 函数之后，程序退出之前被调用。

2.2.2 编译、汇编和连接

在 GCC 的编译和连接中，对于一个只有一个源文件的程序，可以使用 gcc 工具一步完成生成可执行程序的过程：

```
$ <prefix>-gcc main.c
```

这时将直接生成可执行程序为 a.out，放置在当前的目录下。以上的命令等价于：

```
$ <prefix>-gcc main.c -o main.o
```

这个过程将一步完成编译、汇编和连接。使用 -o 可以指定输出的文件名称。如果不指定，将使用默认的文件名 a.out。

使用 GCC 直接从 C 语言源文件生成可执行程序的过程中，结果不会产生目标文件。

以前面的工程为例，对于多个文件的可执行程序的生成过程，也可以在一歩完成，使用的命令如下所示：

```
$ <prefix>-gcc main.c hello.c init.c -o test
```

其中，**test** 为指定的可执行程序的名称，整个过程的结果将只生成 **test** 一个文件。

在一般 GCC 程序的生成过程中，从 C 语言到最终的可执行程序，会使用两个过程：第一个过程为将各个 C 语言的源文件生成目标文件 (*.o)，第二个过程是将各个目标文件进行连接生成可执行程序。

生成目标文件的过程如下所示：

```
$ <prefix>-gcc -pipe -g -Wall -I. -c -o hello.o hello.c
$ <prefix>-gcc -pipe -g -Wall -I. -c -o init.o init.c
$ <prefix>-gcc -pipe -g -Wall -I. -c -o main.o main.c
```

-pipe 表示使用管道替换临时文件；**-I** 表示包含当前目录作为搜索路径，使用这个选项可以增加搜索头文件的路径，其适用于工程较大，文件分布在各个目录的情况；**-g** 表示包含调试信息；**-Wall** 表示输出所有的警告；**-o** 指定目标文件的名称，如果不指定，将使用 C 语言源文件的文件名+.o 作为目标文件名，本例中指定的名称与默认名称相同。

连接的过程需要连接 3 个目标文件，命令如下所示：

```
$ <prefix>-gcc -Wall -g hello.o init.o main.o -o test
```

在使用 **gcc** 连接的过程中，默认将生成可执行程序。这时需要在目标文件中，必须包含 **main** 符号作为程序的入口，各个引用的符号都可以找到，且没有重复。否则，会发生符号未定义（**undefined reference to XXX**）或者符号重定义（**multiple definition XXX**）的连接错误。

在连接的过程中，可以使用以下命令将文件连接成静态可执行程序：

```
$ <prefix>-gcc -Wall -g hello.o init.o main.o -static -o test_static
```

静态连接通过增加 **-static** 参数实现，这时产生的文件 **test_static** 将 C 语言库包含在其中，它的大小将比动态连接 **test** 大很多，**test** 只有 10 多 KB，而 **test_static** 将有几百 KB 乃至 1MB 以上，具体的大小还和 C 语言库的类型有关。静态的程序已经包含了所用到的 C 语言库，运行时不需要 C 语言库的支持。

对 C++ 文件的编译成目标文件的命令则如下所示：

```
$ <prefix>-g++ -pipe -g -Wall -I. -c -o hello.o hello.c
$ <prefix>-g++ -pipe -g -Wall -I. -c -o init.o init.c
$ <prefix>-g++ -pipe -g -Wall -I. -c -o main.o main.c
```

生成静态连接的可执行程序：

```
$ <prefix>-g++ g++ -Wall -g hello.o init.o main.o -o test
```

对于 C++ 的对象，C 的连接器不能找到符号：

```
$ gcc -Wall -g hello.o init.o main.o -o test
```

这是由于 C++ 程序中符号的写法和 C 程序不同。例如对于 `init()` 函数，如果用 C 语言的编译器进行编译，生成的符号就是 `init`；而如果用 C++ 的编译器进行编译，生成的符号就是类似 `_Z4init` 的形式。

2.2.3 预处理和汇编

上面说明了一般情况下，使用 GCC 进行编译和连接的步骤。事实上，前面所说的“编译”过程是由几个步骤组成的。

- 第一步：预处理（pre-process），将从 C 语言文件生成预处理后的 C 语言文件。
- 第二步：编译（compile），将从预处理的 C 语言文件生成汇编程序。
- 第三步：汇编（assemble），将从汇编程序生成二进制的目标代码。

在使用 GCC 进行程序生成的时候，也可将这几个步骤分开使用。

首先，生成汇编程序的命令如下所示：

```
$ <prefix>-gcc -pipe -g -Wall -I. -S -o hello.s hello.c
```

事实上，只需要在 `gcc` 中指定 `-S` 参数，即可以从 C 语言文件生成汇编程序。如果不使用 `-o` 指定输出文件，默认的汇编语言文件名将是 C 语言源程序的文件名+.s。

以上命令生成的汇编文件 `hello.s` 本质上依然是一个文本文件，与体系结构相关。

可以进一步将汇编语言文件生成目标文件，命令和编译 C 语言源文件一样。

```
$ <prefix>-gcc -pipe -g -Wall -I. -c -o hello.o hello.s
```

预处理的过程一般包括了去注释、头文件展开、宏替换等过程，使用 `gcc` 对 C 语言文件进行预处理的命令为：

```
$ <prefix>-gcc -E hello.c
```

`-E` 指定了对 C 语言文件进行预处理，还可以使用 `-o` 指定输出文件。在程序的预处理阶段，将进行如下的工作：注释会被剔除，`define` 的部分将会被预处理器替换，头文件信息会被替换成实体头文件信息等。

预处理阶段输出的文件本质上还是 C 语言的源文件，只是当中的头文件以及宏被展开了。这种文件还可以使用 `-c` 和 `-s` 指定编译或者汇编。

2.2.4 归档工具（ar）和静态库

在 GCC 中，可以使用 `ar` 工具（归档工具）生成静态库（static lib）。静态库的生成需要目标文件，静态库可以被应用程序连接。

按照以上的程序，可以将 `init.o` 和 `hello.o` 归档为一个静态库，命令格式如下所示。

```
$ <prefix>-ar -rv libtest_s.a init.o hello.o
```

生成的静态库的名称为 `libtest_s.a`。选项 `r` 表示在库中插入模块（替换），当插入的模块名已经在库中存在，则替换同名的模块；选项 `v` 用来显示执行操作选项的附加信息。

在命令执行的过程中，显示的内容为：

```
a - init.o
a - hello.o
```

`ar` 工具与 `gcc` 的编译-连接工具稍有区别，其参数中静态库 `libtest_s.a` 是输出，同时也是输入，可以表示对一个已经存在的动态库进行进一步的操作。

`ar` 工具可使用“增量”的方式进行归档，正如选项 `-r` 指定的，当模块存在的时候，使用新的替换。

以上生成静态库的命令与下面的命令等价：

```
$ <prefix>-ar -rv libtest_s.a init.o
$ <prefix>-ar -rv libtest_s.a hello.o
```

由于静态库 `libtest_s.a` 已经包含了 `init.o` 和 `hello.o` 两个目标文件，如果再继续执行以上命令，`libtest_s.a` 静态库本身也不会产生变化。由此可见，`ar` 命令中指定的静态库既是命令的输出，也是命令的输入。

对于替换库的情况，`ar` 命令显示的内容为：

```
r - init.o
r - hello.o
```

可以使用如下命令，查看静态库中的内容：

```
$ <prefix>-ar -t libtest_s.a
init.o
hello.o
```

在静态库中的目标文件也可以被删除，命令如下所示：

```
$ <prefix>-ar -d libtest_s.a init.o
```

删除文件后，再次查看归档文件，如下所示：

```
$ <prefix>-ar -t libtest_s.a
hello.o
```

由此可见，目标文件 `init.o` 已经被移除了。

在生成可执行程序的时候，可以直接连接静态库，而不需要连接目标文件。典型的过程如下所示：

```
$ <prefix>-gcc -Wall main.o -L. -ltest_s -o testbylib
```

在以上的命令中，`-L.` 表示包含当前路径为搜索路径，`-ltest` 的含义为连接名称为 `libtest_s.a` 的静态库或者名称为 `libtest_s.so` 的动态库。

该命令将生成 `testbylib`，一个可执行程序，它和使用目标文件连接而成的基本没有区别。`testbylib` 已经包含了静态库 `libtest_s.a`，它在运行时不需要依赖 `libtest_s.a`。

静态库不是 ELF 格式的文件，在使用静态库的时候，可以只使用静态库生成可执行程序或者动态库，但是静态库有连接顺序的问题。

```
$ <prefix>-ar -rv libtools.a init.o hello.o
$ <prefix>-ar -rv libmain.a main.o
```

一个错误的连接顺序如下所示：

```
$ <prefix>-gcc -L. -ltools -lmain -o test_wrong
```

正确的连接顺序如下所示：

```
$ <prefix>-gcc -L. -lmain -ltools -o test_right
```

从道理上来讲，静态库中也可以包括 `main` 入口，因此连接 `libmain.a` 和 `libtools.a` 可以生成一个可执行程序。由于 `libmain.a` 依赖于 `libtools.a`，因此 `libtools.a` 要写在连接顺序的前面。

2.2.5 动态库

可使用 `gcc` 工具生成动态库（dynamic lib）。与静态库相比，动态库不需要被连接到可执行程序中，不会增加可执行程序的大小，但是运行的时候需要动态库存在。

动态库的生成命令为：

```
$ <prefix>-gcc -shared -Wall -g hello.o init.o -o libtest_d.so
```

该命令主要使用 `-shared` 参数，指定生成动态库，动态库的后缀为 `so`。本命令将 `hello.o` `init.o` 连接成动态库 `libtest_d.so`。

使用以下命令生成应用程序，并连接动态库：

```
$ <prefix>-gcc -Wall -g main.o -L. -ltest_d -o test_exe
```

`test_exe` 是一个可执行程序，它本身不包括动态库 `libtest_d.so`，在运行的时候需要 `libtest_d.so` 的存在。

`test_exe` 生成后，如果直接运行，可能产生如下的错误：

```
error while loading shared libraries: libtest_d.so
```

以上错误表示 `test_exe` 所需要的动态库没有被找到，`test_exe` 在没有动态库的情况下是不能运行的。

如果需要在目标机系统的 Linux 运行，可以通过修改环境变量，增加当前路径为动态库的搜索地址，如下所示：

```
$ export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

此时，可执行程序 `test_exe` 就可以运行了。

动态库的另外一种使用方法是不直接由应用程序连接，而在程序中使用 `dlopen()` 等函数动态使用。

2.2.6 ELF 格式文件信息读取（readelf）

`readelf` 工具主要用于读取 ELF 文件的信息，其使用的方式为：

```
readelf [options] file
```

例如，使用表示读取 ELF 信息的 `-h` 参数，来读取目标文件 `hello.o` 的 ELF 头的命令如

下所示:

```
$ <prefix>-readelf -h hello.o
```

显示的内容如下所示:

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                ARM
  ABI Version:                           0
  Type:                                   REL (Relocatable file)
  Machine:                                ARM
  Version:                                0x1
  Entry point address:                    0x0
  Start of program headers:               0 (bytes into file)
  Start of section headers:              1044 (bytes into file)
  Flags:                                  0x0
  Size of this header:                     52 (bytes)
  Size of program headers:                 0 (bytes)
  Number of program headers:               0
  Size of section headers:                 40 (bytes)
  Number of section headers:               22
  Section header string table index:      19
```

其中比较重要的信息包括, 文件的类型 **Type** 为 **REL (Relocatable file)**, 表示这是一个重定向文件; **Machine** 的类型为 **ARM**, 表示这是一个 ARM 体系结构下的目标文件。

readelf 获取的头信息与 ELF 格式吻合, 主要包括以下内容。

- **Entry point address:** 入口地址。
- **Start of program headers:** 程序头的起始字节。
- **Start of section headers:** 节头的起始字节。
- **Flags:** 标志。
- **Size of this header:** ELF 头的长度。
- **Size of program headers:** 程序头的大小。
- **Number of program headers:** 程序头数目。
- **Size of section headers:** 节头的大小。
- **Number of section headers:** 节头的数目。

对于共享库文件, 显示的内容如下所示:

```
$ readelf -h libtest_d.so
```

命令执行后, 显示的内容如下所示:

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                ARM
  ABI Version:                           0
```

```
Type: DYN (Shared object file)
Machine: ARM
Version: 0x1
Entry point address: 0x518
Start of program headers: 52 (bytes into file)
Start of section headers: 7124 (bytes into file)
Flags: 0x2, has entry point, GNU EABI
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 3
Size of section headers: 40 (bytes)
Number of section headers: 32
Section header string table index: 29
```

与目标文件的主要区别为，此处文件的类型 Type 为 DYN (Shared object file)，表示共享库文件。

对于可执行程序 test，执行如下的命令：

```
$ <prefix>-readelf -h test
```

命令执行后，显示内容如下所示：

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: ARM
ABI Version: 0
Type: EXEC (Executable file)
Machine: ARM
Version: 0x1
Entry point address: 0x83cc
Start of program headers: 52 (bytes into file)
Start of section headers: 10052 (bytes into file)
Flags: 0x2, has entry point, GNU EABI
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 6
Size of section headers: 40 (bytes)
Number of section headers: 34
Section header string table index: 31
```

其文件的类型 Type 为 EXEC (Executable file)，表示可执行文件。

在 ELF 格式中有 3 种文件类型：可重定向文件、共享对象文件、可执行文件。这 3 种文件格式，即分别对应了以上的 hello.o（目标文件）、libtest_d.so（共享库）、test 文件（可执行程序）。它们都是 ELF 格式的文件，但是由于用途不同，内部结构存在差别。

此外，另外一种方法是使用 Linux 下的 file 工具查看 ELF 文件的格式。可以同时查看多个文件：

```
$ file hello.o libtest_d.so test
hello.o: ELF 32-bit LSB relocatable, ARM, version 1 (ARM), not stripped
libtest_d.so: ELF 32-bit LSB shared object, ARM, version 1 (ARM), not stripped
test: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,
dynamically linked (uses shared libs), for GNU/Linux 2.0.0, not stripped
```

使用 `file` 程序可以获得文件的类型、运行结构、动态连接或者静态连接，以及是否被剥离符号的信息。尤其在开发程序的过程中，使用它查看二进制代码类型（Intel 80386 或者 ARM）的信息是很方便的。

如果使用 x86 的编译工具编译以上文件，显示的内容为：

```
hello.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
libtest_d.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped
test:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.6.4, dynamically linked (uses shared libs), for GNU/Linux 2.6.4, not stripped
```

对于上述静态连接的 ARM 版本的可执行程序 `test_static`，其信息为：

```
$ file test_stripped
test_static: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,
statically linked, for GNU/Linux 2.0.0, not stripped
```

与目标文件和动态库的主要差别是，其显示为静态连接 `statically linked`。

2.2.7 符号信息工具（nm）

`nm` 工具用于显示文件中的符号，可以用于各种 ELF 格式的文件。使用 `nm` 查看以上的目标文件（可重定向文件），将显示符号及其相关内容。

`nm` 工具用于显示文件中的符号，可以用于各种 ELF 格式的文件。

`nm` 列出的符号类型主要有以下几种：

- T 表示在代码段中定义的一般变量符号。
- R 表示只读数据段。
- D 表示已初始化的数据段。
- B 表示未初始化的数据段。
- U 表示在这个文件中使用但是没有定义的符号。
- W, `weak` 的缩写，表示如果其他函数库中也有对这个符号的定义，则其他符号的定义可以覆盖这个定义。

在解析的各个符号中，大写字母表示这个符号是全局（`global`）符号，小写字母表示这个符号是局部（`local`）符号。

使用 `nm` 工具查看 3 个目标文件的符号使用如下命令：

```
$ <prefix>-nm main.o hello.o init.o
```

显示的结果如下所示：

```
main.o:
00000000 T aftermain
        U atexit
        U hello
        U init
00000034 T main
        U printf
hello.o:
00000000 T hello
        U printf
```

```
init.o:
00000000 b bss_data
00000000 T init
        U printf
00000000 R ro_data
00000000 d rw_data
```

在以上显示的内容中，主要表示含义如下所示：

- 在 main.o 的符号中，main 和 aftermain 的类型为 T，表示它们是代码，而且是全局的；atexit、hello、init 和 printf 为 U，表示它们是未定义的符号。
- 在 hello.o 的符号中，hello 是代码，printf 是未定义的符号。
- 在 init.o 的符号中，init 为代码；printf 是未定义的符号；bss_data 的类型为 b；是局部的 BSS 数据；ro_data 的类型为 R，是全局的只读数据；rw_data 的类型为 d，是局部的可读写数据。

C++的符号与 C 语言不同，如果使用 g++ 的编译器进行编译，然后使用 nm 查看符号，通常具有如下情况：

```
$ g++ -c main.c hello.c init.c
$ nm main.o hello.o init.o
main.o:
        U _Z4initi
        U _Z5helloiPPc
00000000 T _Z9aftermainv
        U __gxx_personality_v0
        U atexit
0000002c T main
        U putchar
        U puts
hello.o:
00000000 T _Z5helloiPPc
        U __gxx_personality_v0
        U printf
        U puts
init.o:
00000000 T _Z4initi
00000040 r _ZL7ro_data
00000000 d _ZL7rw_data
00000000 b _ZL8bss_data
        U __gxx_personality_v0
        U printf
```

如上面所示，C++程序生成的符号具有前缀，并非原始函数和变量的名字。

2.2.8 字符串工具（strings）

strings 是一个简单的工具，用于查看文件中的字符串。

strings 工具的使用方式如下所示：

```
$ strings [option(s)] [file(s)]
```

strings 工具主要参数的含义如下所示。

- -a / --all: 扫描整个文件，而不是数据段。

- **-f/--print-file-name**: 在字符串前面打印文件的名字。
- **-n --bytes=[number]**: 定位和打印任何以 NUL 为结束的序列, 至少 **number** 个字符, 默认为 4 个。
- **-t --radix={o,x,d}**: 基于八、十、十六进制打印字符串。
- **-o**: 相当于 **--radix=o**。
- **-T/--target=<BFDNAME>**: 指定二进制文件的格式。
- **-e --encoding={s,S,b,l,B,L}**: 选择字符大小, **s** = 7-bit, **S** = 8-bit, **{b,l}** = 16-bit, **{B,L}** = 32-bit。
- **-h/--help**: 显示 **strings** 的所有选项, 然后退出。
- **-v/--version**: 显示 **strings** 的版本号, 然后退出。

使用 **strings** 查看 **main.o** 文件中的字符串:

```
$ <prefix>-strings main.o
ReadWrite Data
ReadOnly Data
===== main begin =====
write sum:%d
BSS Data
ro_data:%x,%s
rw_data:%x,%s
bss_data:%x,%s
===== main end =====
```

从中可见, 目标文件 **main.o** 中的字符串都被显示了出来。

strings 工具附加 **-n** 参数可以在全目标文件搜索字符串。其基本的规则是寻找以 NUL 为结束的字符串。然而这样有可能将文件中不是字符串的部分被识别为字符串, 因此在 **strings** 中, 还可以指定字符串的最短长度, 这时只有字符串大于这个长度的时候, 它才会被识别为字符串而显示出来。

2.2.9 去除符号 (strip)

strip 工具用于去除文件中的符号信息。使用 **strip** 去除符号信息后, 程序的大小可以显著减小, 尤其适合在嵌入式系统中使用, 可以节省存储空间。

使用 **strip** 去除可执行程序 **test** 中的符号信息:

```
$ <prefix>-strip test -o test_stripped
```

去除符号信息后, 使用 **nm** 查看文件的符号:

```
$ <prefix>-nm test_stripped
nm: test_stripped: no symbols
```

由此可见, **strip** 之后的文件将不包含任何符号信息。

使用 **file** 等工具查看文件:

```
$ file test_stripped
test_stripped: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,
dynamically linked (uses shared libs), for GNU/Linux 2.0.0, stripped
```

显示文件的最后一项不再是 `not stripped`，而是 `stripped`。

`strip` 程序也可以使用命令行选项，有选择地删除符号。在使用 `strip` 程序的时候，如果不使用 `-o` 指定输出，将改变输入的文件。

2.2.10 目标文件复制（objcopy）

`objcopy` 用于复制一个目标文件的内容到另一个目标文件中。实际上，`objcopy` 经常作为格式转换工具使用，即将目标文件转换成另一种格式的目标文件。`objcopy` 还可以用于去除文件中的信息。它在进行目标文件的转换时，将生成一个临时文件，转换完成后就将这个临时文件删掉。

使用如下命令将可执行程序 `test` 转换为纯二进制文件：

```
$ <prefix>-objcopy test -O binary test.bin
```

`test` 是一个 ELF 格式的可执行程序，对其进行 `objcopy` 操作，`-O` 指定输出为 `binary`。这时生成的目标文件为 `test.bin`，仅仅包括二进制的內容，即目标机的机器代码和数据。

查看 `test.bin` 的文件格式：

```
$ file test.bin
test.bin: data
```

在嵌入式系统中，有很多程序是在没有操作系统的情况下运行的，如 `BootLoader`，这些程序映像文件的格式应为二进制。

生成 `S-Record` 格式的文件：

```
$ <prefix>-objcopy test -O srec test.srec
```

查看文件信息：

```
$ file test.srec
test.srec: Motorola S-Record; binary data in text format
```

`objcopy` 可以用于部分更改输出文件的信息，需要使用 `-S` 参数，例如：

```
$ <prefix>-objcopy test -S test.stripped
```

此命令去除了 `test` 中的符号信息，功能与 `strip` 类似。

查看输出文件的信息：

```
$ <prefix>-nm test.stripped
nm: test.stripped: no symbols
```

`objcopy` 可以使用 `-R` 指定去除的段，命令如下所示：

```
$ <prefix>-objcopy test -R .data test_removed
```

此命令将去除 `test` 中的读写数据段（`.data`），生成另外一个目标 `test_removed`。

2.2.11 目标文件信息（objdump）

目标文件信息工具 `objdump` 可以显示文件的信息。对于各种目标文件、库文件、可执

行程序均可以使用。对二进制的文件进行解析后，可以获取头信息和对机器代码反汇编。

objdump 常用的参数为 **-R** 和 **-D**。 **-D** 表示显示文件中所有的汇编信息； **-R** 表示显示文件的动态重定位入口，仅仅对于动态目标文件有意义，比如特定类型的共享库。

使用 **objdump**，显示目标文件的头信息，如下所示：

```
$ <prefix>-objdump -h hello.o
hello.o:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          00000080  00000000  00000000  00000034  2**2
    CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data          00000000  00000000  00000000  000000b4  2**0
    CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000000  00000000  00000000  000000b4  2**0
    ALLOC
  3 .debug_abbrev  0000006a  00000000  00000000  000000b4  2**0
    CONTENTS, READONLY, DEBUGGING
  4 .debug_info    0000015d  00000000  00000000  0000011e  2**0
    CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line    00000038  00000000  00000000  0000027b  2**0
    CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata        00000020  00000000  00000000  000002b4  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .debug_frame   00000030  00000000  00000000  000002d4  2**2
    CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_pubnames 0000001c  00000000  00000000  00000304  2**0
    CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_aranges 00000020  00000000  00000000  00000320  2**0
    CONTENTS, RELOC, READONLY, DEBUGGING
10 .debug_str      0000000d  00000000  00000000  00000340  2**0
    CONTENTS, READONLY, DEBUGGING
11 .comment       00000012  00000000  00000000  0000034d  2**0
    CONTENTS, READONLY
```

显示结果中包含了 ELF 格式文件中各个节 (Section) 的信息。例如，**.text** 为文本段，**.data** 为已初始化数据段，**.bss** 为未初始化数据段。

对于可执行文件的头信息，使用 **objdump** 显示如下所示：

```
test:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .interp        00000013  000080f4  000080f4  000000f4  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag  00000020  00008108  00008108  00000108  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA, LINK_ONCE_DISCARD
  2 .hash          00000040  00008128  00008128  00000128  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym         000000b0  00008168  00008168  00000168  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr         000000b8  00008218  00008218  00000218  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version    00000016  000082d0  000082d0  000002d0  2**1
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r  00000030  000082e8  000082e8  000002e8  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn        00000020  00008318  00008318  00000318  2**2
```

```

      CONTENTS, ALLOC, LOAD, READONLY, DATA
8 .rel.plt      00000028 00008338 00008338 00000338 2**2
      CONTENTS, ALLOC, LOAD, READONLY, DATA
9 .init         0000001c 00008360 00008360 00000360 2**2
      CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt         00000050 0000837c 0000837c 0000037c 2**2
      CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text        0000038c 000083cc 000083cc 000003cc 2**2
      CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini        00000018 00008758 00008758 00000758 2**2
      CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .rodata      000004c8 00008770 00008770 00000770 2**2
      CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .eh_frame    00000004 00008c38 00008c38 00000c38 2**2
      CONTENTS, ALLOC, LOAD, READONLY, DATA
15 .ctors       00000008 00011000 00011000 00001000 2**2
      CONTENTS, ALLOC, LOAD, DATA
16 .dtors       00000008 00011008 00011008 00001008 2**2
      CONTENTS, ALLOC, LOAD, DATA
17 .jcr         00000004 00011010 00011010 00001010 2**2
      CONTENTS, ALLOC, LOAD, DATA
18 .dynamic     000000c8 00011014 00011014 00001014 2**2
      CONTENTS, ALLOC, LOAD, DATA
19 .got         00000034 000110dc 000110dc 000010dc 2**2
      CONTENTS, ALLOC, LOAD, DATA
20 .data        0000040c 00011110 00011110 00001110 2**2
      CONTENTS, ALLOC, LOAD, DATA
21 .bss         00000404 0001151c 0001151c 0000151c 2**0
      ALLOC
22 .comment     00000090 00000000 00000000 0000151c 2**0
      CONTENTS, READONLY
23 .debug_aranges 000000c0 00000000 00000000 000015b0 2**3
      CONTENTS, READONLY, DEBUGGING
24 .debug_pubnames 000000e8 00000000 00000000 00001670 2**0
      CONTENTS, READONLY, DEBUGGING
25 .debug_info   0000073d 00000000 00000000 00001758 2**0
      CONTENTS, READONLY, DEBUGGING
26 .debug_abbrev 000002de 00000000 00000000 00001e95 2**0
      CONTENTS, READONLY, DEBUGGING
27 .debug_line   000001d3 00000000 00000000 00002173 2**0
      CONTENTS, READONLY, DEBUGGING
28 .debug_frame  00000124 00000000 00000000 00002348 2**2
      CONTENTS, READONLY, DEBUGGING
29 .debug_str     000001ac 00000000 00000000 0000246c 2**0
      CONTENTS, READONLY, DEBUGGING

```

结果包含了可执行程序各个节（Section）的信息，相比目标文件多了一些特定的节。利用 `objdump` 工具可以将二进制的文件反汇编，命令如下所示：

```
$ <prefix>-objdump -D hello.o
```

显示的内容片断如下所示：

```

hello.o:      file format elf32-littlearm
Disassembly of section .text:
00000000 <hello>:
0:      e1a0c00d mov ip, sp
4:      e92dd800 stmdb    sp!, {fp, ip, lr, pc}
8:      e24cb004 sub fp, ip, $ 4 ; 0x4

```

```

c:      e24dd00c sub    sp, sp, $ 12 ; 0xc
10:     e50b0010 str    r0, [fp, $ -16]
14:     e50b1014 str    r1, [fp, $ -20]
18:     e59f0058 ldr    r0, [pc, $ 88]      ; 78 <.text+0x78>
1c:     ebfffffe bl     0 <printf>
20:     e3a03000 mov    r3, $ 0 ; 0x0
24:     e50b3018 str    r3, [fp, $ -24]
28:     e51b2018 ldr    r2, [fp, $ -24]
2c:     e51b3010 ldr    r3, [fp, $ -16]
30:     e1520003 cmp    r2, r3
34:     aa000018 bge    68 <hello+0x68>
38:     e51b3018 ldr    r3, [fp, $ -24]
3c:     e1a02103 mov    r2, r3, lsl $ 2
40:     e51b3014 ldr    r3, [fp, $ -20]
44:     e0823003 add    r3, r2, r3
48:     e59f002c ldr    r0, [pc, $ 44]      ; 7c <.text+0x7c>
4c:     e51b1018 ldr    r1, [fp, $ -24]
50:     e5932000 ldr    r2, [r3]
54:     ebfffffe bl     0 <printf>
58:     e51b3018 ldr    r3, [fp, $ -24]
5c:     e2833001 add    r3, r3, $ 1 ; 0x1
60:     e50b3018 str    r3, [fp, $ -24]
64:     ea000008 b      28 <hello+0x28>
68:     e3a03001 mov    r3, $ 1 ; 0x1
6c:     e1a00003 mov    r0, r3
70:     e24bd00c sub    sp, fp, $ 12 ; 0xc
74:     e89da800 ldmia   sp, {fp, sp, pc}
78:     00000000 andeq   r0, r0, r0
7c:     00000010 andeq   r0, r0, r0, lsl r0
Disassembly of section .debug_abbrev:

```

以上内容为 `hello.o` 文件的反汇编信息，其中的内容为 ARM 的反汇编。以上的 `<hello>` 节，实际上就是示例程序中的函数：`hello(int argc, char* argv[])`。

对于 `objdump` 命令，处理的结果将二进制的机器代码反汇编成相应体系结构的汇编语言。由此，不同体系结构编译器的 `objdump` 不能通用。从中可以看出 `<printf>` 为对 C 语言库函数 `printf()` 的调用，函数入口处 `mov ip, sp` 等语句实现了对函数压栈。

如果使用 x86 生成目标文件 `hello.o`，然后再使用 `objdump` 查看反汇编信息，命令和反汇编得到的内容如下所示：

```

$ gcc -pipe -g -Wall -I. -c -o hello.o hello.c
$ objdump -D hello.o
hello.o:      file format elf32-i386
Disassembly of section .text:
00000000 <hello>:
0:      55                      push    %ebp
1:      89 e5                   mov     %esp, %ebp
3:      83 ec 28                sub     $0x28, %esp
6:      c7 04 24 00 00 00 00  movl    $0x0, (%esp)
d:      e8 fc ff ff ff         call    e <hello+0xe>
12:     c7 45 fc 00 00 00 00  movl    $0x0, 0xfffffff0(%ebp)
19:     eb 26                  jmp     41 <hello+0x41>
1b:     8b 45 fc                mov     0xfffffff0(%ebp), %eax
1e:     c1 e0 02                shl     $0x2, %eax
21:     03 45 0c                add     0xc(%ebp), %eax
24:     8b 00                  mov     (%eax), %eax
26:     89 44 24 08             mov     %eax, 0x8(%esp)

```

```

2a: 8b 45 fc      mov    0xffffffff(%ebp),%eax
2d: 89 44 24 04    mov    %eax,0x4(%esp)
31: c7 04 24 0d 00 00 00 movl   $0xd, (%esp)
38: e8 fc ff ff ff call   39 <hello+0x39>
3d: 83 45 fc 01    addl   $0x1,0xffffffff(%ebp)
41: 8b 45 fc      mov    0xffffffff(%ebp),%eax
44: 3b 45 08      cmp    0x8(%ebp),%eax
47: 7c d2        jl     1b <hello+0x1b>
49: b8 01 00 00 00 mov    $0x1,%eax
4e: c9          leave
4f: c3          ret

```

Disassembly of section .debug_abbrev:

由此可见，在 x86 和 ARM 生成的目标文件经过反汇编后，得到完全不同的汇编代码。在 x86 的汇编中，call 为函数调用，push 为进入函数的压栈，ret 为函数返回。

第 3 章

工程管理和 make 机制

make 和 Makefile 是 Linux 系统下开发常用的机制。在程序开发的过程中，Makefile 带来的最大好处即自动化编译。根据 Makefile 定义编译规则，在编译的时候只需要一个 make 命令，整个工程根据是否需要更新完成自动编译，极大地提高了软件开发的效率。

使用 GNU autoconf 及 automake 这两套工具可以协助自动产生 Makefile 文件，构建自动化的 Make 过程。

3.1 make 工具

make 工具是实现 Makefile 机制的基础，它是一个 Linux 下的二进制程序。make 工具自动确定工程的哪部分需要重新编译，执行命令去编译它们。虽然 make 多用于 C 程序，然而只要提供命令行的编译器，也可以用于其他编程语言。事实上，make 工具的应用范围不仅限于编程，也可以用于描述一些文件改变时，需要自动更新另一些文件的任务来使用。

make 机制在开发过程中起到辅助的作用，基本的流程如图 3-1 所示。

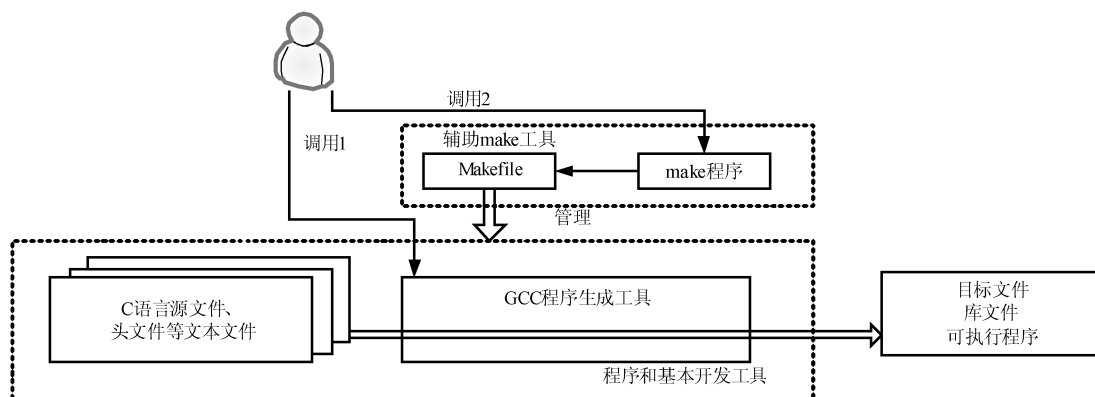


图 3-1 使用 make 的程序生成过程

在 Linux 的命令行（Shell 终端）键入 make 的时候，将依次寻找名称为 GUNmakefile、

makefile 和 Makefile 等的文件作为编译文件。找到编译文件后，make 工具将根据其中的第一个目标自动寻找依赖关系，找出这个目标所需要的其他目标。如果所需要的目标也需要依赖其他的目标，make 工具将一层层直到找到最后一个目标为止。

make 工具的使用格式为：

```
make [options] [target] ...
```

options 为 make 工具的选项，target 为 Makefile 中指定的目标。

在 make 工具中的选项主要包括以下几项。

- -e: 指明环境变量的值覆盖 Makefile 中定义的变量的值。
- -h: 显示帮助信息。
- -f FILE: 指定需要执行的 Makefile。
- -W FILE: 假设目标 FILE 是最新的。

3.2 Makefile 的基本原则

Makefile 文件作为 make 工具的输入，它基本是一个文本文件。与 C 语言文件不同，Makefile 的语法与格式有关，tab 和空格等格式语句均表示了特别的含义。

3.2.1 Makefile 的变量

在 Makefile 中，变量是一个名字（像是 C 语言中的宏），代表一个文本字符串（变量的值）。在 Makefile 的目标、依赖、命令中引用变量的地方，变量会被它的值所取代（与 C 语言中宏引用的方式相同，因此其他版本的 make 也把变量称为“宏”）。

在 Makefile 中变量有以下几个特征：

1) Makefile 中变量和函数的展开（除规则命令行中的变量和函数以外），是在 make 读取 Makefile 文件时进行的，这里的变量包括了使用“=”定义和使用指示符“define”定义的。

2) 变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表和所有我们能够想到的事物。

3) 变量名是不包括“:”、“#”、“=”、前置空白和尾空白的任何字符串。需要注意的是，尽管在 GNU 的 make 机制没有对变量的命名有其他限制，但定义包含除字母、数字和下画线以外的变量也是不好的，因为除字母、数字和下画线以外的其他字符可能会在 make 的后续版本中被赋予特殊含义，并且这样命名的变量对于一些 shell 来说是不能被作为环境变量来使用的。

4) 变量名是大小写敏感的。变量“foo”、“Foo”和“FOO”指的是三个不同的变量。Makefile 的传统做法是变量名全采用大写的方式。一个推荐的做法是：对于内部定义的一般变量（例如：目标文件列表 objects）使用小写方式，而对于一些参数列表（例如：编译选项 CFLAGS）采用大写方式，但这并不是要求的。但需要强调一点：对于一个工程，所有 Makefile 中的变量命名应保持一种风格，否则会显得你是一个蹩脚的程序员（就像代码

的变量命名风格一样)。

5) 另外, 有一些变量名只包含了一个或者很少的几个特殊字符(符号)。它们被称为自动化变量, 如“\$<”、“\$@”、“\$?”、“\$*”等。

Makefile 中的变量定义分为 4 种: 预定义变量、用户变量、自动变量、环境变量。其中, 自动变量是以\$符号开头的一系列在命令行可以替换的变量, 预定义变量有 CC、AR、AS、CFLAGS 等(也可以被进行替换), 环境变量是使用 `export` 在运行的环境中进行导出定义的变量。

在 Makefile 中, 经常使用的变量如表 3-1 所示。

表 3-1 Makefile 中的常用变量

变量	描述
\$@	目标文件名
\$<	规则中的第一个文件名
^	规则中所有相关文件的名称
\$?	规则中日期比目标新的文件列表, 用空格分开
\$(D)	目标文件的目录部分
\$(F)	目标文件的文件名部分

在 Linux 的 Makefile 目标名称指定的时候, 常常有以下惯例。

- `all`: 表示编译所有的内容。
- `clean`: 表示清除目标。
- `distclean`: 表示清除所有的内容。
- `install`: 表示安装内容。

提示: Makefile 中规则的名称是人为指定的, 并无默认字符串。

在 Makefile 中还可以使用“`include`”表示包含其他文件。`include` 指示符告诉 `make` 暂停读取当前的 Makefile, 而转去读取“`include`”指定的一个或者多个文件, 完成以后再继续当前 Makefile 的读取。Makefile 中指示符“`include`”书写在独立的一行, 其形式如下:

```
include FILENAMES...
```

FILENAMES 是 shell 所支持的文件名, 也可以使用通配符。指示符“`include`”和文件名之间、多个文件之间使用空格或者[Tab]键隔开。行尾的空白字符在处理时被忽略。使用指示符包含文件到当前 Makefile 中(如果存在变量或者函数的引用)。

`make` 的执行过程分为两个阶段。

- 第一阶段: 读取所有的 Makefile 文件, 内建所有的变量、明确规则和隐含规则, 并建立所有目标和依赖之间的依赖关系结构链表。
- 第二阶段: 根据第一阶段已经建立的依赖关系结构链表决定哪些目标需要更新, 并使用对应的规则来重建这些目标。

3.2.2 Makefile 的条件执行

条件语句可以根据一个变量的值来控制 `make` 执行或者忽略 `Makefile` 的特定部分。条件语句可以是两个不同变量，或者变量和常量值的比较。要注意的是：条件语句只能用于控制 `make` 实际执行的 `Makefile` 文件部分，它不能控制规则的 `shell` 命令执行过程。`Makefile` 中使用条件控制可以做到处理的灵活性和高效性。

一个简单的不包含“`else`”分支的条件判断语句的语法格式为：

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
endif
```

表达式中“`TEXT-IF-TRUE`”可以是若干任何文本行，当条件为真时它将被 `make` 作为需要执行的一部分。当条件为假时，不作为需要执行的一部分。

包含“`else`”的复杂一点的语法格式为：

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
else
TEXT-IF-FALSE
endif
```

常用的条件语句中几个条件如下所示：

1) “`ifeq`”表示条件语句的开始，并指定了一个表示相等的比较条件，之后使用圆括号包围的、使用逗号“`,`”分隔的两个参数，与关键字“`ifeq`”之间用空格分开。参数中的变量引用在进行变量值比较时被展开。“`ifeq`”之后的内容就是当条件满足时 `make` 所需要执行的，条件不满足时忽略。

2) “`else`”之后就是当条件不满足时的执行部分。不是所有的条件语句都需要此部分。

3) “`endif`”表示一个条件语句的结束，任何一个条件表达式都必须以“`endif`”结束。

3.2.3 Makefile 中的函数

`GNU make` 的函数提供了处理文件名、变量、文本和命令的方法。使用函数，我们的 `Makefile` 可以书写得更加灵活和健壮。可以在需要的地方调用函数来处理指定的文本（需要处理的文本作为函数的参数），函数在调用它的地方被替换为它的处理结果。函数调用（引用）的展开和变量引用的展开方式相同。

`GNU make` 函数的调用格式类似于变量的引用，以“`$`”开始表示一个引用。语法格式如下所示：

```
$ (FUNCTION ARGUMENTS)
```

对于函数调用的格式有以下几点说明：

1) 调用语法格式中“`FUNCTION`”是需要调用的函数名，它应该是 `make` 内嵌的函数名。对于用户自己的函数，需要通过 `make` 的“`call`”函数来间接调用。

2) “`ARGUMENTS`”是函数的参数，参数和函数名之间使用若干个空格或者`[tab]`字符分隔（建议使用一个空格，这样不仅使得在书写上比较直观，更重要的是当你不能确定是

否可以使用[Tab]的时候,避免不必要的麻烦);如果存在多个参数时,参数之间使用逗号“,”分开。

3) 以“\$”开头,使用成对的圆括号或花括号把函数名和参数括起(在 Makefile 中,圆括号和花括号在任何地方必须成对出现)。参数中存在变量或者函数的引用时,对它们所使用的分界符(圆括号或者花括号)建议和引用函数的相同,不要使用两种不同的括号。推荐在变量引用和函数引用中统一使用圆括号;这样在使用 vim 编辑器书写 Makefile 时,使用圆括可以高亮显示 make 机制的内嵌函数名,避免函数名的拼写错误。例如:在 Makefile 中应该这样来书写“\$(sort \$(x))”;而不是“\$(sort \${x})”或其他。

4) 函数处理参数时,参数中如果存在对其他变量或者函数的引用,首先对这些引用进行展开得到参数的实际内容。而后才对它们进行处理。参数的展开顺序是按照参数的先后顺序来进行的。

5) 书写时,函数的参数不能出现逗号“,”和空格。这是因为逗号被作为多个参数的分隔符,前导空格会被忽略。在实际书写 Makefile 时,当有逗号或者空格作为函数的参数时,需要把它们赋值给一个变量,在函数的参数中引用这个变量来实现。

Makefile 中的内嵌函数分为字符串处理函数、文件名处理函数、其他函数等几类。

字符串替换函数如下所示:

```
$(subst FROM,TO,TEXT)
```

模式替换函数如下所示:

```
$(patsubst PATTERN,REPLACEMENT,TEXT)
```

去空格函数如下所示:

```
$(strip STRINT)
```

查找字符串函数如下所示:

```
$(findstring FIND,IN)
```

过滤函数如下所示:

```
$(filter PATTERN...,TEXT)
```

反过滤函数如下所示:

```
$(filter-out PATTERN...,TEXT)
```

排序函数如下所示:

```
$(sort LIST)
```

取单词函数如下所示:

```
$(word N,TEXT)
```

取首单词函数如下所示:

```
$(firstword NAMES...)
```

取目录函数如下所示：

```
$(dir NAMES...)
```

取文件名函数如下所示：

```
$(notdir NAMES...)
```

取后缀函数如下所示：

```
$(suffix NAMES...)
```

取前缀函数如下所示：

```
$(basename NAMES...)
```

加后缀函数如下所示：

```
$(addsuffix SUFFIX,NAMES...)
```

加前缀函数如下所示：

```
$(addprefix PREFIX,NAMES...)
```

单词连接函数如下所示：

```
$(join LIST1,LIST2)
```

获取匹配模式文件名函数如下所示：

```
$(wildcard PATTERN)
```

循环函数如下所示：

```
$(foreach VAR,LIST,TEXT)
```

在函数上下文中实现条件判断如下所示：

```
$(if CONDITION,THEN-PART[,ELSE-PART])
```

创建定制参数化的函数的引用函数如下所示：

```
$(call VARIABLE,PARAM,PARAM,...)
```

直接返回变量“VARIBALE”代表的值如下所示：

```
$(value VARIABLE)
```

eval 函数表示将后面的文本作为 Makefile 的语句来执行，如下所示：

```
$(eval text)
```

origin 函数用于表示变量的来源，通常用于调试 Makefile，如下所示：

```
$(origin VARIABLE)
```

在 **origin** 函数的输出结果中，**file** 表示来自文件自己当中的变量，**environment** 表示来自环境变量，**command line** 表示来自命令行的变量，**override** 表示重定义过的变量。此外，**automatic** 表示自动的变量，**default** 表示默认的变量，**undefined** 表示变量没有定义。

error 函数如下所示：

```
$(error TEXT...)
```

此外，Makefile 中还包含了 shell 命令行当中的函数。

3.3 Makefile 使用示例

3.3.1 简单的 Makefile

Makefile 书写的基本格式如下所示：

```
TARGET... : PREREQUISITES...
      COMMAND
```

TARGET 表示为当前的目标，PREREQUISITES 为目标先决条件，COMMAND 为目标所需要执行的命令。注意：COMMAND 需要使用 TAB 作为开头。

一个简单的 Makefile 内容如下所示：

```
rule:
    @echo "==== Makefile test ====="
```

执行的结果为：

```
$ make
==== Makefile test =====
```

在没有指定目标的情况下，将自动找到第一个目标进行执行。

在 Makefile 具有多个目标的情况下，可以使用如下的命令进行执行：

```
$ make rule
```

3.3.2 依赖关系实例

Makefile 不仅可以用于编译，也可用于处理其他的逻辑。本部分以一个简单的 Makefile 使用为例，说明 Makefile 的处理过程。

本部分所使用的 Makefile 文件如下所示：

```
all:rule0 file.o

rule0:rule1
    @echo "==== rule0 ====="
    @echo 'The deps:$^'
    @echo 'The target:$@'

rule1:rule2
    @echo "==== rule1 ====="

rule2:rule3
    @echo "==== rule2 ====="

rule3:
    @echo "==== rule3 ====="
```

```
file.o:
    @echo "hello:1234567890" > file.o
    @echo "File path: $(@D) File name : $(@F)"

.PHONY : clean rule0 rule1 rule2 rule3

clean:
    @echo "----- clean -----"
    rm -f file.o
```

在这个 Makefile 的路径下，执行 make 命令：

```
$ make
```

执行显示的结果为：

```
===== rule3 =====
===== rule2 =====
===== rule1 =====
===== rule0 =====
The deps:rule1
The target:rule0
File path: . File name : file.o
```

由于 make 没有指定选项和目标，将默认使用 Makefile 文件，并执行其中的 all 目标。在执行的过程中，首先发现 all 目标依赖于 rule0 和 file.o 两个目标，因此需要完成这两个目标的处理。对于 rule0 目标，依次寻找它的依赖关系，直到找到 rule3 目标，然后再从 rule3 目标执行，依次执行 rule3, rule2, rule1, rule0。对于 file.o 目标，将生成 file.o 文件，它由 file.o 目标生成，内容为“hello:1234567890”，显示中使用的变量@D 表示目标所在目录的路径，@F 表示目标的文件名。

在规则 rule0:rule1 中，使用了变量\$^和\$@，前者表示依赖的所有文件，后者表示目标的名称。

Makefile 的执行顺序不是按照每条规则书写的先后，而是由规则之间的依赖关系确定的。由于在本例中 all 规则依赖于 rule0，因此必须在 rule0 的命令执行完成后才能执行 all 规则，这点在编译过程中确定依赖关系是非常有作用的。

在 Makefile 中，将目标 clean rule0 rule1 rule2 rule3 定义为伪目标（.PHONY），这是由于它们不是需要生成的内容的名称；file.o 是实际生成的结果，因此它是真实的目标，而不是伪目标。

在执行过一次 make 之后，再次执行 make 命令，得到的结果如下所示：

```
===== rule3 =====
===== rule2 =====
===== rule1 =====
===== rule0 =====
The deps:rule1
The target:rule0
```

从执行结果中可见，这次的执行只执行了 rule0 及其依赖的目标，没有执行目标 file.o。这是由于目标 file.o 所依赖的内容没有变化，所以这条目标不需要被执行。

执行 `make clean` 的结果如下所示:

```
$ make clean
----- clean -----
rm -f file.o
```

这次执行删除了 `file.o` 文件, 状态已经退回到 `make` 执行之前。因此再次执行 `make` 的时候, 将和首次执行是一致的。

在 `make` 命令的使用中, 可以使用 `-n` 参数显示执行的序列:

```
$ make -n
```

这次执行的结果为:

```
echo "==== rule3 ====="
echo "==== rule2 ====="
echo "==== rule1 ====="
echo "==== rule0 ====="
echo 'The deps:rule1'
echo 'The target:rule0'
echo "hello:1234567890" > file.o
echo "File path: . File name : file.o"
```

由此可见, 在本次的执行中, 只显示了需要执行的命令, 而不真正地执行这些命令。在这个过程中, 寻找依赖关系的过程和直接的 `make` 过程是一致的, 但是只显示要执行命令而不去执行。

在使用 `make` 的过程中, 也可以指定一条单独的目标来执行, 例如:

```
$ make rule2
==== rule3 =====
==== rule2 =====
```

这时, 将指定目标 `rule2` 来执行, 执行的过程发现它依赖于目标 `rule3`, 因此先执行 `rule3` 的内容, 再执行目标 `rule2` 的内容。对于其他的目标, 则不需要执行。

3.3.3 隐含规则的编译实例

本部分仍然以上述程序的编译为例, `Makefile` 和程序在一个文件夹中。

本部分所使用的 `Makefile` 文件如下所示:

```
CC      := gcc
HEAD    := hello.h init.h
SRC      := hello.c init.o main.c
OBJS     := hello.o init.o main.o
TT       := test

Files := $(wildcard ./*)

INC = .
CFLAGS = -pipe -g -Wall -I$(INC)
LDFLAGS = -Wall -g

all:$(TT)

$(TT):$(OBJS)
```

```

@echo "==== Build Standalone application : $@ ====="
$(CC) $(LDFLAGS) $(OBJS) -o $@
libtest_d.so:hello.o init.o
@echo "==== Build dynamic lib : $@ ====="
$(CC) -shared $(LDFLAGS) hello.o init.o -o $@

test_exe:libtest_d.so main.o
@echo "==== Build exe : $@ ====="
$(CC) $(LDFLAGS) main.o -L. -ltest_d -o $@

filelist:
@echo "<<<<<< Files in this folder >>>>>>"
@file $(Files)
.PHONY : clean filelist
%.O:%c
$(CC) $(CFLAGS) -c $< -o $@

clean:
@echo "----- clean -----"
rm -f *.o
rm -f $(TT)
rm -f libtest_d.so
rm -f test_exe

```

在本例中，定义 CC 等变量，在变量的引用时，需要使用\$(CC)等形式。
使用 make 执行：

```

$ make
gcc -pipe -g -Wall -I. -c -o hello.o hello.c
gcc -pipe -g -Wall -I. -c -o init.o init.c
gcc -pipe -g -Wall -I. -c -o main.o main.c
==== Build Standalone application : test =====
gcc -Wall -g hello.o init.o main.o -o test

```

在执行的过程中，默认执行 all 目标。由于 all 目标依赖于变量\$(TT)，\$(TT)实际上是 test，\$(TT)依赖于\$(OBJS)，\$(OBJS)就是 hello.o init.o main.o。由此，需要产生这 3 个目标文件。

在所有的规则中，并没有这 3 个目标文件的生成规则，因此使用默认的目标%.o:%c 中的规则生成这 3 个目标文件。由此，使用 gcc 编译生成这 3 个目标文件。

下一步将执行 test 目标，进行目标文件的连接。

事实上，在以上的执行过程中，只是直接执行了 all 目标，在 Makefile 中还有 libtest_d.so、test_exe 和 filelist 几个目标没有执行，这些目标可以单独执行。

执行单独的目标 filelist:

```
$ make filelist
```

显示的结果如下所示：

```

$ make filelist
<<<<<< Files in this folder >>>>>>
./hello.c:      ASCII C program text
./hello.h:      ASCII text
./hello.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./init.c:       ASCII C program text
./init.h:       ASCII text

```

```
./init.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./main.c:      ASCII C program text
./main.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./Makefile:    ASCII make commands text
./test:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.6.4, dynamically linked (uses shared libs), for GNU/Linux 2.6.4, not stripped

<<<<<< Files in this folder >>>>>>
./hello.c:     ASCII C program text
./hello.h:     ASCII text
./hello.o:     ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./init.c:      ASCII C program text
./init.h:      ASCII text
./init.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./main.c:      ASCII C program text
./main.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
./Makefile:    ASCII make commands text
./test:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.6.4, dynamically linked (uses shared libs), for GNU/Linux 2.6.4, not stripped
```

这条目标执行的命令是使用 `file` 工具查看本文件夹下的所有文件信息, `Files := $(wildcard /*)` 表示使用通配符寻找目录下的所有文件。

执行生成可执行程序 `test_exe` 的命令:

```
$ make test_exe
```

结果如下所示:

```
===== Build dynamic lib : libtest_d.so =====
gcc -shared -Wall -g hello.o init.o -o libtest_d.so
===== Build exe : test_exe =====
gcc -Wall -g main.o -L. -ltest_d -o test_exe
```

`test_exe` 目标是一个可执行程序, 它本身需要连接一个动态库 `libtest_d.so`, 因此它依赖于目标 `libtest_d.so` 和 `main.o` 目标。`main.o` 已经生成了, 这样还需要生成 `libtest_d.so` 目标。在 `libtest_d.so` 目标中, 依赖的文件 `hello.o` 和 `init.o` 都已经生成了, 因此直接生成这个动态库即可。`libtest_d.so` 生成后, 再生成 `test_exe` 可执行程序。

本例中的 `test` 和 `test_exe` 都是可执行的程序, 它们的区别在于前者包含了 3 个目标文件, 可以直接执行, 后者只包括了 `main.o` 一个目标文件, 它的执行必须依赖动态库。

在清除目标之后, 继续执行 `test_exe` 可执行程序:

```
$ make test_exe
===== Build dynamic lib : libtest_d.so =====
gcc -shared -Wall -g hello.o init.o -o libtest_d.so
===== Build exe : test_exe =====
gcc -Wall -g main.o -L. -ltest_d -o test_exe
```

在这次执行的过程中, 由于 `hello.o`、`init.o` 和 `main.o` 这 3 个目标文件还没有生成, 因此在生成库 `libtest_d.so` 之前, 需要先编译生成 `hello.o` 和 `init.o` 两个目标, 它们使用的是默认的规则。在 `libtest_d.so` 生成后, 还需要生成 `main.o` 的过程, 它也需要使用默认的规则。

使用 `clean` 清除目标如下所示:

```
$ make clean
```

```

----- clean -----
rm -f *.o
rm -f test
rm -f libtest_d.so
rm -f test_exe

```

3.3.4 指定依赖的编译实例

本部分仍然使用上述程序的编译为例，介绍一个指定依赖关系的 Makefile 的使用。所使用的 Makefile 文件如下所示：

```

ifeq (arm,${ARCH})
CC    := arm-linux-gcc
else
ARCH  := x86
CC    := gcc
endif

HEAD  := hello.h init.h
SRC   := hello.c init.o main.c
OBJS  := hello.o init.o main.o
TT    := test

INC = .
CFLAGS = -pipe -g -Wall -I$(INC)
LDFLAGS = -Wall -g

all:$(TT)
$(TT):$(OBJS)
    @echo "==== Build Standalone application : $@ ====="
    @echo "Build ARCH = $(ARCH)"
    $(CC) $(LDFLAGS) $(OBJS) -o $@

main.o:main.c hello.h init.h
    $(CC) $(CFLAGS) -c $< -o $@

hello.o:hello.c hello.h
    $(CC) $(CFLAGS) -c $< -o $@

init.o:init.c init.h
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY : clean

clean:
    @echo "----- clean -----"
    rm -f *.o
    rm -f $(TT)

```

在这个过程中，没有使用默认的规则，而是对每一个目标文件实现单独的规则并指定所依赖的文件。

在 Makefile 目录下执行 make 命令：

```

$ make
gcc -pipe -g -Wall -I. -c hello.c -o hello.o
gcc -pipe -g -Wall -I. -c init.c -o init.o
gcc -pipe -g -Wall -I. -c main.c -o main.o
===== Build Standalone application : test =====

```

```
Build ARCH = x86
gcc -Wall -g hello.o init.o main.o -o test
```

这个结果依然是先编译生成目标文件，然后连接生成可执行程序。实际上，这个执行过程依次执行了 `hello.o`、`init.o`、`main.o` 和 `test` 规则。

在目标生成之后，如果再次使用 `make` 命令，将显示以下内容：

```
$ make
make: Nothing to be done for `all'.
```

更新文件 `main.c` 后，继续执行 `make` 命令：

```
$ touch main.c
```

结果如下所示：

```
$ make
gcc -pipe -g -Wall -I. -c main.c -o main.o
===== Build Standalone application : test =====
Build ARCH = x86
gcc -Wall -g hello.o init.o main.o -o test
```

在执行的过程中，可以发现先后执行了 `main.o` 和 `test` 目标中的规则。这是由于 `main.o` 目标依赖于 `main.c` 文件，因此 `main.c` 更新后，这个目标就需要重新生成。`hello.o` 和 `init.o` 目标的规则是不需要执行的，因为它们依赖的文件没有更新。

更新 `hello.h` 文件，然后重新生成，过程如下所示：

```
$ touch hello.h
$ make
gcc -pipe -g -Wall -I. -c hello.c -o hello.o
gcc -pipe -g -Wall -I. -c main.c -o main.o
===== Build Standalone application : test =====
Build ARCH = x86
gcc -Wall -g hello.o init.o main.o -o test
```

在这次执行的过程中，由于 `hello.o` 和 `main.o` 目标都依赖 `hello.h` 文件，因此，这两个目标都需要重新执行。

执行清除命令 `clean`，过程如下所示：

```
$ make clean
----- clean -----
rm -f *.o
rm -f test
```

在 `make` 的时候，可以通过命令行指定变量的名称，例如在本 `Makefile` 中使用变量 `ARCH` 可以在命令行指定它的值：

```
$ make ARCH=arm
arm-linux-gcc -pipe -g -Wall -I. -c hello.c -o hello.o
arm-linux-gcc -pipe -g -Wall -I. -c init.c -o init.o
arm-linux-gcc -pipe -g -Wall -I. -c main.c -o main.o
===== Build Standalone application : test =====
Build ARCH = arm
arm-linux-gcc -Wall -g hello.o init.o main.o -o test
```

变量 ARCH 将传递到 Makefile 之中。因此，以上的命令执行，使用 ARM 的交叉编译工具进行编译和连接。

这种实现通过在 Makefile 中使用 ifeq () 函数来控制：

```
ifeq (arm,${ARCH})
CC      := arm-linux-gcc
else
ARCH     := x86
CC       := gcc
endif
```

如果指定 ARCH 为 arm 时，将 CC 指定为 arm-linux-gcc 交叉编译工具；如果不指定，将 ARCH 定义为 x86，并将 CC 执行为 gcc。

3.4 自动生成 Makefile

在实际的项目中，由于 make 规则的复杂性和不确定性，因此自己编写 Makefile 是一件费时费力的事情。Makefile 本身具有一定的相似性，因此利用 GNU autoconf 及 automake 这两套工具可以协助自动产生 Makefile 文件，并且让开发出来的软件可以像大多数源代码包那样，只需运行命令“./configure”、“make”、“make install”就可以把程序安装到系统中，对于各种源代码包的分发和兼容性具有很好的效果。

3.4.1 autoconf 工具介绍

autoconf 是一个用于产生可以自动配置源代码包，生成 Shell 脚本的工具，它可以适应各种类 UNIX 系统的需要。autoconf 产生的配置脚本在运行时独立于 autoconf，也就是说使用这些脚本的用户不需要安装 autoconf。

autoconf 机制包括以下几个相关的文件。

- autoconf 生成的配置脚本通常名称是 configure，得到这个文件，通常需要以下的依赖文件。
- configure.in 文件：生成 configure 的必需文件，需要手动编写。
- aclocal.m4 和 acsite.m4 文件：在编写了除 autoconf 提供的测试外的其他测试补充时，才会用到这两个文件，也需要手动编写。
- acconfig.h 文件：如果使用了含有 \$ define 指令的头文件，则需要自行编写该文件，一般都需要使用，这个时候会生成另外一个 config.h.in 文件，这个文件需要和软件包一同发布。

总之，在 autoconf 运行完毕后，得到两个需要和软件包同时发布的文件：configure 和 config.h.in，config.h.in 也可以不存在。

3.4.2 automake 工具介绍

automake 是一个从文件 Makefile.am 自动生成 Makefile.in 的工具。每个 Makefile.am 基本上是一系列 make 的宏定义（make 规则也会偶尔出现）。生成的 Makefile.in 也服从 GNU

Makefile 标准。

典型的 automake 输入文件是一系列简单的宏定义。处理所有相关的文件并创建 Makefile.in 文件。在一个项目的每个目录中通常仅包含一个 Makefile.am。

目前 automake 支持 3 种目录层次：平坦模式（flat）、混合模式（shallow）和深层模式（deep）。

1) 平坦模式指的是所有文件都位于同一个目录中。就是所有源文件、头文件及其他库文件都位于当前目录中，且没有子目录。

2) 混合模式指的是主要的源代码都存储在顶层目录，其他各个部分则存储在子目录中。也就是主要源文件在当前目录中，而其他一些实现各部分功能的源文件位于各自不同的目录。

3) 深层模式指的是所有源代码都被存储在子目录中；顶层目录主要包含配置信息。也就是所有源文件及程序员自己写的头文件都位于当前目录的一个子目录中，而当前目录里没有任何源文件。

在这 3 种支持的目录层次中，平坦模式类型是最简单的，深层模式类型是最复杂的。但是这些模式使用 autoconf 和 automake 所遵循的基本原则和流程是一样的。

3.4.3 其他工具

使用自动生成 Makefile 的话，需要涉及下面其他几个工具。

1) autoheader: 能够产生供 configure 脚本使用的 C \$ define 语句模板文件。

2) autom4te: 对文件执行 GNU M4。

3) autoreconf: 如果有多个 autoconf 产生的配置文件，autoreconf 可以保存一些相似的工作，它通过重复运行 autoconf（以及在合适的地方运行 autoheader）以重新产生 autoconf 配置脚本和配置头模板，这些文件保存在以当前目录为根的目录树中。

4) autoscan: 该程序可以用来为软件包创建 configure.in 文件。autoscan 在以命令行参数中指定的目录为根（如果未给定参数，则以当前目录为根）的目录树中检查源文件。为软件包创建一个 configure.scan 文件，该文件就是 configure.in 的前身。

5) autoupdate: 该程序的作用是转换 configure.in，从而使用新的宏名。

3.4.4 自动生成 Makefile 的流程

在进行自动化生成 Makefile 之前，务必要设定好工作的根目录，在当前环境下，至少要保证 autoscan、autoconf、aclocal、automake 这些命令能够正常运行。在这一节中，我们就以一个最简单的示例来说明 automake 和 autoconf 的基本使用方法，这个例子是一个平坦模式的模型。

使用 autoconf 和 automake 来进行自动化配置和生成 Makefile 的流程可以概括如下：

1) 运行 autoscan 命令。

2) 将 configure.scan 文件重命名为 configure.in，并修改 configure.in 文件，如图 3-2 所示。

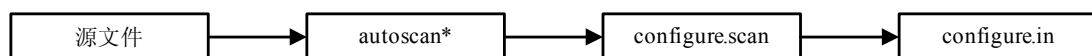


图 3-2 configure.in 文件的生成

- 3) 运行 `aclocal` 命令得到 `aclocal.m4` 文件。
 - 4) 运行 `autoconf` 命令得到 `configure` 文件。
 - 5) 在工程目录下新建 `Makefile.am` 文件，如果存在子目录，子目录中也要有此文件。
- 步骤 3 至步骤 5 的执行如图 3-3 所示。

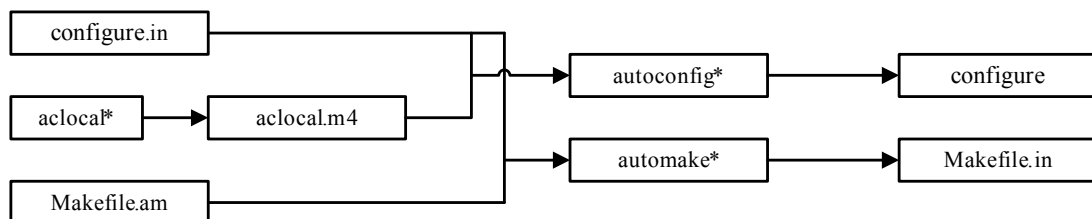


图 3-3 configure 文件和 Makefile.in 的生成

- 6) 将 `/usr/share/automake-{}/` 目录下的 `depcomp` 和 `compile` 文件复制到需要处理的目录下。
- 7) 运行 `automake -a` 命令得到 `Makefile.in` 文件。
- 8) 运行 `./configure` 脚本，执行编译，如图 3-4 所示。

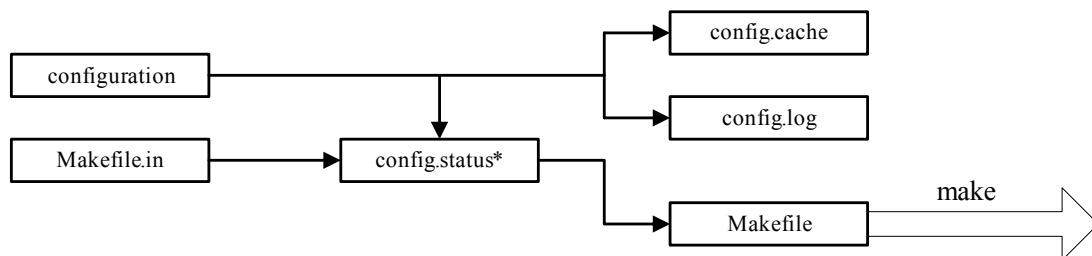


图 3-4 Makefile 的生成

第 4 章

调试和 GDB

4.1 嵌入式系统的调试技术

4.1.1 调试技术

嵌入式系统的特点：目标系统不具有自举开发的能力，性能也较 PC 系统为低。嵌入式系统通常需要使用一种称为交叉调试（cross debug），也称为远程调试（remote debug）的技术。

交叉调试的概念与交叉编译类似，在交叉调试技术中，运行调试器软件的机器称为主机，例如一个运行 Linux 系统或 Windows 系统的 PC；被调试的系统（即开发中的嵌入式系统）称为目标机；主机与目标机之间可以通过串行接口、网络接口或特殊的硬件调试接口等方式进行连接。

从调试的技术实现途径以及它的应用情景两个角度看，其通常可以分为两种技术：

- 硬件级调试工具技术。
- 源码级软件调试工具技术。

例如：最常见的在线仿真器 ICE 和 JTAG 仿真器就属于硬件级调试工具，而 GDB（包括运行在目标机上的 gdbserver 或 gdbstub 程序）则属于源码级软件调试工具。

硬件级调试工具与源码级软件调试工具在许多方面均存在很大的区别。硬件级调试工具通常适用于嵌入式系统的硬件设计与调试阶段，以及嵌入式系统的系统软件环境（包括 bootloader、操作系统内核和库等）搭建阶段，例如：在一块嵌入式板实现一个 bootloader 程序，或者将 Linux 内核移植到一种新型体系结构的 CPU 上，等等。当使用硬件级调试工具技术时，主机与目标机之间一般是通过特殊的硬件调试接口来连接的。

而源码级的软件调试工具则一般使用串口或 TCP/IP 网络接口来实现主机到目标机的连接，它的应用一般必须得到目标机上系统软件环境的支持，包括：内核、交叉编译器、库程序、shell 交互程序、终端仿真程序等。通常可以用源码级软件调试工具来调试运行在

目标上的应用软件。

4.1.2 硬件调试

硬件调试技术常用于调试嵌入式系统的 CPU 以及相关硬件，如下所示。

- 在线仿真器（In-Circuit Emulators, ICE）：代替了物理上的目标机的处理器。
- 片上调试器：封装在 CPU 内部。
- JTAG 技术（Joint Test Action Group，联合测试行动小组）：通过特殊的引脚检测。IEEE 1149.1-1990 标准，可对具有 JTAG 接口芯片的硬件电路进行边界扫描和故障检测。

1. 在线仿真器

在线仿真器（In-Circuit Emulators, ICE）是在嵌入式系统领域使用得最多的调试手段之一。对在线仿真器的一个比较恰当的描述是：在线仿真器是一个用来设计其他计算机系统的计算机，它代替了物理上的目标机的处理器或微控制器，其表现与被代替的目标机处理器完全一样，即它们可以运行相同的程序，引脚的定义也是相同的。在线仿真器的好处是用户可以查看处理器内部的数据或代码并控制 CPU 的运行。

2. 片上调试器

随着现代的微处理器封装越来越表贴化，例如：LQFP（Low profile Quad Flat Package，薄型四方扁平封装）和 FBGA（Fine Ball Grid Array，底部球型引脚封装）封装形式，仿真器探头的实现也就越来越困难。但是如果不能对微处理器实现探测，也就不能知道微处理器上正在执行的一切操作细节信息，从而也不能实现测试与调试的目的。另一方面，嵌入式系统调试统计数字也表明：在大约 95% 的调试过程中用户仅仅使用了简单断点、单步以及访问处理器资源、内存和外设等一些运行控制方面的基本调试手段。

与 ICE 或逻辑分析仪相比，片上调试器不存在任何因 CPU 封装或 CPU 速度而带来的那些问题。CPU 与调试器之间的连接问题也不存在，因为调试器只占用几根专用的 CPU 引脚。通过片上调试逻辑实现的运行控制，可以对目标机 CPU 进行设置断点、单步执行以及资源读写访问等绝大多数的基本调试动作。此外，片上调试器同样也不占用任何目标系统的资源。

相比仿真器，片上调试器而不是仿真器时也具有如下的限制：

- 片上调试器是不存在重叠 RAM（Overlay RAM）的。这给开发人员带来一些不便，因为为了验证修改后的代码，开发人员不得将它们重新下载到目标机的 ROM 或 Flash 上。
- 断点功能没有 ICE 强大。大多数片上调试器不允许复杂的嵌套断点条件，有些甚至不允许硬件断点。
- 实时跟踪功能也不再可用了。这是从 ICE 转到片上调试器所带来的一个最大损失（尤其对于实时的嵌入式应用而言）。

3. JTAG

JTAG 是 1985 年制定的检测 PCB 和 IC 芯片的一个标准，1990 年被修改后成为 IEEE 的一个标准，即 IEEE 1149.1-1990。通过这个标准，可对具有 JTAG 接口芯片的硬件电路进行边界扫描和故障检测。

IEEE 1149.1 标准中规定对应于数字集成电路芯片的每个引脚都设有一个移位寄存单元，称为边界扫描单元（Boundary Scan Cell，BSC），它将 JTAG 电路与内核逻辑电路联系起来，同时隔离内核逻辑电路和芯片引脚。由集成电路的所有边界扫描单元（BSC）构成边界扫描寄存器（Boundary Scan Register，BSR）。边界扫描寄存器电路仅在 JTAG 测试时有效，在集成电路正常工作时无效，不影响集成电路的功能。

4.1.3 代码调试

代码程序的调试方式完全不同于硬件调试的方式，属于软件调试的范畴。通常的源代码调试分成两个阶段：首先，基本内容是调试信息，需要支持程序中打印的信息；进一步可以使用调试工具进行调试，支持断点、单步和获取程序运行时状态的功能。

在 Linux 系统当中，用户空间程序的调试与内核的调试不同。

- 调试信息

用户空间的程序使用 `printf()`，内核中使用 `printk()`。

- 工具的调试

调试工具为 GDB，在内核中需要特殊的机制支持，包括 KDB（用于转存的交互式内核调试器）、KGDB（内核中的 GDB）。

4.2 Linux 的基本信息

在 Linux 环境中，最基本的 C 语言程序的调试是通过增加调试信息完成的。

在 Linux 中获取时间处理的方法，使用 Linux 特定头文件 `sys/time.h` 中的内容。

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv,
                 const struct timezone *tz);
struct timeval {
    time_t      tv_sec;        /* seconds */
    suseconds_t tv_usec;      /* microseconds */
};
struct timezone {
    int tz_minuteswest;        /* minutes west of Greenwich */
    int tz_dsttime;           /* type of DST correction */
};
```

`gettimeofday()` 函数用于获取时间，第一个参数表示时间，第二个参数表示时间区域。在获取时间信息的时候，只使用第一个参数即可。

在程序中，通常可以有如下的写法：

```
#include <sys/time.h>
```

```
struct timeval tv;
gettimeofday(&tv, NULL);
printf("Time = %d %d \n", (int)tv.tv_sec, (int)tv.tv_usec);
```

gettimeofday()函数被调用之后, 信息从 timeval 结构的指针中返回, tv_sec 表示秒, 而 tv_usec 表示微秒, 它们的本质都是整数值。

4.3 GDB 调试和远程调试

从原理上看, GDB 工具只要经过嵌入式系统交叉编译工具的编译, 就可以在嵌入式系统中直接使用, 使用的方式和在桌面电脑中的使用方式基本相同, 但是由于嵌入式系统性能的局限, 会给 GDB 的使用带来一些困难。因此, 在嵌入式系统中使用更多的是远程 GDB。

GDB 是一个强大的命令行调试工具。命令行的强大就在于其可以形成执行序列, 形成脚本。UNIX 下的软件全是命令行的, 这给程序开发提供了极大的便利。命令行软件的优势在于, 它们可以非常容易地集成在一起, 使用几个简单的已有工具的命令, 就可以做出一个非常强大的功能。

GDB 主要提供以下几个方面的功能:

- 启动的程序, 可以按照自定义的要求运行程序。
- 可让被调试的程序在指定的断点处停住 (断点可以是条件表达式)。
- 当程序被停住时, 可以检查这个时候程序中所发生的事。
- 动态地改变程序的执行环境。

在嵌入式系统使用 GDB 的时候, 目标程序包含几个种类, 对它们需要分别进行编译:

- 运行于主机 (x86) 的主机调试工具。
- 运行于主机 (x86) 的目标机调试工具 (交叉调试工具)。
- 与目标机程序编译在一起的 gdbstub。
- 运行于目标机的 gdbserver。

其中, 最常用的手段是: 主机运行交叉调试工具, 目标机运行 gdbserver 或者加入了 gdbstub 的被调试程序。

1. GDB 调试和命令

GDB 调试的根本原理是通过 gdb 运行被调试的程序。在使用 gdb 调试的时候, 在编译用户的应用程序时应该加上 -g 选项, 以便指示 gcc 交叉编译器在编译时向目标文件中添加调试信息。进一步, 还可以使用 -ggdb 选项指示 gcc 交叉编译器生成更多专用于 GDB 调试器的调试信息。

GDB 程序的可执行程序是 gdb, 命令的格式如下所示:

```
gdb [options] [executable-file [core-file or process-id]]
gdb [options] --args executable-file [inferior-arguments ...]
```

GDB 本身的参数不多, 一般使用 gdb 以要调试可执行程序为参数运行即可, 运行后会出现调试的交互界面, 因此主要使用的是交互的命令。

GDB 主要的交互命令如下所示。

- **backtrace**: 显示程序中的当前位置和表示如何到达当前位置的栈跟踪。
- **breakpoint**: 在程序中设置一个断点。
- **cd**: 改变当前工作目录。
- **clear**: 删除刚才停止处的断点。
- **commands**: 命中断点时, 列出将要执行的命令。
- **continue**: 从断点开始继续执行。
- **delete**: 删除一个断点或监测点。
- **display**: 程序停止时显示变量和表达式。
- **down**: 下移栈帧, 使得另一个函数成为当前函数。
- **frame**: 选择下一条 **continue** 命令的帧。
- **info**: 显示与该程序有关的各种信息。
- **jump**: 在源程序中的另一点开始运行。
- **kill**: 异常终止在 **gdb** 控制下运行的程序。
- **list**: 列出正在执行的程序的源文件内容。
- **next**: 执行下一个源程序行, 从而执行其整体中的一个函数。
- **print**: 显示变量或表达式的值。
- **pwd**: 显示当前工作目录。
- **ptype**: 显示一个数据结构的内容。
- **quit**: 退出 **gdb**。
- **reverse-search**: 在源文件中反向搜索正规表达式。
- **run**: 执行该程序。
- **search**: 在源文件中搜索正规表达式。
- **set variable**: 给变量赋值。
- **signal**: 将一个信号发送到正在运行的进程。
- **step**: 执行下一个源程序行, 必要时进入下一个函数。
- **undisplay**: **display** 命令的反命令, 不要显示表达式。
- **until**: 结束当前循环。
- **up**: 上移栈帧, 使另一函数成为当前函数。
- **watch**: 在程序中设置一个监测点 (即数据断点)。
- **whatis**: 显示变量或函数类型。

很多 GDB 的命令还有缩写, 例如: **c** 表示执行 **continue**, **r** 表示执行 **run**。在调试的时候, 可以简化命令的执行。

GDB 调试可以直接在命令行运行程序, 也可以使用 **attach** 的方式连接到某个已经运行的进程上。在实际的应用中, 更多使用 **attach** 的方式。

GDB 调试与原始程序的行号相关, 因此程序的书写格式 (C 语言源代码) 也会影响调试, 尤其需要注意宏展开的问题。

提示：如果程序进行优化编译（-O2 等），编译结果和原始的行号可能错位，由此将会对 GDB 调试有所影响。

一段循环程序如下所示：

```

1  #include <stdio.h>
2  #include <unistd.h>
3
4  void dosth()
5  {
6      printf("File:%s Function:%s Line:%d\n",
7             __FILE__, __FUNCTION__, __LINE__);
8  }
9
10 int main (int argc, char* argv[])
11 {
12     int i;
13     for(i=0; i<100; i++){
14         printf("number:%d\n", i);
15         dosth();
16         printf("...sleep...\n");
17         sleep(1);
18     }
19     return 0;
20 }
```

因为需要调试，使用编译工具增加-g 参数，编译和运行如下所示：

```

$ gcc -g sleep_test.c -o sleep_test
$ ./sleep_test
number:3
File:sleep_test.c Function:dosth Line:7
...sleep...
number:4
```

在程序运行时，就可以使用 gdb 进行调试，此时需要知道所运行程序的进程 id，可以通过 ps 命令得到。使用 gdb 调试一个已经运行进程的方式如下所示：

```

$ gdb attach 19082
(gdb) break dosth
Breakpoint 1 at 0x804845a: file sleep_test.c, line 6.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) continue
Continuing.
Breakpoint 1, dosth () at sleep_test.c:6
```

进行 attach 之后，被 attach 的程序可以被停住，然后将可以进行各种 gdb 命令的操作，如设置断点等。以上的程序设置断点后，被重新运行，然后停留在断点上，此时进一步执行 gdb 命令的调试如下所示：

```

(gdb) continue
Continuing.
Breakpoint 1, dosth () at sleep_test.c:6
6      printf("File:%s Function:%s Line:%d\n",
```

```
(gdb) list
1  #include <stdio.h>
2  #include <unistd.h>
3
4  void dosth()
5  {
6      printf("File:%s Function:%s Line:%d\n",
7             __FILE__, __FUNCTION__, __LINE__);
8  }
9
10 int main (int argc, char* argv[])
```

在断点停住之后，可以使用 `list` 命令列出当前源代码的情况，并可以进行下一步的调试。

GNU DDD (Data Display Debugger) 为数据可视化调试器，可以作为 GDB、JDB 等调试器的前端使用。DDD 的运行界面如图 4-1 所示。

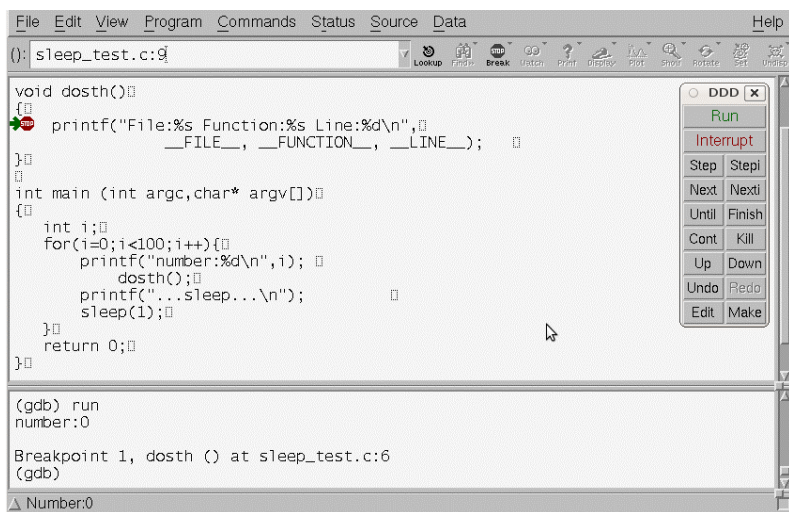


图 4-1 DDD 的运行界面

DDD 的使用方式和命令行的 `gdb` 类似，只是很多命令一般都可以通过图形化的界面运行。在 DDD 的图形界面中，同样也可以调用出 `gdb` 的命令行。

2. 远程 GDB 调试

在远程 GDB 中，有一小段驻留在目标机上的代码，它被称为调试桩 (debugging stub)，也被称为调试代理 (debugging agent)。它的责任就是在目标机上实现由主机上的调试器发送过来的调试命令，例如：读写内存、读写寄存器、设置断点以及运行被调试程序等。此外，还要向主机调试器报告目标机上发生的异常事件，例如：断点命中，以及被除 0 等这样的程序错误等。调试代理与 GDB 主机之间的通信遵循 GDB 远程串行协议 (Remote Serial Protocol)，简称 GDB RSP 协议。关于 GDB RSP 协议的具体内容可以参考 GDB 用户手册 *Debugging With GDB* 的附录 D。

GDB 和 GDB 调试代理进行远程调试的系统框架结构如图 4-2 所示。

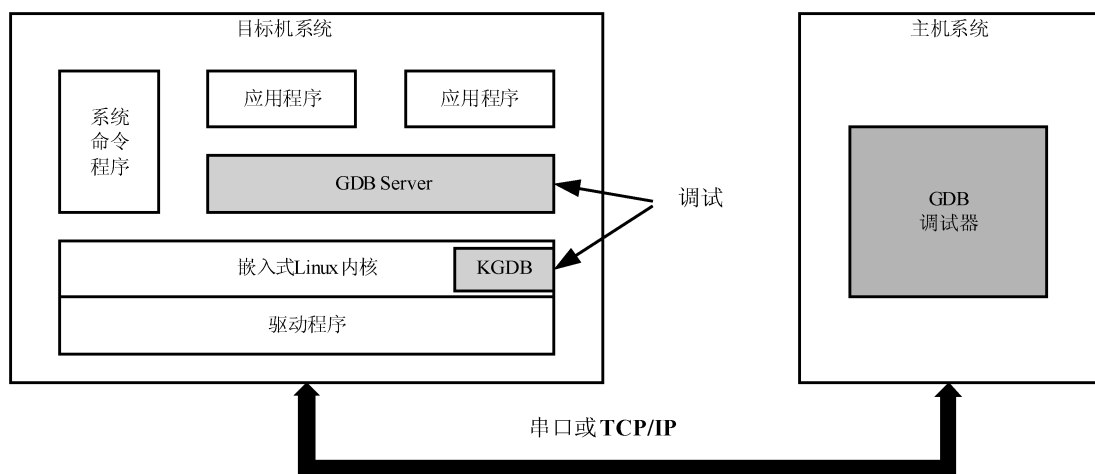


图 4-2 GDB 进行远程调试的系统框架结构

在 GDB 调试代理已经驻留在目标机上的情况下，一个典型的交叉调试会话一般都遵循如下这样的 3 个步骤。

第一步：主机上的 GDB 调试器向调试代理发送一系列的内存写命令，以便把即将被调试的应用程序从主机下载到目标机的内存中。显然，如果应用程序已经通过其他方法安装到目标机上的话，那么这一步可以省去。

第二步：GDB 调试器向调试代理发出断点设置命令，以便在应用程序合适的位置设置断点。断点设置好后，调试器就可以命令调试代理跳转到应用程序的入口点处开始执行程序，从而启动调试会话。

第三步：当运行中的应用程序遇到断点时，其执行路径被中断暂停，目标机的控制权重新回到调试代理手中。调试代理首先必须将被调试程序因遇到断点而暂停执行这一事件通知主机上的调试器，然后它就一直等待主机调试器给它发出的下一个命令。主机上的调试器在得知目标机上的被调试程序遇到断点而暂停执行后，就可以像在单机进行正常的调试一样，查询或修改目标机上的内存，读写目标机处理器的寄存器等。当然，这一切也都是通过向目标机上的调试代理发送调试命令来实现的。但是，主机上的调试器最终必须告诉调试代理继续被调试应用程序的执行。

当前已经实现的 GDB 调试代理主要有以下 3 种。

- 最基本的 gdbstub 实现。

`gdbstub` 源程序需要和目标程序编译在一起，其特点是对目标系统没有要求，只要可以编译 `gdbstub` 源代码即可。

- `gdbserver` 程序。

`gdbserver` 是一个支持多种体系结构处理器的工具，在目标机使用 `gdbserver` 的特点是不需要更改被调试的源程序，但是目标机必须能够运行 `gdbserver` 程序。

- `KGDB` 程序。

`KGDB` 程序是一种专用于调试 Linux 内核的 `gdb` 调试代理实现。

实际上 gdbserver 和 KGDB 这样的实现都必须以 gdbstub 的基本实现为基础。

4.4 GDB 的安装与使用

在使用 GDB 之前，需要获取 GDB 的源代码包，可以从以下 FTP 服务器中下载，其目录为：<ftp://ftp.gnu.org/pub/gnu/gdb/>。

将 GDB 源代码解压缩到当前目录：

```
$ tar -xjvf gdb_src.tar.bz2
```

在 GDB 的源代码包中，主要的目录和文件如表 4-1 所示。

表 4-1 gdb_src 目录结构

目录或文件	类型	描述
configure	脚本	配置脚本，用于生成 Makefile
include/	目录	头文件目录
gdb/	目录	源代码文件夹
gdb/gdbserver/	目录	gdbserver 源代码文件夹

4.4.1 使用 gdbstub 实现调试用户程序

对于远程 GDB 调试的实现，gdbstub 是一种基本、直接的方式。

1. gdbstub 调试的实现

目前在 GDB 的源代码中，实现了的几种体系结构 gdbstub 如表 4-2 所示。

表 4-2 几种体系结构 gdbstub 的实现

文件	说明
i386-stub.c	Intel i386 体系结构的处理器实现的基本调试代理
m68k-stub.c	Motorola m68k 体系结构的处理器实现的基本调试代理
sh-stub.c	Hitachi 公司 SH 体系结构的处理器实现的基本调试代理
Sparc-stub.c	SPARC 体系结构的处理器实现的基本调试代理
Wince-stub.c	WinCE 设备的基本调试代理

这些基本的 gdbstub 实现都是被设计用来独立运行于目标板上的，也就是说它的运行不需要系统软件环境的支持。实际上，这些源文件都必须与被调试程序编译、连接在一起，才能实现用 GDB 远程调试目标机上的程序。而 GDB 主机端与调试代理通信的接口均以现在 GDB 源码包的 remote.c 源文件为实现基础。

首先，所有这些基本的 gdbstub 实现都必须提供的 3 个上层函数接口，如表 4-3 所示。

表 4-3 gdbstub 调试的 3 个上层函数

函数	描述
set_debug_traps()	当被调试程序终止时，该函数将显式地安排 handle_exception 例程来运行。被调试程序在一开始就必须显式地调用这个函数
handle_exception()	本函数是整个 gdbstub 的中心，被调试程序永远不应显式地调用这个函数接口。当被调试程序遇到一个陷入（trap）异常而终止运行时，该函数将被安排运行，接管整个目标机上的控制权，并负责与主机上的 GDB 调试器按照 RSP 协议进行通信。实际上，handle_exception 函数必须在目标机上实现 GDB 的 RSP 协议，它实际上是 GDB 在目标机上的代理人。当然 handle_exception 函数最终必须在接收到 GDB 主机发出的继续执行命令后将目标机的控制权返回给被调试程序，从而继续被调试程序的运行
breakpoint()	这是一个辅助函数接口，可以实现在被调试程序中插入断点。在某些系统中，简单地串口接收字符也会触发一个陷入（trap），那么对于这种目标机就不需要在被调试程序中显式地调用这个函数了。而在有些情况下，这也许是 GDB 获得控制权的唯一方法

除了以上 3 个高层接口外，在基本的 gdbstub 实现中还必须有一些必要的底层函数，以支持整个 gdbstub 的运行。

首先，必须有两个接口告诉 gdbstub 怎样读写目标机上的串口。

- **getDebugChar():** 从目标机上的串口中读取一个字符。这个函数其实与目标机系统软件上的 getchar 函数相似，这里只不过是为了相互区别而换一个不同的名字罢了。
- **putDebugChar():** 向目标机的串口中写入一个字符。其实它与目标机系统软件上的 putchar 函数相似，这里同样只不过是为了相互区别而换一个不同的名字罢了。

如果需要 GDB 能够在被调试程序运行期间通过 Ctrl-C 组合键来停止被调试程序，那么还必须在目标机上实现一个中断驱动型的简单串口驱动程序，以便由串口驱动程序来停止被调试程序。

其次，必须实现一些 gdbstub 专用的、必要的 C 语言库程序接口，这里面包含着下面的两层意思：

第一，专用的，即 gdbstub 最好不要和被调试程序共用一个 C 语言库，以避免在调试期间发生 gdbstub 代码和被调试代码之间相互交叉的情况；例如，用户也许会单步跟踪进入 strlen 函数内部。

第二，必要的，对于那些 gdbstub 没有用到的库函数不需要实现。常用到的库函数有 memset()、strcpy()、strlen()等。

以下两个支持接口通常也是必需的。

- **flush_i_cache():** 用于刷新目标机处理器的指令 cache。刷新指令 cache 的目的是确保被调试程序有一个稳定的状态。对于没有指令 cache 的处理器，它可以是个空函数。
- **exceptionHandler(int exception_number, void *exception_address):** 用于在目标机处理器的异常处理表中安装异常处理程序。参数 exception_address 表示异常处理程序的入口地址，参数 exception_number 表示异常号。异常号的具体意义依赖于处理器的

体系结构。当该异常发生时，控制权应该直接转到 `exception_address` 地址处，而且跳转到 `exception_address` 处的处理器状态必须和异常发生时的处理器状态完全相同。

当需要给一种新体系结构的处理器使用时，从已有的一个 `gdbstub` 实现开始能大大节省时间。例如：根目录下的 `sparc-stub.c` 是针对 SPARC 体系处理器的 `gdbstub` 源文件；根目录下的 `i386-stub.c` 是针对 x86 体系处理器的 `gdbstub` 源文件。

关于 `gdbstub`，要获得更多体系结构下的基本 `gdbstub` 实现的详细信息，可以访问以下 Web 站点：<http://sourceforge.net/projects/gdbstubs/>。

2. 使用 `gdbstub` 调试的步骤

使用 `gdbstub` 调试目标机上的用户程序的方法，可以按照如下步骤进行。

第一步，在用户程序的一开始插入下列代码行：

```
set_debug_traps();
breakpoint();
```

第二步，将所有的源代码和库放在一起编译和连接，包括：用户程序、`gdbstub` 源文件，以及相应的 C 语言库。

第三步，把编译好的用户程序下载到目标机上，运行它。

第四步，在主机上运行 GDB 调试器，并把正在目标机上运行的用户程序指定为可执行文件。这就告诉 GDB 调试器如何寻找被调试程序的符号以及程序正文的内容。

第五步，在主机上用 GDB 的 `target remote` 命令和目标机建立调试会话。命令 `target remote` 的参数取决于主机和目标机之间采用何种连接——串口连接或 TCP 端口连接。如果需要指定主机与目标机之间通过串口建立连接，可以使用命令 `target remote /dev/ttyS0`。如果要用 TCP 连接，则可以使用这样的参数形式：`host:port` 或 `tcp:host:port`，其中 `host` 是主机的名字，`port` 是端口号。

调试会话 (debugging session) 建立后，就可以像单机调试那样远程调试目标机上的用户程序了。

4.4.2 GDB 和 GDB Server 的编译

在 Linux 系统的交叉调试中，需要使用 `gdb` 和 `gdbserver`，需要进行生成几个方面的程序，如下所示。

- x86 的 GDB 调试器 (`gdb`)：运行于 x86 的主机。
- x86 的 GDB 服务器 (`gdbserver`)：运行于 x86 的目标机。
- x86->ARM 的 GDB 调试器 (`gdb`)：运行于 x86 主机，调试 ARM 目标机程序。
- ARM 的 GDB 服务器 (`gdbserver`)：运行于 ARM 目标机。

前面两个程序用于主机和目标机都是 x86 的调试，后面两个程序用于 x86 主机到 ARM 目标机的调试。x86 主机到其他体系结构的交叉调试与 ARM 类似。

GDB 的源代码包可以支持在另外的一个目录编译，源代码目录可以不受影响。在编译之前，需要使用 `configure` 命令进行配置，主要的选项如下所示。

- **--target:** 与特定目标机体系结构相对应的名字。
- **--prefix:** 指定交叉调试器的安装目录。与其他 GNU 工具链程序一样，安装后的二进制程序名依赖于选项**--target** 的值。

1. x86 版本的 GDB 编译

可以在 GDB 源代码目录的平行目录中，新建一个目录作为编译结果，如下所示：

```
$ mkdir build-gdb
$ cd build-gdb
$ ../gdb_src/configure
```

使用 `gdb_src` 目录下的 `configure` 脚本，在编译 GDB 的根目录下，配置之后，在 `build-gdb` 目录中将生成 `Makefile`。此时进行编译，只需要使用 `make` 即可：

```
$ make
```

编译的结果主要将生成 `gdb` 工具和 `gdbserver`，如下所示：

- `gdb` 工具在 `build-gdb/gdb` 目录中，运行于 x86 主机。
- `gdbserver` 在 `build-gdb/gdb/gdbserver` 目录中，运行于 x86 目标机。

编译完成后，可以进行安装：

```
$ make install
```

默认的安装目录为 `/usr/local/bin`、`/usr/local/lib` 等。

2. ARM 版本的 GDB 编译

需要生成 ARM 体系结构交叉调试工具，配置的命令如下所示：

```
$ mkdir build-arm-gdb
$ cd build-arm-gdb
$ ../gdb_src/configure --target=arm-linux
```

调用 `make` 进行编译之后，编译的结果主要将生成 `gdb` 调试器。此处的 `gdb` 调试器是 x86 主机上运行的程序，用于交叉调试 ARM 程序。

3. ARM 版本的 GDB Server 编译

需要生成 ARM 版本的 `gdbserver`，编译之前需要配置成 ARM 的版本：

```
$ cd gdb_src/
$ cd ..
$ mkdir build-arm-gdbserver
$ cd build-arm-gdbserver
$ ../gdb_src/gdb/gdbserver/configure --host=arm-linux --prefix=<prefix>
```

此时的编译是交叉编译，如果使用特殊的交叉编译器，则需要指定其前缀。编译的结果主要将生成 `gdbserver`，`gdbserver` 是运行于 ARM 系统的程序。

4.5 使用 gdbserver 调试

`gdbserver` 程序比 `gdb` 程序更小巧，因此，它更适合作为 GDB 调试器的代理在系统资源紧张的嵌入式目标系统上运行。

1. 调试的步骤

通常情况下，含有调试信息的应用程序比不含有调试信息的应用程序大很多。实际上，`gdbserver` 程序并不要求目标机上被调试的应用程序一定要包含调试信息，而只要保证被调试应用程序留在主机上的那份副本中含有调试信息就可以了，因为 GDB 调试器可以利用主机上含有调试信息的程序副本去进行一切符号处理。因此，当在主机上生成含有调试信息的应用程序后，可以用 `strip` 命令去除程序中的调试信息，然后将 `strip` 过后的应用程序下载到目标机上，从而节省目标机有限的资源空间。

在以上工作完成后，接下来就可以开始 GDB 与 `gdbserver` 之间的调试会话了。GDB 与 `gdbserver` 交互的大概步骤如下所示：

- 第一，在目标机系统上手工启动 `gdbserver` 程序，它将控制被调试程序的运行。
- 第二，在主机上启动 GDB 调试器。
- 第三，使用 `target remote` 命令连接目标机上的 `gdbserver` 进程。连接成功后，调试会话就开始了。

下面需要在目标机系统上手工启动 `gdbserver` 程序，因此在主机和目标机之间必须有两条在物理上相互独立的连接，其中一条用于 GDB 调试器与 `gdbserver` 之间的调试会话，另一条用于主机与目标机之间的虚拟终端连接。在实际使用的过程中，这两条连接的形式均可以是串口连接或以太网连接。

`gdbserver` 程序的命令行格式是：

```
gdbserver comm program [args ...]
```

- 参数 `comm`：用于指定调试会话的连接方式。

对于串口连接，其形式为 `/dev/ttyXX`，例如：串行端口 `/dev/ttyS0`。在使用串口调试的时候，需要为 GDB 提供一个单独的端口，例如：主机与目标机之间的虚拟终端连接通过串行端口 `/dev/ttyS0`，而 GDB 调试器与 `gdbserver` 程序之间的调试会话则使用串行端口 `/dev/ttyS1` 或以太网连接。

对于 TCP/IP 网络连接，其形式为 `host:port`，当前的 `gdbserver` 程序实际上忽略 `host` 的值，而端口号 `port` 则可以是任何不冲突的 TCP/IP 端口号，且它必须与主机 GDB 调试器的 `target remote` 命令中所使用的端口号一致。

- 参数 `program`：用于指定被调试应用程序的全路径名。
- 参数 `args`：用于指定运行被调试应用程序 `program` 本身所需要的参数

2. GDB 远程调试结构

`gdbserver` 更多用于主机-目标机的调试中。在一个典型的嵌入式系统中，主机为 x86，

目标机为 ARM 体系。

如果使用网络连接的调试方法，大部分步骤和 x86 本机调试类似，其区别主要体现在以下几个方面：

第一，在主机端使用的 GDB 程序为交叉版本的 GDB。

第二，在目标机到主机需要有一个终端（通常是串口），还需要和主机具有网络连接作为 GDB 调试端口。

第三，gdbserver 需要使用交叉编译工具进行编辑，生成目标机的版本。

第四，为了节省目标机的资源，目标机运行的程序可以不包含调试信息，只需要主机的程序包含调试信息即可。即在目标机使用 gdbserver 启动的程序，可以不包含调试信息；而在主机端使用 gdb 启动的程序，应该是包含调试信息的副本。

如果使用串口连接的调试方法，则一般需要两个串口：一个串口作为目标机-主机的终端；另一个是 GDB 所需要的端口。

在目标机上，使用串口作为端口，例如：

```
# gdbserver /dev/ttyS1 sum
```

由于 ttyS0 一般作为目标机-主机的终端，因此可使用 ttyS1 作为 GDB 连接。

在主机端，使用 GDB 启动程序后，也需要指定串口远程连接：

```
(gdb) target /dev/ttyS1
```

上述命令将建立主机-目标机的基于串口的 GDB 连接。

3. gdbserver 的调试实例

本实例以 x86 的 GDB 远程调试为例，gdb 和 gdbserver 运行在一台计算机上，但是它们之间使用 TCP/IP 网络连接方式。在这种情况下，本机调试和多机调试的过程是类似的。

使用 gcc -g 将程序进行编译和连接。

```
1  #include <stdio.h>
2
3  int info(int argc,char* argv[])
4  {
5      printf("<<<< get Sum >>>>\n");
6      printf("==== begin ==== \n");
7
8      return 0;
9  }
10
11 int main (int argc,char* argv[])
12 {
13     int num1, num2, total ;
14
15     info(argc,argv);
16
17     num1 = 100;
18     printf("The first number : %d \n",num1);
19
20     num2 = 200;
21     printf("The second number : %d \n",num2);
```

```

22
23     total = num1 + num2;
24     printf("\nThe sum is : %d\n", total);
25
26     return 0;
27 }

```

使用 `gcc -g` 将程序进行编译和连接，如下所示：

```
$ gcc -g sum.c -o sum
```

在远程调试的时候，实际上只有 `gdb` 调试器需要调试信息。因此，运行于目标机的可执行程序可以运行不经过 `-g` 编译的，而在主机运行的是经过 `-g` 编译的。

后面就可以进行 `GDB` 远程调试了。在调试的时候，需要使用两个终端，一个是所谓的“目标机”，另一个是运行 `GDB` 程序的主机。

在“目标机”当中，通过 `gdbserver` 运行被调试的可执行程序，如下所示：

```

# gdbserver 127.0.0.1:6789 sum

Process exam created; pid = 14106

Listening on port 6789

```

由于本例中主机和目标机实际上是同一台机器，因此目标机需要使用本机的 IP 地址 127.0.0.1，端口选一个可以使用的即可。如果对远程的系统调试，将 127.0.0.1 改称目标机的 IP 地址即可。

在主机的控制台中，使用 `gdb` 调试被调试的程序，如下所示：

```
$ gdb sum
```

在 `GDB` 的信息显示完成后，可以使用 `GDB` 命令继续调试。首先需要使用 `target` 命令连接 `GDB Server`，如下所示：

```

(gdb) target remote 127.0.0.1:6789
Remote debugging using 127.0.0.1:6789

```

连接 `GDB Server` 之后，可以使用通用 `GDB` 命令进行调试：

```

(gdb) break main
Breakpoint 1 at 0x8048381: file sum.c, line 15.
(gdb) continue
Continuing.
Breakpoint 1, main (argc=1, argv=0xbfff9944) at sum.c:15
15     info(argc,argv);
(gdb) next
17     num1 = 100;
(gdb) next
18     printf("The first number : %d \n",num1);
(gdb) next
20     num2 = 200;
(gdb) next
21     printf("The second number : %d \n",num2);
(gdb) next
23     total = num1 + num2;
(gdb) next

```

```
24     printf("\nThe sum is : %d\n", total);
(gdb) next
26     return 0;
(gdb) next
27     }
(gdb) next
0x00b8abba in __libc_start_main () from usr/lib/libc.so.6
(gdb) next
Single stepping until exit from function __libc_start_main,
which has no line number information.
Program exited normally.
(gdb) quit
```

在以上的调试中，在主机端（gdb）使用命令可以控制目标机程序的执行。这种调试方式和直接使用 GDB 是一样的。

与使用 gdb 直接进行调试类似，gdbserver 也可以使用 attach 的方式，实现对一个已经运行的进程进行调试。

第 5 章

Linux 系统的结构

5.1 Linux 操作系统基本概念

Linux 类似于 UNIX。Linux 是免费的、源代码开放的、符合 POSIX 标准规范的操作系统。Linux 拥有现代操作系统所具有的内容，例如：真正的抢先式多任务处理、多用户、内存保护、虚拟内存、SMP（Symmetric Multi Processing，对称多处理机）、符合 POSIX 标准、TCP/IP，支持绝大多数的 32 位和 64 位 CPU。

Linux 系统运行时最基本的概念是运行的空间、文件系统和进程。

Linux 系统运行后分为内核空间和用户空间。

- 内核空间（Kernel Space）：主要由内核代码运行。
- 用户空间（User Space）：主要由用户空间代码运行，但在进行系统调用时切换到内核空间。

文件系统是 Linux 程序和数据存储主体，除了介质上的常规文件和目录，设备和通信手段也被表示成文件。

进程是 Linux 运行时的主体，进程分为内核进程和用户进程。每个进程可以分裂（fork）子进程，进程可包括若干个线程。从本质上来说，Linux 内核的进程和线程是同一种东西，用户空间看到的是一个进程。如果有的进程调用了 POSIX 线程库创建线程之后，也可以看到进程下面的线程。

5.1.1 Linux 的进程信息

进程是 Linux 程序运行的主体，进程分为内核空间的进程和用户空间的进程。

Linux 中一个进程的主要属性如下所示：

- 命令行（cmdline）。
- 名称、状态和调度情况等。
- 进程 id 和父进程 id。

- 用户 (Uid)、组 (Gid) 和所属组 (Groups)。
- 所打开的文件 (fd 和 fdinfo)。
- 内存使用情况 (mem)。
- 进程当中的线程 (task)。

Linux 系统的进程信息可以通过各种命令行获取，基本的手段如下所示：

- 使用 ps 工具查看进程的信息。
- /proc/<pid>/表示进程的信息。

在程序运行时，可以用 ps 查看其进程信息，然后根据 pid 查看 proc 文件系统中的内容。命令行中的 ps 工具用于查看进程的信息。每个进程中的几个项目如下所示。

- UID: 用户的 ID，本质是一个整数。
- PID: 进程的 ID，本质是一个整数。
- PPID: 父进程的 ID，本质是一个整数。
- CMD: 所执行的命令，本质是一个字符串。

```
$ ps -ef
UID          PID  PPID  C  STIME TTY          TIME CMD
```

PID 为 1 的进程是 init，用户空间的第一个进程；PID 为 2 的进程是 kthreadd，内核空间的第一个进程。其他进程直接或间接诞生于此二者。

proc 文件系统的/proc/<pid>/目录为每个进程的信息。其中的几个重要文件如下所示。

- cmdline: 进程启动参数。
- environ: 环境变量。
- limits: 进程限制信息。
- status: 进程状态。
- stat: 进程统计（提供数值信息，比较难于阅读）。
- mem: 进程占用的内存。
- maps: 进程使用的动态连接库文件。

另有一个名称为 task 的目录，表示子进程（线程）的信息，其中的子目录就是子进程的进程 id，每个子进程目录的结构和父进程类似。

一个用户空间的可执行程序运行后，就会形成一个进程。例如，一个循环的示例程序 HelloLoop.c 如下所示：

```
#include <stdio.h>
#include <sys/time.h>
int main(int argc, char**argv)
{
    struct timeval tv;
    int i = 0;
    for(;;){
        gettimeofday(&tv, NULL);
        printf("Loop %d: Time = %d %d \n", i,
            (int)tv.tv_sec, (int)tv.tv_usec);
        sleep(1);
        i++;
    }
}
```

```

    }
    return 0;
}

```

HelloLoop.c 源文件经过编译和程序的执行如下所示:

```

$ gcc HelloLoop.c -o HelloLoop
$ ./HelloLoop
Loop 0: Time = 1333938454 978950
Loop 1: Time = 1333938455 979051
Loop 2: Time = 1333938456 979125
Loop 3: Time = 1333938457 979201

```

由于生成的 HelloLoop 可执行程序是一个循环, 因此可以一直运行, 不会自动退出。运行这个程序后, 使用 `ps -ef` 查看系统的进程, 根据名称可以找到 HelloLoop, 那一行就是该进程的信息, 由此也可以得到其进程的 id。

进一步, 可以根据进程 id 查看 `proc` 进程目录信息, 查看命令行信息如下所示:

```

$ cat /proc/23779/cmdline
./HelloLoop

```

查看进程目录中的状态信息如下所示:

```

$ cat /proc/23779/status
Name:      HelloLoop
State:     S (sleeping)
Tgid:      23779
Pid:       23779
PPid:      23305
TracerPid: 0
Uid:1000 1000 1000 1000
Gid:1000 1000 1000 1000
FDSize:    256
Groups:    4 20 24 25 29 30 44 46 107 109 115 126 1000
VmPeak:    1720 kB
VmSize:    1620 kB
VmLck:     0 kB
VmHWM:     364 kB
VmRSS:     364 kB
VmData:    32 kB
VmStk:     88 kB
VmExe:     4 kB
VmLib:     1464 kB
VmPTE:     24 kB
Threads:   1

```

`status` 文件提供了更详细的信息, 并且可读性也较强, 几个主要的内容是: `Name` 表示进程名称; `Tgid` 表示线程组 id; `Uid` 表示运行的用户 id; `Gid` 表示运行的用户组 id; `Groups` 表示运行的用户所属于的其他组; `Vm` 表示虚拟内存的信息; `Threads` 表示进程当中所有线程的个数。

在上面的程序中, 可以看出进程的 `Pid` 和进程的目录的名称相同; `Tgid` 表示组进程 id, 由于是一个进程, 也与之相同; `PPid` 表示父进程的 ID; `Uid` 和 `Gid` 表示的是当前进程所属的用户和组, 与执行程序的命令行有关; `Groups` 当中包括了若干个整数, 表示当前程序所属的组; 由于程序没有启动额外的线程, `Threads` 的数值为 1。

提示：Gid 和 Groups 当中表示的整数都是用户组，同一数值含义相同。

在桌面的 Linux 中，通过/etc/passwd 文件可以获取用户名与 id 的对应关系。查看的一个结果如下所示：

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
hanchao:x:1000:1000:chan,,,:/home/hanchao:/bin/bash
```

第一列是名称，第三列和第四列分别表示它对应的用户 id 和组 id，例如 root 表示的就是 id 为 0 的用户和 id 为 0 的组；1000 则通常代表主用户。

提示：/etc/passwd 文件在不同的 Linux 系统中不一定存在。

一个多线程的程序 HelloThread.c 如下所示：

```
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
void *thread_body(void* param) {
    struct timeval tv;
    int i = 0;
    for(;;){
        gettimeofday(&tv,NULL);
        printf("Thread %d: Time = %d %d \n",i,
            (int)tv.tv_sec,(int)tv.tv_usec);
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
int main(int argc,char**argv)
{
    int i = 0;
    pthread_t thr;
    pthread_create(&thr, NULL,thread_body,NULL);
    for(;;){
        printf("Main Loop %d: \n",i);
        sleep(2);
        i++;
    }
    return 0;
}
```

在 HelloThread.c 程序中建立一个线程，程序运行的过程中将会有不只一个线程同时运行。对此程序进行编译并运行的过程如下所示：

```
$ gcc HelloThread.c -lpthread -o HelloThread
$ ./HelloThread
Main Loop 0:
Thread 0: Time = 1333940614 978883
Thread 1: Time = 1333940615 978987
Main Loop 1:
```

```

Thread 2: Time = 1333940616 979062
Thread 3: Time = 1333940617 979132
Main Loop 2:
Thread 4: Time = 1333940618 979206
Thread 5: Time = 1333940619 979274
Main Loop 3:
Thread 6: Time = 1333940620 979348
Thread 7: Time = 1333940621 979416
Main Loop 4:
Thread 8: Time = 1333940622 979492
Thread 9: Time = 1333940623 979559
Main Loop 5:
Thread 10: Time = 1333940624 979634

```

程序当中建立了一个线程，因此主入口当中的循环和线程中的循环将同时运行，二者分别延迟 1 秒和 2 秒，因此线程打印两次，主程序打印一次。

在 HelloThread 运行的过程中，使用 `ps` 找到其进程 `id`，然后通过 `proc` 文件系统可以进一步查看信息，如下所示：

```

$ cat /proc/24451/status
Name:      HelloThread
State:     S (sleeping)
Tgid:      24451
Pid: 24451
PPid:      23305
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize:    256
Groups:    4 20 24 25 29 30 44 46 107 109 115 126 1000
VmPeak:    10052 kB
VmSize:    10052 kB
VmLck:     0 kB
VmHWM:     528 kB
VmRSS:     528 kB
VmData:    8372 kB
VmStk:     88 kB
VmExe:     4 kB
VmLib:     1548 kB
VmPTE:     24 kB
Threads: 2

```

从中可见，`Threads` 的值为 2，表示程序当中具有两个线程。

在 Linux 用户空间中查看进程的时候，只有 `/proc/` 当中的 `id` 表示的是进程，而程序中创建的线程的本质是子进程（线程），不会包含在 `/proc/` 的子目录中，也不能通过 `ps` 命令直接列出。但是在 `/proc/<pid>/task/` 中可以列出每个子进程（线程）的信息。

上述程序的查看结果如下所示：

```

$ ls /proc/24451/task/
24451 24452

```

24451 和 24452 是两个子目录，表示 24451 进程中包括的子进程，目录结构与进程目录 `/proc/<pid>` 当中的内容相同。

查看 24452 当中的 `status` 进程信息文件如下所示：

```
$ cat /proc/24451/task/24452/status
Name:      HelloThread
State:     S (sleeping)
Tgid:      24451
Pid:       24452
PPid:      23305
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize:    256
Groups:    4 20 24 25 29 30 44 46 107 109 115 126 1000
```

此时的 Tgid (24451) 是线程组的 ID, 数值与该子进程的父进程 ID 相同, Pid (24452) 表示自己的进程 id, 而 PPid (23305) 父进程 ID 实际上是这个线程组 (24451) 的父进程 ID。

5.1.2 Linux 的文件系统和文件信息

Linux 操作系统的文件系统是一个广义的文件系统, 磁盘等物理介质可以被挂接成目录树结构, 类似于 proc、sys、temp 等虚拟的文件系统也可以表示成目录树结构。

文件是文件系统的目录树中的一个节点, Linux 当中也将很多内容都表示成文件, 其中既包括普通文件, 也包括特殊的设备文件。Linux 当中的文件包括以下几个属性:

- 文件类型。
- 用户 id (uid) 和组 id (gid)。
- 访问权限 (表示为矩阵[读|写|执行]×[本用户|组内用户|其他])。
- 修改时间。

Linux 当中的文件类型有以下 8 种。

- 常规文件 (REG, 0x10): 代表字符为"-"。
- 目录 (DIR, 0x4): 代表字符为"d"。
- 字符设备 (CHR, 0x2): 代表字符为"c"。
- 块设备 (BLK, 0x6): 代表字符为"b"。
- 管道和 FIFO (FIFO, 0x1): 代表字符为"p"。
- 连接 (LNK, 0x12): 代表字符为"l"。
- 套接字 (SOCK, 0x14): 代表字符为"s"。

使用 ls -l 命令查看文件的时候, 首字符显示为"-"、"d"和"c"等, 表示文件的类型。查看几种文件的情况如下所示:

```
$ ls -l
total 8
brw-r--r-- 1 root root 1, 1 2010-06-11 21:12 block_dev
crw-r--r-- 1 root root 1, 1 2010-06-11 21:12 char_dev
drwxr-xr-x 2 root root 4096 2010-06-11 21:11 dir
prw-r--r-- 1 root root 0 2010-06-11 21:13 fifo
-rw-r--r-- 1 root root 344 2010-06-11 21:11 hello.txt
lrwxrwxrwx 1 root root 9 2010-06-11 21:14 link -> hello.txt
```

第一列的首字母就是文件的类型, 然后是使用 9 个字母表示的文件属性, 后面的依次

是用户名和组名，然后是文件的大小或者主次设备号（不同类型的内容不同），然后是修改时间，最后一列表示文件的名称。

Linux 的文件权限使用 9 个字母表示，访问权限有读（数值为 4）、写（数值为 2）和执行（数值为 1）三种，根据本用户，组内用户和其他情况分别应用三组原则，每组原则都有三个字母。由字母（"r"、"w"或"x"）表示权限存在，"- "表示无权限。

Linux 的文件访问权限问题，以进程的为主体，以文件的为客体，根据情况分别应用三种原则。

- 第一组原则（本用户）：如果进程的 id 和文件的 id 相等，应用第一组原则，也就是首三个字母表示的权限。
- 第二组原则（组内用户）：第一组原则不满足，且进程的 gid 等于文件的 gid 或者进程 Groups 包含文件的 gid，则应用第二组原则，也就是中间三个字母表示的权限。
- 第三组原则（其他用户）：如果第一组原则和第二组原则都不满足，应用第三组原则，也就是后三个字母表示的权限。

三组原则肯定是排他的，任何一个进程对任何一个文件的访问只能应用三组原则中的一组。值得注意的是：进程的 Groups 是一个整数类型的数组，其中包含的也是组 id，因此在组用户不相等的情况下，只要 Groups 包含文件的组 id，也满足第二组原则。

表示文件访问权限几种可能数值的含义如下所示。

- `rwxr-xr-x` (755)：本用户可读可写可执行；如果不是本用户，就可读可执行。
- `rw-r--r--` (644)：本用户可读可写；如果不是本用户，就只可读。
- `rwxr----x` (741)：本用户可读可写可执行，组内用户可读，其他用户可执行。
- `rwX-W----` (720)：本用户可读可写可执行，组内用户可写，其他用户无权限。
- `r---w---x` (421)：本用户可读，组内用户可写，其他用户可执行。
- `---rw---x` (061)：本用户无权限，组内用户写读写，其他用户可执行。

提示：本用户的权限不一定高于其他用户，而在于原则的设置。

在 Linux 命令行中，可以使用 `mknod` 命令建立特殊的文件，类型 `c` 或者 `u` 表示字符设备，类型 `b` 表示块设备，此二者需要指定主次设备号，类型 `p` 表示 FIFO。使用 `mkfifo` 命令可建立 FIFO。

建立字符设备、块设备和 FIFO 的命令分别如下所示：

```
$ sudo mknod char_dev c 1 1
$ sudo mknod block_dev b 1 1
$ mknod fifo p
$ mkfifo fifo2 p
```

`chown` 和 `chgrp` 分别用于改变文件用户 id (uid) 和组 id (gid)。使用 `chmod` 命令改变文件的访问权限，值是读 (4)、写 (2)、执行 (1) 三个位相与的结果。

5.1.3 文件的另外三位属性

Linux 的文件权限除了 3×3 的权限位之外，还有三位，分别为 SUID (数值为 4)、SGID

(数值为 2) 和 Sticky (数值为 1), 这三个数值相与之后就是文件属性的高 3 位。

如果一个可执行文件设置了 SUID 和 SGID, 其运行后 `uid` 和 `egid` 就会变成其所有者的 `uid` 和 `gid`。

Sticky 位只针对目录有效, 在具有 Stick 位的目录中, 用户如果在该目录下具有 `w` 及 `x` 权限, 则当用户在该目录下建立文件或目录时, 只有文件拥有者与 `root` 才有权力删除。

Linux 进程的属性除了 `uid` 和 `gid` 之外, 还有 `uid` 和 `egid`, 分别表示有效的用户和有效的组。默认 `uid` 等于 `uid`, 而 `egid` 等于 `gid`。这种情况只有在可执行的文件具有 SUID 和 SGID 才会发生变化。

在文件系统显示的时候, 通常采用 9 个字母来表示 12 位的文件属性, 字母 "s"、"S"、"t"、"T" 出现在执行位所在位置的时候具有特殊的含义。"s" 和 "S" 可能出现在第一个和第二个执行位, "t" 和 "T" 可以出现在第三个执行位。

表示文件访问权限的几个数值的含义如下所示。

- `rws--s---` (6710): 如果运行文件, 进程 `uid` 为文件的 `uid`, 进程 `egid` 为文件的 `gid`, 本用户和组内用户可运行文件。
- `rws--x---` (4710): 如果运行文件, 进程 `uid` 为文件的 `uid`, 本用户和组内用户可运行文件。
- `rwX--s---` (2710): 如果运行文件, 进程 `egid` 为文件的 `gid`, 本用户和组内用户可运行文件。
- `rw-rwS---` (2660): 如果运行文件, 进程 `egid` 为文件的 `gid`, 本用户可运行文件。
- `rwXrwx--t` (1771): 文件的 Sticky 位被设置, 其他用户可运行文件。
- `rw-rw-rwT` (1666): 文件的 Sticky 位被设置, 其他用户不可运行文件。

实际上, 出现在第一个 "x" 位置上的 "s"="x"+有 SUID, 出现在第二个 "x" 位置上的 "s" 表示有 SGID 位, 出现在第一个 "x" 位置上的 "S"="-"+有 SUID 位, 出现在第二个 "x" 位置上的 "S"="-"+有 SGID 位。

"s" 位的典型应用场景就是用于执行超级用户的可执行程序 `su`, 它的权限通常为:

```
$ ls -l /bin/su
-rwsr-xr-x 1 root root 31100 2010-02-15 06:11 /bin/su
```

如果有用户执行 `su`, 这个进程的 `uid` 将变成 `root`, 由此可以执行一些超级用户的操作。

5.2 Linux 系统的组成和构建

5.2.1 Linux 系统的组成

Linux 系统一般来说由 3 个部分组成: BootLoader、Linux 内核、根文件系统。

BootLoader (引导加载器) 是系统的引导程序, 是整个系统运行的第一个部分。在一般情况下, BootLoader 会以二进制的形式被烧写到系统的启动地址处。当系统启动后, 首先运行 BootLoader, 在 BootLoader 的前面包含了系统的启动代码, 完成系统硬件的初始化

工作，之后进入 BootLoader 的环境。在 BootLoader 运行的情况下，用户可以根据它的功能选择进行相应的操作，Bootloader 通常都会提供下载、烧写、简单用户界面等功能。BootLoader 最基本的功能是加载 Linux 内核并运行。

Linux 内核实际上也是经过编译生成的一段可执行程序。在完整的系统中，Linux 内核会被烧写在系统的某段地址内，内核由 BootLoader 加载运行。在嵌入式的 Linux 中，Linux 内核有可能被压缩在系统的内存中，也可以直接放置在可运行的地址处。BootLoader 加载内核可以有几种不同的模式，但是最终运行 Linux 内核的形式都是跳转到 Linux 内核的起始地址运行。内核运行时，可能需要从外部获取启动参数，因此在 BootLoader 将会保存一段参数区，在 Linux 内核启动的时候，将该参数传递给内核。

根文件系统是 Linux 内核启动之后首先需要加载的文件系统，一般来说，根文件系统也需要烧写到可以固化的存储器（如嵌入式系统的 Flash 存储器当中）中，由内核挂接。本身 Linux 内核运行并不依赖于根文件系统，但是要实现一些基本的功能都需要根文件系统的支持。当根文件系统挂接完成后，内核可能会运行根文件系统中的程序。由于根文件系统中的内容是被内核加载起来的，因此根文件系统中可以包含 ELF（Executable and Linking Format，可执行链接格式）格式的程序。

5.2.2 嵌入式 Linux 的构建

在嵌入式 Linux 的构建中，BootLoader 和 Linux 内核一般都有相对成熟的代码。对这两个部分的工作主要是移植和交叉编译：第一步是根据本系统硬件平台的状况进行移植；第二步是采取交叉编译对源代码进行编译，形成运行时需要的映像文件。

对于 BootLoader，使用者需要针对本硬件平台完成移植。完成移植后，将移植部分和不需要改动的部分一起做交叉编译，生成可执行的二进制机器代码文件。不同 BootLoader 的功能不同，因此需要的硬件支持也不同，移植的部分也不一样。BootLoader 的移植结构如图 5-1 所示。

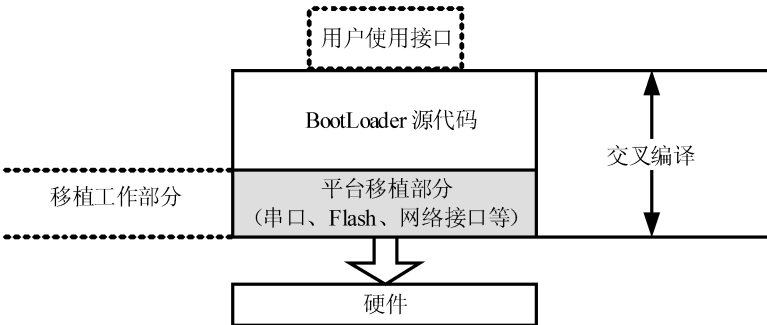


图 5-1 BootLoader 的移植

Linux 内核支持多种体系的处理器，在 Linux 的内核代码中，包括与体系结构无关和与体系结构相关两个部分。Linux 的内核代码中有对各种体系结构的支持（例如 i386、ARM、MIPS 等），但是针对本系统的平台还需要完成针对硬件的移植，主要工作包括配置（包括

本系统的内存映射等)、定时器及中断系统的移植(用以完成系统所需要的时钟),以及串口驱动(作为系统的标准输出,显示调试信息)。移植完成后,选择需要的硬件平台及相关配置,进行交叉编译形成内核映像文件。Linux 内核的移植结构如图 5-2 所示。

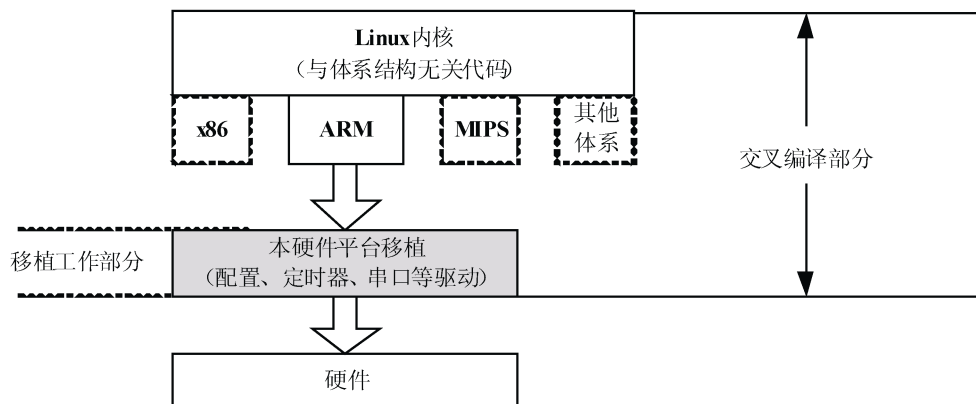


图 5-2 Linux 内核的移植

Linux 内核的启动并不依赖于根文件系统(rootfs),但是要保证 Linux 内核的正常运作,仍需要根文件系统的支持。因此,在构建系统的过程中需要为系统生成根文件系统的映像文件,让 Linux 内核在启动的时候挂接(mount)根文件系统。

5.3 Linux 系统的启动流程

Linux 系统的启动流程,可以分成三个阶段:

- BootLoader 运行阶段。
- Linux 初始化阶段。
- 系统的正常运行阶段。

在 x86 平台上的 BIOS 启动加载阶段,在 Linux 系统启动之前发生,这个过程和 Linux 系统自身并无关系。在 BIOS 运行完成之后,进入 Linux 的 BootLoader 运行。

Linux 系统的启动流程,分成三个阶段,如图 5-3 所示。

Linux 系统的启动流程,分成以下三个阶段。

第一个阶段: BootLoader 启动,初始化硬件,加载 Linux 内核,启动 Linux 内核,并传递 Linux 内核所需要的启动参数。此后 BootLoader 交出系统的控制权,以后的步骤再和 BootLoader 无关。

第二个阶段: Linux 内核启动,完成初始化工作后,加载根文件系统,之后运行根文件系统中的 init 作为第一个进程(运行于用户空间),并启动内核守护进程(kthreadd)作为第二个进程(运行于内核空间)。

第三个阶段: 系统进入正常运行状态,用户空间的各个进程由 1 号进程启动,内核空间的各个进程由 2 号进程启动。并可以由程序加载不同的文件系统以及运行不同的文

件系统中的程序，当用户空间的程序进行系统调用（system call）的时候，将切换到内核空间执行。

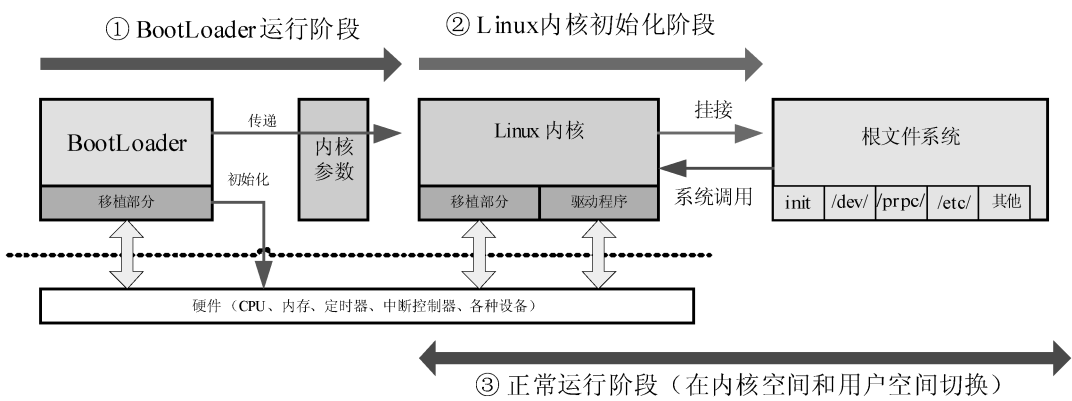


图 5-3 Linux 系统启动流程

在系统正常的情况下，已经进入第 3 个运行阶段。在这个阶段中，Linux 内核空间与用户空间程序交替在 CPU 执行代码。

第 6 章

BootLoader 及其构建

6.1 嵌入式 Linux 的 BootLoader

在桌面的 Linux 中，BootLoader 的功能比较统一。在系统启动阶段，BIOS（Basic Input/Output System）加载 BootLoader 并运行，BootLoader 加载 Linux 内核并运行，交出系统的控制权。桌面 Linux 的 BootLoader 从 LILO（Linux Loader）发展到现在的 GRUB（Grand Unified BootLoader），除了引导系统之外，通常也提供了命令行的调试功能。

在嵌入式系统中，由于不具有自举开发的能力，其 BootLoader 除了引导操作系统之外，还要担负辅助开发的责任，如与主机通信、与用户交互、更新系统的功能。虽然嵌入式系统不可能实现通用的 BootLoader，但是各系统的 BootLoader 依然具有一定的相通性。由此，嵌入式系统中常用的 BootLoader 也都具有一定的可移植性，可以在大部分代码不更改的情况下，根据本系统的情况，通过修改具体硬件相关的代码并进行相应的配置来使用。

6.1.1 BootLoader 的开发要点

BootLoader 不但是嵌入式 Linux 系统运行的必备部件，也在系统开发的过程中起着重要的辅助作用。

在嵌入式系统中，通常并没有像 BIOS 那样的固件程序，BootLoader 就是系统运行的第一个程序，负责整个系统的加载启动任务。各个系统一般将首先运行 BootLoader，然后可以对 BootLoader 进行简单的操作，主要功能是由 BootLoader 加载操作系统运行。在嵌入式系统中，BootLoader 实际上充当着两个角色，一个功能是系统的主要运行程序——操作系统的加载器，另一个是系统的辅助开发工具。

嵌入式系统的 BootLoader 的运行方式主要有两种：分别对应于嵌入式系统产品阶段和开发阶段的运行。

在嵌入式系统产品阶段的时候，BootLoader 的引导过程相对简单，一般都是将操作系统自动加载到内存，并运行操作系统。在这种情况下，是不需要用户干预的，操作系统一

般存放在可固化存储器（如 Flash）中，加载到 RAM 后，就可以跳转到操作系统运行。此时，系统的控制权已经交给了操作系统，BootLoader 的功能完成。

在嵌入式系统的开发阶段，所使用的 BootLoader 的功能较多，一般在 BootLoader 引导后，不会直接运行操作系统，而是进入人机交互界面，由开发者决定系统的行为。这时，可以执行载入内核、烧写 Flash、运行操作系统等动作。

BootLoader 不但是嵌入式系统产品必要的引导程序，也是嵌入式系统开发阶段必要的辅助手段。一般来说，在操作系统的基本功能完成后，整个系统可以利用操作系统提供的程序下载、通信、运行程序、调试程序等功能。在操作系统的基本功能实现之前，尤其在调试操作系统本身的时候，使用 BootLoader 就是必不可少的。

在嵌入式系统中，BootLoader 的功能强弱不同，但是其基本功能一般主要在以下方面实现：

- 引导系统（完成硬件的初始化）。
- 运行操作系统（加载 Linux 内核并运行）。
- Flash 烧写（更新系统 Flash 存储器的内容）。
- 通信（串口、网络、USB）。
- 人机交互界面（例如：基于命令行的交互功能）。

在 BootLoader 的各项功能中，有一些与具体硬件无关的，有一些是与硬件相关的。从嵌入式系统开发的角度，其整体开发流程如图 6-1 所示。

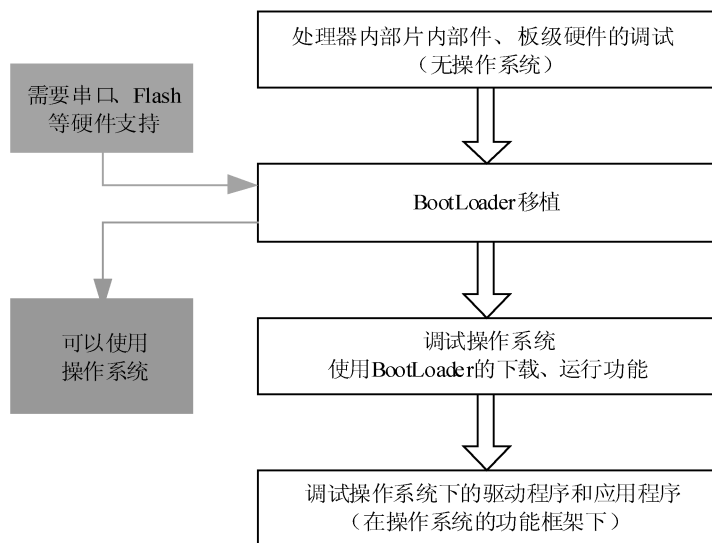


图 6-1 GDB BootLoader 在嵌入式系统开发中的作用

在嵌入式系统的软件开发中，首先需要调试的是系统的基本硬件，这时可以利用集成开发环境。当一些基本的硬件（如：内存、串口）调试完成后，就可以试图移植 BootLoader，在移植过程中，也不需要实现 BootLoader 的所有功能，一般来说具有启动代码、串口通信功能、加载操作系统功能后，BootLoader 就可以运行操作系统，这时就可以移植和调试操

作系统了。在操作系统移植完成后，就可以在操作系统的环境下调试硬件的驱动程序和应用程序。

对于 BootLoader，只需要实现在 RAM 中运行操作系统，运行操作系统的功能后，就可以调试操作系统。而在产品化的阶段，必须使用 BootLoader 或者其他烧写手段将操作系统固化到 Flash 中。因此，在开发阶段只要 BootLoader 能从 RAM 中加载和运行操作系统就可以，并不需要实现烧写功能，甚至可以没有 Flash。

在产品阶段，一般使用 Nor Flash，这是因为 Nor Flash 可以支持直接运行，即 XIP (eXecute In Place)。操作系统的内核和根文件系统一般也在 Flash 中，BootLoader 可以将它们加载到 RAM 中运行。嵌入式系统的内存结构如图 6-2 所示。

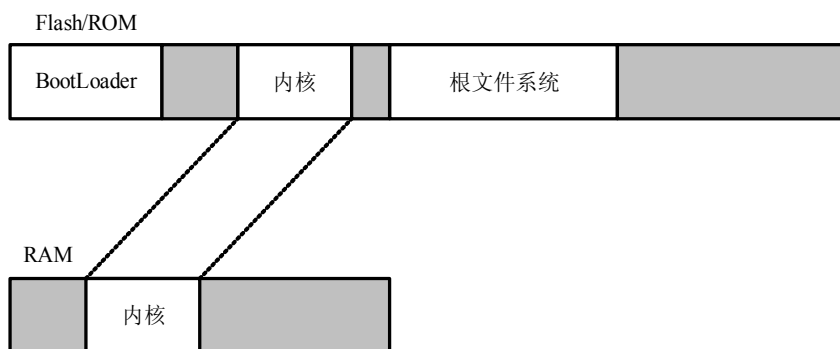


图 6-2 嵌入式系统的内存结构

BootLoader 和操作系统的内核、文件系统都存储于 Flash 中，系统上电后，首先运行 BootLoader，由 BootLoader 实现运行操作系统的功能，运行的方式一般是将操作系统的内核加载到 RAM 中，创建运行环境（一般将完成读写数据段、未初始化数据段的初始化），跳转到操作系统运行。

6.1.2 BootLoader 的结构

从结构上，BootLoader 的各项功能之间有一定的依赖关系，某些功能是与硬件相关的，某些功能是纯软件的。

一般 BootLoader 的功能框架如图 6-3 所示（与硬件相关的功能使用灰色表示）。

芯片的启动代码是 BootLoader 必备的基础，不同处理器启动需要的设置是不同的，即使相同处理器其内存空间的配置也是不一样的。因此，各个嵌入式系统的启动代码一般都是不相同的。

运行操作系统是 BootLoader 的核心功能，包括将操作系统加载到内存，开辟操作系统所需要的数据代码区域，然后跳转到操作系统的代码处运行。

在嵌入式系统中，BootLoader 运行操作系统和操作系统运行应用程序的过程有所不同：操作系统内核一般被编译成纯二进制代码，BootLoader 运行操作系统内核主要是内存加载和跳转两个步骤；操作系统运行应用程序则复杂得多，需要对程序头进行解析，然后才能够执行，如 Linux 运行 ELF 格式的应用程序。

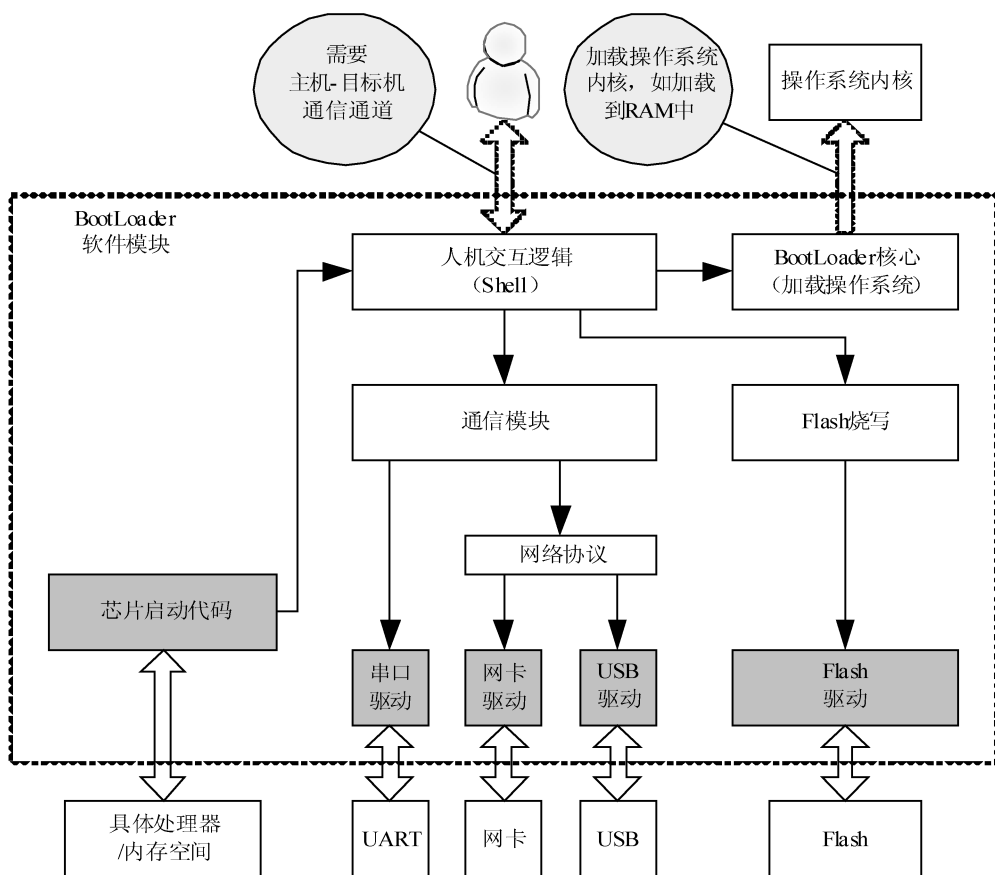


图 6-3 BootLoader 的功能框架

根据 BootLoader 的功能，可以提供支持不同操作系统的引导。在一般情况下，BootLoader 在引导操作系统的时候，是运行纯二进制操作系统映像，本质是将其加载到内存、创建运行环境和跳转运行，附加的功能还可能包括传递一些参数。对于这种引导方式，运行不同操作系统映像文件的差异不大。某些功能强大的 BootLoader 可以支持不同格式的操作系统映像（如 ELF 格式）的加载。在这种情况下，BootLoader 对操作系统的加载过程类似于 BootLoader 运行一个应用程序。

在一般 BootLoader 的功能框架中：人机交互的功能是系统的逻辑核心，它将 BootLoader 各部分的功能组织在一起，并向外部提供交互的接口。由此，用户可以通过命令行控制 BootLoader。人机交互功能本身是一个与具体硬件相关的功能，但是它一般需要建立在目标机-主机通信机制之上，如使用系统的串口（UART）。

BootLoader 的通信功能主要完成目标机到主机的通信，通信模块主要依赖的硬件机制包括串口（UART）、网络、USB 等。一般来说，串口可以实现目标机-主机控制台的功能，是实现人机交互的基础；而网络和 USB 的速度较快，可以实现较大文件的传输，它们通常建立在 TCP/IP 网络协议之上。在通信功能中，通信层接口、网络协议等功能是和硬件无关的；而串口、网络、USB 等模块的驱动是与硬件相关的，需要不同的嵌入式系统根据自身

的情况实现或者移植。

Flash 相关的功能也是 BootLoader 常见的功能之一，这是由于在嵌入式系统中 BootLoader 需要烧写自身系统，完成系统的更新。功能较为强大的 BootLoader 还可以支持 Flash 上的分区和文件系统的功能。

嵌入式 BootLoader 的核心部分是启动系统和运行操作系统内核，最简单的 BootLoader 即启动系统后，直接加载操作系统运行，这是嵌入式系统产品化阶段的主要运行方式。BootLoader 的网络通信、用户界面、Flash 烧写等功能属于附加的扩展功能，这些功能一般用于开发阶段。

6.2 U-Boot 的使用

6.2.1 U-Boot 概述

U-Boot (Universal Boot Loader, 通用引导加载程序) 是遵循 GPL 条款的开源软件项目。U-Boot 正在支持越来越多的 CPU 体系结构、开发板以及不同操作系统的引导，逐渐成为嵌入式 Linux 系统中最常用的引导加载器。

U-Boot 源代码在 sourceforge 的项目网页为 <http://sourceforge.net/projects/U-BOOT>。

U-Boot 的 FTP 地址为 <ftp://ftp.denx.de/pub/u-boot/>。

U-Boot 从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似。某些 U-Boot 源码是相应的 Linux 内核源程序的简化，这点尤其体现在一些设备的驱动程序上。

在不同操作系统的引导方面，U-Boot 不仅仅支持嵌入式 Linux 系统的引导，还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。

在对不同体系结构的支持方面，U-Boot 除了支持 PowerPC 和 ARM 之外，还支持 mips、blackfin、avr32、lib_m68k、nios、nios2 等众多常用系列的处理器。

U-Boot 项目最初是由 PPCBoot 和 ArmBoot 合并发展而来的。PPCBoot 是 PowerPC 体系结构的引导加载程序，最初是为 PowerPC-Linux 系统使用的。ArmBoot 是针对 ARM CPU 编写的通用 BootLoader，其作用是对 CPU 以及主板进行初始化，然后加载操作系统或其他 boot image (引导映像)，支持多种 ARM 开发板，并且可以很容易地移植到新的开发板上。

由此，U-Boot 对 PowerPC-Linux 和 ARM-Linux 的支持是最稳定的。尤其对于目前使用众多的 ARM-Linux 系统，U-Boot 已经真正成为这种系统“通用”的引导加载程序，实现了 U-Boot 名称中 Universal 的含义。

从嵌入式系统开发的角度，U-Boot 具有以下几项优势：

- 开放源代码。
- 支持多种操作系统。
- 支持多种 CPU 体系结构。
- 支持多种具体的开发板，可以方便移植到新的硬件平台。

- 众多的设备驱动程序。
- 具有良好的可配置性。
- 具有良好的可扩展性。

事实上，在嵌入式系统的引导中，不仅和 CPU 有关，还和处理器的片内部件以及具体的硬件平台（开发板）相关。由此，U-Boot 提供了对很多开发板的板级支持。对于相近系统的移植（例如：使用相同的处理器，但使用不同的开发板），只需要更改少量的配置代码就可以完成。

U-Boot 是一个功能强大的 BootLoader，除了可以支持众多的操作系统和 CPU 外，其功能的优势主要体现在以下两个方面：在逻辑功能上，U-Boot 强大的 Shell 界面，可以方便实现目标机-主机的交互；在硬件驱动上，U-Boot 支持众多的嵌入式系统的硬件。

U-Boot 强大的功能主要体现在以下几个方面：

- 系统引导支持 NFS 挂载、RAMDISK（压缩或非压缩）格式的根文件系统。
- 支持 NFS 挂载、从 Flash 中引导压缩或非压缩系统内核。
- 具有强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤其对 Linux 支持最好。
- 支持目标板环境参数的多种存储方式，如 Flash、NVRAM、EEPROM。
- CRC32 校验，可校验 Flash 中内核、RAMDisk 镜像文件是否完好。
- 众多的设备驱动，如：串口、SDRAM、Flash、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持。
- 上电自检功能：SDRAM、Flash 大小自动检测、SDRAM 故障检测，以及 CPU 型号检测。
- 特殊功能 XIP（eXecute In Place）内核引导。

6.2.2 U-Boot 的结构

根据嵌入式 BootLoader 的理念，它不仅需要支持嵌入式产品阶段的操作系统的引导，更需要在嵌入式系统的开发阶段提供方便的手段。U-Boot 具有这些特点，尤其是用户接口的功能很方便，并且非常容易扩展。

U-Boot 的目录具有结构清晰、模块化的特点，在其根目录下，包含了以下文件。

- Makefile: 编译的 U-Boot 根 Makefile。
- MAKEALL: 编译过程 Makefile 的辅助文件。
- *.mk: 各个体系结构的配置文件。
- README: 介绍文件。
- COPYING: GPL 文件。
- CHANGELOG: 改动记录。

其中，还有*.mk 若干个文件，分别代表了对应体系结构的编译配置选项。

U-Boot 根目录的一级子目录的情况如表 6-1 所示。

表 6-1 U-Boot 的目录结构

文件夹	性质	描述
board	开发板相关	各个开发板相关的目录文件，例如：RFXlite (mpc8xx)、ep7312 (ARM720T 的 EP7312)、smdk2410 (ARM920T 的 S3C2410)、sc520_cdp (x86) 等目录
cpu	处理器相关	处理器相关的目录文件，例如：mpc8xx、ppc4xx、arm720t、arm920t、xscale、i386 等目录
lib_XXX	体系结构相关	各个体系结构的相关文件，有几个目录，例如：lib_ppc (存放对 PowerPC 体系结构通用的文件)、lib_arm (存放对 ARM 体系结构通用的文件)、lib_i386 (存放对 X86 体系结构通用的文件) 等。其中很多代码是各体系结构的汇编代码
lib_generic	通用	通用库函数的实现
include	头文件	包含各部分的头文件，也划分为各个子文件夹，其中 configs 子目录下与各个开发板相关的配置头文件是移植过程中经常要修改的文件
common	通用	独立于处理器体系结构的通用代码，如命令行的解析等
net	通用	网络相关的代码
fs	通用	文件系统相关的代码
post	通用	上电自检程序
drivers	通用	通用的设备驱动程序，如：led 驱动、以太网接口驱动、usb 驱动、flash 驱动等，这个文件夹的内容和硬件相关，但是和具体的 CPU 体系结构以及处理器无关
disk	通用	硬盘接口程序
rtc	通用	RTC (实时时钟) 的驱动程序
dtc	通用	数字温度测量器或者传感器的驱动
examples	示例	一些独立运行的应用程序的例子，例如 helloworld
tools	工具	制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage
doc	文档	开发使用文档，包含了各种 README.* 文件

在 U-Boot 的发展过程中，其源代码结构和 Linux 内核的源代码越来越接近。在较新的 U-Boot 结构中，增加了一个表示体系结构的 arch 目录，由此 U-Boot 的代码结构与 Linux 源代码的目录结构更类似。arch 目录中分为各个体系结构的子目录，如 arm、mips 等。原来 U-Boot 的 lib<arch> 目录、cpu 目录和 include 目录中的相关内容被挪到了这个目录中。

在 tools 目录下还有一些 U-Boot 的工具。

- bmp_logo: 制作标记的位图结构体。
- envcrc: 校验 u-boot 内部嵌入的环境变量。
- gen_eth_addr: 生成以太网的 MAC 地址。
- img2srec: 转换 SREC 格式映像。
- mkimage: 转换 U-Boot 格式映像。
- updater: U-Boot 自动更新升级工具。

这些工具都有源代码，可以依次为参考改写成其他工具。其中，mkimage 是很常用的

一个工具，用于将 Linux 内核映像和 ramdisk 文件系统映像转换成 U-Boot 的格式。此外，tools 目录下还包含了 scripts 目录，这是一些常用到的脚本，脚本的命令是 U-Boot 的 Shell 界面中的命令组合。

对于 include 目标，这个 Linux 系统源代码的结构类似：inlcude 目录下的头文件是各种体系结构通用的头文件；以 asm 为前缀的各个文件夹，包含了各个体系结构的头文件；在 config 文件夹中，包含各个开发板不同的配置文件。

6.2.3 U-Boot 的生成

U-Boot 的编译过程分为两步。

第一步用于配置编译选项：

```
$ make <board> CROSS_COMPILE=<prefix>
```

<board>参数用于使用一个预定义支持的板的名称。

第二步开始编译：

```
$ make CROSS_COMPILE=<prefix>
```

CROSS_COMPILE 用于指定交叉工具链的前缀，<prefix>的值可以是 arm-linux- 等形式，也可以包含工具全路径。

连接的结果 u-boot 是一个 ELF 格式的文件，在编译的最后阶段还将通过二进制转换工具生成 S-Record 格式的文件和纯二进制文件。

```
<prefix>-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec
<prefix>-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

include/config.h 不是源代码包中包含的，而是在配置-编译过程中自动生成的，其内容通常如下所示：

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

config.h 中包含了某个板子的配置头文件。而在 U-Boot 主要的源代码中，都可能包含 config.h，使用这个文件完成配置和条件编译的功能。在这个文件中，需要定义 U-Boot 所需要的宏和编译选项。为了实现 U-Boot 的可配置性，在 U-Boot 的源代码中，并没有这个文件，而是需要通过各个开发板根据自身的情况自动生成。生成的方式很简单，即包含各个开发板的头文件，这些头文件在/include/configs 文件夹中，实际上是通过 config.h 作为 U-Boot 源代码和各个开发板的配置文件的中间层。

U-Boot 经过构建之后，将生成以下几个文件。

- system.map: U-Boot 映像的符号表。
- u-boot: U-Boot 映像的 ELF 格式。
- u-boot.bin: U-Boot 映像原始的二进制格式。
- u-boot.srec: U-Boot 映像的 S-Record 格式。

U-Boot 构建的过程为：首先使用交叉编译工具将各个目录下的源文件生成目标文件

(*o)，目标文件生成后，会将若干个目标文件组合生成静态库文件 (*.a)，最后通过连接各个静态库文件生成 ELF 格式的可执行文件。

在连接的过程中，需要根据连接脚本文件（一般是各个以 `lds` 为后缀的文本文件）确定目标文件的各个段，连接脚本文件通常是 `board/<board>/目录中的 u-boot.lds` 文件。在连接的过程中，会包含各个静态库 (*.a) 文件。

连接的结果 `u-boot` 是一个 ELF 格式的文件，编译的最后，还将通过二进制转换工具 `objcopy` 生成 S-Record 格式的文件和纯二进制文件。

`u-boot.bin` 即最终的二进制文件。根据嵌入式系统的运行原则，一般系统在上电后首先运行的程序就是 `BootLoader`，因此 `BootLoader` 在嵌入式系统中使用的一般都是二进制的文件。

如上所述，如果在嵌入式系统中使用 `U-Boot`，一般使用纯二进制的文件 `u-boot.bin`。`U-Boot` 也同时支持从网络启动方式和 `Flash` 启动方式：前者的工作原理和无盘工作站类似，自动通过网络下载操作系统内核映像和文件系统映像；后者则属于本地启动的方式，一般使用支持直接运行的 `Nor Flash`。

6.2.4 U-Boot 的启动流程

从大的方面来说，`U-Boot` 等支持 C 语言的 `BootLoader` 的启动分成两个阶段。

第一阶段：本部分程序比较简单，主要的职责是准备初始化的环境。`BootLoader` 的汇编代码从 `ROM` 或者 `Nor Flash` 等可运行的存储器中直接执行，此时的存储器中机器代码被依次取出执行，此时系统还没有进入 C 语言的运行阶段，并没有堆 (`heap`) 和栈 (`stack`)。也就不需要额外的 `RAM`。在准备好环境之后，才能进入第二阶段。

第二阶段：进入 C 语言的运行阶段。在上一阶段 `BootLoader` 通常已经将代码段复制到 `RAM` 当中，并且在 `RAM` 准备好了内存资源，C 语言程序中的内容由此才可以运行。

之所以将 `BootLoader` 的启动分成两个阶段，主要是由于 C 语言的运行需要较为复杂的环境，而这个环境必须由 `BootLoader` 自己准备。

对于 `U-Boot`，它开始运行之后首先进入汇编文件 `start.s` 进行启动、复位，然后执行板级别的启动（对于 `ARM` 平台就是 `start_armboot()` 函数），根据系统初始化函数指针列表，进入 `main_loop` 循环。

针对 `ARM` 平台的 `U-Boot` 的启动流程如图 6-4 所示。

`ARM` 平台的一个连接脚本文件 `u-boot.lds` 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 = .;
}

```

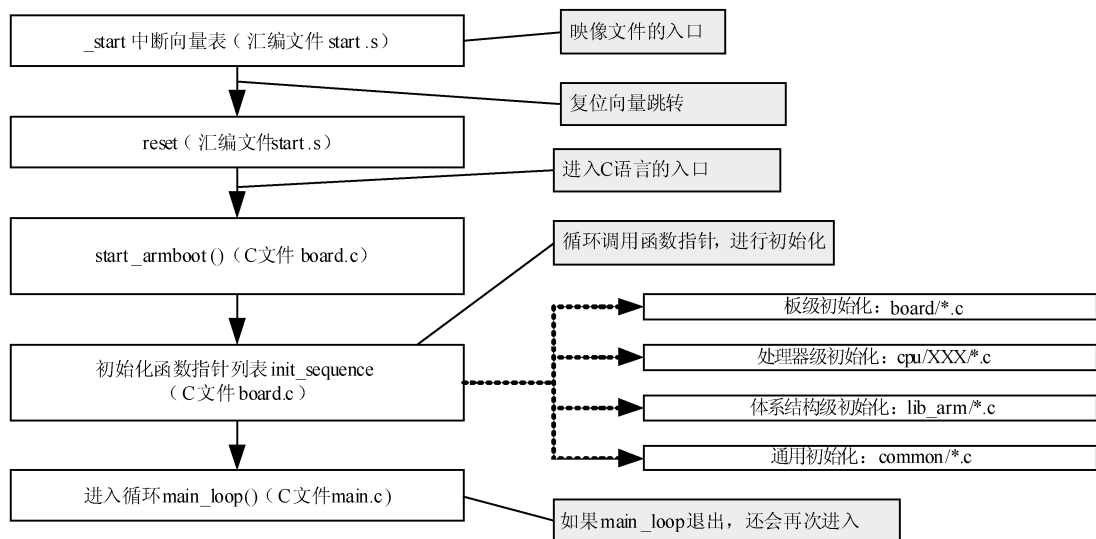


图 6-4 ARM 平台的 U-Boot 的启动流程

在本连接脚本文件中，定义了起始地址为 0x00000000，每个段使用 4 字节对齐 ALIGN(4)。几个段分别为文本段 (.text)、只读数据段 (.rodata)、数据段 (.data)、全局变量表段 (.got) 和比较特殊的命令段 (.u_boot_cmd)。其中，文本段的第一个目标为：cpu/arm920t/start.o，在其中定义映像文件的入口。

在运行 U-Boot 时，根据连接脚本 u-boot.lds，start.o 连接在头部，ENTRY(_start)指定入口也在 start.o 之中。因此整个程序启动后，首先运行的是 start.S 文件。在 start.S 中会进入 lib_arm/board.c 中，执行 start_armboot()函数。

在执行完初始化工作后，将进入 common 目录下 main.c 文件中的 main_loop()函数。由于使用了 for (;;){}，将进入 U-Boot 的命令行循环，等待用户输入参数来完成和系统交互的工作。main_loop()函数返回的时候会重新运行。

6.3 U-Boot 的命令

6.3.1 U-Boot 命令概述

在开发阶段，需要经常使用 U-Boot 的各种命令，这样可以让开发者使用交互式的方式和嵌入式系统交互。

在 U-Boot 启动后，将出现命令的提示符，用户可以在提示符下输入命令：

```
=>
```

使用 **help** 命令可以查看单个命令的详细信息，其格式为：

```
help <command>
```

例如，显示 **echo** 命令的帮助信息：

```
=> help echo
echo [args...]
echo args to console; \c suppresses newline
```

U-Boot 的常用命令如表 6-2 所示。

表 6-2 U-Boot 的命令

命令	种类	描述
? /help	信息命令	打印在线帮助信息
autoscr	运行命令	在内存中运行脚本
base	内存命令	打印或者设置地址偏移量
bdinfo	信息命令	打印开发板信息结构
boot	运行控制命令	默认运行
bootd	运行控制命令	默认运行
bootelf	运行控制命令	运行内存中的 ELF 映像
bootm	运行控制命令	运行内存中的应用程序映像，例如： 从某地址引导 Linux 内核：=> bootm 40020000
bootp	运行控制命令/网络命令	使用 BOOTP/TFTP 通过网络运行映像
bootvx	运行控制命令	从 ELF 中运行 vxWorks
cmp	内存命令	内存比较
coninfo	信息命令	打印控制台设备信息
cp	内存命令	内存复制
crc32	内存命令	校验命令
date	杂项命令	显示或设置时间
echo	杂项命令	显示控制台参数
erase	文件系统命令	擦除 Flash
flinfo	信息命令	打印 Flash 内存信息
go	运行控制命令	从地址处运行应用
icrc32	I2C 命令	i2c 校验

续表

命令	种类	描述
iprobe	I2C 命令	探测发现 i2c 设备有效地址
iloop	内存命令	无限内存循环
imd	I2C 命令	i2c 内存显示
iminfo	信息命令	打印应用映像头信息
imls	信息命令	列出 Flash 中的映像
imm	I2C 命令	i2c 内存修改（自动增量地址）
imw	I2C 命令	i2c 内存写
inm	I2C 命令	i2c 内存修改（固定地址）
loadb	网络命令	连续载入 S-Record 文件
loads	网络命令	连续载入二进制文件
loop	内存命令	地址范围内无限循环
md	内存命令	内存显示
mm	内存命令	内存修改（自动增量）
nm	内存命令	内存修改（固定地址）
mtest	内存命令	简单 RAM 测试
mw	内存命令	内存写——填充
nfs	网络命令	使用 NFS 协议从网络引导映像
printenv	环境变量命令	打印环境变量，例如： 打印所有环境变量：=> printenv 打印某个环境变量：=> printenv bootargs
protect	内存命令	使能或禁止 Flash 写保护
rarpboot	网络命令	通过 RARP/TFTP 协议从网络引导映像
reset	杂项命令	执行 CPU 复位
run	环境变量命令	在一个环境变量中运行命令
saveenv	环境变量命令	保存环境变量
setenv	环境变量命令	设置环境变量，例如： 设置 IP 地址：=>setenv ipaddr 192.168.1.1 设置 TFTP 服务器地址：=>setenv serverip 192.168.1.254
sleep	杂项命令	延迟一段时间执行
tftpboot	网络命令	通过 TFTP 协议从网络引导映像
version	杂项命令	显示版本信息

printenv、setenv、saveenv 等命令用于操作 U-Boot 的环境变量，环境变量存储之后可以保存在 Nand Flash 等存储器上，启动之后依然有效。

U-Boot 的环境变量如表 6-3 所示。

表 6-3 U-Boot 的环境变量

环境变量	描述
bootdelay	执行自动启动的等候时间

续表

环境变量	描述
bootcmd	只有在使能 CONFIG_BOOTDELAY 的时候有效，定义自动执行的命令字符串，这将在复位后引导延迟（Boot Delay）之后执行
bootargs	引导 RTOS（实时操作系统）的参数
bootfile	使用 TFTP 时引导文件的名字
bootargs	传递给 Linux 内核的命令行参数
baudrate	串口控制台的波特率
loadaddr	使用"bootp"、"rarpboot"、"tftpboot"、"loadb"或"diskboot"等命令时默认的加载地址
ipaddr	本地的 IP 地址
netmask	以太网接口的掩码
ethaddr	以太网接口的 MAC 地址
stdin	标准输入设备，一般是串口
stdout	标准输出设备，一般是串口
stderr	标准出错信息输出设备，一般是串口

U-Boot 除了可以支持各种命令外，还可以支持更复杂的命令行（command line），这种命令行类似 Linux 下的 Shell，可以使用环境变量和脚本。U-Boot 支持两种命令行解析器，一种是旧的简单命令行解析器，另一种是强大的 Hush Shell 解析器。

旧的简单命令行解析器：

- 支持环境变量（通过使用 setenv / saveenv 命令）。
- 在一行中使用几个命令，通过\来分隔。
- 特殊的字符，（例如：'\$', ';'），需要通过前缀\书写，例如：

```
setenv bootcmd bootm \${address}
```

可以将文本放入单引号，例如：

```
setenv addip 'setenv bootargs $bootargs ip=$ipaddr:$serverip:$gatewayip:$netmask:
$hostname::off'
```

Hush Shell 与 Bourne Shell 类似，支持控制结构，例如：

```
if...then...else...fi, for...do...done; while...do...done,
until...do...done, ...
```

其中支持全局变量（通过 setenv/saveenv 命令）和局部变量（通过 Shell 语法，类似"name=value"），只有环境变量可以使用 run 命令。

6.3.2 增加命令

在 U-Boot 增加新的命令是很容易的，只需要按照格式实现函数，并完成相应的命令行解析功能即可，而不需要更改 U-Boot 的核心代码。这种实现方式与 U-Boot 映像文件的数据结构有关。

在公共的头文件 include/command.h 中定义命令相关的数据结构：

```

struct cmd_tbl_s {
    char      *name;          // 命令名称
    int       maxargs;        // 最大的参数个数
    int       repeatable;     // 是否允许自动重复
    int       (*cmd)(struct cmd_tbl_s *, int, int, char *[]); // 实现函数
    char *usage;              // 使用信息      (短)
#ifdef CFG_LONGHELP
    char *help;               // 帮助信息
#endif
#ifdef CONFIG_AUTO_COMPLETE
    int (*complete)(int argc, char *argv[],                // 自动补全参数
                    char last_char, int maxv, char *cmdv[]);
#endif
};
typedef struct cmd_tbl_s cmd_tbl_t;

```

对于 U-Boot 的一个命令，需要实现的就是一个数据结构 `struct cmd_tbl_s`。这个数据结构最主要的成员是命令的名称（`name`）和实现函数的指针（`cmd`）：命令的名称是在 U-Boot 的命令行中键入的内容，U-Boot 使用一定机制根据命令行输入的内容执行相应的功能，即执行 `cmd` 函数。在 `cmd` 函数中，需要完成命令行参数的解析。

在 `struct cmd_tbl_s` 中，包含的成员还包括：最大的参数个数 `maxargs`；是否允许自动重复 `repeatable`；使用信息(短)`usage`；帮助信息(长)`help`；自动补全参数的函数指针 `complete`。其中使用信息是使用 `help` 命令的时候，该命令显示的帮助信息；帮助信息与之类似，但是包括了更丰富的内容，受到条件编译宏 `CFG_LONGHELP` 的控制；自动补全参数函数指针实现了命令行中的自动补全功能，受到条件编译宏 `CONFIG_AUTO_COMPLETE` 的控制。

利用数据结构 `struct cmd_tbl_s`，在增加实现 U-Boot 命令的过程中，可以实现命令增加的模块化，也就是说增加一个命令并不需要更改其他的数据结构。

在实际的使用中，U-Boot 提供了一个方便的宏定义 `U_BOOT_CMD`，这也在 `include/command.h` 中定义：

```

#define Struct_Section __attribute__((unused,section(".u_boot_cmd")))
#ifdef CFG_LONGHELP //使用长帮助信息
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage, help}
#else // 不使用长帮助信息
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage}
#endif // CFG_LONGHELP

```

此处使用宏 `CFG_LONGHELP` 分两种情况定义了宏 `U_BOOT_CMD`，它们之间的差别仅仅在于是否使用长帮助信息。在使用的过程中，使用一次宏 `U_BOOT_CMD` 即相当于定义了一个 `struct cmd_tbl_s` 结构体类型的实例，其中将包含结构体需要的 6 个参数。

在使用宏 `U_BOOT_CMD` 的时候，将 `__u_boot_cmd_` 与参数 `name` 连接形成结构体的名称，由此，结构体的名称为 `__u_boot_cmd_(name)`。例如：如果 `name` 为 `version`，那么结构体的名称就是 `__u_boot_cmd_version`。结构体的成员由参数 `#name`、`maxargs`、`rep`、`cmd`、`usage` 指定，其中 `name` 参数为字符串。利用这种方式可以实现结构体的自动生成。在 GCC 的编译系统中，可以使用 `#` 表示替代当前的字符串，`##` 是 C 语言中的连接操作符，可以实现预

备处理阶段的连接。

使用宏 `U_BOOT_CMD` 生成的结构体有一个特点，它是放在一个特殊的段中，这通过 `Struct_Section` 定义：

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

即使用宏 `U_BOOT_CMD` 生成的结构体都在命令段（`.u_boot_cmd`）中，注意：如果不指定段，这个段将在读写数据段（`.data`）中。事实上，在连接脚本中将指定的所有命令段（`.u_boot_cmd`）连接在一起，如下所示：

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

在命令的增加过程中，可以参考 U-Boot 内置的命令。例如：`version` 命令使用如下方式实现：

```
int do_version (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
    extern char version_string[];
    printf ("\n%s\n", version_string);
    return 0;
}

U_BOOT_CMD(
    version, 1,          1,  do_version,
    "version - print monitor version\n",
    NULL
);
```

在 `version` 命令的结构体中，`version` 是一个字符串，即在 U-Boot 命令行使用的命令名；第一个 1 表示该命令的最大参数是一个；第二个 1 表示允许自动重复；`do_version` 是函数指针，也是本命令的执行函数；字符串“`version……`”是一个本命令的使用方法；`NULL` 表示本命令没有长帮助信息。

`do_version()` 是 `version` 命令的执行函数，即当用户在 U-Boot 的命令行中输入“`version`”的时候，U-Boot 将解析执行这个函数，在这个函数的参数中，同时可以得到本命令 `cmd_tbl_t` 结构体的指针 `cmdtp`、命令标志 `flag`、命令的参数个数 `argc`、命令参数向量 `argv`。本命令函数只是简单地打印信息，没有做其他操作，因此没有使用参数。

对于 U-Boot 命令，`argc` 和 `argv` 的含义与 C 语言的主函数 `main()` 中的含义是一样的。

另外一个稍微复杂的命令 `echo`，使用如下的方式实现：

```
int do_echo (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
    int i, putnl = 1;
    for (i = 1; i < argc; i++) {
        char *p = argv[i], c;
        if (i > 1)
            putc(' ');
        while ((c = *p++) != '\0') {
            if (c == '\\\ ' && *p == 'c') {
                putnl = 0;
            }
        }
    }
}
```

```

        p++;
    } else {
        putc(c);
    }
}
}
if (putnl)
    putc('\n');
return 0;
}
U_BOOT_CMD(
    echo,      CFG_MAXARGS, 1,  do_echo,
    "echo      - echo args to console\n",
    "[args..]\n"
    "    - echo args to console; \\c suppresses newline\n"
);

```

在 echo 命令的结构体中，echo 是一个字符串，即在 U-Boot 命令行使用的命令名；第一个 CFG_MAXARGS 是一个宏，表示该命令的最大参数是在别的地方定义；第二个 1 表示允许自动重复；do_echo 是函数指针，也是本命令的执行函数；字符串 "echo " 是一个本命令的使用方法；" [args..]..... " 表示是本命令的长帮助信息，注意，"[args..]\n" 后面没有逗号，它与下面一行 " - echo args" 两个字符串连接起来，表示一个字符串。

do_echo() 是 echo 命令的执行函数，即当用户在 U-Boot 的命令行中输入 "echo" 的时候，U-Boot 将解析执行这个函数。echo 命令可以接收若干个参数，do_echo() 函数根据参数个数 argc 和参数向量 argv 解析执行。

由于 U-Boot 的命令解析程序，将从固定命令段 (.u_boot_cmd) 中解析命令。因此，对于实现一个命令，只需要使用 U_BOOT_CMD 宏构造一组结构即可，不需要再更改其他代码。由于命令的执行名称和命令的执行函数只在一个 U_BOOT_CMD 宏中有对应关系，也不必担心由于粗心引起的不匹配问题。

6.4 U-Boot 的移植

从整体上看，U-Boot 是一个层次式结构。对于某一个特定平台而言，其源代码是 U-Boot 代码的一部分，一般将包含以下几个部分。

- 与硬件无关的通用代码：这部分代码一般在 common 文件夹中，这些代码和具体的硬件平台无关，但是可以根据平台的情况采用条件编译的方式进行裁减。这种方式为 U-Boot 提供了很强的可配制性。
- 硬件驱动程序：这部分代码一般在 driver 文件夹中，这些硬件一般和具体的处理器关系不大。虽然嵌入式的硬件平台（开发板）很多，但是它们的硬件系统无非是各种硬件的排列组合。因此 driver 文件夹包含各种类型的硬件驱动，各个平台可以根据自身的情况进行选择 and 配置。
- CPU 源文件：根据 lib_xxx 等各个文件夹进行配置。
- 开发板源文件：board 文件夹中的各种文件。
- 包含头文件：在 include 文件夹中，包括通用头文件、开发板相关头文件、开发板配

置头文件。

6.4.1 U-Boot 的移植概述

根据嵌入式系统 BootLoader 移植的特点, U-Boot 与硬件移植相关的目录结构也包含了以下三个层次。

- 第一个层次是对应不同体系结构的几个以 lib 开头的目录, 如: PowerPC、ARM、MIPS、x86 等。
- 第二个层次是对处理器的支持, 在 cpu 文件夹中的各个子目录, 如: mcf52x2(Freescale ColdFire MCF52x2 处理器)、ppc4xx (AMCC PowerPC 4xx 处理器)、arm720t (ARM720T 内核的处理器)、arm920t (ARM920T 内核的处理器)、i386 (i386 兼容处理器)、pxa (Intel XScale PXA 处理器)。
- 第三个层次是对不同开发板的支持, 在 board 文件夹中的各个子目录, 如: RPXlite (mpc8xx)、dave/B2(ARM7TDMI 的 S3C44B0X)、ep7312 (ARM720T 的 EP7312)、smdk2410 (ARM920T 的 S3C2410)、sc520_cdp (x86) 等。

根据移植的三个层次, 第一个层次是和汇编代码相关的, 各个体系结构包含了不同的汇编代码; 第二个层次是对处理器的移植, 一般不包含汇编代码, 只是各个处理器具有不同的外设; 第三个层次是对不同开发板的移植。

6.4.2 U-Boot 的扩展

1. 增加开发板的支持

在嵌入式系统的开发中, 将一个已有 BootLoader 移植到自己的系统中, 针对开发板的扩展是最常用的使用方法。因此, 增加 U-Boot 对一个新的开发板的支持, 是开发中最常见的任务。顶层目录下的 Makefile 负责 U-Boot 整体配置编译。按照配置的顺序阅读其中关键的几行。每一种开发板在 Makefile 都需要有板子配置的定义。

对于这项任务, 需要做的事情包括: 更改顶层目录的 Makefile, 增加 config 文件, 增加目标开发板文件夹。

首先需要更改根目录中的 Makefile, 增加处理器的编译命令, 格式如下所示:

```
$ (配置名称)      :      unconfig
                   @$(MKCONFIG) $(@:_config=) $(体系结构) $(处理器) $(开发板)
```

编译过程在具体开发板的目录中实现, 其中包含了如下内容。

- **configs/XXX.h:** 这是相关开发板的配置文件, 将在编译过程中被自动生成的 config.h 文件包含。
- 在 board/目录下增加这个开发板的文件夹, 和 board/目录下其他的文件夹是平级关系。
- 在 board 目录中, 需要包含 u-boot.lds 和一些 c 文件。

需要实现的功能包括开发板初始化函数和动态内存初始化函数:

```
int board_init (void)
```

```

{
    // .....开发板初始化
}
int dram_init (void)
{
    // .....动态内存初始化
}

```

除了以上的 C 语言文件外，还需要包含一个汇编语言的源文件：`lowlevel_init.S`，在这个文件中需要实现低级初始化函数：

```
lowlevel_init
```

在增加一个新的开发板支持的过程中，最好的方法是选择一个类似的开发板作为模板，复制它的文件和目录，在相应的地方进行修改。由于嵌入式系统处理器的集成度较高，因此对于同一个处理器，不同的开发板之间的差别不会很大。在移植的过程中，主要需要考虑内存地址空间等问题。

2. 增加处理器的支持

在 U-Boot 中增加新的处理器，主要的工作是在 `cpu` 中增加一个处理器相关目录，在这个目录项中应包含 `start.S`、`cpu.c`、`interrupts.c` 和 `serial.c` 等文件。

主要需要实现的内容包括以下几个方面：

- CPU 的初始化 (`cpu_init`)。
- 中断的初始化 (`interrupt_init`)。
- 串口相关函数 (`serial_XXX`)。

`serial.c` 中的函数将被 U-Boot 的通用代码调用，这个文件中实现的功能实际上是 U-Boot 的抽象层。`start.S` 文件是一个汇编语言写的，它将是整个 BootLoader 程序的入口。

`cpu.c` 文件的主要内容如下所示：

```

int cpu_init (void)
{
    // CPU 的初始化 .....
}

```

`interrupts.c` 文件的主要内容如下所示：

本文件中必须实现的是中断的初始化函数，这个函数将在系统的初始化阶段被调用。

```

int interrupt_init (void)
{
    // 中断初始化 .....
}

```

`serial.c` 文件主要包括了以下内容：

- 嵌入式的 BootLoader 通常使用串口作为输入/输出端口，在这个文件中实现了输入/输出的功能，主要需要实现串口的初始化以及字符的发送和接收等几个函数。
- `serial.c` 中实现的函数将被 U-Boot 的通用代码调用，因此其中实现的功能实际上是 U-Boot 的抽象层。

在处理器级别的移植过程中，以上几个函数是必需的。无论系统是否需要这些功能，

这些函数都要实现（即具有函数体，但是可以为空）。事实上，在 U-Boot 的其他代码中，这些函数将被直接或者间接地调用，因此如果没有实现将会发生连接错误。

6.4.3 板级支持

1. B2 板（ARM7 处理器）

三星公司 S3C44B0X 处理器的 B2 开发板是一个基于 ARM7 体系结构的简单系统，不支持 MMU。U-Boot 对它的支持包括以下几个部分。

- common/：通用目录。
- driver/：驱动目录。
- arch/arm/lib：ARM 体系结构目录。
- arch/arm/cpu/s3c44b0/：处理器目录。
- board/dave/B2/：开发板目录。
- include/：通用头文件。
- include/asm-arm/arch-s3c44b0/：头文件。
- include/configs/B2.h：开发板配置头文件。

B2.h 是 B2 开发板配置的头文件，其中主要是一些对 U-Boot 代码通用的部分进行条件编译的内容。其中配置了一些条件编译的宏：

```
#define CONFIG_ARM7          1    // ARM7 CPU
#define CONFIG_B2            1    // B2 开发板
#define CONFIG_ARM_THUMB     1    // CPU 内核 ARM7TDMI
#define CONFIG_ARM7_REV D    // 禁止 ARM720 REV.D
#define CONFIG_S3C44B0_CLOCK_SPEED 75 // S3C44B0 的频率为 75MHz
#define CONFIG_USE_IRQ       // 不使用 IRQ
```

include/configs/目录下的各个配置文件，经过编译后，将自动包含在 config.h 文件中，再由 U-Boot 各部分源代码包含，这样即可以起到使用宏配置 U-Boot 源代码的目的。

以 S3C44B0X 处理器的 B2 开发板为例，其编译的命令在根目录的 Makefile 中指定：

```
#####
## S3C44B0 Systems
#####
B2_config      :    unconfig
                  @$(MKCONFIG) $(@:_config=) arm s3c44b0 B2 dave
```

第一个选项为 arm，指定了体系结构为 ARM；第二个选项为 s3c44b0，表示处理器使用 cpu 目录下的 s3c44b0 目录；第三个、第四个选项为 B2 dave，它们表示开发板使用 board 目录下的 dave/B2 目录。

2. S3C64xx 板（ARM11 处理器）

S3C64xx 是一个三星公司的 ARM11 处理器，有 S3C6400 和 S3C6410 等子型号，其参考平台包括 SMDK6400 和 SMDK6410。

U-Boot 对该平台支持的主要目录和内容如下所示。

- board/samsung/smdk6400/：板级支持目录，u-boot-nand.lds 是 Nand 的连接脚本。

- arch/arm/cpu/arm1176/s3c64xx: CPU 的支持目录。
- include/asm-arm/: ARM 体系结构的头文件。
- include/configs/smdk6400.h: 配置头文件。

在顶级 Makefile 中，配置的相关内容如下所示：

```
smdk6400_config : unconfig
@mkdir -p $(obj)include $(obj)board/samsung/smdk6400
@mkdir -p $(obj)nand_spl/board/samsung/smdk6400
@echo "#define CONFIG_NAND_U_BOOT" > $(obj)include/config.hv \
echo "RAM_TEXT = 0xc7e00000" \
>> $(obj)board/samsung/smdk6400/config.tmp; \
$(MKCONFIG) $(@:_noUSB_config=) \
arm arm1176 smdk6400 samsung s3c64xx; \
@echo "CONFIG_NAND_U_BOOT = y" >> $(obj)include/config.mk
```

在 smdk6400.h 配置头文件中，包括了如下的内容：

```
#ifndef __CONFIG_H
#define __CONFIG_H
#define CONFIG_S3C6400 1 /* in a SAMSUNG S3C6400 SoC */
#define CONFIG_S3C64XX 1 /* in a SAMSUNG S3C64XX Family */
#define CONFIG_SMDK6400 1 /* on a SAMSUNG SMDK6400 Board */
#define CONFIG_SKIP_RELOCATE_UBOOT
```

S3C64xx 处理器中包括一个内部的只读存储器 iROM，系统上电之后，首先运行的代码是 iROM 中的固化代码，iROM 将会从 SD/MMC、OneNand 或者 Nand 等几种介质中查找 BootLoader，然后再运行 BootLoader，这样的典型好处就是可以让系统自动支持多种启动介质，并可以根据优先级寻找启动方式。

S3C6410 处理器与启动相关的结构如图 6-5 所示。

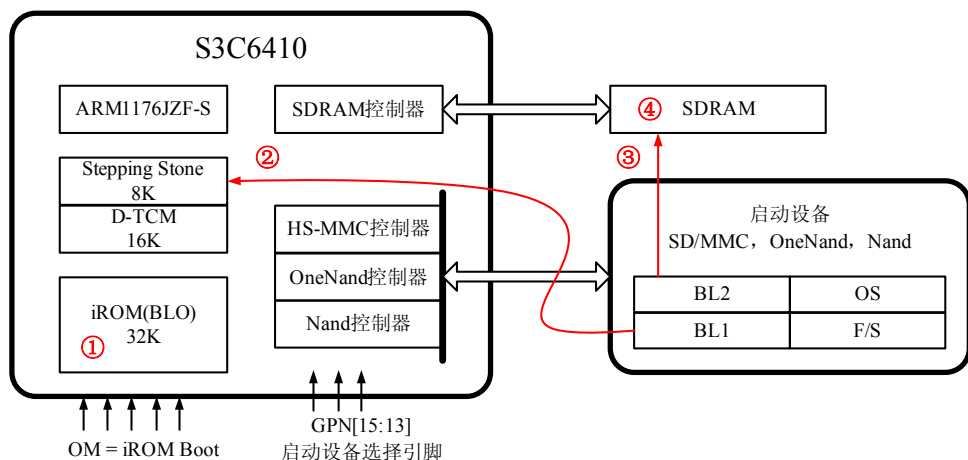


图 6-5 S3C6410 处理器与启动相关的结构

从图 6-5 中可见，S3C64xx 的启动分成了四个步骤。

- 第一步：iROM 当中的固化代码运行，根据硬件的配置等因素进行存储介质的选择，确定介质之后将 BootLoader 的第一阶段复制到内部的 Stepping Stone 存储器中。

- 第二步: BootLoader 的第一阶段在 Stepping Stone 存储器中运行。
- 第三步: BootLoader 的第一阶段将 BootLoader 的第二阶段复制到 SDRAM 中。
- 第四步: BootLoader 的第二阶段在 SDRAM 存储器中运行。

iROM 当中的代码是处理器固化的内部代码, S3C64xx 启动的第一步就是其中的一段核心的执行逻辑, 如图 6-6 所示。

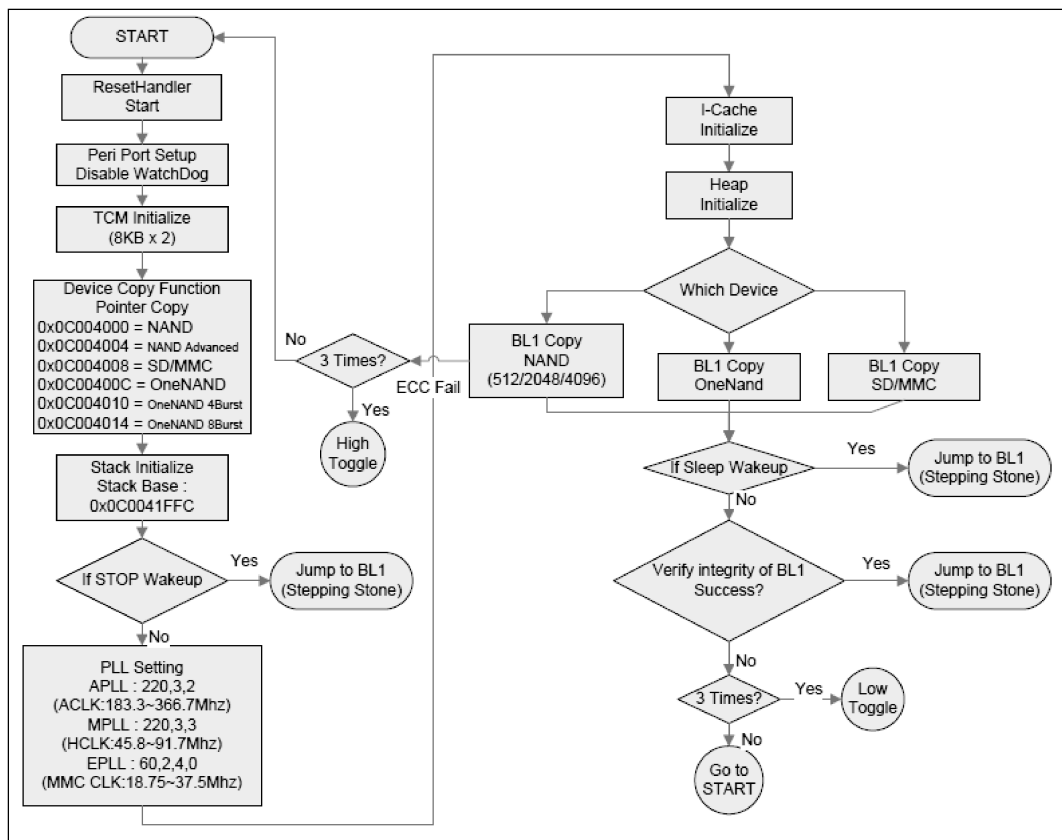


图 6-6 S3C64xx 的核心执行逻辑

iROM 当中的代码是首先运行的, 它需要进行系统的初始化, 其中的一个特定逻辑就是判断 BootLoader 所在的存储器, 其中考虑了硬件引脚的配置因素。它的执行以跳转到 Stepping Stone 存储器中的 BootLoader 运行第一个阶段为目的。考虑到存储器的坏块情况, 它可以进行多次尝试。

S3C64xx 系统 BootLoader 的第一个阶段将完成后续的工作。由于某些硬件已经被初始化过, 因此 BootLoader 的第一阶段不需要再做重复处理。进入_start()复位之后, S3C64xx 将初始化 ARM11 处理器 MMU, 然后是 UART、内存等部分。之后还需要判断启动的设备, 找到启动设备之后, 将 BootLoader 的第一个阶段加载到 RAM 中运行。

S3C64xx 系统 BootLoader 的第二个阶段的 C 代码根据需要将 Flash、SD/MMC 以及网络等硬件初始化, 然后进入循环。

第 7 章

Linux 内核及其构建

7.1 Linux 内核概述

7.1.1 Linux 内核结构

Linux 是一个比较庞大的操作系统，从操作系统的角度 Linux 内核属于宏内核（Monolithic Kernel），而非微内核（Micro Kernel）。

Linux 内核的网站为：<http://www.kernel.org/>。

Linux 内核下载的 FTP 地址为：<ftp://ftp.kernel.org/pub/>。

Linux 内核下载的 RSYNC 地址为：<rsync://rsync.kernel.org/pub/>。

Linux 的内核从逻辑上可以分成 5 个部分。Linux 内核的子系统之间的关系如图 7-1 所示。

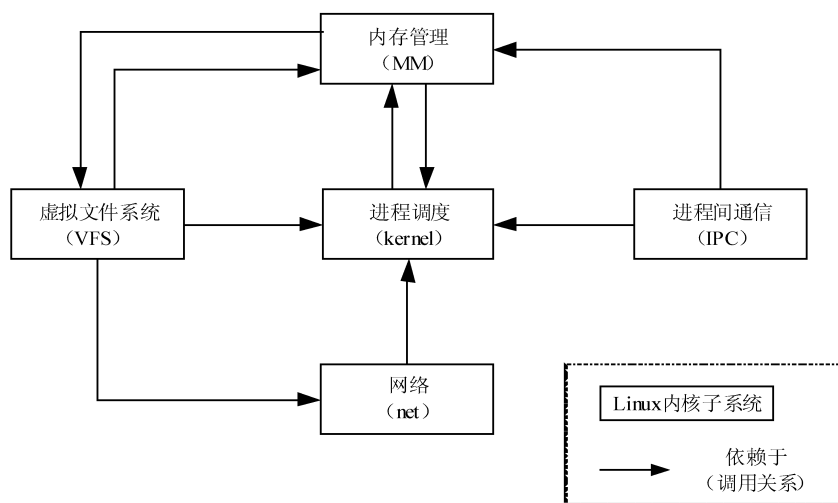


图 7-1 Linux 内核的子系统之间的关系

● 进程调度（Process Schedule）

进程调度控制进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际上是仅等待 CPU 资源的进程。如果某个进程在等待其他资源，则该进程是不可运行进程。Linux 使用了基于优先级的进程调度算法选择新的运行进程。

● 进程间通信（IPC，Intre-Process Communication）

Linux 的进程间通信包括 FIFO、管道（pipe）等机制以及 System V IPC 的共享内存（shm）、消息队列（msg）和信号灯（sem）。

● 内存管理（MM，Memory Management）

内存管理允许多个进程安全地共享主内存区域。Linux 的内存管理支持虚拟内存，即在计算机中运行的程序，它的代码、数据和堆栈的总量可以超过实际内存的大小，操作系统只是把当前使用的程序块保留在内存中，其余的程序块则保留在磁盘中。在必要时，操作系统负责在磁盘和内存间交换程序块。内存管理从逻辑上分为硬件无关部分和硬件有关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关的部分为内存管理硬件提供了虚拟接口。

● 虚拟文件系统（VFS，Virtual File System）

虚拟文件系统隐藏了各种硬件的具体细节，为所有的设备提供了统一的接口，其提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，包括 ext2、fat、NFS 等；设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

● 网络（net）

Linux 是源于网络的操作系统，提供了大量的内置网络功能，并且网络功能和内核的联系非常紧密。Linux 的网络功能包括各种网络协议和对网络硬件的访问。

7.1.2 Linux 源文件结构

Linux 2.6 之后的源代码分为几个部分：与体系结构无关的核心 C 语言代码（包括内核、进程间通信、内存管理、文件系统和网络）、驱动程序、初始化代码、与体系结构相关的 C 和汇编代码以及各部分头文件。

Linux 源代码结构如表 7-1 所示。

表 7-1 Linux 源代码结构

代码部分	程序语言	描述
init	C 语言	初始化代码，包括 C 语言入口函数 main.c
kernel	C 语言	内核的核心代码，包括进程调度等
ipc	C 语言	进程间（InterProcess Communication）通信代码
mm	C 语言	内存管理代码
fs	C 语言	文件系统代码
net	C 语言	网络代码

续表

代码部分	程序语言	描述
drivers	C 语言	各种驱动程序代码
include	C 语言头文件	各部分代码头文件
arch	C 语言/汇编语言	与体系结构相关的 C 和汇编代码

根据 Linux 源文件的结构, kernel、ipc、mm、fs 和 net 这 5 个子目录, 分别对应了 Linux 内核的 5 大模块: 进程调度、进程通信、内存管理、文件系统和网络。

Linux 另外的几个主要源代码目录如下所示。

- drivers 是 Linux 驱动程序的目录。随着 Linux 操作系统支持的设备越来越多, 驱动程序占 Linux 源文件的比例也越来越大。
- 在 init 目录中包含了 Linux 操作系统入口的函数 main.c。
- 在 arch 目录中包含了按照体系结构分类的文件夹, 例如: alpha、arm、arm26、i386、m68k、ppc、ppc64、sparc、sparc64 和 x86_64 等文件夹。这些文件夹是相互独立的, 每个文件夹代表一种体系结构。

此外, 在 Linux 的源代码中, 还包含了大量的脚本文件, 如 Makefile 和 Kconfig 等。

7.2 嵌入式 Linux 的配置和编译

7.2.1 Linux 内核配置结构

Linux 2.6 之后的内核源代码的配置使用 Kconfig 机制, 从各层子目录中逐层进行递归的配置和编译。在配置和编译的过程中, 将从根目录下的 Makefile 开始对于一个体系结构的配置和编译。

根据文件进行配置的结果, 将在 Linux 源代码根目录下生成.config 文件, 本文件是内核在编译的过程中实际需要使用的配置文件。

由于 Linux 支持多种平台, 还涉及功能的增加和裁减方面, 因此构建 Linux 内核需要先进行配置, 然后进行生成。

根目录下的 Makefile 是 Linux 内核编译的全局入口, 实际的编译动作需要根据具体体系结构的相关文件进行。

对于一个体系结构的配置和编译, 从该体系结构的目录 (即 arch 目录下的一个目录) 开始, 主要需要涉及以下内容。

- arch/\$(XXX): 体系结构的目录。
- arch/\$(XXX)/Kconfig: 用于决定某个体系结构的配置选项的文件。
- arch/\$(XXX)/config: 体系结构的各种默认文件, 将决定初始配置的选项。
- arch/\$(XXX)/configs/*_defconfig: 具体平台的配置文件。

7.2.2 Linux 内核的配置

在内核源代码目录下输入 “make XXXconfig” 等命令之一就可以对内核进行配置。

- **make config**: 提供了一个命令行界面，然后对每一个内核选项依次询问用户的选择。
- **make menuconfig**: 提供了一个基于终端 `curses` 的图形界面配置菜单。
- **make xconfig**: 提供了一个基于 `GTK` 的 `X Window` 图形界面配置菜单。
- **make oldconfig**: 与 **make config** 类似，但只提示用户设置之前没有配置过的选项。

config 只提供简单的命令行界面，**xconfig** 提供了最友好的用户界面，但是受系统界面的限制，在一般情况下，使用 **menuconfig** 即可。

配置完成之后，**make menuconfig** 将在内核源代码根目录下生成一个 **.config** 配置文件，该文件包括了当前配置所有选项的设置信息。该文件既是配置阶段的输出，同时也是实际编译过程的输入。

根据以上文件进行配置，将在 Linux 源代码根目录下生成 **.config** 文件，**.config** 文件是内核在编译的过程中实际需要使用的配置文件。它将控制整个编译过程的进行，涉及的内容包括：使用内核的哪些功能，使用哪个平台的文件，使用哪些驱动程序，对于某些功能还需要指定使用模块（**module**）或者直接编入内核。**.config** 文件和具体机器的配置文件 ***_defconfig** 是同一种格式，是根据 ***_defconfig** 自动生成的。

Linux 内核的 **make .config** 和 **make menuconfig** 流程如图 7-2 所示。

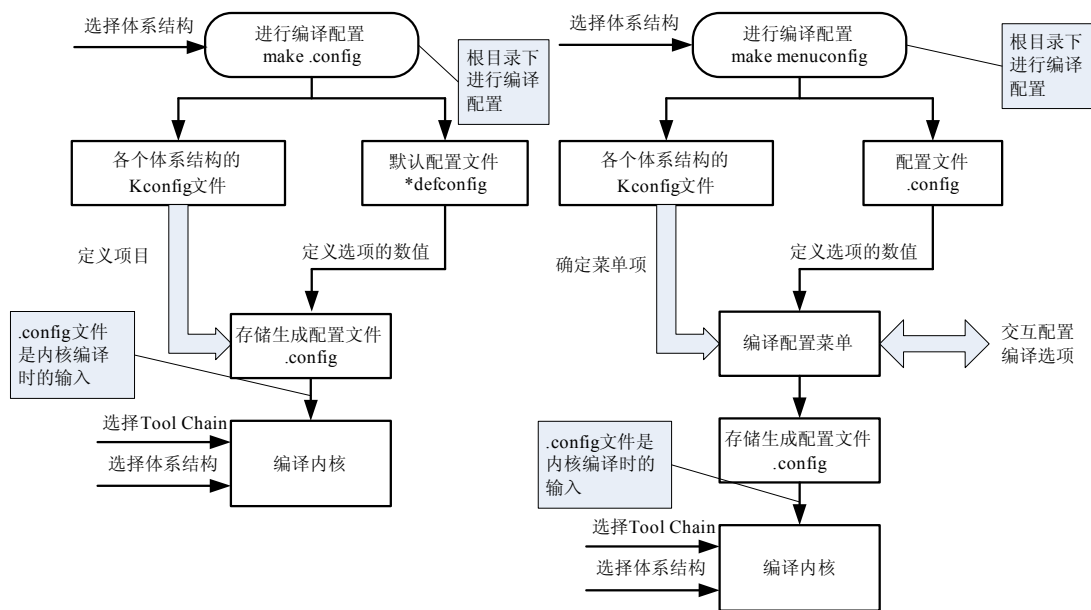


图 7-2 Linux 内核的 **make .config** 和 **make menuconfig** 流程

提示： **make .config** 并不是简单的复制，**Kconfig** 中具有选项，不一定在 **defconfig** 中有定义，但是配置之后在 **.config** 当中会有定义。

1. 主配置菜单

对于 `x86` 体系结构，使用如下的命令进行 **menuconfig**：

```
# make ARCH=i386 menuconfig
```

命令执行后，出现编译选项的列表，如图 7-3 所示。

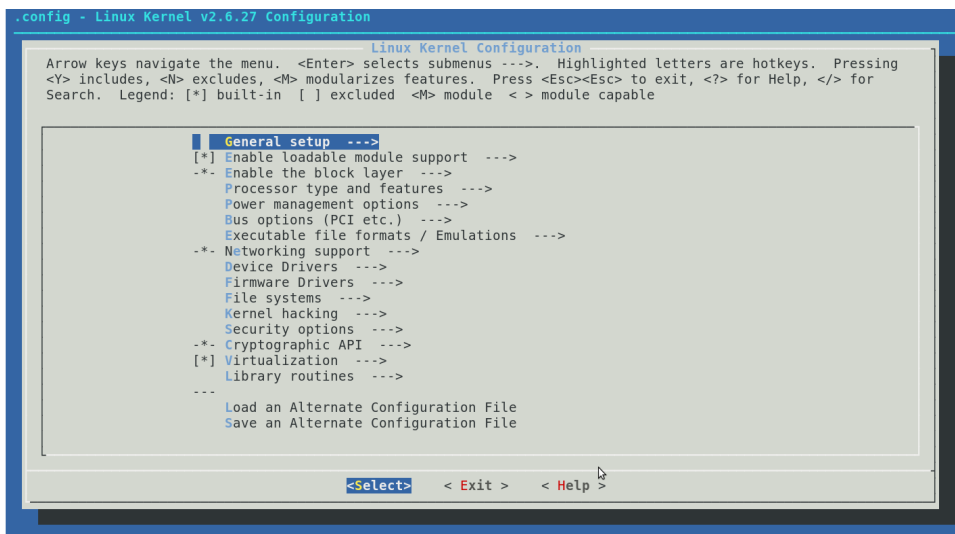


图 7-3 Linux 的主配置菜单 (x86)

对于 ARM 体系结构，可以使用如下的命令进行 menuconfig:

```
# make ARCH=arm menuconfig
```

命令执行后，出现编译选项的列表，如图 7-4 所示。

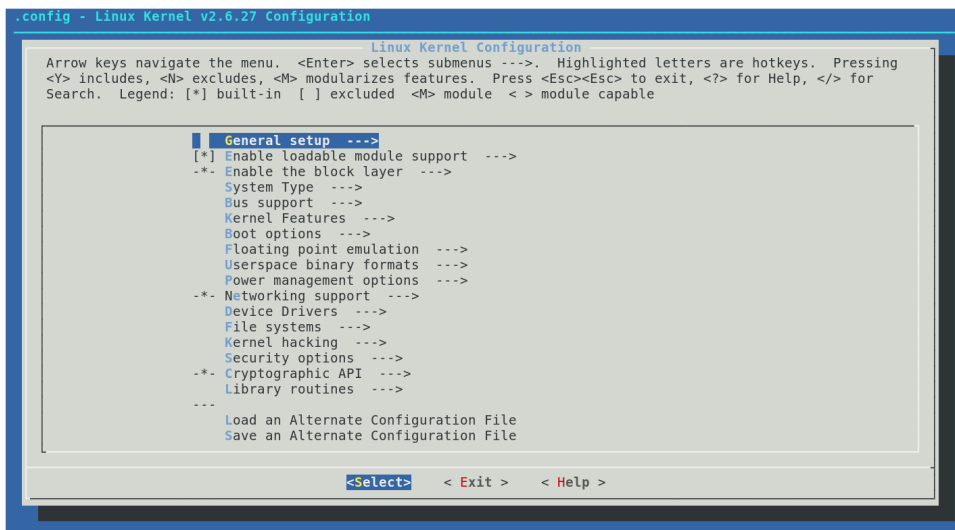


图 7-4 Linux 的主配置菜单 (ARM)

编译菜单出现后，列出了编译配置的一级菜单。对于不同的体系结构，菜单及其下几级菜单的选项是不相同的，一些通用选项如下所示：

- General Setup（通用设置）。
- System Type（系统类型）。
- Kernel Features（内核特性）。
- Devices Drivers（设备驱动）。
- File Systems（文件系统）。

在本菜单下，可以进行以下操作：左右键选择 **Select**（选择）、**Exit**（退出）、**Help**（帮助）几项，按回车键或使用快捷字母进入相应的选项。

- **Select**（选择）用于进入下级菜单。
- **Exit**（退出）用于退出本菜单。
- **Help**（帮助）用于显示各项的帮助信息。

当第一次配置的时候，将首先生成一些在主机运行的工具，生成过程的信息如下所示：

```
HOSTCC scripts/basic/fixdep
HOSTCC scripts/basic/docproc
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/kxgettext.o
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/util.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/mconf.o
SHIPPED scripts/kconfig/zconf.tab.c
SHIPPED scripts/kconfig/lex.zconf.c
SHIPPED scripts/kconfig/zconf.hash.c
HOSTCC scripts/kconfig/zconf.tab.o
HOSTLD scripts/kconfig/mconf
```

信息生成完成后，将根据 **arch=**所指定的内容，根据 **arch/{arch}** 目录下面的 **Makefile** 文件，执行配置的过程。因此，对于 **x86** 体系结构使用的是 **arch/i386/Makefile**；对于 **ARM** 体系结构，使用的是 **arch/arm/Makefile** 文件。

在配置完成后，需要选择 **Exit** 退出主菜单，系统将提示是否存储配置信息，这时将生成 **.config** 文件。这个文件是进行内核编译时的输入。

在编译配置的时候，也可以选择将当前配置项存储成其他的文件，方法为选择主菜单下面的 **Save**（存储）和 **Load**（加载）选项，具体步骤如下所示。

选择 **Save** 后，存储配置文件对话框如图 7-5 所示。

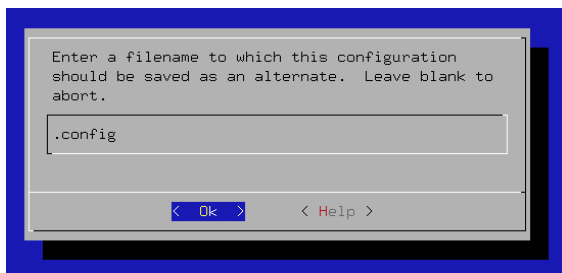


图 7-5 存储配置文件对话框

在提示框中，输入存储的配置文件的名称，这个文件将被存储在 Linux 2.6 源代码的根目录下。

选择 Load（加载）选项的对话框，如图 7-6 所示。

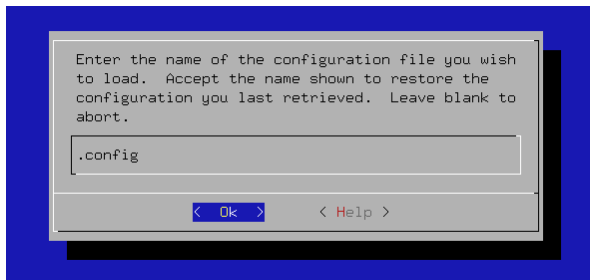


图 7-6 加载配置文件对话框

使用配置文件，可以将不同的配置存储成不同的文件，在以后编译的时候只需要加载这个文件即可。存储和加载的编译配置文件是临时文件，最终配置需要生成.config 文件。

当.config 文件已经存在的时候，再次运行“make menuconfig”命令的话，该命令将以.config 为模板设置各选项的当前值。如果需要在上次的基础上修改配置，则直接运行“make menuconfig”就可以了。内核配置菜单还提供了保存配置文件和载入配置文件的功能，可以把设置好的配置保存到一个指定的文件中，也可以从一个已有的配置文件中装载先前的配置，然后以此为基础进行设置。

.config 指定的各个选项将在各级目录的 Makefile 中使用。这个文件也可手工修改。修改的过程通常是在 CONFIG_XXX=y 和# CONFIG_XXX is not set 两种写法中修改，来确定配置功能。值得注意的是，配置中的依赖关系是由 Kconfig 文件确定的，.config 不考虑依赖关系。

2. 使用 def 配置文件

对于特定的处理器，可以使用预定义的配置文件进行配置。例如：对于 S3C2410 处理器的配置，可以使用如下的方式：

```
$ make ARCH=arm s3c2410_defconfig .config
#
# configuration written to .config
#
make[1]: Nothing to be done for `'.config'.
```

本过程实际上复制了 arch/arm/configs 目录中的 s3c2410_defconfig 文件为根目录中的.config 文件。

.config 文件生成后，可以继续使用 menuconfig 进行菜单配置。以上的两个步骤可以合二为一，使用如下命令：

```
$ make ARCH=arm s3c2410_defconfig menuconfig
```

这些选项是在 arch/arm/Kconfig 文件中定义，然后根据 s3c2410_defconfig 中设置的选

项来生成的。

arch/arm/Kconfig 这部分的片断如下所示:

```
menu "System Type"
choice
    prompt "ARM system type"
    default ARCH_VERSATILE
```

"System Type"确定生成第一级菜单。

Choice 确定项目的几个选项, 本例中使用的是如下的项目:

```
config ARCH_S3C2410
bool "Samsung S3C2410, S3C2412, S3C2413, S3C2440, S3C2442, S3C2443"
select GENERIC_GPIO
select HAVE_CLK
help
    Samsung S3C2410X CPU based systems, such as the Simtec Electronics
    BAST (<http://www.simtec.co.uk/products/EB110ITX/>), the IPAQ 1940 or
    the Samsung SMDK2410 development board (and derivatives).
```

s3c2410_defconfig 中定义内容的片断如下所示:

```
CONFIG_ARCH_S3C2410=y
# CONFIG_ARCH_SHARK is not set
# .....
```

这里使用的 CONFIG_* 的各个选项, 对应了 Kconfig 中 config 后面的内容。
s3c2410_defconfig 定义了 ARCH_S3C2410 选项, 在 Kconfig 中选定了相应的内容。

arm/Kconfig 进一步引用了其他目录中的 Kconfig 文件, 例如它包含了 arm/mm/Kconfig 文件:

```
source "mm/Kconfig"
```

在 arm/mm/Kconfig 中, 有以下的片断:

```
# ARM920T
config CPU_ARM920T
    bool "Support ARM920T processor"
    depends on ARCH_EP93XX || ARCH_INTEGRATOR || CPU_S3C2410 || CPU_S3C2440 ||
CPU_S3C2442 || ARCH_IMX || ARCH_AAEC2000 || ARCH_AT91RM9200
    default y if CPU_S3C2410 || CPU_S3C2440 || CPU_S3C2442 || ARCH_AT91RM9200
    select CPU_32v4T
    select CPU_ABRT_EV4T
    select CPU_PABRT_NOIFAR
    select CPU_CACHE_V4WT
    select CPU_CACHE_VIVT
    select CPU_CP15_MMU
    select CPU_COPY_V4WB if MMU
    select CPU_TLB_V4WBI if MMU
    help
    .....
```

在 s3c2410_defconfig 中设置了相应的内容:

```
#
# S3C2400 Machines
#
```

```
CONFIG_CPU_S3C2410=y
CONFIG_CPU_S3C2410_DMA=y
```

由于选定了 `CONFIG_CPU_S3C2410`，因此 `CPU_ARM920T` 实际上也被打开。

Kconfig 配置的选项具有依赖关系：`depends on` 表示只有当后面的条件满足的时候，这个选项才会出现在菜单中；`default y` 表示后面的条件满足后，这个选项被开启；`select` 表示当这个选项开启的时候，进而选择其他的选项。

3. 手动配置选项

在 `menuconfig` 的时候，可以在菜单中手动选择配置的内容，几种选项的情况如下所示。

- `[*]`：表示以内建（build-in）的形式进行编译。

- `[]`：表示不编译。

- `[M]`：表示以模块（module）的形式进行编译。

在进行配置的时候，可以使用下面几个按键。

- **Y 键**：以内建的形式编译。

- **N 键**：不编译。

- **M 键**：以模块的形式编译。

在 Linux 内核编译的时候，有一些内容可以选择内建（build-in）和模块（module）两种形式。使用内建形式，将把相应文件的目标文件编入 Linux 内核映像中，并随内核的加载而加载；使用模块方式，将把相应文件的目标文件作为独立的 `*.ko`，在内核启动之后，使用 `insmod` 和 `rmmmod` 的方式进行加载和卸载。

注意，只有主菜单中的 `Enable loadable module support` 选项被打开之后，才能支持模块的功能，否则不会出现使用 `M` 的选项。

按照上面的例子，`System Type`（系统类型）菜单下面的“`PWM device support`”选项是在 `arch/arm/plat-s3c24xx/Kconfig` 中定义的，内容如下所示：

```
if PLAT_S3C24XX
config S3C24XX_PWM
    bool "PWM device support"
    select HAVE_PWM
    help
        Support for exporting the PWM timer blocks via the pwm device
        system.
Endif
```

当使用 **Y 键** 选择了“`PWM device support`”选项之后，存储并退出，`.config` 文件将出现如下的变化：

```
CONFIG_ARM=y
+CONFIG_HAVE_PWM=y
CONFIG_SYS_SUPPORTS_APM_EMULATION=y
CONFIG_GENERIC_GPIO=y

CONFIG_PLAT_S3C24XX=y
CONFIG_CPU_S3C244X=y
-# CONFIG_S3C24XX_PWM is not set
+CONFIG_S3C24XX_PWM=y
```

```
CONFIG_PM_SIMTEC=y
CONFIG_S3C2410_DMA=y
```

原本的# CONFIG_S3C24XX_PWM is not set 属于注释, CONFIG_S3C24XX 值没有被设置, 当手动设置后, CONFIG_S3C24XX_PWM 被设置为 y, 并增加了 CONFIG_HAVE_PWM 选项, 且设置为 y。

这些.config 文件中的配置选项最终是将在 Makefile 中被使用的, 主要用来确定哪些文件被使用, 用何种方式使用。在 Makefile 中 obj-y 表示作为内建的被编译; obj-m 表示作为模块 (module) 的被编译。也可以使用 obj-{} 选项, 从选项中读取数值, 确定用何种方式编译。

例如 arch/arm/plat-s3c24xx/Makefile 文件中, 具有如下片断:

```
obj-$(CONFIG_HAVE_PWM) += pwm.o
```

这就决定了之后当 CONFIG_HAVE_PWM 为 y 的时候, pwm.c 文件才被使用, 编译成 pwm.o。

对于可以设置成模块的选项, 还多了一种选择, 例如 Device Drivers 中的 I2C support。显示的内容如图 7-7 所示。

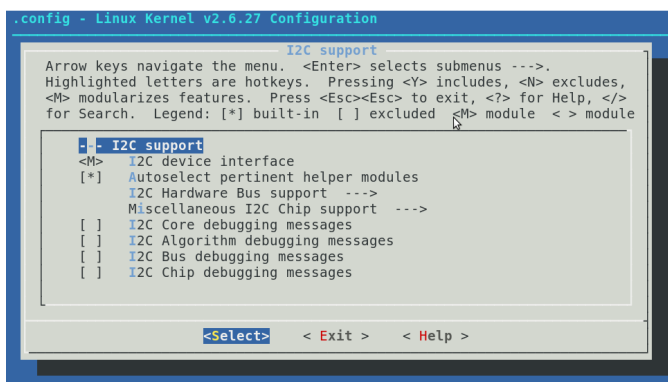


图 7-7 I2C 的配置菜单

其中"I2C device interface"选项就被设置成了模块的形式, 这段内容是在 drivers/i2c/Kconfig 文件中定义的, 如下所示:

```
if I2C
config I2C_CHARDEV
    tristate "I2C device interface"
endif # I2C
```

drivers/i2c/Makefile 中定义了相关的内容:

```
obj-$(CONFIG_I2C_CHARDEV) += i2c-dev.o
```

事实上, I2C_CHARDEV 变量的数值可以设置为 y 或者 n。

如果需要对 Linux 内核进行剪裁, 适应更小的系统需求, 则需要从配置菜单上将一些选项从 Y 选定为 N。这里需要特别注意的是依赖关系的问题。因此, 更简易地使用

menuconfig 进行配置，而不是使用手动修改.config 文件。因为如果使用手动修改.config，则可能造成依赖关系没有修改的问题。

剪裁的方式可以根据系统的需要进行，必要的剪裁可以减少系统的尺寸。由于驱动程序（配置项目为 Device Drivers）部分占用的比例较大，因此重点可以去除不必要的驱动程序支持，尤其是仅有驱动程序框架，而并无实际驱动程序的情况。网络部分（配置项目为 Networking support）也可以通过去掉不必要的协议来缩减系统的尺寸。

7.2.3 Linux 内核的生成

Linux 内核默认编译的方式如下所示：

```
$ make ARCH=<arch> CROSS_COMPILE=<prefix>
```

编译开始之后，将通过各个目录中的 Makefile 结合各个 CONFIG_*配置选项对各个文件进行编译，生成各个.o 文件，然后目录中的各个目标文件将被连接，之后生成名称为 built-in.o 的文件。编译的过程也会产生一些自动生成的头文件，如：include/linux/version.h 表示内核的版本。

```
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
MODPOST vmlinux
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
SHIPPED arch/arm/boot/compressed/liblfuncs.S
AS      arch/arm/boot/compressed/liblfuncs.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

提示：内核编译过程中显示的 CC、LD、AS 等命令表示都是调用了交叉工具，而 HOSTCC 等则是调用主机本身的工具。

内核生成的几个文件在根目录、arch/arm/boot/目录和 arch/arm/boot/compressed/目录中：

- vmlinux.o 是连接后内核的目标文件。
- vmlinux（根目录）是 ELF 格式的内核。
- System.map 是内核的符号表。
- Image 是二进制的内核文件，由 vmlinux 转换而成。
- vmlinux（compressed 目录）是经过压缩的 vmlinux 映像（通过 gzip 算法压缩）和解压缩程序的组合体，也是 ELF 格式。
- zImage 是解压缩程序和压缩内核的结合。

内核编译后，也会生成相关的模块，模块以 ko 为后缀名。模块不是内核映像的一部分。在键入 make 的时候可以增加一个目标为参数。

- **zImage**: 表示生成内核。
- **modules**: 表示生成各种内核模块。
- **uImage**: 表示生成要通过 U-Boot 启动的内核。

如果需要生成 **uImage**，则需要系统中具有 **mkimage** 工具，这个工具的格式如下所示：

```
$ mkimage
Usage: mkimage -l image
      -l ==> list image header information
      mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d
data_file[:data_file...] image
      -A ==> set architecture to 'arch'
      -O ==> set operating system to 'os'
      -T ==> set image type to 'type'
      -C ==> set compression type 'comp'
      -a ==> set load address to 'addr' (hex)
      -e ==> set entry point to 'ep' (hex)
      -n ==> set image name to 'name'
      -d ==> use image data from 'datafile'
      -x ==> set XIP (execute in place)
```

uImage 常常从 **zImage** 生成，通常是在 **uImage** 上增加一个 64 字节（0x40）的头信息。描述这个内核的版本、加载位置、生成时间、大小等信息。

利用 **mkimage** 在命令行直接生成 **zImage** 的一个命令如下所示：

```
$ mkimage -n 'linux-2.6.35' -A arm -O linux -T kernel -C none -a 0x30008000 -e 0x30008040
-n "S3C2440" -d zImage uImage
```

其中最重要的是由 **-a** 指定的加载地址和由 **-e** 指定的运行地址，它们需要与内核的编译信息相符合。

7.3 Linux 内核的启动过程

在 Linux 的启动过程中，从程序运行时间的角度上来说分成两个阶段，**BootLoader** 阶段和内核阶段。系统开始运行后，首先运行 **BootLoader**，**BootLoader** 完成初始化后，调用 Linux 内核（加载和运行），进入内核阶段。

Linux 内核本身的启动又分为压缩内核和非压缩两种。从 Linux 内核程序的结构上，具有如下的特点：

压缩内核 = 解压缩程序 + 压缩后的内核映像

当压缩内核运行后，将运行一段解压缩程序，得到真正的内核映像，然后跳转到内核映像运行。此时，Linux 进入非压缩内核的入口，在非压缩的内核入口中，系统完成各种初始化任务后，跳转到 C 语言的入口处运行。

以 ARM 为例，Linux 内核的启动流程如图 7-8 所示。

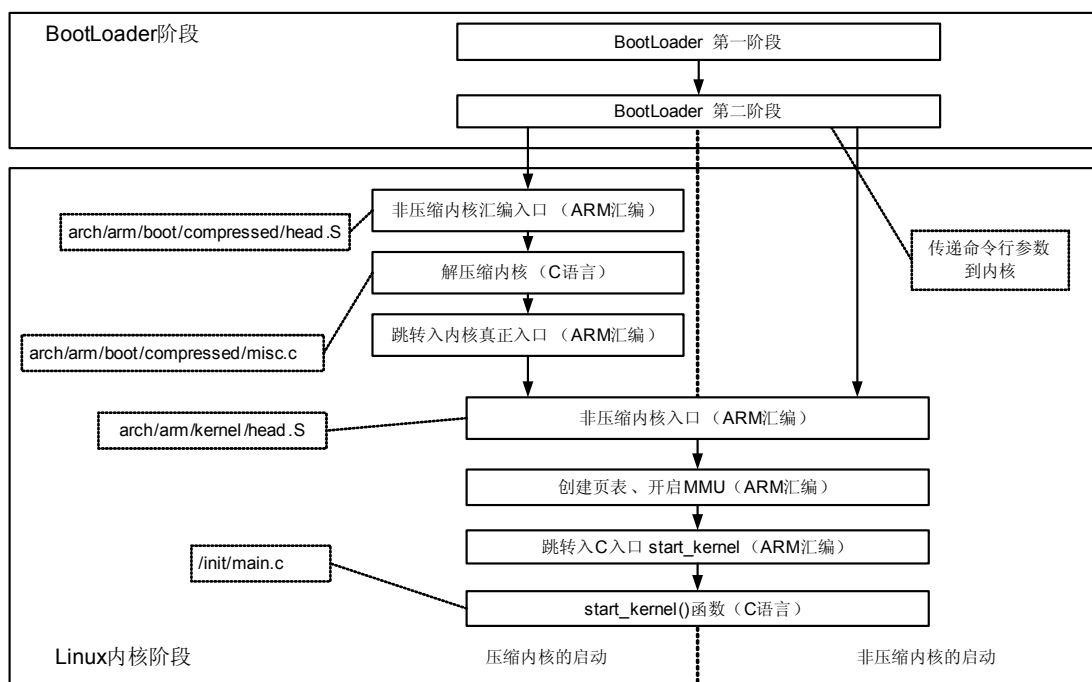


图 7-8 Linux 的启动流程

1. 压缩内核的入口

对于 Linux 2.6 之后的 ARM 体系结构，解压缩程序通常在 `arch/arm/boot/compressed/` 目录中，通常包含了 `head.S`、`piggy.S`、`misc.c` 和 `piggy.gz` 等几个文件，它们经过编译后，生成的内容独立于真正的 Linux 内核，这部分内容的功能就是初始化环境、解压缩和运行真正的 Linux 内核。

在压缩内核启动的时候，首先进入 `arch/arm/boot/compressed` 目录中的 `head.S` 文件。这是一个 ARM 汇编的程序文件，在所有 ARM 处理器上是共用的。

`head.S` 文件入口部分如下所示。在进入内核的时候，一般 `r0` 为 0，`r1` 为处理器的类型，MMU 和数据 Cache 处于关闭状态。

```

start:
    .type    start, #function
    .rept    8
    mov     r0, r0
    .endr

    b       1f
    .word   0x016f2818    @ 帮助装载器运行的幻数
    .word   start         @ zImage 加载和运行的绝对地址
    .word   _edata        @ zImage 结束地址
1:  mov     r7, r1        @ 保存体系结构的 ID
    mov     r8, #0        @ 保存 r0

```

`start` 是 `head.S` 的程序起始运行处，在此之前都是一些宏定义。在程序的 1 处保存由

BootLoader 传递下的参数，这个参数是内核启动中需要的。

```
#ifndef __ARM_ARCH_2__
    mrs r2, cpsr           @ 获得当前的模式
    tst r2, #3             @ 测试是否为 user 模式
    bne not_angel
    mov r0, #0x17          @ angel_SWIreason_EnterSVC
    swi 0x123456          @ angel_SWI_ARM
not_angel:
    mrs r2, cpsr           @ 关闭中断，在运行中禁止 angel
    orr r2, r2, #0xc0
    msr cpsr_c, r2
#else
    teqppc, #0x0c000003    @ 关闭中断
#endif
```

这段程序进行了测试，如果内核从 angel 运行，进入的状态将是用户模式（usr），这时需要进入监管模式（svc mode）并禁止所有 FIQ 和 IRQ 中断。这些只有在进入时处于用户模式下的时候才会进行。正常情况下，将运行 not_angel 处关闭中断的代码，程序继续向下运行。

```
.text
adr r0, LC0
ldmia r0, {r1, r2, r3, r4, r5, r6, ip, sp}
subs r0, r0, r1           @ 计算地址变化量

beq not_relocated

/*
 * 在不同的地址运行，需要修正以下变量：
 * r5 - zImage base address
 * r6 - GOT start
 * ip - GOT end
 */
add r5, r5, r0
add r6, r6, r0
add ip, ip, r0
```

由于程序连接时的地址和它运行的地址有可能不相同，以上程序的作用主要是计算地址的偏移量，这个过程被称为重定向（relocate）。如果连接地址 = 运行地址，则不用使用重定向，即跳转到 not_relocated 运行。如果运行地址和连接不相同，总需要重新计算地址。计算的方法为：

r5 = zImage 基地址
r6 = GOT 起始地址
r5 = GOT 结束地址

```
#ifndef CONFIG_ZBOOT_ROM
    add r2, r2, r0
    add r3, r3, r0
    add sp, sp, r0
1:  ldr r1, [r6, #0]        @ 在 GOT 中重定向入口表
    add r1, r1, r0          @ 修正 C 语言的引用
    str r1, [r6], #4
```

```

        cmp r6, ip
        blo lb
    #else
1:  ldr r1, [r6, #0]          @ 在 GOT 中重定向入口表
        cmp r1, r2          @ 如果 entry < bss_start ||
        cmphs r3, r1        @ _end < entry table
        addlo r1, r1, r0    @ 修正 C 语言的引用
        str r1, [r6], #4
        cmp r6, ip
        blo lb
    #endif

```

以上程序通过修改 GOT（Global Offset Table）完成了重定向的工作，后面的程序将按照重定向之后的情况运行：

```

not_relocated:  mov r0, #0
1:  str r0, [r2], #4        @ 清除 bss（未初始化数据段）
        str r0, [r2], #4
        str r0, [r2], #4
        str r0, [r2], #4
        cmp r2, r3
        blo lb

        bl  cache_on
        mov r1, sp          @ 在栈上分配 64KB 空间
        add r2, sp, #0x10000

```

程序进入重定向阶段之后，首先清空未初始化数据段，这是在所有 C 语言环境运行之前都需要做的，之后 C 语言的运行环境就建立完成了，打开 Cache 提高效率。在 `cache_on` 上，需要针对处理器的情况开启 MMU，从而打开 Cache。

开启 MMU 的结果在于处理器将开启虚拟内存，处理器访问的地址将不再直接是物理地址。然而，在以上程序的 `cache_on` 中，使用 `flat` 的模式开启 MMU。这时虽然已经存在内存映射，但实际上虚拟地址 = 物理地址。从程序运行的角度来说地址没有变化，但是通过开启 MMU，可以使用 Cache，提高程序的效率。这段在 ARM 体系结构中开启 Cache 的代码如下所示：

```

        cmp r4, r2
        bhs wont_overwrite
        add r0, r4, #4096*1024    @ 4MB 最大内核大小
        cmp r0, r5
        bls wont_overwrite

        mov r5, r2                @ 在分配空间后解压缩内核
        mov r0, r5
        mov r3, r7
        bl  decompress_kernel

        add r0, r0, #127
        bic r0, r0, #127          @ 对齐内核长度

        add r1, r5, r0            @ 解压缩内核完成
        adr r2, reloc_start
        ldr r3, LC1
        add r3, r2, r3
1:  ldmia r2!, {r8 - r13}        @ 复制重定向代码

```

```

stmia    r1!, {r8 - r13}
ldmia    r2!, {r8 - r13}
stmia    r1!, {r8 - r13}
cmp      r2, r3
blo      1b

bl       cache_clean_flush
add      pc, r5, r0           @ call 调用重定向代码

```

上面的一段程序是在检查是否在程序复制的过程中将自己的代码段覆盖了。计算的方法如下所示：

- r4 = 最终内核地址。
- r5 = 这段映像地址。
- r2 = malloc 空间结束地址。

计算期望的结果是：

- $r4 \geq r2$ 。
- $r4 + \text{image 的长度} \leq r5$ 。

判定条件的道理很清楚：malloc 的空间不能和内核代码地址相重，内核的长度不能超过本段映像。一般情况下，覆盖情况不会发生，然后将运行以下代码：

```

wont_overwrite:  mov r0, r4
                 mov r3, r7
                 bl  decompress_kernel
                 b   call_kernel

```

在 decompress_kernel 中，将调用 arch/arm/boot/compressed 下 misc.c 文件中的解压缩程序，将内核解压缩到适当的位置，然后跳转到解压缩以后的内核（非压缩内核）运行。由此，程序进入真正的内核（非压缩）入口。

2. 非压缩内核的入口

在 ARM 的 Linux 系统中，非压缩内核的入口是 linux/arch/arm/kernel/head.S，这是内核的真正入口，实现了内核启动以后、C 语言入口之前的功能。

```

ENTRY(stext)
msr     cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE
                                     @ 确保 svc 模式和禁止中断
mrc     p15, 0, r9, c0, c0          @ 得到 CPU 的 id
bl      __lookup_processor_type      @ r5=处理器信息 r9=cpu ID
movs    r10, r5                      @ 如果 r5==0，就为无效处理器
beq     __error_p                    @ 'p'为错误代码
bl      __lookup_machine_type        @ r5=机器信息
movs    r8, r5                      @ 如果 r5==0，就为无效的机器信息
beq     __error_a                    @ 'a'为错误代码
bl      __vet_atags
bl      __create_page_tables

```

在此处，主要对处理器类型信息和机器类型信息进行检查。其基本原理是用从 BootLoader 传入的信息和编入 Linux 内核的信息进行比较。如果发现支持该处理器和机器，继续运行；否则，执行错误代码。

在一般情况下，程序将不执行 `__error` 函数，直接执行下面的创建页表功能：

```
b1 __create_page_tables
```

在进入创建页表之前，ARM 的寄存器中 r8 为机器信息(machinfo)，r9 为 CPU ID(cpuid)，r10 为处理器信息(procinfo)。跳转程序返回后，r4 保存页表的物理地址，r0、r3、r5、r6、r7 被破坏。

本段程序是一段公共的程序，但是具体页表如何创建，需要根据不同内核的 ARM 处理器和机器信息来查找。也就是说，`__create_page_tables` 程序的实际参数是通过以上信息找到的。之所以有这样的区别，是因为各种 ARM 内核的 MMU 存在差别，各种平台的物理内存地址也存在差别。

创建页表是为了真正开启 MMU，进行内存映射做出准备，即准备使用虚拟内存。在前面的程序中，虽然 MMU 也可以开启，但是只是把物理地址映射到相同的逻辑地址，即 Flat 映射方式。在这种方式下，程序运行的逻辑地址=物理地址，因此不需要页表。在真正开启 MMU 后，页表将被放入内核空间内。

在 Linux 的虚拟内存中，通常 0-3G 的地址为用户空间，3G-4G 的地址为内核空间。在 ARM 体系中，物理地址和逻辑地址均为 32 位。在内存映射完成后，由于以上初始化等程序都属于内核的代码，因此程序即将转入 3G-4G 的内核空间运行。

在随后的程序中，将开启 MMU，并利用绝对地址跳转的方式跳转到内核空间运行，其地址由 `__switch_data` 确定。

```
ldr r13, __switch_data      @ 在 MMU 开启后，需要跳转的地址
adr lr, __enable_mmu        @ 返回 PIC（与位置无关）的地址
add pc, r10, #PROCINFO_INITFUNC
```

在后面的程序中，进入 `head-common.S` 文件中的内容，定义了 MMU 开启过程中各个寄存器中需要保存的信息，如下所示：

```
.type __switch_data, %object
__switch_data:
    .long __mmap_switched
    .long __data_loc          @ r4
    .long __data_start        @ r5
    .long __bss_start         @ r6
    .long _end                @ r7
    .long processor_id        @ r4
    .long __machine_arch_type @ r5
    .long cr_alignment        @ r6
    .long init_thread_union+8192@ sp
```

各个数值（交换数据，`switch_data`）为与内存地址和处理器有关的信息，从各种平台的链接过程中得到，这些信息将在下面的程序中使用：

```
.type __mmap_switched, %function
__mmap_switched:
    adrr3, __switch_data + 4
    ldmia r3!, {r4, r5, r6, r7}
```

在上述程序中，`__mmap_switched+4` 表示将以上定义的 4 个数据载到寄存器 r4~r7 之

中。在后面的程序中，根据这些信息，准备 C 语言的运行环境：

```

        cmp    r4, r5                @ 复制所需的数据段
1:   cmpne    r5, r6
        ldrne  fp, [r4], #4
        strne  fp, [r5], #4
        bne   1b

        mov    fp, #0                @ 清空 BSS
1:   cmp    r6, r7
        strcc  fp, [r6], #4
        bcc   1b

        ldmiar3, {r4, r5, r6, sp}
        str    r9, [r4]              @ 保存处理器 ID
        str    r1, [r5]              @ 保存机器类型
        bic    r4, r0, #CR_A         @ 清空 'A' 位
        stmia  r6, {r0, r4}          @ 保存寄存器值

        b      start_kernel
    
```

以上程序和一般 C 语言运行程序的创建很类似，即复制 RW data（可读写数据段）到 RAM，开辟 BSS 空间并清零，随后程序跳转到 C 语言的入口 `start_kernel` 中运行。

`start_kernel()` 在 `init` 目录的 `main.c` 文件中。在该函数中，将完成进入 C 语言环境以后的初始化工作，如：

```

setup_arch(&command_line);    \* 设置体系结构信息 *\
init_IRQ();                   \* 初始化中断 *\
time_init();                  \* 初始化时间 *\
console_init();               \* 控制台初始化 *\
    
```

在 `start_kernel()` 的最后将调用 `rest_init()`，解锁内核进入进程调度阶段。在 `start_kernel()` 的初始化工作中，主体运行于 C 语言环境中，但依然会调用汇编语言的代码。这是由于有一些对处理器核心的寄存器（不是芯片内部部件寄存器）操作需要通过汇编语言完成。

7.4 特定系统的 Linux 的构建

7.4.1 Linux 内核的移植

Linux 可以支持多种硬件平台，Linux 内核移植的概念是在使用其大部分公用代码的情况下，增加如下的两个部分：

- 针对处理器移植的部分。
- 针对各种驱动程序的构建部分。

这两部分的工作都是硬件相关的，前者是对处理器核心运行的部分，通常涉及 CPU、内存、中断、定时器、DMA 等部分；后者是针对平台各种设备的 Linux 驱动程序的实现。

对于某个硬件平台的 Linux 移植工作如图 7-9 所示。

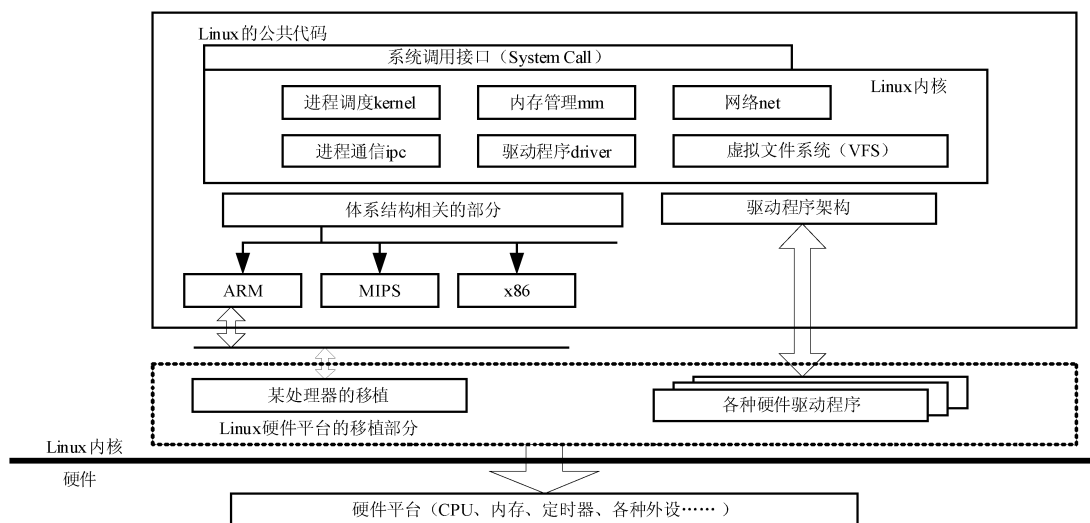


图 7-9 某个硬件平台的 Linux 移植

对于特定硬件平台的 Linux 系统的构建主要包括以下几个部分：

- 全局的配置文件是特定平台的 defconfig 文件，它可以生成.config 文件，但不是简单的复制关系。
- 公共和特有的内容均在 defconfig 文件中定义。
- 在 arch/<arch>/目录中包含其特定的内容，其中的 mach-<mach>表示某个机器的目录；plat-<plat>表示某个平台的目录，包括自身头文件和实现文件。
- 每个体系结构均有移植的核心结构。ARM 体系结构的核心是 arch/arm/include/asm/mach/arch.h 文件中的名为 machine_desc 的结构体。

7.4.2 ARM 处理器上运行的 Linux 系统

在 ARM 处理器运行的 Linux 系统也称为 ARM-Linux。ARM 是标准 Linux 支持的几种主要体系结构之一。Linux 是目前非常流行的开源操作系统，而 ARM 是目前嵌入式领域中最流行的 32 位处理器，ARM 与 Linux 的结合在嵌入式系统中有着非常广泛的应用空间。

在 ARM 处理器上运行的标准 Linux 与 PC 的 Linux 使用着基本相同的内核（包括进程调度、内存管理、进程间通信、虚拟文件系统、网络几个部分）。在 Linux 操作系统中，无论基于 ARM 的 Linux 和基于 x86 PC 的 Linux，绝大多数使用 C 语言编写的操作系统内核都是相同的，只有部分与体系结构相关的代码使用相应的汇编语言编写。因此，这使得很多在桌面 Linux 的程序都可以很容易移植到嵌入式的 ARM 系统中。

基于 ARM 的标准 Linux 和基于 x86 的桌面 Linux 使用类似的内存管理系统，使用虚拟内存到物理内存的映射。当然，它们之间也存在区别，这主要源于 ARM 的 MMU 与 i386 体系结构的 MMU 在工作原理和逻辑上有着一些差别。

i386 的 MMU 使用 48 位逻辑地址到 32 位线性地址，再到 32 位物理地址的转换。

ARM 的 MMU 使用 32 位线性地址到 32 位物理地址的转换。在 ARM MMU 的硬件中，

分成段模式和二级页模式两种，在标准 Linux 中，使用粗页表（coarse page）的二级页模式。在二级的页表中，使用 4KB 的小页。因此，ARM-Linux 使用的 ARM MMU 硬件为粗页表-小页的方式，操作系统的代码也与之对应。

Linux 将每种体系结构组成一个文件夹。以 ARM 体系结构为例，又包含了各种不同的处理器，操作对于它们的代码又不相同。因此，在 arch/arm 下包含了两方面的代码：与处理器无关的公共代码以及与处理器相关的部分。

根据代码的适用范围分类，Linux 各个部分的源文件与功能关系如图 7-10 所示。

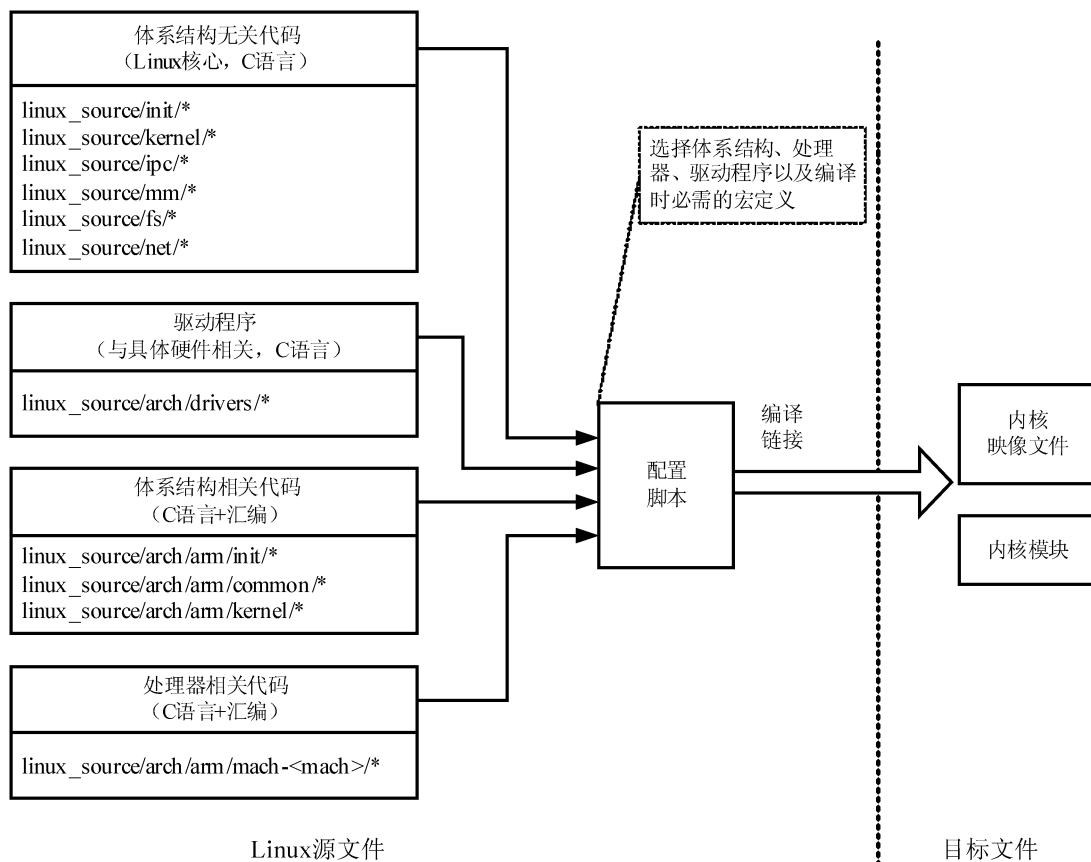


图 7-10 Linux 源文件与功能关系

arch/arm 是 ARM 体系结构的专用目录，其中主要包含的内容如下所示。

- init: 启动汇编入口，文件夹 compressed 中包含了压缩程序。
- configs: 配置脚本。
- mm: 与体系结构相关的内存管理代码。
- tools: 包含各种体系结构的定义。
- mach-<mach>: 各种体系结构的代码，例如：mach-sa1100 是对 Intel Strong ARM 内核的 SA1100 的支持，mach-at91rm9200 是对 Atmel ARM920T 内核的 AT91rm9200 处理器的支持。

一般来说, 每个 `mach-<mach>` 对应一种处理器, 可能几种相似的处理器使用一个文件夹。对处理器进行操作系统移植的时候, 实际上就可以根据相近的处理器出发, 新建一个 `mach-<mach>` 文件夹。在进行整个 Linux 编译的时候, 实际上所有的代码包括了与体系结构无关的部分+与体系结构相关的部分+与处理器相关的部分。

在 ARM 体系结构中, Linux 可以支持的种类包括 StrongARM、ARM720T、ARM920T 以及 XScale 等。但是在移植的代码结构中, 并没有对它们进行区分。事实上, 这些 ARM 的体系结构更多体现在性能上, 对于内核功能的差别固然有, 但是比较小。相比之下, 各种 ARM 处理器外围部件的差别却很大, 尤其定时器和中断控制器等关系到内核运行的部件都不相同, 这是对移植影响比较大的方面。

同样, 对于同一种处理器还可以有不同的系统。在移植的过程中, 针对相同处理器的不同系统, 还需要做一定的改动。一般来说, 这些改动体现在内存的基地址和容量上。

7.4.3 S3C6410 Linux 内核的构建

S3C6410 是一个以 ARM11 为核心的处理器, S3C6410 系统的 Linux 内核是 ARM 体系结构中的一种 Linux 实现, 具体的内容还包含了一些板级相关的内容。

选择配置文件的过程如下所示。

S3C6410 系统的 Linux 配置命令如下所示:

```
$ make ARCH=arm smdk6410_dev_defconfig .config
```

S3C6410 的主配置菜单如图 7-11 所示。

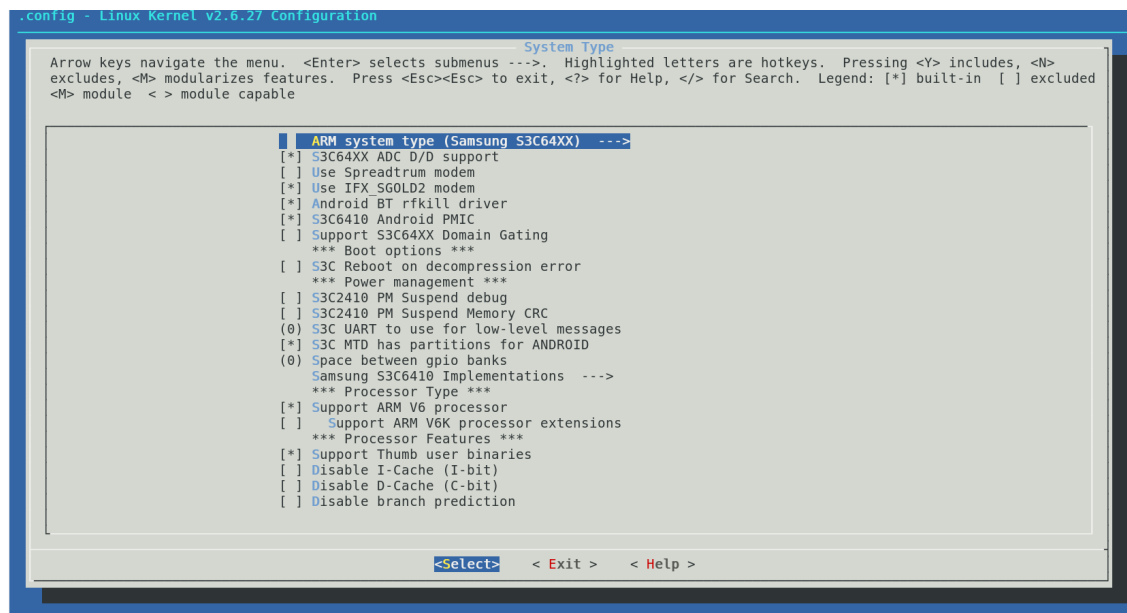


图 7-11 S3C6410 的主配置菜单

选择处理器的菜单如图 7-12 所示。

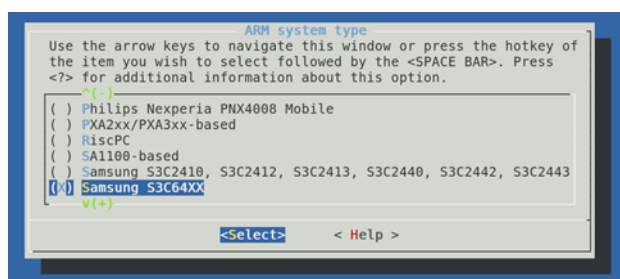


图 7-12 选择处理器的菜单

S3C6410 选择的系统类型为 Samsung S3C6400。三星的其他处理器是 ARM9 系列的，而 S3C64xx 是 ARM11 系列的，它们的系统类型是分开的。

进一步，选择机器类型为 SMDK6410，选择的内容如图 7-13 所示。

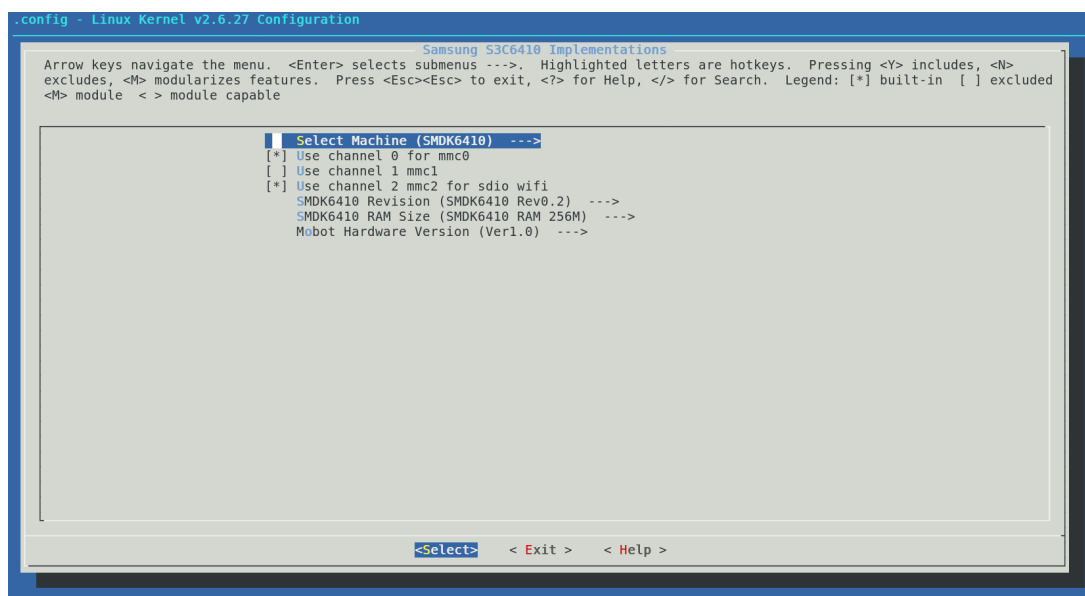


图 7-13 选择机器类型为 SMDK6410

Linux 内核配置的机器（machine）部分分成系统类型和机器类型。Samsung S3C6400 的系统类型中目前包含了 S3C6400 和 S3C6410 两种机器。

7.4.4 S3C6410 Linux 内核的移植内容

S3C6410 系统的 Linux 的构建命令如下所示，选择指定的编译工具进行编辑即可：

```
$ make ARCH=arm CROSS_COMPILE={path}/arm-none-linux-gnueabi-
```

编译的最终结果如下所示：

```
GEN    .version
CHK    include/linux/compile.h
UPD    include/linux/compile.h
CC     init/version.o
```

```

LD      init/built-in.o
LD      .tmp_vmlinux1
KSYM    .tmp_kallsyms1.S
AS       .tmp_kallsyms1.o
LD      .tmp_vmlinux2
KSYM    .tmp_kallsyms2.S
AS       .tmp_kallsyms2.o
LD      vmlinux
SYSMAP   System.map
SYSMAP   .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS       arch/arm/boot/compressed/head.o
GZIP     arch/arm/boot/compressed/piggy.gz
AS       arch/arm/boot/compressed/piggy.o
CC       arch/arm/boot/compressed/misc.o
LD       arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
Building modules, stage 2.
MODPOST 1 modules
CC       drivers/scsi/scsi_wait_scan.mod.o
LD [M]   drivers/scsi/scsi_wait_scan.ko

```

与标准 Linux 编译的流程类似：根目录中的 `vmlinux.o` 是连接后内核的目标文件；`vmlinux` 是 ELF 格式的内核（没有经过压缩）；`arch/arm/boot/` 目录中的 `Image` 是二进制的内核文件，由根目录中的 `vmlinux` 转换而成。`arch/arm/boot/compressed/` 目录中的 `vmlinux` 是经过压缩的 `vmlinux`（gzip 算法压缩）和解压缩程序的组合体，也是 ELF 格式；`zImage` 是经过 `vmlinux` 转换而成的二进制文件。

具体将生成哪些模块，由配置的时候决定，这里生成了 `scsi_wait_scan.ko` 模块。

S3C6410 系统的 Linux 内核特有的内容主要包含了几个目录，如下所示。

- `arch/arm/plat-s3c/`：三星的 S3C 平台公共的内容。
- `arch/arm/mach-s3c64xx/`：s3c64xx 平台的移植内容。
- `arch/arm/plat-s3c64xx/include/plat/`：主要包含了各个模块的寄存器定义、DMA、中断等头文件。

S3C6410 处理器主要的文件如下所示：

- `arch/arm/mach-s3c64xx/cpu.c`：处理器移植文件。
- `arch/arm/mach-s3c64xx/irq.c`：中断移植文件。
- `arch/arm/mach-s3c64xx/pm.c`：能源管理相关文件。
- `arch/arm/mach-s3c64xx/dma.c`：DMA 移植文件。
- `arch/arm/mach-s3c64xx/mach-smdk6410.c`：当前硬件系统支持文件。

控制 S3C6410 Linux 内核配置的是其默认的 `defconfig` 文件，这个文件的主要内容如下所示：

```

CONFIG_ARM=y
#
# System Type
#
CONFIG_ARCH_S3C64XX=y

```

```
CONFIG_PLAT_S3C64XX=y
CONFIG_CPU_S3C6400_INIT=y
CONFIG_CPU_S3C6400_CLOCK=y
CONFIG_S3C64XX_ADC=y
CONFIG_CPU_S3C6410=y
```

这里主要指定了为 ARM 体系结构，平台为 PLAT_S3C64XX，机器为 CPU_S3C6410，在编译的过程中，Linux 源代码各个目录中的 Makefile 根据这些选项决定哪些文件被编译，哪些不需要编译。

mach-smdk6410.c 是当前机器实现的核心文件，每一个处理器实现移植的入口是机器类型的定义部分。SMDK6410 机器类型的定义部分如下所示：

```
MACHINE_START(SMDK6410, "SMDK6410")
    .phys_io = S3C_PA_UART & 0xffff0000, //物理端口
    .io_pg_offst = (((u32)S3C_VA_UART) >> 18) & 0xfffc, // IO 地址
    .boot_params = S3C64XX_PA_SDRAM + 0x100, //启动参数
    .fixup = smdk6410_fixup, // 修定的信息
    .init_irq = s3c6410_init_irq, // IRQ 初始化
    .map_io = smdk6410_map_io, // IO 映射
    .init_machine = smdk6410_machine_init, // 机器初始化
    .timer = &s3c64xx_timer, // 定时器
MACHINE_END
```

在 MACHINE_START 和 MACHINE_END 之间的内容为当前机器的信息。这里实现的结构是 arch/arm/include/asm/mach/arch.h 中定义的 struct machine_desc。这里赋值了定时器、物理 IO 等内容以及初始化机器、初始化 irq、IO 映射等函数指针。

其中，mach-smdk6410.c 文件中定义了各个平台设备（platform_device）的列表。定义的 platform_device 类型的数组结构如下所示：

```
static struct platform_device *smdk6410_devices[] __initdata = {
    &s3c6410_pmic_device,
#ifdef CONFIG_SMDK_64xx_I2C_0
    &s3c_device_i2c0,
#endif
#ifdef CONFIG_SMDK_64xx_I2C_1
    &s3c_device_i2c1,
#endif
    // &s3c_device_spi0,
    &s3c_device_spi1,
#ifdef CONFIG_TOUCHSCREEN_S3C
    &s3c_device_ts,
#endif
    &s3c_device_smc911x,
    &s3c_device_lcd,
    &s3c_device_onenand,
    &s3c_device_keypad,
    &s3c_device_mobot_keypad,
    &s3c_device_mobot_ts,
    &s3c_device_usb,
    &s3c_device_usb gadget,
#ifdef CONFIG_USB_S3C_OTG_HOST
    &s3c_device_usb_otghcd,
#endif
}
```

```
// 省略部分内容, 设备的定义
};
```

`smdk6410_devices` 数组表示了平台中的各个设备信息 (包括资源信息), 在初始化的阶段被注册到系统中, 注册的过程如下所示:

```
static void __init smdk6410_machine_init(void)
{
    // ..... 基本的初始化步骤
    platform_add_devices(smdk6410_devices, ARRAY_SIZE(smdk6410_devices));
    s3c6410_add_mem_devices (&pmem_setting);
    s3c6410_pm_init();      // 设备电源管理相关的初始化
    smdk6410_set_qos();
}
```

在 Linux 的驱动程序架构中, `platform_device` (平台设备) 和 `platform_driver` (平台驱动) 需要匹配, `platform_device` 定义系统中的设备和相关的资源, `platform_driver` 定义某个设备的驱动程序。`platform_device` 和 `platform_driver` 匹配的方式是其 `name`。

例如: S3C64xx 的平台设备的定义在 `arch/arm/plat-s3c64xx/devs.c` 文件中, 其中有关 LCD 的设备的定义如下所示:

```
static struct resource s3c_lcd_resource[] = {
    [0] = {
        .start = S3C64XX_PA_LCD,          // LCD 的内存资源
        .end   = S3C64XX_PA_LCD + SZ_1M - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_LCD_VSYNC,           // LCD 的中断资源
        .end   = IRQ_LCD_SYSTEM,
        .flags = IORESOURCE_IRQ,
    }
};
static u64 s3c_device_lcd_dmamask = 0xffffffffUL;
struct platform_device s3c_device_lcd = {
    .name       = "s3c-lcd", // 设备的名称
    .id        = -1,
    .num_resources = ARRAY_SIZE(s3c_lcd_resource), // 资源大小
    .resource   = s3c_lcd_resource, // 资源
    .dev        = {
        .dma_mask      = &s3c_device_lcd_dmamask,
        .coherent_dma_mask = 0xffffffffUL
    }
};
```

在 `platform_device` 类型的结构 `s3c_device_lcd` 中, 成员 `name` 为 "s3c-lcd", 表示定义了设备名称。

在驱动程序的目录中, 文件 `driver/video/samsung/s3cfb.c` 是 LCD 的驱动程序, 其内容如下所示:

```
static struct platform_driver s3cfb_driver = {
    .probe      = s3cfb_probe,
    .remove     = s3cfb_remove,
    .suspend    = s3cfb_suspend,
```

```
.resume      = s3cfb_resume,  
.driver      = {  
    .name     = "s3c-lcd",           // device_driver 结构  
    .owner    = THIS_MODULE,  
},  
};
```

在这里，定义了 `platform_driver` 当中 `device_driver` 的 `name` 也是 "s3c-lcd"，因此在注册这个驱动程序的时候，将和 `platform_device` 完成匹配。在设备中所定义的资源 `s3c_lcd_resource`，也将在驱动程序中被使用。

第 8 章

文件系统及其构建

8.1 Linux 文件系统特性

文件系统（File System）是文件存放在存储设备上的组织方法。主要体现在对文件和目录的组织上。在 UNIX 系统中，文件系统是最基本的资源。在内核和文件系统之间通过制定一个标准的接口实现，不同文件结构之间可以通过该接口方便地交换数据。

Linux 正是使用这种方式，在内核和文件系统之间提供了标准的接口——VFS（Virtual File System，虚拟文件系统）。

文件系统（File System）是文件存放在磁盘等存储设备上的组织方法。主要体现在对文件和目录的组织上。目录提供了管理文件的一个方便而有效的途径。本身 Linux 的内核运行是不依赖于文件系统的，但是文件系统可以让嵌入式 Linux 操作系统拥有很多与 PC 相似的强大功能。

文件系统是 UNIX 系统最基本的资源。最初的 UNIX 系统一般都只支持一种单一类型的文件系统，文件系统的结构与系统内核紧密联系在了一起，这对于升级或更换文件系统十分不利。要改变这种情况，就需要按照上面的方式分离内核与文件系统。

文件系统的代码分成了两部分：上层用于处理系统内核的各种表格和数据结构；而下层用来实现文件系统本身的函数，并通过 VFS 来调用。

在 Linux 中，不同的文件系统连接成一个单一的树型结构，用一个统一的单个实体表示文件系统。Linux 在文件系统安装的时候，将其加载到这个单一的文件系统树上。所有的文件系统，不管什么类型，都安装在一个目录下，安装好的文件系统的文件掩盖了这个目录原来存在的内容。这个目录叫作安装目录或安装点。当这个文件系统卸载的时候，安装目录自己的文件又可以显现出来。加载文件系统的用户命令是 `mount`，该命令的实现正是基于上面的道理。

在 Linux 的嵌入式应用中，目前常用的有 EXT2、CRAMFS、JFFS2、NFTL、NFS 和 RAM 磁盘文件系统等。对于嵌入式文件系统，需要考虑如下几个特性。

- 可写入

只有当嵌入式系统需要更新文件系统中的数据时，它才需要一个具有可写入能力的文件系统。

- 可更新

该特性是指文件系统能保持重启前的更改，只有当嵌入式系统需要让它写入的内容在重启之后仍然有效，才需要文件系统具有可更新的性质。显然，可更新的基础是文件系统可写入。

- 掉电可靠性

文件系统在发生掉电的时候是否可以恢复原数据。该特性不仅指文件系统在掉电时数据可以保持，而且还要保证数据是可靠的。

- 可压缩

该特性是指文件系统内容是否可被压缩。在嵌入式系统中，文件系统的压缩特性可以让同样的存储器存储更多的数据，只有在使用时才将数据解压，这样可以节省系统的成本。

- RAM 启动

文件系统加载的方式是文件系统需要考虑的另一个问题。大多数文件系统是从其存储器中直接被加载的。但有时受到 ROM 或 Flash 大小的限制，文件系统可能要经过压缩放在存储器中，这时就需要在 RAM 中分配出一块区域将文件系统解压后，再将这块内存加载到系统中。这里就需要一个被称为 RAMDisk 的技术（也称“RAM 盘”技术）。

RAMDisk 技术的目的是将内存变成一个附加的硬盘以提高系统的运行速度。正如 RAM 盘这个名字所描述的，这项技术是基于 RAM 的块设备。同一时间内可以有多个 RAM 盘。由于它们表现为块设备，因此任何文件系统可以配合 RAM 盘使用。但是需要注意盘中内容可以保持的时间仅仅截至系统重新引导。RAM 盘通常被组装上磁盘文件系统的的一个压缩映像。

这种压缩的 RAMDisk 映像对于嵌入式 Linux 最有吸引力的地方之一就是系统的安装。大体上，内核可以从存储器中导出 RAMDisk 映像，作为它的根文件系统来使用。在启动时，内核首先检查引导选项是否存在一个 `initrd`。如果有，内核就从指定的存储介质中获得压缩的或是未压缩的文件系统的映像，并且导入到 RAM 盘中，将其挂载作为自己的根文件系统。`initrd` 的结构实际上是为内核提供根文件系统的最简单方法，同时也提高了系统的执行效率。

简单的 RAMDisk 启动映像的制作方式如下所示：

```
# dd if=/dev/zero of=./initrd.img count=2048 bs=1024
# mke2fs -F -m0 initrd.img
# mount initrd.img /mnt/ -o loop
# 向/mnt/目录中增加内容
# umount /mnt/
```

8.2 Linux 文件系统的结构

Linux 的文件系统结构分成虚拟文件系统和具体文件系统两个层次。虚拟文件系统（VFS）为 Linux 上层提供统一的文件系统的接口。每一种对同一接口的实现就是具体的文件系统，例如：RAMFS、EXT、FAT 等。具体的文件系统通常又基于块设备。

Linux 的文件系统结构如图 8-1 所示。

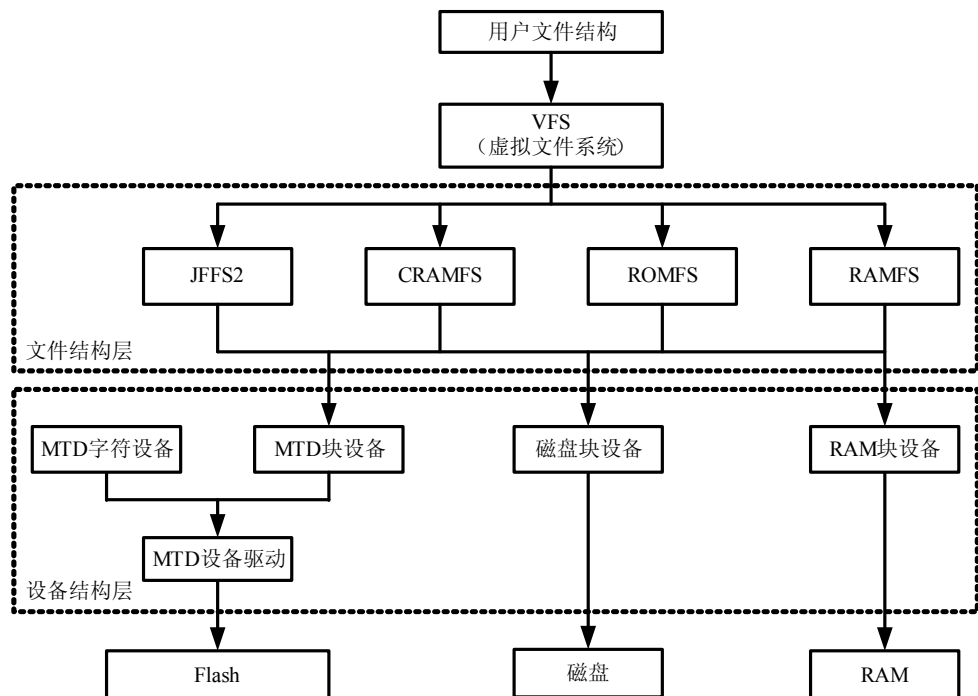


图 8-1 Linux 的文件系统结构

文件系统是 Linux 内核的关键部分，Linux 内核源代码的 fs 目录是文件系统的核心实现，fs 目录中的各个子目录是具体文件系统的实现。include/linux/fs.h 头文件是文件系统主要的头文件。

8.2.1 文件系统的主要接口

Linux 文件系统的主要接口包括表示文件系统中文件的 inode 及其操作 inode_operations、表示文件系统超级块的 superblock 及其操作 super_operations、表示进程打开文件的 file 及其操作 file_operations 等几个结构体。

1. 索引节点 inode

inode 结构的含义为索引节点，它表示文件在文件系统中存储信息。其中的主要信息包括：大小、访问权限、拥有者（gid 和 uid）、时间戳、连接到这个文件数目等。常规文件、目录、设备节点等类型的文件在 Linux 内核都用 inode 来表示。

inode 结构如下所示:

```
struct inode {
    struct hlist_node    i_hash;
    struct list_head     i_list;
    struct list_head     i_sb_list;           // 表示 super_block 的链表
    struct list_head     i_dentry;
    unsigned long        i_ino;
    atomic_t             i_count;
    unsigned int         i_nlink;
    uid_t               i_uid;               // 用户 id
    gid_t               i_gid;               // 用户组 id
    dev_t               i_rdev;
    unsigned int         i_blkbits;
    u64                 i_version;
    loff_t              i_size;
    // ..... 省略部分内容
    const struct inode_operations *i_op;    // 表示索引节点的操作
    const struct file_operations *i_fop;    // 表示该索引节点被打开后的文件操作
    struct super_block   *i_sb;             // 表示指向的超级块
    struct file_lock     *i_flock;
    // ..... 省略部分内容
};
```

除了文件的基本信息之外, inode 当中还有一个 inode_operations 结构, 它表示了对这个文件节点的操作。

inode_operations 结构如下所示:

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *,int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*check_acl) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range) (struct inode *, loff_t, loff_t);
    long (*fallocate) (struct inode *inode, int mode, loff_t offset,
                       loff_t len);
    int (*fiemap) (struct inode *, struct fiemap_extent_info *, u64 start,
                  u64 len);
};
```

`inode_operations` 结构当中包括了众多的函数指针，这些函数指针表示的就是对文件系统中文件的操作。例如 `create` 表示创建，`link` 和 `unlink` 表示建立连接和删除连接，`mknod` 表示建立一个节点，`mkdir` 和 `rmdir` 表示创建和删除目录，`rename` 表示重新命名，`setattr` 和 `getattr` 表示设置和获取属性。

`inode` 的几个操作函数如下所示：

```
extern int inode_init_always(struct super_block *, struct inode *);
extern void inode_init_once(struct inode *);
extern void inode_add_to_lists(struct super_block *, struct inode *);
extern void iput(struct inode *);
extern struct inode * igrab(struct inode *);
extern ino_t iunique(struct super_block *, ino_t);
extern int inode_needs_sync(struct inode *inode);
extern void generic_delete_inode(struct inode *inode);
extern void generic_drop_inode(struct inode *inode);
extern int generic_detach_inode(struct inode *inode);
```

表示文件的 `inode` 属于表示文件系统的 `super_block`，因此 `inode_init_always()` 函数表示从文件系统中初始化一个 `inode`，`inode_add_to_lists()` 表示将 `inode` 加入文件系统。其他的函数是对 `inode` 本身的操作。

几个虚拟文件系统的操作函数如下所示：

```
extern int vfs_create(struct inode *, struct dentry *, int, struct nameidata *);
extern int vfs_mkdir(struct inode *, struct dentry *, int);
extern int vfs_mknod(struct inode *, struct dentry *, int, dev_t);
extern int vfs_symlink(struct inode *, struct dentry *, const char *);
extern int vfs_link(struct dentry *, struct inode *, struct dentry *);
extern int vfs_rmdir(struct inode *, struct dentry *);
extern int vfs_unlink(struct inode *, struct dentry *);
extern int vfs_rename(struct inode *, struct dentry *,
                      struct inode *, struct dentry *);
```

几个以 `vfs` 为开头的函数作为对虚拟文件系统的操作，它们实际上就是调用 `inode_operations` 结构当中的函数指针来实现的。

2. 超级块 `super_block`

`super_block` 的含义为超级块，超级块的本身含义就是某个磁盘分区最开始的一段内容。超级块中的数据其实就是文件系统的元数据，也就是一个文件系统最根本的内容。

```
struct super_block {
    struct list_head s_list;        // 首项为链表头
    dev_t            s_dev;         /* search index; _not_ kdev_t */
    unsigned char    s_dirt;
    unsigned char    s_blocksize_bits; // 块的移位数
    unsigned long    s_blocksize;     // 块的大小
    loff_t           s_maxbytes;     /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations *s_op; // 超级块的操作
    const struct dquot_operations *dq_op;
    const struct quotactl_ops *s_qcop;
    const struct export_operations *s_export_op;
    unsigned long    s_flags;
    unsigned long    s_magic;
```

```

    struct dentry      *s_root;                // 表示文件系统的根
// .....省略部分内容
    struct list_head s_inodes;    /* all inodes */
    struct hlist_head s_anon;     /* anonymous dentries for (nfs) exporting */
    struct list_head s_files;
    struct list_head s_dentry_lru; /* unused dentry lru */
    int s_nr_dentry_unused; /* # of dentry on lru */
// .....省略部分内容
};

```

`super_block` 当中除了表示大小信息之外, `s_inodes` 成员还表示了超级块中所有 `inode` 的链表。主要成员 `s_op` 的类型为 `super_operations` 结构体, 表示文件系统的各种操作。

`super_operations` 结构如下所示:

```

struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options)(struct seq_file *, struct vfsmount *);
    int (*show_stats)(struct seq_file *, struct vfsmount *);
#ifdef CONFIG_QUOTA
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
#endif
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};

```

`super_operations` 由一系列的函数指针组成, 这些函数指针用于操作一个文件系统。例如, `alloc_inode` 表示的就是从超级块 `super_block` 中得到一个索引节点 `inode`。`super_operations` 的主要函数指针都用于操作 `inode`, 它们用于联系文件系统和其中的文件。

以下的两个函数表示锁住和解锁文件系统:

```

extern void lock_super(struct super_block *);
extern void unlock_super(struct super_block *);

```

对文件系统的一些操作常常夹在 `lock_super()` 和 `unlock_super()` 两个函数之间。

3. 文件 file

`file` 结构体表示进程打开的文件, 正对应了用户空间中的文件描述符。进程打开的文件通常来自文件系统, 但是二者并不是直接的对应关系, 一个文件系统中的文件可以被打开多次, 得到不同的进程打开的文件。

file 结构如下所示:

```
struct file {
    union {
        struct list_head    fu_list;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path              f_path;
#define f_dentry             f_path.dentry
#define f_vfsmnt             f_path.mnt
    const struct             file_operations *f_op;      // 进程打开文件的文件操作
    atomic_long_t            f_count;
    unsigned int             f_flags;
    mode_t                   f_mode;
    loff_t                   f_pos;                     // 文件的操作位置
    struct fown_struct       f_owner;                   // 文件的所有者
    unsigned int             f_uid, f_gid;              // 文件本身的 uid 和 gid
    struct file_ra_state     f_ra;
    u64                      f_version;
    // ..... 省略部分内容
    void                    *private_data;              // 文件的私有数据
    // ..... 省略部分内容
};
```

除了这个文件在磁盘中的相关信息之外, file 结构中还包括一个 `file_operations` 结构体, 表示这个被打开的文件所支持的操作。

表示文件操作的结构 `file_operations` 主要由函数指针组成, 如下所示:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    // ..... 省略部分内容: file_operations 当中的函数指针
};
```

`file_operations` 结构中的各个函数指针表示进程打开文件之后的各种操作, 它们基本和用户空间的文件操作函数相对应。`open` 函数指针的第一个参数是 `inode` 类型的指针, 第二个参数是 `file` 类型的指针, 表示从一个 `inode` 得到 `file`; 其他各个函数指针的第一个成员大都是 `file` 类型的指针, 表示进程打开文件的操作句柄。

提示: `inode` 当中表示文件操作的 `file_operations` 结构, 它通常在文件打开时, 赋值给 `file` 中的对应结构。

8.2.2 文件系统的实现

1. 基本的实现结构

对于一个具体的文件系统的实现，核心的内容就是实现表示操作的 `inode_operations` 和 `super_operations`，然后将其向系统注册。

具体文件系统的实现还需要一些封装的结构和注册函数来完成，其中重要的是 `file_system_type` 结构。Linux 具体文件系统的实现结构如图 8-2 所示。

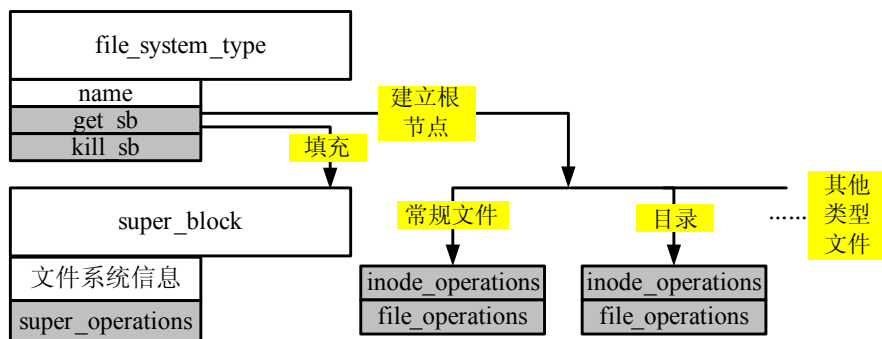


图 8-2 Linux 具体文件系统的实现结构

表示文件系统的 `file_system_type` 结构如下所示：

```

struct file_system_type {
    const char *name;           // 文件系统的名称
    int fs_flags;
    int (*get_sb) (struct file_system_type *, int,
                  const char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next; // 指向下一个文件系统
    struct list_head fs_supers;
    // .....省略部分内容: 各个 lock_class_key
};
    
```

`file_system_type` 是文件系统的实现接口，其中的 `name` 表示文件系统的名字。`get_sb` 函数指针用于得到一个文件系统上的 `super_block`，第一个参数是表示文件系统的 `file_system_type` 本身，第二个参数是整型的 `flag`，第三个参数是名称，第四个参数表示设备，最后一个参数表示在虚拟文件系统上的挂接信息。

以下的两个函数完成文件系统的注册和注销，如下所示：

```

extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
    
```

Linux 各种文件系统的实现在 `fs` 的各个子目录中，它们都通过定义 `file_system_type` 结构来实现。

2. RAMFS 的实现

RAMFS 是在内存中建立的文件系统，这也是 Linux 众多文件系统中最为简单的一种，因为其实现的“介质”只是内存，没有特殊的硬件。

RAMFS 实现的源代码在 fs/ramfs/ 目录中，包括 inode.c 和 file-mmu.c 两个文件。

file_system_type 类型结构的定义如下所示：

```
static struct file_system_type ramfs_fs_type = {
    .name      = "ramfs",
    .get_sb     = ramfs_get_sb,
    .kill_sb    = ramfs_kill_sb,
};
```

ramfs_fs_type 只实现了 file_system_type 当中的两个函数指针，“ramfs”是这种文件系统的名称。

ramfs_fs_type 的注册和注销如下所示：

```
static int __init init_ramfs_fs(void)
{
    return register_filesystem(&ramfs_fs_type);
}
static void __exit exit_ramfs_fs(void)
{
    unregister_filesystem(&ramfs_fs_type);
}
```

表示获得超级块的 ramfs_get_sb() 函数如下所示：

```
int ramfs_get_sb(struct file_system_type *fs_type,
    int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
    return get_sb_nodev(fs_type, flags, data, ramfs_fill_super, mnt);
}
```

get_sb_nodev() 是文件系统的核心函数，其中的第 4 个参数 ramfs_fill_super 是一个函数指针，用于填充一个超级块的结构。

ramfs_fill_super() 函数如下所示：

```
int ramfs_fill_super(struct super_block *sb, void *data, int silent)
{
    struct ramfs_fs_info *fsi;           // ramfs 的私有数据结构
    struct inode *inode = NULL;
    struct dentry *root;
    int err;
    save_mount_options(sb, data);
    fsi = kzalloc(sizeof(struct ramfs_fs_info), GFP_KERNEL);
    sb->s_fs_info = fsi;
    // .....省略错误处理
    err = ramfs_parse_options(data, &fsi->mount_opts); // 处理参数信息
    if (err)
        goto fail;
    sb->s_maxbytes      = MAX_LFS_FILESIZE; // 填充超级块 super_block 的成员
    sb->s_blocksize     = PAGE_CACHE_SIZE;
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic          = RAMFS_MAGIC;
```

```

        sb->s_op          = &ramfs_ops;          // 设置 super_operations
        sb->s_time_gran    = 1;
        inode = ramfs_get_inode(sb, NULL, S_IFDIR | fsi->mount_opts.mode, 0);
// .....省略错误处理
        root = d_alloc_root(inode);            // 获得根节点
        sb->s_root = root;
// .....省略错误处理
        return 0;
// .....省略错误处理
    }

```

ramfs_fill_super()函数主要调用 ramfs_get_inode()函数用于建立首个 inode，并设置 super_block 中用于操作的 super_operations 结构。首个 inode 建立之后，将其设置为 super_block 当中的根 (root)，首个 inode 的类型显然是目录。

super_operations 类型的结构 ramfs_ops 如下所示：

```

static const struct super_operations ramfs_ops = {
    .statfs      = simple_statfs,
    .drop_inode  = generic_delete_inode,
    .show_options= generic_show_options,
};

```

ramfs_ops 中只实现了 super_operations 中的 3 个函数指针，它们 3 个都是文件系统的默认实现。

ramfs_get_inode()函数的实现如下所示：

```

struct inode *ramfs_get_inode(struct super_block *sb,
                              const struct inode *dir, int mode, dev_t dev)
{
    struct inode *inode = new_inode(sb); // 从超级块中建立一个 inode
    if (inode) {
        inode_init_owner(inode, dir, mode); // 初始化 inode
        inode->i_mapping->a_ops = &ramfs_aops; // inode 的操作
        inode->i_mapping->backing_dev_info = &ramfs_backing_dev_info;
        mapping_set_gfp_mask(inode->i_mapping, GFP_HIGHUSER);
        mapping_set_unevictable(inode->i_mapping);
        inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
        switch (mode & S_IFMT) { // 针对不同类型文件的操作
            default: // 完成其他类型文件的初始化
                init_special_inode(inode, mode, dev); // 建立特殊文件
                break;
            case S_IFREG: // 常规文件 inode 的处理
                inode->i_op = &ramfs_file_inode_operations;
                inode->i_fop = &ramfs_file_operations;
                break;
            case S_IFDIR: // 目录 inode 的处理
                inode->i_op = &ramfs_dir_inode_operations;
                inode->i_fop = &simple_dir_operations;
                inc_nlink(inode);
                break;
            case S_IFLNK: // 符号连接 inode 的处理
                inode->i_op = &page_symlink_inode_operations;
                break;
        }
    }
}

```

```

    return inode;    // 返回所建立的 inode
}

```

对于 `inode` 的建立，主要就是要设置其中的索引节点操作 `inode_operations` 和文件操作 `file_operations`。常规文件的两个操作各有不同的实现；目录的索引节点操作有特定实现，文件操作则使用默认的实现；对于符号连接只有索引节点操作，文件操作来自连接到的文件；对于其他类型，用系统的默认实现。`init_special_inode()`用于建立字符设备、块设备、FIFO 设备和 `Socket` 设备几种特殊文件类型的默认操作。

用于目录操作的索引节点操作的 `ramfs_dir_inode_operations` 结构如下所示。

```

static const struct inode_operations ramfs_dir_inode_operations = {
    .create      = ramfs_create,    // inode 的创建函数
    .lookup      = simple_lookup,
    .link        = simple_link,
    .unlink      = simple_unlink,
    .symlink     = ramfs_symlink,   // 建立 inode 软连接的函数
    .mkdir       = ramfs_mkdir,     // 建立 inode 目录的函数
    .rmdir       = simple_rmdir,
    .mknod       = ramfs_mknod,     // 建立 inode 节点的函数
    .rename      = simple_rename,
};
static int

```

`ramfs_dir_inode_operations` 当中只有少数几个函数指针是 `ramfs` 的特定实现，其他均使用系统的默认实现。

用于建立节点的 `ramfs_mknod()`函数如下所示：

```

static ramfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
{
    struct inode * inode = ramfs_get_inode(dir->i_sb, dir, mode, dev);
    int error = -ENOSPC;    // 如果没有建立，就返回没有磁盘空间的错误
    if (inode) {
        d_instantiate(dentry, inode);
        dget(dentry); /* Extra count - pin the dentry in core */
        error = 0;
        dir->i_mtime = dir->i_ctime = CURRENT_TIME; // 设置目录的更新时间
    }
    return error;
}

```

`ramfs_mknod()`函数的主体也是调用 `ramfs_get_inode()`函数实现，调用完成之后，还需要设置这个节点所在目录的信息。

另外的几个函数的实现如下所示：

```

static int ramfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
    int retval = ramfs_mknod(dir, dentry, mode | S_IFDIR, 0);
    if (!retval)
        inc_nlink(dir);
    return retval;
}
static int ramfs_create(struct inode *dir, struct dentry *dentry,
                        int mode, struct nameidata *nd)

```

```
{
    return ramfs_mknod(dir, dentry, mode | S_IFREG, 0);
}
static int ramfs_symlink(struct inode *dir, struct dentry *dentry, const char *symname)
{
    struct inode *inode;
    int error = -ENOSPC;
    inode = ramfs_get_inode(dir->i_sb, dir, S_IFLNK|S_IRWXUGO, 0);
    if (inode) {
        int l = strlen(symname)+1;
        error = page_symlink(inode, symname, l);
        if (!error) {
            d_instantiate(dentry, inode);
            dget(dentry);
            dir->i_mtime = dir->i_ctime = CURRENT_TIME; // 设置时间
        } else
            iput(inode);
    }
    return error;
}
```

ramfs_create() 函数和 ramfs_mkdir() 函数都是通过 ramfs_mknod() 函数实现的；ramfs_symlink() 的实现稍微复杂一些，其通过调用 ramfs_get_inode() 函数，并传入 S_IFLNK 类型参数实现。

实际上，RAMFS 的各种操作最后基本都归结到文件系统的通用实现上。文件系统特定的实现只是处理了少量附加信息。

表示 RAMFS 中常规文件的索引节点操作和文件操作在 file-mmuc 当中定义，如下所示：

```
const struct file_operations ramfs_file_operations = {
    .read      = do_sync_read,
    .aio_read  = generic_file_aio_read,
    .write     = do_sync_write,
    .aio_write = generic_file_aio_write,
    .mmap      = generic_file_mmap,
    .fsync     = noop_fsync,
    .splice_read = generic_file_splice_read,
    .splice_write = generic_file_splice_write,
    .llseek    = generic_file_llseek,
};
const struct inode_operations ramfs_file_inode_operations = {
    .setattr    = simple_setattr,
    .getattr    = simple_getattr,
};
```

这里的几个实现函数都是通用的实现。Linux 文件系统的核心提供了多个通用实现的函数，每种文件系统可以不同的实现来完成自己的功能。例如：对于此处 RAMFS 的实现，由于文件不存在向真正磁盘同步的问题，因此用于同步的 fsync 就不用实现。

8.2.3 默认的公共实现

fs 目录中的 libfs.c 文件实现了一些简单的操作函数指针，这些实现通常以 simple_ 为函数开头，可以作为通用的实现来使用。inode.c 文件是 inode 的通用实现。

fs 目录当中，还为不同类型文件实现了不同的独立文件操作和不同的 file_operations，

如下所示。

- **read_write.c**: 普通文件的实现，有众多以 `generic_` 为开头的函数，作为文件的通用操作实现，`generic_ro_fops` 实现了作为通用只读文件的文件操作。
- **seq_file.c**: 以 `seq_` 为开头的函数作为顺序读写文件的实现。
- **fifo.c**: FIFO 的实现，`fifo_open()` 函数为从 `inode` 打开得到 FIFO 的打开函数，`def_fifo_fops` 实现了作为默认的管道文件的文件操作。
- **pipe.c**: 管道的实现，`read_pipefifo_fops` 实现了作为可读的管道文件的文件操作；`write_pipefifo_fops` 实现了作为可写的管道文件的文件操作；`rdwr_pipefifo_fops` 实现了作为可读可写的管道文件的文件操作。
- **char_dev.c**: `def_chr_fops` 实现了作为默认的字符设备文件的文件操作。
- **block_dev.c**: `def_blk_fops` 实现了作为默认的块设备文件的文件操作。

例如：`read_write.c` 文件当中实现 `file_operations` 的 `llseek` 如下所示：

```
loff_t generic_file_llseek_unlocked(struct file *file, loff_t offset, int origin)
{
    struct inode *inode = file->f_mapping->host; // 从 file 得到 inode
    switch (origin) { // 区分参数中的 SEEK_SET (首)、SEEK_CUR (当前)、SEEK_END (尾)
    case SEEK_END:
        offset += inode->i_size; // 偏移量加上文件大小得到绝对偏移量
        break;
    case SEEK_CUR:
        if (offset == 0)
            return file->f_pos;
        offset += file->f_pos; // 偏移量加上当前的位置得到绝对偏移量
        break;
    }
    if (offset < 0 || offset > inode->i_sb->s_maxbytes) // 大于超级块定义的文件大小
        return -EINVAL;
    if (offset != file->f_pos) {
        file->f_pos = offset; // 设置文件中的位置
        file->f_version = 0;
    }
    return offset;
}
EXPORT_SYMBOL(generic_file_llseek_unlocked);
loff_t generic_file_llseek(struct file *file, loff_t offset, int origin)
{
    loff_t rval;
    mutex_lock(&file->f_dentry->d_inode->i_mutex); // 对打开的文件上锁
    rval = generic_file_llseek_unlocked(file, offset, origin);
    mutex_unlock(&file->f_dentry->d_inode->i_mutex); // 对打开的文件解锁
    return rval;
}
EXPORT_SYMBOL(generic_file_llseek);
```

在以上操作中，从 `file` 结构中得到 `inode` 结构，然后得到文件在文件系统的大小，之后再设置 `file` 结构中的位置 (`f_pos`)。考虑上锁情况的 `generic_file_llseek()` 函数通过调用不考虑上锁的 `generic_file_llseek_unlocked()` 函数来实现。

8.3 几种 Linux 使用的文件系统

在基于 Linux 的嵌入式系统中，常见的文件系统有以下几种：

- EXT2/3（Extended File System 2/3，扩展文件系统 2/3）。
- ROMFS（Read Only Memory File System，只读文件系统）。
- CRAMFS（Compressed ROM File System，压缩 ROM 文件系统）。
- JFFS2（Journaling Flash File System 2，日志 Flash 文件系统 2）。
- NFS（Network File System，网络文件系统）。
- YAFFS（Yet Another Flash File System，另一种 Flash 文件系统）。
- UBIFS（Unsorted Block Image File System，非排序块映像文件系统）。

8.3.1 EXT 2/3（扩展文件系统 2/3）

EXT2（Second Extended File System，扩展文件系统 2）是目前 Linux 操作系统上使用的一种高性能的磁盘文件系统。它可以支持 4TB 的硬盘分区和 256B 的长文件名，并且在速度和 CPU 利用方面有突出表现，是一种性能优异的文件系统。与大多数文件系统一样，EXT2 文件系统建立在文件数据存放于数据块的前提上。这些数据块都是相同长度的。虽然不同 EXT2 文件系统的块长度可以不同，但是对于一个特定的 EXT2 文件系统，它的块长度在创建的时候就确定了。

EXT3 是一种日志式文件系统，是对 EXT2 系统的扩展，它兼容 EXT2。所谓日志文件系统，就是可以将文件系统改动（例如写入）的过程记录在文件系统的介质当中。这样当由于外部因素（比如断电）造成文件系统写入中断的时候，可以根据记录重整文件系统。

事实上，EXT2/3 文件系统具有稳定性、可靠性和健壮性的特点，适合基于 Linux 的桌面系统、工作站、服务器使用。但是，EXT2/3 文件系统本身是基于 IDE 硬盘设计的。如果在嵌入式系统常用的 Flash 存储介质中使用，则在扇区管理、平衡损耗方面存在缺陷。因此，它在嵌入式系统中的使用并不是很广泛。

EXT 文件系统在 Linux 内核的实现代码在 fs/ext2、fs/ext3、fs/ext4 中。

8.3.2 NFS（网络文件系统）

NFS（网络文件系统）是一种以网络协议为基础的文件系统。其他的文件系统一般是基于存储器（RAM 或者 ROM）或者磁盘构建的，但是 NFS 是利用网络实现的文件系统。

NFS 文件系统由原 Sun 公司发展的，并于 1984 年推出。NFS 是一个 RPC service，它可以实现文件的共享。NFS 的设计是为了在不同的系统间使用，所以它的通信协议设计与主机及作业系统无关。当使用者想用远端文件的时候，只要用“mount”就可把远程的文件系统挂接在自己的文件系统之下，使得远端文件在使用上和本机器的文件一样。

NFS 文件系统的目的就是促使服务器上的文件能被其他的机器访问，从而可以资源共享。使用这些文件的机器被称为 NFS 客户端，一个 NFS 客户端可以从 NFS 服务器上 mount

一个文档或一个目录。实际上，任何一台机器都可以做 NFS 服务器或 NFS 客户端，甚至可以同时作为 NFS 服务器和 NFS 客户端。

在嵌入式系统中，网络文件系统常用于系统的调试阶段。在调试的时候，可以将目标机执行的程序放在主机上，并且在主机上开设 NFS 服务器；在目标机上，通过网络挂接主机的 NFS 中的目录。这样，目标机可以直接运行主机上的程序，就像将程序放入目标机一样。利用网络文件系统，甚至可以作为目标机的根文件系统。

NFS 文件系统在 Linux 内核的实现代码在 fs/nfs、fs/nfscommon 中。

8.3.3 ROMFS（只读文件系统）

ROMFS 是一个轻量级的文件系统，常在嵌入式设备上使用，具备体积小、可靠性好、读取速度快等优点。ROMFS 也支持目录、符号链接、硬链接、设备文件。ROMFS 的局限性在于，这是一种只读的文件系统，而且最大文件不超过 256MB。

ROMFS 对存储空间的节约来自于两个方面：首先，内核支持 ROMFS 文件系统比支持 EXT2 文件系统需要更少的代码；其次 ROMFS 文件系统相对简单，在建立文件系统超级块（superblock）时需要更少的存储空间。

事实上，正是由于 ROMFS 使用了简化的顺序存储方式，因此其数据本身就是连续存放的，读取效率较高，但是显然不能支持更改。ROMFS 也没有使用支持压缩，由此也支持在 Nor Flash 存储设备上直接运行程序（XIP 运行方式）。

使用 genromfs 工具创建一个 ROMFS 类型的文件系统映像。

```
$ genromfs [OPTIONS] -f IMAGE
```

例如：要将目录 src 压缩为映像 romfs.img，可以使用如下命令：

```
$ genromfs -d src -f romfs.img
```

ROMFS 文件系统在 Linux 内核的实现代码在 fs/romfs 中。

8.3.4 CRAMFS（压缩 ROM 文件系统）

CRAMFS（Compressed ROM File System）是一个简单、压缩和只读的文件系统。CRAMFS 通过优化索引节点表的尺寸和除去传统文件系统中文件之间的空间浪费，从而实现节约空间的目的。CRAMFS 还使用了 zlib 压缩，实现优于 2:1 的压缩比例。

CRAMFS 文件系统还具有以下一些特点：

- 支持实时解压缩，但是解压缩有延迟。
- 文件的最大尺寸为 16MB，文件系统的最大尺寸是 256MB。
- 在目录中，没有表示当前目录的节点（.）和表示父目录的节点（..）。
- 16 位的文件 UID 域和 8 位的 GID 域。
- 要求内核使用 4096B 的内存页面大小（PAGE_CACHE_SIZE 的值为 4096B），写文件系统的机器和读文件系统的机器必须具有同样的大小端（endian）。
- 支持硬连接，但是所有文件的链接计数均固定为 1。

在 CRAMFS 中, 不会保存文件的时间戳(timestamps)信息。正在使用的文件由于 inode 保存在内存中, 因此它的时间可以暂时地变更为最新时间, 但是不会保存到 CRAMFS 文件系统中去。

CRAMFS 文件系统是用于保存只读的根文件系统内容的一个很好的方案。CRAMFS 主要的优点是将文件数据以压缩形式存储, 在需要运行的时候进行解压缩。由于它存储的文件形式是压缩的格式, 所以文件系统不能直接在 Flash 上运行。虽然这样可以节约很多 Flash 存储空间, 但是文件系统运行需要将大量的数据拷贝进 RAM 中, 造成一定的浪费。

在嵌入式系统中, CRAMFS 经常与 RAMDisk 协同使用。常把启动代码压缩存储到 CRAMFS 中, 在启动后将内容解压缩到 RAMDisk 中运行。作为压缩式的文件系统, CRAMFS 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, 只是在系统需要访问某个位置的数据的时候, 立刻计算出该数据在 CRAMFS 中的位置, 实时地将其解压缩到内存之中, 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。对于用户, 这个过程是透明的, 并不需要了解具体的实现过程。

CRAMFS 文件系统在 Linux 内核的实现代码在 fs/cramfs 中。

8.3.5 JFFS2 (日志 Flash 文件系统)

JFFS (Journaling Flash File System, 日志文件系统) 是开源的日志文件系统, 它在设计时充分考虑了 Flash 的读写特性和电池供电的嵌入式系统的特点。

瑞典 Axis 通信公司在 2000 年发布了 JFFS。2001 年, Red Hat 公司的 David Woodhouse 在 JFFS 基础上进行了改进, 并将之称为 JFFS2。JFFS2 直接在 Flash 上实现, 作为用于嵌入式设备的基于原始 Flash 芯片的文件系统的实现。

Linux 2.4 内核开始支持 JFFS2 文件系统, 它采用了成熟稳定的 MTD 技术, 因此要比 JFFS 稳定。JFFS2 文件系统是日志结构化的, 这意味着它基本上是一长列节点。每个节点包含有关文件的部分信息: 可能是文件的名称, 也可能是一些数据。

JFFS2 文件系统是专门为像 Flash 芯片那样的嵌入式设备创建的, 所以其整个设计在 Flash 管理方面更为优秀。JFFS 和 JFFS2 文件系统是特别为 Flash 存储器设计的, 它们都具有一种被称为“损耗平衡”的特点, 也就是说 Flash 的所有被擦写的单元都保持相同的擦写次数。利用这种特有的保护措施, Flash 的使用周期得到相当大的提升。JFFS2 使用了压缩的文件格式, 为 Flash 节省了大量的存储空间, 它更优于 JFFS 格式在系统中的使用。

JFFS2 文件系统在 Linux 内核的实现代码在 fs/jffs 中。

8.3.6 YAFFS (另一种 Flash 文件系统)

YAFFS (Yet Another Flash File System, 另一种 Flash 文件系统) 是嵌入式系统中一种针对 Nand Flash 的日志文件系统。

与 JFFS/JFFS2 非常类似, YAFFS 是专门为 Nand Flash 设计的嵌入式文件系统, 适用于大容量的存储设备。它是日志结构的文件系统, 提供了损耗平衡和掉电保护, 可以有效

地避免意外掉电对文件系统一致性和完整性的影响。

YAFFS 文件系统是按层次结构设计的,分为文件系统管理层接口、YAFFS 内部实现层和 Nand 接口层。这样就简化了其与系统的接口设计,可以方便地集成到系统中去。与 JFFS 相比,它减少了一些功能,因此速度更快,占用内存更少。

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

YAFFS 目前已经移植到 WinCE 和 Linux 中。此外,YAFFS 可以方便地移植到各种 RTOS 和嵌入式 OS 中,也支持 Nor Flash 和 RAM。

提示: Linux 内核中默认没有包含 YAFFS 的支持,需要单独增加代码。

YAFFS 源代码包内除了本身文件系统代码外,utils 目录下还包含了 mkyaffsimage 和 mkyaffs2image 的代码,这是两个可执行程序。其中,mkyaffsimage 用于制作 512 B 的小页 YAFFS 文件系统,mkyaffs2image 用于制作 2KB 以上的大页 YAFFS 文件系统。输入以下格式命令,制作出支持大页的 YAFFS2 文件系统映像。

mkyaffs2image 的使用方法如下所示:

```
mkyaffs2image: image building tool for YAFFS2
usage: mkyaffs2image [-f] dir image_file [convert]
        -f          fix file stat (mods, user, group) for device
        dir         the directory tree to be converted
        image_file  the output file to hold the image
        'convert'   produce a big-endian image from a little-endian machine
```

mkyaffs2image 需要指定一个目录 (dir) 和映像文件 (image_file) 的名称,即可以生成 YAFFS 文件系统。

8.3.7 UBIFS (非排序块映像文件系统)

UBIFS 的全称是 Unsorted Block Image File System,是一种 Flash 的文件系统,在较新的 Linux 内核 (Linux 2.6.27) 加入。UBIFS 不是像 Linux 的传统文件系统一样基于块设备,也就是不会去操作块设备。传统的文件系统实现了 FTL (Flash Translation Layer, Flash 转换层),将 Flash 表现成块设备的形式,而 UBIFS 基于原始的 Flash。

UBIFS 是对传统的 Flash 文件系统 JFFS2、YAFFS2 的改进。因此,UBIFS 有时被称为下一代的 JFFS2。UBIFS 在内存消耗、Flash 的均衡写等方面有所改进。JFFS2 工作在 MTD 设备层;而 UBIFS 依赖 UBI 卷 (volumes)。JFFS2 是写穿 (write-through) 方式,也就是不使用缓存直接写入;而 UBIFS 是写回 (write-back) 方式,使用缓存优化写入。UBIFS 基于 UBI 层,在用户空间中,通常表现为 drivers/mtd/ubi 设备。

挂接一个 UBIFS 的 0 号分区方式如下所示:

```
$ mount -t ubifs ubi0_0 /mnt/ubifs
```

挂载 UBIFS 的根分区的方式如下所示：

```
$ mount -t ubifs ubi0:rootfs /mnt/ubifs
```

UBIFS 在 Linux 内核的实现代码在 fs/ubifs 中。

8.4 Linux 文件系统的构建

8.4.1 根文件系统的结构

根文件系统是 Linux 内核启动后挂载（mount）的文件系统。它包含内核可能需要使用的文件和可执行文件，还有用于系统管理的可执行文件。Linux 内核在启动的时候将挂载根文件系统。

挂载完成后，Linux 将拥有根文件系统的目录结构，此时，Linux 可以执行根文件系统的二进制代码。Linux 系统还可以将其他文件系统挂载根文件系统到某个目录中。例如：在默认情况下，Linux 内核会寻找/bin 等目录中名为 init 的可执行程序，作为用户空间中运行的第一个进程。

Linux 操作系统的根文件系统以树的结构方式组织，最上层的目录名为“/”，其下面的顶层目录多为有不同意义和用途的系统目录。典型的根文件系统目录结构如下所示。

- /bin/：包含基本的用户命令工具程序。
- /sbin/：包含基本的系统管理程序。
- /boot/：包含内核映像及启动相关文件。
- /etc/：（executive time config）系统运行时配置文件。
- /lib/：包含系统库和内核模块。
- /usr/：用户程序及库目录。
- /home/：用户主目录。
- /root/：root 用户主目录。
- /dev/：设备文件目录，目录下的每个文件代表一个设备。
- /mnt/：文件系统临时挂装目录。
- /var/：包含运行时改变的文件，例如 lock 和 log 文件。
- /proc/：存放运行时系统信息的文件系统。
- /tmp/：临时文件目录。

一个嵌入式 Linux 系统的根文件系统没有如此复杂，因为它通常不支持多用户登录，功能和用途相对简单。一般来讲，只有/bin、/sbin、/etc、/lib、/dev、/usr、/var 和/proc 目录及其下的内容是需要。而且，嵌入式 Linux 根文件系统的组成比较灵活，开发者可以根据项目的实际情况来决定采用什么样的结构来组织文件系统的内容。

在 Linux 文件系统中，有一些文件只具有操作系统的节点，并不具有实际的存储数据。如 dev 下的目录文件、proc 下的系统信息。

主机到目标机的文件系统构建如图 8-3 所示。

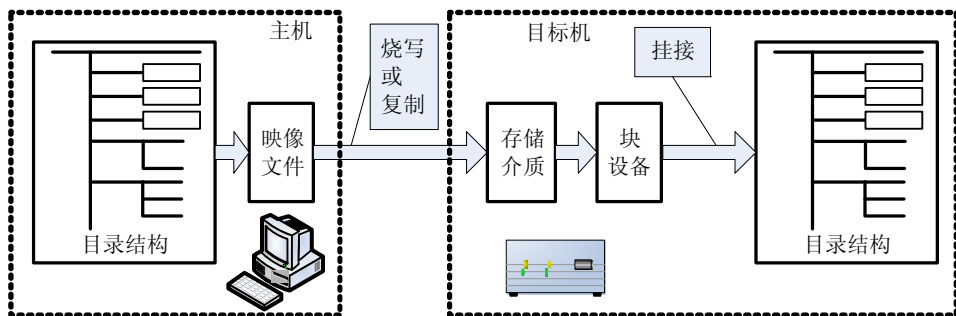


图 8-3 主机到目标机的文件系统构建

主机到目标机的文件系统构建包括了以下几个步骤：

- 主机构建目标机文件系统目录结构。
- 目录结构生成某种格式的映像文件。
- 映像文件烧写或复制到目标机的磁盘等存储介质。
- 目标系统将介质上的内容表示为块设备。
- 目标系统将块设备挂载成某种格式的文件系统。

一般情况下，主机文件系统的目录结构都需要生成映像文件，映像文件烧写到目标机上，目标机再将烧写的映像挂载成文件系统。只有使用 NFS 的时候，不需要生成映像文件。

在嵌入式 Linux 系统中，需要精简根文件系统，来适应嵌入式系统特定应用的需要。所有适合提供给多用户扩展环境的目录，如/home、/mnt、/opt 和/root，可以被忽略。甚至可以进一步删除/tmp 和/var 以精简根文件系统，但这些裁剪有可能会使某些程序不能正常执行。因此，除非清楚地知道所做的这些操作，否则不要轻易删掉/tmp 和/var 目录。目标系统可能不需要/boot 目录，这与 BootLoader 及其配置有关。这依赖于在目标系统的内核启动之前，引导装载机（BootLoader）能否从根文件系统中找到内核映像。

8.4.2 制作根文件系统映像

在 Linux 的主机中可以使用 `mkfs.<type>` 命令来制作文件系统，例如 `mkfs.cramfs`、`mkfs.ext2`、`mkfs.vfat` 等，`mke2fs` 和 `mkcramfs` 等命令也与之类似。

例如：创建一个 `cramfs` 格式的映像文件：

```
$ mkfs.cramfs ${dir}/ root.img
```

`mkfs.ext2` 可以直接在映像文件中建立文件系统。可以先使用 `dd` 生成空文件，然后再建立文件系统：

```
$ dd if=/dev/zero of=root.img bs=1024 count=4096
$ mkfs.ext2 root.img
```

创建后可以使用回环的方式把映像文件挂接到文件系统中：

```
$ mount -o loop root.img /mnt/
```

这样可以在挂接的目录中，复制和修改内容，然后使用 `umount` 卸载文件系统：

```
$ umount /mnt/
```

卸载后，内容将保存到文件系统的映像中。

8.4.3 内核启动中根文件系统的参数

Linux 操作系统中包含了文件系统的操作，内核启动后需要加载根文件系统（rootfs）。除了根文件系统外，Linux 操作系统可以支持多个文件系统。

Linux 启动参数也用于指定内核启动时使用的根文件系统。

内核启动参数（bootargs）设置 root（根设备），通常为一个块设备。有一些特殊的量，如果设置为 `root=/dev/nfs`，表示使用 NFS 作为根文件系统（需要指定 IP 地址，`nfsroot=`）；如果设置为 `root=/dev/ramX`，表示使用 RAMDisk 构建根文件系统。

例如，以下设置表示将 `initrd` 作为内存盘，启动根文件系统。

```
root=/dev/ram0 console=ttyS0 initrd=0xa0800000,0x00800000 rw mem=64M
```

其中，指定的 `root=` 为根文件系统的设备名称。

在通常情况下，如果通过 u-boot 启动 Linux 内核，则可以通过在 u-boot 中设置 bootargs 参数来完成。其中，设置根文件系统的选项是通过指定 `root=` 的内容。

例如：通过 NFS（网络文件系统）作为根文件系统的启动方式的 u-boot 启动参数如下所示。

```
# setenv bootargs=console=ttyS2,115200n8=noinitrd mem=64M root=/dev/nfs rw
nfsroot=192.168.0.1:/nfs/,ip=192.168.0.2 init=/init
```

这里指定了 `root=/dev/nfs` 表示使用 NFS 作为根文件系统，使用 `nfsroot=192.168.0.1` 指定根文件系统的路径。

通过 Flash 的 MTD 分区作为根文件系统的启动方式的 u-boot 启动参数如下所示：

```
# setenv bootargs=console=ttyS2,115200n8 mem=64M rdinit=/init mtdparts=nand:8m@8m(kernel),
32m@16m(user)
```

如果使用 Flash 的 MTD 分区作为根文件系统，则需要指定根文件系统的分区情况。在以上的内容中，`32m@16m(user)` 表示分区的名称为 `user`，从 Flash 的地址 16M 中开始，包含 32M 的大小。这个配置的内容需要和真实的 Flash 分区中的内容相匹配。也就是说在真实 Flash 的 16M 开始的 32M 的内容，应该烧写对应 `user` 这个文件系统的映像。

通过 MMC 或者 SD 卡作为根文件系统的启动方式的 u-boot 启动参数如下所示：

```
# setenv bootargs console=ttyS2,115200n8 rw root=/dev/mmcblk0p1 mem=128M devfs=mount
rootdelay=1 init=/init
```

这里指定的 `root=/dev/mmcblk0p1`，表示根文件系统使用 MMC 块设备的第一个分区。

第 9 章

Linux 用户空间的核心

9.1 嵌入式系统中的操作系统和系统关系

嵌入式操作系统核心部分的内容与通用操作系统类似，包含了进程调度、进程通信、内存管理、设备管理等部分。嵌入式操作系统通常包括移植层和驱动程序，它们是嵌入式操作系统适配各种不同硬件的接口。

嵌入式的完整系统通常具有操作系统、中间件和应用程序层这几个层次，如图 9-1 所示。

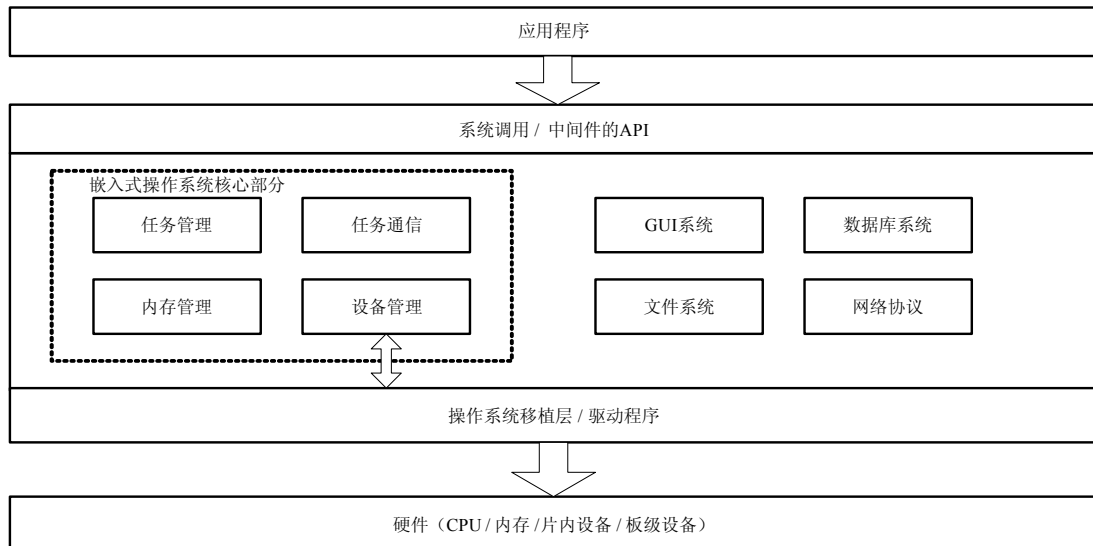


图 9-1 嵌入式系统的层次架构

嵌入式操作系统中的扩展模块通常有文件系统、网络协议、GUI 系统、数据库系统等模块。它们的功能和通用操作系统类似，但是每种嵌入式操作系统包含的内容不定。如果

不具有但需要，则作为中间件实现。中间件是整个系统中承上启下的重要软件层。根据操作系统的不同，中间件的实现层次不同，有些是在操作系统提供的，有些则不是。

对于 Linux 系统，内核当中包含了文件系统部分和网络部分，但是不包括 GUI 等部分。GUI 和其他各种中间件都在用户空间提供。中间件提供的就是软件层的 API。

如果需要基于 Linux 操作系统，构建一个完整的系统，在技术上具有以下内容：

- Linux 的内核及其硬件相关的移植。
- 各种设备的驱动程序。
- 用户空间 C 库、Shell。
- 用户空间的各种中间件。
- 用户空间的应用程序。

在特定的开发环境中，将内核空间和用户空间的各个部分整合之后，并经过优化，才能完成基于 Linux 的完整系统。

Linux 系统的用户空间基于 Linux 操作系统，内核空间提供给用户空间的接口是用户空间的基础。嵌入式 Linux 系统内核空间和用户空间的接口包括了以下几个部分。

- 系统调用：POSIX 标准的系统调用。
- 基于设备节点：包括字符设备和块设备，通常作为硬件相关的特殊接口。
- 基于 Socket 的网络：可以在内核中通过网络协议结构自定义。
- 基于 proc 或者 sys 文件系统：Linux 提供的特殊文件系统。

系统调用是所有 UNIX 系统中近似的内容，但不同体系结构略有不同；其他的几种内核空间到用户空间的接口，则是 Linux 系统特有的。

提示：Linux 用户空间的程序中 C 库和 Shell 程序通常是最基础的。

9.2 C 语言库

C 语言库（简称 C 库）是 C 语言程序的运行基础，所有的 C 语言程序都需要依赖 C 语言函数库。C 库建立于操作系统的系统调用之上，提供 C 语言的函数接口。不同的体系结构中，C 语言函数库也是不相同的。在 Linux 当中，用户空间编程的基础也是 C 语言库。

C 语言库本身基于系统调用实现，对上层提供库接口，其中的关系如图 9-2 所示。

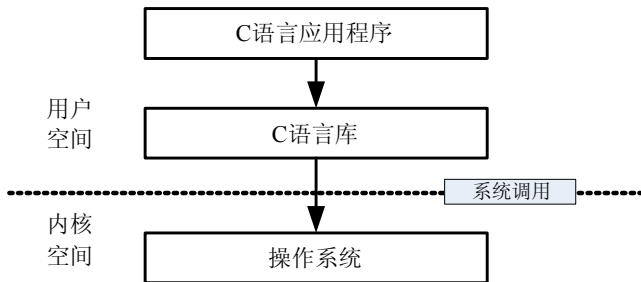


图 9-2 C 语言库与操作系统的关系

C 语言库提供给应用程序一些 C 语言标准的函数接口，它本身是一个函数库。C 语言编程当中很多的功能靠 C 语言库中的函数直接实现，如字符处理函数 `strcpy()`、`strcmp()` 等。C 语言库中的另外一些函数需要调用操作系统的系统调用来实现，例如：在 C 语言库中实现的 `printf()` 函数需要向标准输出设备输出字符，这显然需要硬件的支持。由于 C 语言库不需要知道系统的硬件情况，调用 `write` 这个系统调用即可。

提示：C 语言库本身很大程度上也是由 C 语言源程序实现的。

C 语言库作为操作系统和应用程序间统一的接口，其实现方式可能不同。在桌面 Linux 中通常使用 `glibc`，而在嵌入式 Linux 中通常使用 `uclibc`。

`glibc` 是一种按照 LGPL 许可协议发布的 C 函数库，是程序运行时使用到的一些 API 集合，它们一般是已预先编译好，以二进制代码形式存在于 Linux 类系统中。`glibc` 通常作为 GNU C 编译程序的一个部分发布。它最初是自由软件基金会为其 GNU 操作系统所写，但目前最主要的应用是配合 Linux 内核，成为 GNU/Linux 操作系统的一个重要组成部分。

`glibc` 的网址：<http://www.gnu.org/s/libc/>。

`uclibc` 是一个面向嵌入式 Linux 系统的小型 C 标准库。最初 `uclibc` 是为了支持 `uClinux` 而开发的。`uClinux` 是一个不需要内存管理单元的 Linux 版本，因此可以在微控制器系统（没有内存管理单元 MMU 的嵌入式处理器）中使用。

`uclibc` 的网址：<http://www.uclibc.org/>。

提示：`uclibc` 和 `uClibc` 是嵌入式 Linux 中使用的两个 C 语言库。`uClibc` 的出现更早，更轻量级，其中有些接口不是标准的。`uclibc` 则更好地考虑了和 `glibc` 的兼容问题，并且具有更多的功能。

在嵌入式系统的开发过程中，使用 C 语言库有两种方式：一种方式是直接使用交叉编译工具中自带的 C 语言库，另外一种方式就是从源代码生成 C 语言库。相比 PC 的 Linux 上常用的 `glibc` 库，嵌入式系统使用得更多的是 `uclibc` 库。

`uclibc` 在经过编译之后，可以生成二进制 C 语言库，然后根据二进制 C 语言库开发各种用户空间的应用程序。

首先将 `uclibc` 的源代码包（例如：`uClibc-0.9.28.3.tar.tar`）解压缩到某个目录，在这个目录中运行配置程序：

```
# make CROSS=<cross-prefix> menuconfig
```

`<cross-prefix>` 表示检查编译工具链的前缀。由于需要生成 ARM 版本的 C 语言库，因此可以使用 ARM 的编译作为交叉编译工具（例如：以 `arm-linux-` 为前缀）。执行完 `menuconfig` 之后，将弹出 `uclibc` 配置界面。

`uclibc` 配置主界面如图 9-3 所示。

`uclibc` 的配置选项较少，一般情况下使用默认的即可。一般情况下，需要进行配置的是 Target Architecture Feature and Options（目标体系结构特性和选项）。选中这个选项之后，配置的界面如图 9-4 所示。

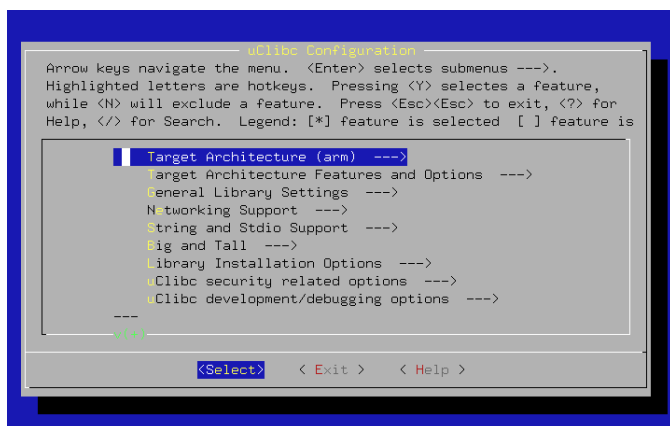


图 9-3 μlibc 配置主界面

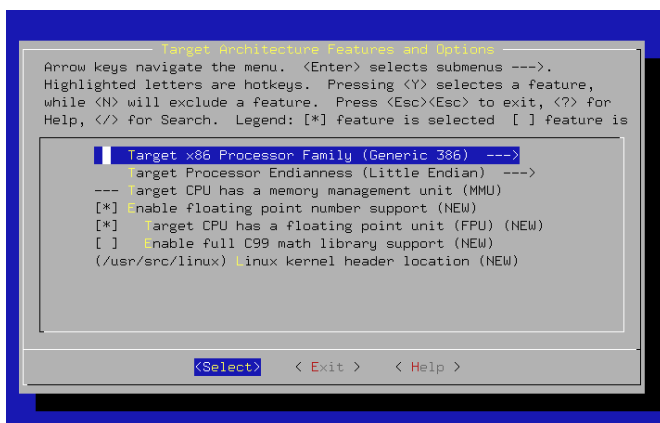


图 9-4 μlibc 体系结构特性和选项配置主界面

在 Target Processor Family（目标处理器家族）中，需要选择 ARM 处理器家族。在目标处理器端模式（Target Processor Endianness）中，各具体系统处理器的情况选择，对于应用的 ARM 处理器，一般使用小端模式（Little Endian）。

另一个重要的选项是 Linux 内核的头文件，即 Linux Kernel header location 选项。选择对话框如图 9-5 所示。

Linux 内核的头文件需要填入一个目录，这个目录是 Linux 内核源文件的根目录。编译的时候将使用这个文件夹中的头文件。事实上，C 语言库本身需要调用 Linux 操作系统的接口实现某些功能。这里面需要填入的目录必须和本系统所编译 Linux 内核的源代码一致。

由于嵌入式 Linux 系统组成的映像包括操作系统内核映像和文件系统映像两部分，因此，如果使用编译好的 C 语言库也需要放入文件系统映像中。C 语言库本身需要依赖 Linux 操作系统内核，因此在一个嵌入式系统构建的时候，C 语言库执行的系统调用必须与内核中的实现相一致。这样才能保证文件系统上的 C 语言库和操作系统的内核相匹配。

提示：由于需要依赖 Linux 内核中的头文件，因此 C 库的编译通常比较烦琐。

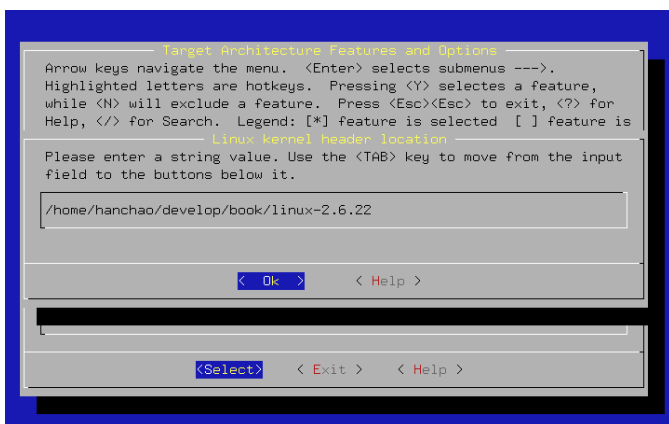


图 9-5 Linux 内核目录

编译完成后，可以使用 `make install` 将目标安装到某个文件夹中：

```
# make PREFIX=<path> install
```

其中，`<path>`是需要安装的目录，一般情况下，默认的安装目录为`/usr/arm-linux-uclibc/`。

在这个目录中包含 `lib` 和 `usr` 两个文件夹。在 `lib` 中，包含 `ld-uClibc-<version>.so` 等函数库和一些连接文件；在 `usr` 的 `include` 目录中包含 C 语言库的头文件，在 `usr` 的 `lib` 目录包含一些连接。

这些文件生成 ARM 体系结构的 `uclibc` 库文件，这些 C 语言库文件需要和交叉编译工具结合使用。

9.3 Shell 工具 Busybox

在嵌入式 Linux 系统中，用户终端（Shell）是操作系统之外的最基本软件，终端程序提供一个可以进行人机交互的界面，包含了 Linux 中常用的命令。尤其在开发调试阶段，用户终端在嵌入式系统中起到了很重要的作用。相比 Linux 中的各种中间件，用户终端更像一个工具。这个模块通常要包含一个可执行程序。

在 Linux 中，命令行工具通常需要建立在一个系统的标准终端中，这个终端通常是一个 Linux 中的 `tty` 设备，用户终端程序使用这个设备作为输入/输出的基础，并提供一个基本的界面（通常是一个提示符，表示为`#`或者`$`）。一些比较好的终端可以实现历史记录、自动完成命令等功能。用户终端所支持的各种命令（比如 `mv`、`ls`、`cp`、`ifconfig` 等）可以通过调用 Linux 的系统调用实现，例如完成 `mkdir` 命令，可以通过 `mkdir` 函数完成。

嵌入式 Linux 系统 Shell 工具结构如图 9-6 所示。

在嵌入式系统的开发中，在开发的最初阶段就需要具有人机交互界面的程序（Shell）。嵌入式 Linux 的 Shell 为系统提供和桌面 Linux Shell 类似的功能，在开发调试的过程中起到非常重要的作用。

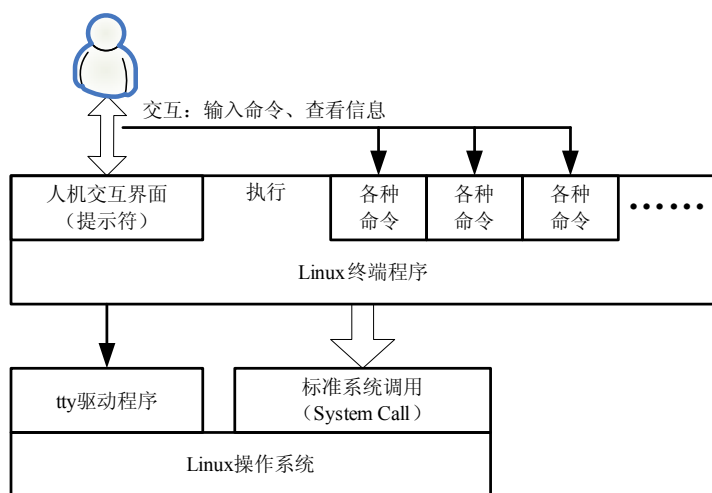


图 9-6 嵌入式 Linux 系统 Shell 工具结构

在嵌入式系统中，Busybox 是一个多功能应用软件，被称为“瑞士军刀”，意思是小巧但功能繁多，特别适合于嵌入式系统使用。Busybox 是一组小程序，可以提供一些在命令行使用的工具，实现 Shell 人机交互界面的功能，为任何一个小型的或者嵌入式系统提供了相当完整的环境。

BusyBox 把很多通用 UNIX 命令的微型版整合到一个很小的单一可执行文件中。这些命令通常可以用于替换 GNU 的 fileutils 和 shellutils 等中的大部分命令，一般情况下这些命令比对应的完整版 GNU 命令要少很多选项。

Busybox 的网站为 <http://busybox.net/>。

9.3.1 Busybox 配置和编译

解压缩 Busybox 软件包后，在根目录下执行 make 命令，出现 Busybox 的主配置菜单，如图 9-7 所示。

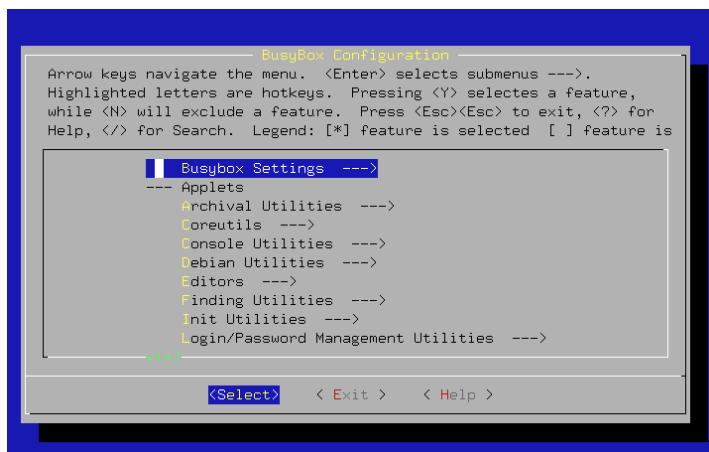


图 9-7 Busybox 的主配置菜单

在 Busybox 的配置菜单中，主要包含 Busybox 全局的设置（Busybox Settings）和小程序（Applets）两项。选择 Busybox Settings 后出现的界面如图 9-8 所示。

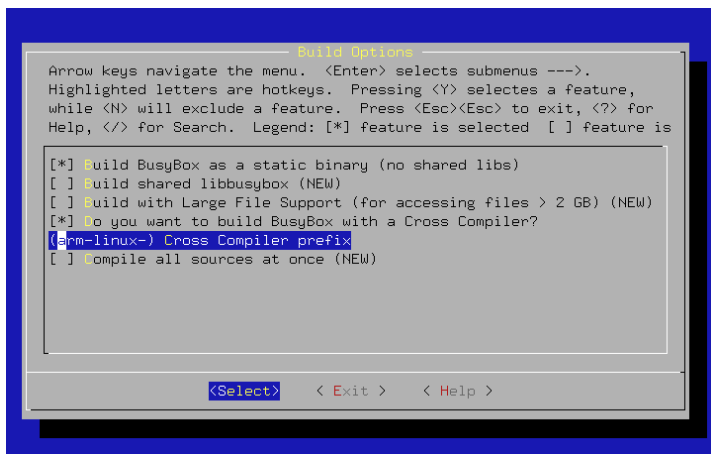


图 9-8 Busybox 的设置菜单

在 Busybox Settings 菜单中，主要需要选择的内容是 Build Busybox as a static binary，选择这个选项将把 Busybox 编成没有共享库的可执行程序。

提示：Busybox 需要依赖 C 语言函数库，在编译的时候选择静态库可以将 Busybox 所需要的 C 语言函数库和 Busybox 编译成一个可执行程序，这个可执行程序自己就可以运行，不需要再依赖 C 语言的函数库。

在交叉编译工具上，填入交叉编译工具的前缀（例如：arm-linux-）。

在小程序（Applets）选项中，可以根据需要增加命令程序，这些命令程序一般与 PC 上 Linux 的 Shell 的含义相同。

- Shells: Shell 工具本身的形式，一般选择 ash。
- Core Utilities（核心工具）：包括 cat、mv、ls、cp、mv 等命令，根据需要选择。
- Init Utilities（初始化工具）：用户空间的初始化程序 init。
- Process Utilities（进程工具）：包括 ps、kill、top 等。
- Networking Utilities（网络工具）：包括 ping、ifconfig、netstat、wget 等。
- Archival Utilities（归档工具）：包括 gzip、tar、rmp 等。
- Console Utilities（控制台工具）：包括 clear、setFont 等。
- Editor（编辑器）：包括 vi、patch、diff 等。
- Finding Utilities（查找工具）：包括 find、grep、xarg 等。
- Login/Password Management Utilities（登录和密码工具）：包括 login、passwd 等。
- Linux Module Utilities（Linux 模块工具）：包括动态的模块支持。
- Miscellaneous Utilities（杂项工具）：包括 man、time 等。
- Print Utilities（打印工具）：包括 lpd、lpr、lpq 等。

- Mail Utilities（邮件工具）：包括 mkmime、sendmain 等。
 - Runit Utilities（运行工具）：包括 runsv、sv、setuidgid 等。
 - System Logging Utilities（系统日志工具）：系统日志的支持。
- Busybox 各种工具的实现基本是和硬件无关的。

提示：init 要运行为用户空间的第一个进程，可以有不同的实现，例如：传统的 System V init, Ubuntu 的 upstart, RedHat 的 systemd, Busybox 可以提供嵌入式系统的 init。

编译 Busybox 的时候键入 make 即可：

```
$ make
```

Busybox 的编译结果将主要生成以下两个文件：busybox 和 busybox_unstripped，busybox_unstripped 是生成的可执行程序，busybox 是经过 strip（剥离）生成的可执行程序。可以通过 file 命令查看它们的属性：

```
$ file busybox busybox_unstripped
busybox:          ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,
statically linked, for GNU/Linux 2.0.0, stripped
busybox_unstripped: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux
2.0.0, statically linked, for GNU/Linux 2.0.0, not stripped
```

这两个文件都是静态连接的文件，一般使用的可执行程序是 busybox。由于剥离了符号，因此它占用的空间更小。

Busybox 在编译完成后，还可以进行安装，安装使用的命令如下所示：

```
$ make PREFIX=./target install
```

使用 PREFIX=选择安装的路径。如果不选择，默认的安装目录为 ./_INSTALL。

在安装目录下将生成 bin、sbin 和 usr 3 个目录，可以进一步根据这些目录和文件生成文件系统。注意，在 Busybox 的安装目录中，主要的内容为 busybox 可执行程序，在 bin 目录下包含的一些命令实际上是到 busybox 的连接。

提示：Busybox 安装的目的是构成一个基本的根文件系统目录。在不同嵌入式系统的构建中，也可以不使用此步骤。

9.3.2 Busybox 的源代码结构

Busybox 的源代码采用模块化的方式组织，各个模块位于不同的目录中。Busybox 的核心部分是 libbb（bb 的意思就是 Busy Box）。

Busybox 的全局头文件为 include/libbb.h，libbb 目录是 libbb.so 库的实现。

Busybox 的各个命令是单独实现的，核心的实现在 coreutils/目录中。其中的每一个文件都是一个命令的实现。各个命令都以 bb_xxx()为函数名。

例如 cat.c 文件为 cat 命令的实现，实现的内容如下所示：

```
#include "libbb.h"
int bb_cat(char **argv)
```

```

{
    int fd;
    int retval = EXIT_SUCCESS;
    if (!*argv)
        argv = (char**) &bb_argv_dash;           // 对参数的处理
    do {
        fd = open_or_warn_stdin(*argv);           // 以参数作为输入
        if (fd >= 0) {
            off_t r = bb_copyfd_eof(fd, STDOUT_FILENO); // 调用库中的内容
            if (fd != STDIN_FILENO)    close(fd);
            if (r >= 0)    continue;
        }
        retval = EXIT_FAILURE;
    } while (*++argv);
    return retval;
}
int cat_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE; // 声明入口函数
int cat_main(int argc UNUSED_PARAM, char **argv)
{
    getopt32(argv, "u");
    argv += optind;
    return bb_cat(argv);                          // 进行参数的处理
}

```

`MAIN_EXTERNALLY_VISIBLE` 宏表示了一个函数作为一个命令的入口，也就是可以在命令行直接执行的。各个命令都按照 C 语言主函数的形式，以 `argc` 和 `argv` 作为参数，参数内容由用户输入的命令所定。

第 10 章

Linux 用户空间的编程

10.1 Linux 用户空间编程概述

Linux 用户空间的编程以 C 语言为基础。可以使用 C 语言的各种标准库，需要在源代码中使用 `include` 包含其头文件，在编译选项中使用 `-l` 连接使用的库。Linux 中的库包括以下几个方面。

- ISO 标准 C 语言的接口：通常指 ANSI 标准化的 C99。
- POSIX 标准的 C 语言接口：UNIX 系统普遍使用的接口。
- Linux 特殊的 C 语言接口：Linux 操作系统的特殊接口。
- 体系结构和硬件相关的接口：不同体系结构和硬件的特殊接口。

UNIX 的 C 语言编程通常指 ISO 标准 C 语言的接口和 POSIX 标准的 C 语言接口。除此之外，Linux 当中的编程还有一些特殊的接口以及硬件相关的接口。POSIX（Portable Operating System Interface of UNIX）是 IEEE 为要在各种 UNIX 操作系统上运行的软件，而定义 API 的一系列互相关联的标准的总称，其正式名称为 IEEE 1003，而国际标准名称为 ISO/IEC 9945。

Linux 环境中可以使用的 C 语言标准的头文件，具体包括以下内容。

- `<assert.h>`：定义了 C 的预编译宏 `assert`，包括一些错误码。
- `<complex.h>`：用于操作复数运算的一组函数。
- `<ctype.h>`：一些数据类型定义和字符处理的函数。
- `<errno.h>`：检测程序错误的返回值并定义错误类型。
- `<fenv.h>`：用来控制浮点操作环境。
- `<float.h>`：实数运算的系统参数。
- `<inttypes.h>`：不依赖于硬件平台的数据类型定义。
- `<iso646.h>`：在 ISO646 字符集下面编程的接口。
- `<limits.h>`：整型运算的系统参数。

- <locale.h>: 设定本地化参数。
- <math.h>: 数学函数。
- <setjmp.h>: 非局部分支和跳转处理。
- <signal.h>: 信号处理函数。
- <stdarg.h>: 可变长度参数表。
- <stdbool.h>: 布尔值操作函数。
- <stddef.h>: 标准定义。
- <stdint.h>: 定义整型的各种变量函数。
- <stdio.h>: 定义输入/输出函数、类型和宏。size_t 是由 sizeof 产生的无符号整数类型；fpos_t 类型说明文件中的每个位置，宏 EOF 表示文件的结尾。
- <stdlib.h>: 标准库工具。
- <string.h>: 字符串操作。
- <tgmath.h>: 通用类型数学宏。
- <time.h>: 时间与日期相关的数据结构和函数。
- <wchar.h>: 扩展多字节及宽字符支持。
- <wctype.h>: 宽字符分类及映射支持。

UNIX 的标准 C 语言编程头文件 unistd.h 是 POSIX 标准的 UNIX 类系统定义符号常量的头文件，包含了许多 UNIX 系统服务的函数原型。unistd.h 本身从 POSIX 定义的系统调用转换而来。每一个系统调用号对应一个函数，系统调用号和函数类型（参数和返回值类型）都是标准的，在不同的体系结构中基本相同。

Linux 系统编程环境中还包含一些特殊的头文件。通常是 sys 目录中的 C 语言头文件，其中一些常用的文件如下所示。

- <sys/types.h>: 基本系统数据类型。
- <sys/stat.h>: 关于描述文件结构。
- <sys/file.h>: 关于文件的定义。
- <sys/ioctl.h>: 关于端口控制。
- <sys/mman.h>: 关于内存方面的。
- <sys/time.h>: 关于时间方面的。

除此之外，Linux 编程的每个功能方面还包括一些专用的头文件和接口，例如：系统特殊控制的、文件系统的、网络的、硬件的、多媒体的。

在编程的时候，主要的帮助文档可以在 Linux 的命令行中得到，也就是 *Linux Programmer's Manual*，利用 man 命令可以查看函数的帮助。

对于一个函数的描述，一般有如下的内容。

- 名称 (NAME): 包括描述。
- 摘要 (SYNOPSIS): 包括包含的头文件和函数原型。
- 描述 (DESCRIPTION): 功能的文字性描述。
- 返回值 (RETURN VALUE): 函数本身的返回值。

- 错误码 (ERRORS): 存储在系统错误中的错误码。

例如: 通过 `man` 命令查看 `ioctl()` 函数的内容如下所示:

```
$ man ioctl
NAME
    ioctl - control device

SYNOPSIS
    #include <sys/ioctl.h>
    int ioctl(int d, int request, ...);

DESCRIPTION
    The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument d must be an open file descriptor.

    The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally char *argp (from the days before void * was valid C), and will be so named for this discussion.

    An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl() request are located in the file <sys/ioctl.h>.

RETURN VALUE
    Usually, on success zero is returned. A few ioctl() requests use the return value as an output parameter and return a non-negative value on success. On error, -1 is returned, and errno is set appropriately.

ERRORS
    EBADF d is not a valid descriptor.
    EFAULT argp references an inaccessible memory area.
    EINVAL Request or argp is not valid.
    ENOTTY d is not associated with a character special device.
    ENOTTY The specified request does not apply to the kind of object that the descriptor d references.
```

某些函数的帮助中还会列出相关的函数，也有的包含一些示例代码。

提示: Linux 的错误码 EFAULT、ENOTTY 等本身为正整数，在应用中通常以其相反数作为错误值的表示。

某些帮助的内容具有多入口，需要使用 `man <N>` 查找帮助（因为某些名称既代表一个函数，也可能代表一个命令行的命令）。

例如，对于 `sleep` 的帮助，就有多个入口。直接使用 `man` 将得到 `sleep` 命令的帮助。如果得到 `sleep` 函数的帮助，需要如此使用：

```
$ man 3 sleep
```

10.2 文件的相关内容

文件操作是 Linux 最基本的操作，其中基本的操作方式为：一个被打开的文件被称为

文件描述符 (File Descriptor), 使用一个整数来描述, 这个文件描述符就是打开文件的句柄, 用于控制文件的操作和关闭。

10.2.1 文件的打开、关闭和读写等

Linux 中文件操作的打开函数如下所示:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

`open()` 用于通过文件的路径打开或者创建一个文件, 打开后即获得了文件描述符, 可以对其进行各种操作。

`open()` 函数 `flags` 参数的互斥参数:

- `O_RDONLY` 表示以只读方式打开文件。
- `O_WRONLY` 表示以只写方式打开文件。
- `O_RDWR` 表示以可读写方式打开文件。

非互斥参数中包括下面几个:

- `O_CREAT` 表示如果打开的文件不存在, 则自动建立该文件。
- `O_APPEND` 表示当读写文件时会从文件尾开始移动。
- `O_NONBLOCK` 表示以不可阻断的方式打开文件。

`mode` 参数表示建立文件的属性:

- `S_IRWXU 00700`, 所有者具有可读、可写及可执行的。
- `S_IRUSR` 或 `S_IREAD 00400`, 所有者具有可读取的。
- `S_IWUSR` 或 `S_IWRITE 00200`, 所有者具有可写入的。
- `S_IXUSR` 或 `S_IEXEC 00100`, 所有者具有可执行的。
- `S_IRWXG 00070`, 用户组具有可读、可写及可执行的。
- `S_IRGRP 00040`, 用户组具有可读的。
- `S_IWGRP 00020`, 用户组具有可写入的。
- `S_IXGRP 00010`, 用户组具有可执行的。
- `S_IRWXO 00007`, 其他用户具有可读、可写及可执行的。
- `S_IROTH 00004`, 其他用户具有可读的。
- `S_IWOTH 00002`, 其他用户具有可写入的。
- `S_IXOTH 00001`, 其他用户具有可执行的。

Linux 中文件操作的关闭、读、写和同步函数如下所示:

```
#include <unistd.h>
int close(int fd);
ssize_t read(int fd, void * buf, size_t count);
ssize_t write (int fd, const void * buf, size_t count);
int fsync(int fd);
```

`close()`函数用于关闭一个打开的文件，关闭后数据写回磁盘，并释放该文件所占用的资源。

`read()`函数将参数 `fd` 所指的文件传送 `count` 个字节到 `buf` 指针所指的内存中。返回值为实际读取到的字节数。如果返回 0，表示已到达文件尾或是无可读取的数据。此外，文件读写位置会随读取到的字节移动。

`write()`函数将参数 `buf` 所指的内存写入 `count` 个字节到参数 `fd` 所指的文件内，文件读写位置也会随之移动。

每一个已打开的文件都有一个读写位置，当打开文件时，通常其读写位置指向文件开头；若是以附加的方式打开文件（如 `O_APPEND`），则读写位置会指向文件尾。当 `read()` 或 `write()` 时，读写位置会随之增加，`lseek()` 便是用来控制该文件读写位置的。

```
#include<sys/types.h>
#include<unistd.h>
off_t lseek(int fildes,off_t offset ,int whence);
```

第 3 个参数 `whence` 的取值有以下几个：

- `SEEK_SET`（值为 0）指向文件头后再增加 `offset` 个位移量。
- `SEEK_CUR`（值为 1）以目前的读写位置向后增加 `offset` 个位移量。
- `SEEK_END`（值为 2）指向文件尾后再增加 `offset` 个位移量。

获取文件大小，并以文件开始位置作为当前位置的实现代码，如下所示：

```
int start,end,filesize = 0;
end = lseek(fd,0,SEEK_END);           // 得到文件尾的偏移量
start = lseek(fd,0,SEEK_SET);         // 得到文件头的偏移量
filesize = end - start;
```

注意，以上代码中 `lseek()` 函数的 `SEEK_END` 和 `SEEK_SET` 两行不可颠倒。因为只有 `SEEK_SET` 后执行的情况下，才能保证获取完文件的大小，文件的操作位置还在文件的头部，否则文件操作就在文件尾部。

在代码中，上面的内容也可以写为：

```
filesize = lseek(fd,0,SEEK_END) - lseek(fd,0,SEEK_SET);
```

`lseek()` 函数的 `SEEK_SET` 操作同样是后执行，保证文件的操作位置在文件头。

一个文件的操作函数如下所示：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc,char* argv[]){
    int fd,size;
    char str[]="File Operation";
    char* filename;
    char buffer[80];
    if(1 == argc){           // 根据第一个参数操作某个文件，或当前目录的 tempfile 文件
        filename = "tempfile";
    } else {
```

```

    filename = argv[1];
}
fd = open(filename,O_RDWR|O_CREAT,S_IRWXU|S_IXGRP|S_IROTH); // 可以新建
size = write(fd, str, sizeof(str)); // 执行写操作
printf("[Write] size = %d \n",size);
close(fd); // 关闭文件描述符
fd = open(filename, O_RDONLY); // 再次打开
size = read(fd, buffer, 20); // 执行读操作
printf("[Total] size = %d [%s]\n",size,buffer);
lseek(fd,5,SEEK_SET); // 移动操作的位置
size = read(fd,buffer,sizeof(buffer));
printf("[Seek] size = %d [%s]\n",size,buffer);
close(fd); // 关闭文件描述符
return 0;
}

```

这段代码的程序，在不加参数的情况下的执行结果如下所示：

```

[Write] size = 15
[Total] size = 15 [File Operation]
[Seek] size = 10 [Operation]

```

执行后将生成 `tempfile` 文件，这个文件的大小为 15 字节，文件的属性为 `-rwx--xr--`，这个属性源于第一个 `open()` 调用的执行。

10.2.2 文件的控制、映射和查询等

Linux 当中的另外几个重要文件操作函数是 `ioctl()`、`mmap()` 和 `poll()`。

用于控制的 `ioctl()` 函数如下所示：

```

#include<sys/ioctl.h>
int ioctl(int fd ,unsigned int cmd,...);

```

`ioctl()` 函数是一个对设备文件操作的特殊函数，其中的 `cmd` 表示操作的命令号，命令号可以是自定义的整数，后面的参数是可变的。对于普通的文件，一般不使用 `ioctl()`，其主要针对设备文件的特殊操作使用。

用于文件映射的 `mmap()` 函数和 `munmap()` 函数如下所示：

```

#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *start, size_t length);

```

`mmap()` 函数将一个文件或者其他对象映射进内存。函数执行相反的操作，删除特定地址区域的对象映射。

用于查询的 `poll()` 函数如下所示：

```

#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events */
    short revents; /* returned events */
};

```

`poll()`函数用于等待事件和查询，`pollfd` 结构体用于描述 `poll` 调用时的参数，其中包含了文件描述符和表示事件的标志。

在通过 `poll()`实现调用的时候，参数中包括 `pollfd` 结构体类型的 `fds` 和表示这个结构体数目的 `nfds`。`pollfd` 结构体指定了一个被监视的文件描述符，可以传递多个结构体，表示监视多个文件描述符。每个 `pollfd` 结构体的 `events` 成员是监视该文件描述符的事件掩码，由调用者来设置。`revents` 成员是文件描述符的操作结果事件掩码。内核在调用返回时设置这个成员。`events` 成员中请求的任何事件都可能在 `revents` 成员中返回。

`pollfd` 结构的 `events` 和 `revents` 中一些可用的事件如下所示。

- `POLLIN`：有数据可读。
- `POLLRDNORM`：有普通数据可读。
- `POLLRDBAND`：有优先数据可读。
- `POLLPRI`：有紧迫数据可读。
- `POLLOUT`：写数据不会导致阻塞。
- `POLLWRNORM`：写普通数据不会导致阻塞。
- `POLLWRBAND`：写优先数据不会导致阻塞。
- `POLLMSG`：`SIGPOLL` 消息可用。
- `POLLER`：指定的文件描述符发生错误（仅用于 `revents`）。
- `POLLHUP`：指定的文件描述符挂起事件（仅用于 `revents`）。
- `POLLNVAL`：指定的文件描述符非法（仅用于 `revents`）。

提示：对普通文件使用 `poll()`函数没有效果，将直接返回。`poll()`函数主要用于管道和套接字类型的操作，可以实现无开销的阻塞。

10.2.3 文件的其他操作

涉及文件访问和连接操作的几个函数如下所示：

```
#include<unistd.h>
int access(const char * pathname,int mode);
int link (const char * oldpath,const char * newpath);
int unlink(const char * pathname);
```

`access()`函数用于检查是否可以读、写某一文件或者测试某一文件是否存在。如果被测试的文件是一个符号连接，将测试这个符号连接引用的文件。

`link()`函数用于创建连接，参数 `newpath` 表示连接文件路径，`oldpath` 参数表示被建立到连接的原有文件的路径。

`unlink()`函数用于删除参数 `pathname` 指定的文件。如果该文件为最后连接点，就会被删除，但如果有进程打开了此文件，则要在都关闭后才会删除此文件。

提示：当用于删除文件的时候，`unlink()`函数相当于标准库当中的 `remove()`函数。

用于统计文件信息的几个函数如下所示：

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *path, struct stat *buf);
struct stat {
    dev_t    st_dev;        /* 文件所在设备的标识 */
    ino_t    st_ino;        /* 文件节点号 */
    mode_t    st_mode;      /* 文件的模式 */
    nlink_t   st_nlink;     /* 硬连接个数 */
    uid_t     st_uid;       /* 文件用户标识 */
    gid_t     st_gid;       /* 文件用户组标识 */
    dev_t     st_rdev;      /* 文件所表示的特殊设备文件的设备标识 */
    off_t     st_size;      /* 总大小，以字节为单位 */
    blksize_t st_blksize;   /* 文件系统的块大小 */
    blkcnt_t  st_blocks;    /* 分配给文件的块的数量，以 512 字节为单元 */
    time_t    st_atime;     /* 最后访问时间 */
    time_t    st_mtime;     /* 最后修改时间 */
    time_t    st_ctime;     /* 最后状态改变时间 */
};

```

stat()函数通过路径得到其文件系统信息，fstat()通过文件描述符得到信息，结果从 struct stat 类型的结构中返回。其中的 st_uid 和 st_gid 表示用户名和组名；st_mode 表示文件类型，也就是常规文件（REG，0x10）、目录（DIR，0x4）、字符设备（CHR，0x2）等类型。

10.3 进程相关的内容

Linux 中的程序运行以进程为主体，进程之间相互隔离，独立运行，进程之间的通信需要使用特殊的 IPC（Inter-Process Communication，进程间通信）机制。

10.3.1 fork 和 exec

Linux 当中主要通过 fork()函数和 exec 函数族用于产生新进程，前者用于在当前代码运行环境中分裂进程，后者用于使用新的代码产生新的进程。

fork()函数和 exec 函数族如下所示：

```

#include <unistd.h>
pid_t fork(void);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [], char *const envp[]);

```

fork()函数用于从已存在的进程中创建一个新进程。此新进程被称为子进程，而原进程被称为父进程。通过检查返回值可以得知该进程是父进程，还是子进程。

exec 函数族将当前进程替换成所执行的文件。在调用 exec 进程的实体中，代码段、数

据段和堆栈等都被新的内容取代，只有进程 ID 是不变的。exec 函数族的函数执行成功后不会返回。execl 和 execv 两组函数的命令行参数的格式有区别，前者使用可变参数，后者使用字符串数组。execlp() 和 execvp() 函数的第一个参数不用输入完整路径，只要给出执行的文件名即可，它会在环境变量 PATH 当中查找命令。

进程具有 uid、gid 和 groups 几个属性，在运行中可以通过函数获取和设置，这些函数如下所示：

```
#include<unistd.h>
#include<sys/types.h>
gid_t getgid(void);
int setgid(gid_t gid);
uid_t getuid(void);
int setuid(uid_t uid);
int getgroups(int size,gid_t list[]);
#include <grp.h>
int setgroups(size_t size,const gid_t * list);
```

关于 euid 和 egid 的获取和设置函数如下所示：

```
#include<unistd.h>
#include<sys/types.h>
int seteuid(uid_t euid);
int setegid(gid_t egid);
uid_t geteuid(void);
gid_t getegid(void);
```

某些设置的函数对运行的进程有权限的要求，例如要求用户识别码为 0（表示 root），才能够成功进行。

进程在分裂时的几个要点如下所示。

- fork() 函数的返回值不同：对于父进程，返回子进程的 pid；对于子进程，返回 0。
- 变量在父子进程不能共享：变量的存储空间也将分成两个。
- 打开的文件在子进程也有效：表示打开文件的文件描述符依旧可用。

一段使用 fork() 创建线程的程序如下所示：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
static int time = 0;
int loop(char * str,int interval){ // 分裂进程的循环
    for(;;){
        time++;
        printf("[%s]:[%d]\n",str,time);
        sleep(interval); // 休眠 1 秒
    }
    return 0;
}
int main(int argc,char* argv[]){
    pid_t pid;
    if(-1 == (pid=fork())) { // 分裂进程
        perror("fork error\n");
        exit(EXIT_FAILURE);
    } else if(0 == pid) { // 子进程当中的处理
        printf("[Child] pid=%d ppid=%d\n",getpid(),getppid());
```

```

        loop("Child",1);
    } else {
        // 父进程当中的处理
        printf("[Parent] retpid=%d pid=%d ppid=%d\n",pid,getpid(),getppid());
        loop("Parent",2);
    }
    return 0;
}

```

程序的运行结果如下所示（进程 id 的情况会随环境而不同）：

```

[Parent] pid=28080 ppid=30599
[Parent]:[1]
[Child] pid=28081 ppid=28080
[Child]:[1]
[Child]:[2]
[Parent]:[2]
[Child]:[3]
[Child]:[4]
[Parent]:[3]
[Child]:[5]
[Child]:[6]

```

`fork()`函数的调用是本段程序的关键，返回值有 3 种情况，-1 表示出错，0 表示子进程，正数表示父进程。正常情况下，`fork()`函数不会出错，会产生父进程和子进程两个分支。

正常运行的过程中，可以根据 `fork()`返回的是否为 0，进入子进程或者父进程的执行代码中，父进程可以得到由自己分出的子进程的进程 id。在上述程序中作为可写静态变量的 `time` 在子进程和父进程中，被赋成了不同的数值。因此，子进程和父进程不可能使用全局或者静态的变量作为通信的手段。

10.3.2 管道

管道是 UNIX 中基本的 IPC（进程间通信）方式，有名管道被称为 FIFO，无名管道被称为 `pipe`。FIFO 的全称是“First In, First Out”，含义为“先入先出”。

在程序中建立 `pipe` 的函数如下所示：

```

#include<unistd.h>
int pipe(int fd[2])

```

`pipe()`参数中的内容作为一对文件描述符返回，读管道由 `fd[0]`表示，写管道由 `fd[1]`表示。管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；只能用于具有亲缘关系进程之间的通信。

所建立的 `pipe` 是阻塞关系，`pipe` 建立的是文件描述符 0 永远用于读，1 永远用于写，无论单独的读，还是单独的写都会引发阻塞。直到有对应的写操作，才能让读操作继续；直到有对应的读操作，才能让写操作继续。

父子进程使用管道的读写方式如下所示：

```

#include <string.h>
#include <stdio.h>
const char HELLO[] = "Hello";           // 发送的字符串
const char BACK[] = "Back";             // 返回的字符串
int main(int argc,char* argv[]){

```

```

int pipe_fd[2]; // 用于父子进程通信的无名管道
pid_t pid;
char buffer[16];
int size;
memset(buffer, 0, sizeof(buffer));
if(pipe(pipe_fd) < 0) { // 建立管道对
    perror("Create Pipe\n");
    return -1;
}
if((pid = fork()) == 0) { // 分裂进程、子进程当中的处理
    sleep(1);
    size = read(pipe_fd[0], buffer, sizeof(HELLO)); // 子进程读管道 (2)
    printf("[Child] Get size = %d [%s]\n", size, buffer);
    write(pipe_fd[1], BACK, sizeof(BACK)); // 子进程写管道 (3)
    sleep(1);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    printf("[Child] Exit\n"); // 子进程退出, 前面流程: 休眠 1 秒-写-读-休眠 1 秒
    return 0;
} else if(pid > 0) { // 分裂进程、父进程当中的处理
    printf("[Parent] Put\n");
    write(pipe_fd[1], HELLO, sizeof(HELLO)); // 父进程写管道 (1)
    sleep(2);
    size = read(pipe_fd[0], buffer, sizeof(BACK)); // 父进程读管道 (4)
    printf("[Parent] Get size = %d [%s]\n", size, buffer);
    sleep(2);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    printf("[Parent] Exit\n"); // 父进程退出, 前面流程: 写-休眠 2 秒-读-休眠 2 秒
    return 0;
}
return 0;
}

```

程序的执行结果如下所示:

```

[Parent] Put
[Child] Get size = 6 [Hello]
[Parent] Get size = 5 [Back]
[Child] Exit
[Parent] Exit

```

根据程序当中的睡眠延迟时间, 两个进程执行的流程是: 父进程写、子进程读、子进程写、父进程读。在前面的程序中, 使用 `pipe()` 函数建立的管道就是用于两个进程通信的, 父进程写入的内容是 "Hello", 子进程写入的内容是 "Back"。

使用 `pipe()` 建立的是无名管道, 只能在有亲缘的进程中进行通信。这是因为一个已经存在的无名管道不能被再次打开, 没有亲缘关系的进程不能共享文件描述符, 也就不能共享同一个管道。而如果是在文件系统中建立的 FIFO 管道文件, 就可以用于任意进程的通信。二者的存在形式上有差别, 一旦打开后就得到了其文件描述符, 之后二者就具有相同的操作和行为了。

10.3.3 System V IPC

System V IPC 是 UNIX 操作系统当中一种常用的 IPC 通信方式, 自 Columbus Unix (简

称 CB Unix) 引入。System V IPC 包括消息 (message)、信号量 (semaphore) 以及共享内存 (shared memory)。

Linux 系统的编程中, 可以使用 System V, 主要包括在 `sys/ipc.h` (公共内容)、`sys/msg.h` (用于消息)、`sys/sem.h` (用于信号量)、`sys/shm.h` (用于共享内存) 等几个头文件中。

System V 三种类型的 IPC 使用 `key_t` 值表示一个实例, 也称为 IPC 键。根据头文件 `sys/types.h` 的定义, `key_t` 这个数据类型就是一个整数。

`ftok()` 函数把一个已经存在的路径和一个整数标识符转换为一个 `key_t` 值, 由此得到一个 IPC 键。函数的原型如下所示:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

`ftok` 典型实现调用 `stat` 函数, 然后组合以下 3 个量: `pathname` 所在文件系统的信息、该文件在本文件系统内的索引节点号及 `id` 的低序 8 位。

System V 消息 (message) 简称 `msg`, 其相关的函数如下所示:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

`msgget()` 函数用于获得一个消息, `msgctl()` 用于一个消息的控制, `msgsnd()` 和 `msgrcv()` 函数用于在消息队列上进行收发消息。

`msqid_ds` 数据结构用于描述一个实际的消息, 如下所示:

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /* 所有者和权限 */
    time_t          msg_stime;     /* 最后的 msgsnd() 时间 */
    time_t          msg_rtime;     /* 最后的 msgrcv () 时间 */
    time_t          msg_ctime;     /* 最后的改变时间 */
    unsigned long   __msg_cbytes;  /* 队列中的字节数 */
    msgqnum_t       msg_qnum;      /* 队列中的消息数 */
    msglen_t        msg_qbytes;    /* 队列中最大的字节数 */
    pid_t           msg_lspid;     /* 最后的 msgsnd() 的 PID */
    pid_t           msg_lrpid;    /* 最后的 msgrcv() 的 PID */
};
```

System V 消息的本质是消息队列, 主要使用方法就是 `msgsnd()` 和 `msgrcv()` 函数当中的 `msgp` 指针, 它指向消息缓冲区的指针, 此位置用来作为临时的消息, 可以用于发送者和接收者之间的信息传递。

System V 信号量 (semaphore) 简称 `sem`, 其相关的函数如下所示:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
```

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
int semtimedop(int semid, struct sembuf *sops,
               unsigned nsops, struct timespec *timeout);
struct sembuf{
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
}
```

`semget()` 函数用于获得一个信号量，`semctl()` 用于一个信号量的控制，`semop()` 和 `semtimedop()` 函数用于操作一个信号量。`sembuf` 结构表示一个信号量中存储的内容，其中 `sem_num` 表示操作的数目，而 `sem_op` 表示信号量数值的增加和减少。

`semid_ds` 数据结构用于描述一个实际的信号量，如下所示：

```
struct semid_ds {
    struct ipc_perm sem_perm; /* 所有者和权限 */
    time_t          sem_otime; /* 最后 semop() 的时间 */
    time_t          sem_ctime; /* 最后的更改时间 */
    unsigned short  sem_nsems; /* 信号量的数据 */
};
```

System V 信号量是整数型的信号量（非布尔型），它的值表示资源的使用情况：当值大于 0 时，表示当前可用资源数的数量；当值小于 0 时，其绝对值表示等待使用该资源的进程个数。

System V 共享内存（shared memory）简称 shm，其相关的函数如下所示：

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

`shmget()` 函数用于获得一个信号量，`shmctl()` 函数用于一个信号量的控制，`shmat()` 函数用于把共享内存区对象映射到调用进程的地址空间，`shmdt()` 函数用于断开共享内存连接。

`shmid_ds` 数据结构用于描述一个实际的信号量，如下所示：

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* 所有者和权限 */
    size_t          shm_segsz; /* segment 的大小（字节数） */
    time_t          shm_atime; /* 最后一次连接的时间 */
    time_t          shm_dtime; /* 最后一次解除连接的时间 */
    time_t          shm_ctime; /* 最后一次改变的时间 */
    pid_t           shm_cpid; /* 创建者的 PID */
    pid_t           shm_lpid; /* 最后一次调用 shmat() 和 shmdt() 的 PID */
    shmatt_t        shm_nattch; /* No. of current attaches */
    // .....省略部分内容
};
```

System V 共享内存就是通过获取一块内存地址来进行通信的。也就是 `shmat()` 函数执行后连接成功，然后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问。Linux 系统特别将共享内存空间初始化为 0。

System V 三种 IPC 都有一个共同的结构 `ipc_perm`，如下所示：

```
struct ipc_perm {
    key_t    __key;    /* 获得 IPC 之后得到的 IPC 键值 */
    uid_t    uid;      /* 拥有者的有效 UID */
    gid_t    gid;      /* 拥有者的有效 GID */
    uid_t    cuid;     /* 创建者的有效 UID */
    gid_t    cgid;     /* 创建者的有效 GID */
    unsigned short mode; /* 表示权限的值 */
    unsigned short __seq; /* 序列数字 */
};
```

10.3.4 POSIX IPC

POSIX IPC 包含消息队列（`mqueue`）、信号量（`semaphore`）、共享内存区（POSIX shared memory）。

POSIX IPC 都是用 IPC 名字来表示的。IPC 名字的共同属性：用于标识的路径名、打开或创建时指定的标志以及访问权限。

`S_TYPEISMQ(buf)`、`S_TYPEISSEM(buf)`、`S_TYPEISSHM(buf)`三个宏用于操作 POSIX IPC 对象。它们的单个参数是指向某个 `stat` 结构的指针，其内容由 `fstat`、`lstat` 或 `stat` 这三个函数填入。如果所指定的 IPC 对象（消息队列、信号量或共享内存区对象）是作为一种独特的文件类型实现的，而且参数所指向的 `stat` 结构访问这样的文件类型，那么这三个宏计算出一个非零值；否则，计算出的值为 0。

POSIX 消息队列的相关函数如下所示：

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
mqd_t mq_close(mqd_t mqdes);
mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr *oldattr);
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
mqd_t mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

`mq_open()`函数用于根据 IPC 名字打开（包括新建）一个消息队列。`mq_send()`和 `mq_receive()`函数用于消息队列的收发，其中的内容用一块内存来表示。

POSIX 信号量的相关函数如下所示：

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

```
int sem_getvalue(sem_t *sem, int *sval);
```

`sem_open()`函数用于根据 IPC 名字打开（包括新建）一个信含量，`sem_init()`函数用于初始化一个信含量。`sem_wait()`函数将从信号量的值减 1，但它永远会先等待该信号量为一个非零值才开始做减法。`sem_post()`函数的作用是给信号量的值加 1。`sem_getvalue()`函数将信号量当前值设置成 `sval` 指向的整数上。

POSIX 共享内存的相关函数如下所示：

```
#include <sys/mman.h>
#include <sys/stat.h>      /* For mode constants */
#include <fcntl.h>         /* For O_* constants */
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

`shm_open()`函数的返回值是一个文件描述符，也就是将一个 POSIX 共享内存表示成一个打开的文件。`ftruncate()`函数用于调整文件共享内存的空间，`fstat()`函数用于获取到文件状态，`mmap()`和 `munmap()`函数进行映射。

在 Linux 系统中，POSIX 共享内存对象驻留在 `tmpfs` 伪文件系统中。系统默认挂载在 `/dev/shm` 目录下。当调用 `shm_open()`函数创建或打开 POSIX 共享内存对象时，系统会将创建/打开的共享内存文件放到 `/dev/shm` 目录下，也就是 `tmpfs` 伪文件系统中。

10.4 信号相关的内容

信号（signal）是 UNIX 中的一种异步机制，可以在进程运行的任意时刻发生。信号发生的原因可能来自进程的內部，也可能来自进程的外部。

信号本质就是一个整数，Linux 系统当中所有的信号及其代表的含义如下所示。

- [1] SIGHUP：本信号在用户终端连接结束时发出。
- [2] SIGINT：程序中止（interrupt）信号。
- [3] SIGQUIT：与 SIGINT 类似，但由 QUIT 字符来控制。
- [4] SIGILL：执行了非法指令。
- [5] SIGTRAP：由断点指令或其他 trap 指令产生。
- [6] SIGABRT：程序自己发现错误并调用 `abort` 时产生。
- [7] SIGBUS：在 PDP-11 上由 iot 指令产生，在其他机器上和 SIGABRT 一样。
- [8] SIGFPE：非法地址，包括内存地址对齐出错。
- [9] SIGKILL：用于立即结束程序的运行，本信号不能被阻塞、处理和忽略。
- [10] SIGUSR1：保留给用户。
- [11] SIGSEGV：用于立即结束程序的运行，本信号不能被阻塞、处理和忽略。
- [12] SIGUSR2：保留给用户。
- [13] SIGPIPE：流水线被打断。
- [14] SIGALRM：时钟定时信号。
- [15] SIGTERM：程序结束信号，与 SIGKILL 不同的是，可以被阻塞和处理。

- [17] SIGCHLD: 子进程结束时, 父进程会收到这个信号。
- [18] SIGCONT: 让一个停止的进程继续执行, 本信号不能被阻塞。
- [19] SIGSTOP: 停止进程, 不能被阻塞、处理或忽略。
- [20] SIGTSTP: 停止进程, 但该信号可以被处理和忽略。
- [21] SIGTTIN: 当后台作业要从用户终端读数据时产生此信号。
- [22] SIGTTOU: 类似于 SIGTTIN, 但在写终端或修改终端模式时收到。
- [23] SIGURG: 有: 紧急数据或 out-of-band 数据到达 socket 时产生。
- [24] SIGXCPU: 超过 CPU 时间资源限制。
- [25] SIGXFSZ: 超过文件大小资源限制。
- [26] SIGVTALRM: 虚拟时钟信号, 类似于 SIGALRM, 但是计算的是该进程占用的 CPU 时间。
- [27] SIGPROF: 类似于 SIGALRM/SIGVTALRM, 但包括该进程用的 CPU 时间以及系统调用的时间。
- [28] SIGWINCH: 窗口大小改变时发出。
- [29] SIGIO: 文件描述符准备就绪。
- [30] SIGPWR: 电源错误。

用于信号处理的函数如下所示:

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid,int sig);
int raise(int sig);
unsigned int alarm (unsigned int seconds);
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

kill()函数可以用来把参数 sig 指定的信号送给参数 pid 指定的进程。raise()函数向当前的进程发送一个 sig 信号, 相当于 pid 设置为自身的 kill()函数。

alarm()函数用于在进程中设置一个定时器, 当定时时间到的情况下将会向进程发送 SIGALRM 信号。如果进程捕获了这个信号, 将使用安装的信号处理器处理这个函数。如果进程没有捕获这个函数, 将使用默认的行为, 即终止进程。

signal()用于确定以后当信号 signum 出现时的处理方法。如果 handler 的值是 SIG_DFL, 就采用预先定义的默认行为。如果 handler 的值是 SIG_IGN, 就忽略该信号, 否则调用 handler 所指向的函数 (参数为信号类型)。

一段进行了信号处理的代码如下所示:

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
void alarm_handler(){ // 信号处理器的实现函数
    printf("----- alarm -----\n");
}
int main(int argc,char* argv[]){
    int time = 0;
    signal(SIGALRM,alarm_handler); // 注册 14 号信号 (SIGALRM) 处理器
```

```
alarm(3); // 3 秒后发送一个 SIGALRM 信号
for(;;){
    time++;
    printf("[LOOP] %d\n",time);
    sleep(1);
}
return 0;
}
```

程序的运行结果如下所示：

```
[LOOP] 1
[LOOP] 2
[LOOP] 3
----- alarm -----
[LOOP] 4
[LOOP] 5
```

其中注册了 SIGALRM（数值为 14）信号的处理程序，因此调用的 alarm(3)将在 3 秒之后引发信号处理程序的调用。这种代码的实现可以在用户空间构建异步的定时器。

在上面运行的过程中，可用命令行向进程发送信号：

```
$ kill -14 <pid>
```

程序将有类似如下的运行方式：

```
[LOOP] 20
[LOOP] 21
----- alarm -----
[LOOP] 22
```

如上，程序当中注册的信号处理器同样也可以接受来自别处发生的信号。

10.5 pthread 线程

在 Linux 系统中，线程可以视为轻量级的进程。Linux 系统下的多线程遵循 POSIX 线程接口，其被称为 pthread，也就是 POSIX thread。Linux 中用户空间的线程实际上基于内核级别的线程机制来实现。在 Linux 内核中，线程和进程的本质基本相同。

Linux 线程编程引用 pthread.h 头文件，使用线程的程序需要连接 pthread 库，在命令行使用 -lpthread 来完成。

pthread 相关的基本类型如下所示。

- pthread_t：表示线程的句柄。
- pthread_attr_t：表示线程属性的句柄。
- pthread_mutex_t：表示互斥量的句柄。
- pthread_cond_t：表示条件量的句柄。

提示：Linux 命令行当中的 ps 只能看到进程的信息，而 /proc/<pid>/是某个进程的信息目录，/proc/<pid>/task/<pid>/是一个进程中线程的信息的目录，二者结构相同。

10.5.1 线程的基本使用

POSIX 线程库当中的几个主要函数如下所示：

```
#include <pthread.h>
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
void pthread_exit(void *value_ptr);
int pthread_join(pthread_t thread, void **value_ptr);
int pthread_kill(pthread_t thread, int sig);
int pthread_equal(pthread_t t1, pthread_t t2);
int pthread_detach(pthread_t thread);
pthread_t pthread_self(void);
```

pthread_create()函数用于创建一个线程，线程基本的执行内容需要与一个函数联系在一起，就是参数 **start_routine** 表示函数指针，它就是线程的运行体。

pthread_exit()函数用于退出当前的进程。**pthread_self()**函数用于得到自身函数句柄。**pthread_equal()**函数用于比较两个线程是否相同。

pthread_join()函数用于等待一个线程的退出；**pthread_detach()**函数与之相反，可以解除这种对线程等待的绑定。**pthread_kill()**函数用于向一个线程发送信号。

一段基本的线程使用代码如下所示：

```
#include <stdio.h>
#include <pthread.h>
static pthread_t thr1, thr2;
void *thread_body1(void* param) { // 第一个线程的执行体
    int i = 0;
    pthread_t thr;
    thr = pthread_self(); // 获取自身线程句柄，并判断相等
    printf("This thread == thr1 ? %d\n", pthread_equal(thr, thr1));
    printf("This thread == thr2 ? %d\n", pthread_equal(thr, thr2));
    while(i < 2){ // 执行二次之后要退出
        printf("Thread 1 count=%d \n", i);
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
void *thread_body2(void* param) { // 第二个线程的执行体
    int i = 0;
    while(i < 4){ // 执行四次之后要退出
        printf("Thread 2 count=%d \n", i);
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
int main(int argc, char**argv){
    int ret = 0;
    pthread_create(&thr1, NULL, thread_body1, NULL); // 建立第一个线程
    pthread_create(&thr2, NULL, thread_body2, NULL); // 建立第二个线程

    pthread_join(thr1, (void**)&ret); // 等待第一个线程退出
    printf("pthread_join thread 1 = %d\n", ret);
    pthread_join(thr2, (void**)&ret); // 等待第二个线程退出
```

```
printf("pthread_join thread 2 = %d \n",ret);
return 0;
}
```

`thread_body1()`和 `thread_body2()`就是两个线程的运行体，二者都是循环，前者中检查了线程句柄是否相等。主进程当中没有循环运行体，却使用了 `pthread_join()`等待两个线程的退出。

程序的运行结果如下所示：

```
This thread == thr1 ? 1
This thread == thr2 ? 0
Thread 1 count=0
Thread 2 count=0
Thread 1 count=1
Thread 2 count=1
Thread 2 count=2
pthread_join thread 1 = 0
Thread 2 count=3
pthread_join thread 2 = 0
```

从程序的运行结果中可以看出，`pthread_join()`行为就是会等待线程的退出。如果线程在运行中，函数会阻塞；如果线程没有运行则会直接返回，并得到返回值。返回值的类型就是 `pthread_exit()`当中的 `void*`类型的指针。

一段向线程发送信号的代码如下所示：

```
#include <stdio.h>
#include <signal.h>
#include <pthread.h>
void sig_handler(int signo){                                // 线程信号的处理
    printf("sig %d \n", signo);                             // 退出当前的线程
    pthread_exit(NULL);
}
void *thread_body1(void* param) {                            // 第一个线程的执行体
    int i = 0;
    signal(SIGQUIT,sig_handler );                          // 注册信号处理器
    while(1){
        printf("Thread 1 count=%d \n",i);
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
int main(int argc,char**argv){
    int i = 0;
    pthread_t thr1;
    pthread_create(&thr1, NULL,thread_body1,NULL);
    while(1){
        printf("Main count=%d \n",i);
        sleep(1);
        if(i == 4 ){                                        // 执行到第四次，发送信号
            pthread_kill(thr1, SIGQUIT);
        }
        i++;
    }
    return 0;
}
```

在本处的程序中，主进程和建立的一个线程都有循环，当运行到第 4 次循环的时候，主进程向线程发送了信号 SIGQUIT，在线程的信号处理器中进行处理。

程序的运行结果如下所示：

```
Main count=0
Thread 1 count=0
Main count=1
Thread 1 count=1
Main count=2
Thread 1 count=2
Main count=3
Thread 1 count=3
Main count=4
Thread 1 count=4
Main count=5
sig 3
Main count=6
Main count=7
```

pthread_kill()的返回值主要有 ESRCH (3) 表示没有进程，EINVAL (22) 表示参数无效。线程当中的信号操作其实与进程的信号操作类似。

10.5.2 线程的属性

使用 pthread_create()函数建立线程的时候，可选设置一个线程属性的参数 (attr)。线程属性是一个特殊的结构。

POSIX 线程库几个线程属性相关的函数如下所示：

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

pthread_attr_t 结构当中主要的内容包括以下几项。

- **__detachstate**: 表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用 pthread_join() 来同步，且在退出时自行释放所占用的资源。默认值为 PTHREAD_CREATE_JOINABLE 状态。线程运行后，使用 pthread_detach() 函数可以将其设置为 PTHREAD_CREATE_DETACH 状态。
- **__schedpolicy**: 表示新线程的调度策略，可选的值包括 SCHED_OTHER (默认、非实时)、SCHED_RR (实时、轮转法) 和 SCHED_FIFO (实时、先入先出)，后两种调度策略仅对超级用户有效。线程运行后调用 pthread_setschedparam() 会改变。
- **__schedparam**: struct sched_param 结构，其中名为 sched_priority 的整型成员表示线程的运行优先级。这个参数仅当调度策略为实时 (即 SCHED_RR 或 SCHED_FIFO) 时才有效，并可以在运行时通过 pthread_setschedparam() 函数来改变，默认为 0。
- **__inheritsched**，有两种整数值：PTHREAD_EXPLICIT_SCHED (默认，显式指定调度策略和调度参数) 和 PTHREAD_INHERIT_SCHED (继承调用者线程的值)。
- **__scope**: 表示线程间竞争 CPU 的范围，也就是说线程优先级的有效范围，有两个

整数值：PTHREAD_SCOPE_SYSTEM（表示与系统中的所有线程一起竞争 CPU 时间）和 PTHREAD_SCOPE_PROCESS（后者表示仅与同进程中的线程竞争 CPU），在目前 Linux 当中实际上只有 PTHREAD_SCOPE_SYSTEM。

10.5.3 线程互斥量

线程的互斥量（mutex）是一种锁，用于控制有竞争关系的资源访问。

POSIX 线程库中几个互斥量的函数如下所示：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

pthread_mutex_t 结构表示线程的互斥量，调用者不需要知道这个结构的内部信息，可以通过 pthread_mutex_init() 函数建立，使用 pthread_mutex_destroy() 函数删除。此外，如果调用 pthread_mutex_destroy() 销毁一个互斥量，则表示释放它所占用的资源，且要求锁当前处于开放状态。

对互斥量主要有锁和解锁两个操作。pthread_mutex_lock() 函数操作用于获取一个信号量并加锁；如果不能得到，则会形成阻塞。pthread_mutex_trylock() 函数与之类似，但当锁不能得到的时候，会直接返回而不会阻塞。pthread_mutex_unlock() 函数用于释放一个获取的锁。

一个使用加锁线程的程序如下所示：

```
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 定义一个互斥量
static int value = 0;
void *thread_body1(void* param) { // 第一个线程的执行体
    int i = 0;
    struct timeval tv;
    while(1){
        pthread_mutex_lock(&mutex); // 锁定互斥量
        gettimeofday(&tv, NULL);
        printf("Thread Count=%d: Time=%d %d value=%d\n",
              i, (int)tv.tv_sec, (int)tv.tv_usec, value);
        pthread_mutex_unlock(&mutex); // 解锁互斥量
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
int main(int argc, char**argv) {
    int ret = 0;
    pthread_t thr1, thr2;
    pthread_create(&thr1, NULL, thread_body1, NULL);
    sleep(2); // 执行流程：休眠 2 秒-锁定-休眠 3 秒-解锁
```

```

printf("Main mutex_lock\n");
pthread_mutex_lock(&mutex);           // 锁定互斥量
sleep(3);
value = 20;
pthread_mutex_unlock(&mutex);
printf("Main mutex_unlock\n");       // 解锁互斥量
pthread_join(thr1, NULL);
return 0;
}

```

该程序描述了主进程和所建立的线程之间有一种竞争关系，因此二者在访问某个资源的时候都需要遵从加锁和解锁的流程，主线程在 2 秒之后，对资源加锁，并且持有锁的时候有 3 秒。

程序的运行结果如下所示：

```

Thread Count=0: Time=1378275250 579061 value=0
Thread Count=1: Time=1378275251 579153 value=0
Main mutex_lock
Main mutex_unlock
Thread Count=2: Time=1378275255 579427 value=20
Thread Count=3: Time=1378275256 579494 value=20
Thread Count=4: Time=1378275257 579569 value=20

```

从运行结果中可以得知，主线程获取资源后，将会阻塞线程的操作，因此线程再次运行的时候已经是几秒之后。

提示：在 `pthread_mutex_lock()` 和 `pthread_mutex_unlock()` 之间的代码应当简短，不应当长期持有一个锁。

10.5.4 线程条件量

线程的条件量（`cond`）可以用于异步的通知机制，等待条件量的线程可以实现阻塞，当另外一个线程发出信号时让其继续执行。

POSIX 线程库中几个条件量的函数如下所示：

```

#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);

```

`pthread_cond_t` 表示线程条件量的结构，可以使用 `pthread_cond_init()` 函数建立，使用 `pthread_cond_destroy()` 函数删除。条件量的基本使用方式是一个线程使用 `pthread_cond_wait()` 函数等待一个信号，此时调用将进入阻塞的状态，当另一个线程调用了 `pthread_`

`cond_signal()`函数或者 `pthread_cond_broadcast()`函数发送信号之后,等待的线程将解除阻塞。一个使用条件量进行线程等待的程序如下所示:

```
#include <stdio.h>
#include <pthread.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void *thread_body1(void* param) {                // 第一个线程的执行体
    printf("Thread 1 wait \n");
    pthread_cond_wait(&cond,&mutex);              // 等待条件量
    printf("Thread 1 exit \n");
    pthread_exit(NULL);                          // 线程退出
}
void *thread_body2(void* param) {                // 第二个线程的执行体
    int i = 0;
    while(1){
        printf("Thread 2 count=%d \n",i);
        sleep(1);
        i++;
        if(i==4){
            pthread_cond_signal(&cond);          // 发送条件信号
        }
    }
    pthread_exit(NULL);                          // 线程退出
}
int main(int argc,char**argv) {
    int ret = 0;
    pthread_t thr1,thr2;
    pthread_create(&thr1, NULL,thread_body1,NULL);
    pthread_create(&thr2, NULL,thread_body2,NULL);
    pthread_join(thr1, (void**)&ret);
    printf("pthread_join thread 1 = %d\n",ret);
    pthread_join(thr2, (void**)&ret);
    printf("pthread_join thread 2 = %d \n",ret);
    return 0;
}
```

在程序中建立了两个线程,第一个线程直接等待一个条件量,第二个线程向此条件量发送信号,而主进程也在等待两个线程的调用。

程序的运行结果如下所示:

```
Thread 1 wait
Thread 2 count=0
Thread 2 count=1
Thread 2 count=2
Thread 2 count=3
Thread 2 count=4
Thread 1 exit
pthread_join thread 1 = 0
Thread 2 count=5
Thread 2 count=6
```

首先打印的信息是在第一个线程等待条件量之前打出的,当运行到第四次循环的时候,第二个线程向条件量发送了信号,因此第一个线程解除了阻塞,并且直接退出。在主进程中对第一个线程的 `pthread_join()`函数也将返回,而第二个线程的循环继续执行。

10.5.5 线程取消

线程的取消是可以在外部结束线程的手段。线程取消不是发生在调用取消的时候，而是必须发生在取消点。

POSIX 线程库当中与线程取消相关的函数如下所示：

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

`pthread_cancel()` 函数发送终止信号给 `thread` 线程，如果成功则返回 0，否则为非 0 值。发送成功并不意味着线程会终止。

`pthread_setcancelstate()` 函数用于设置本线程对 Cancel 信号的反应，参数 `state` 的值为：`PTHREAD_CANCEL_ENABLE`（默认，分别表示收到信号后设置为 Canceled 状态）和 `PTHREAD_CANCEL_DISABLE`（忽略 Cancel 信号继续运行）；`oldstate` 如果不为 NULL，则存入原来的 Cancel 状态以便恢复。

`pthread_setcanceltype()` 函数用于设置本线程取消动作的执行时机，参数 `type` 的值为：`PTHREAD_CANCEL_DEFERRED`（收到信号后继续运行至下一个取消点）和 `PTHREAD_CANCEL_ASYNCRONOUS`（立即执行取消动作），仅当 Cancel 状态为 Enable 时有效。

`pthread_testcancel(void)` 函数用于检查本线程是否处于 Canceled 状态。如果是，则进行取消动作；否则直接返回。

根据 POSIX 所定义的标准，`pthread_join()`、`pthread_testcancel()`、`pthread_cond_wait()`、`pthread_cond_timedwait()`、`sem_wait()`、`sigwait()` 等函数以及 `read()`、`write()` 等会引起阻塞的系统调用才是取消点。

在 Linux 系统当中并非所有标准的函数都是取消点，一般通过调用 `pthread_testcancel()` 函数作为取消点。一般在函数当中通过调用 `pthread_testcancel()` 表示驱动点。

一个使用线程取消点的程序如下所示：

```
#include <stdio.h>
#include <pthread.h>
static pthread_t thr1, thr2;
void *thread_body1(void* param) {                                // 第一个线程的执行体
    int i = 0;
    while(1){
        printf("pthread_testcancel before %d ...\n", i);
        pthread_testcancel();                                   // 取消点的一个测试调用
        printf("... pthread_testcancel %d ...\n", i);
        pthread_testcancel();                                   // 取消点的一个测试调用
        printf("pthread_testcancel after %d ...\n", i);
        sleep(1);
        i++;
    }
    pthread_exit(NULL);
}
void *thread_body2(void* param) {                                // 第二个线程的执行体
    int i = 0;
```

```

        while(1){
            printf("Thread 2 count=%d \n",i);
            if(i == 2){
                pthread_cancel(thr1);           // 第二次执行后，取消第一个线程
            }
            sleep(1);
            i++;
        }
        pthread_exit(NULL);
    }
int main(int argc, char**argv) {
    pthread_create(&thr1, NULL, thread_body1, NULL);
    pthread_create(&thr2, NULL, thread_body2, NULL);

    pthread_join(thr1, NULL);
    printf("Thread 1 Exited \n");
    pthread_join(thr2, NULL);
    printf("Thread 2 Exited \n");
    return 0;
}

```

在第一个线程体 `thread_body1()` 当中调用了两次取消点, `thread_body2()` 在执行中取消了调用取消操作。

程序的运行结果如下所示:

```

pthread_testcancel before 0 ...
... pthread_testcancel 0 ...
pthread_testcancel after 0 ...
Thread 2 count=0
pthread_testcancel before 1 ...
... pthread_testcancel 1 ...
pthread_testcancel after 1 ...
Thread 2 count=1
pthread_testcancel before 2 ...
... pthread_testcancel 2 ...
pthread_testcancel after 2 ...
Thread 2 count=2
Thread 1 Exited
Thread 2 count=3
Thread 2 count=4
Thread 2 count=5

```

从上面的程序可见, 第一个线程在运行到一定阶段后自行退出, 主进程中可得到第一个线程的退出信息。

10.6 dlopen 机制

10.6.1 dlopen 的结构和意义

`dlopen` 机制是 Linux 系统所提供的动态库的打开机制。使用 `dlopen` 机制可以在运行时打开 (`dlopen`) 一个动态库 (`*.so`), 将其装入内存后, 可以取出 (`dlsym`) 其中的符号动态使用。符号可以是函数和全局的变量。使用 `dlopen` 机制需要使用 `libdl.so`, 因此在连接的时

候需要指定-l`ldl`。

`dlopen` 机制的几个函数如下所示：

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
char *dlerror(void);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

`dlopen()` 用于打开一个动态库，参数 `filename` 表示库的路径。参数 `flag` 的含义如下所示。

- `RTLD_LAZY`：在返回前，对于动态库中未定义的符号不执行解析。
- `RTLD_NOW`：在返回前，解析出所有未定义符号。如果解析不出来，`dlopen()` 会返回 `NULL`。
- `RTLD_GLOBAL`：符号可被其后打开的其他库重定位。
- `RTLD_LOCAL`：动态库中定义的符号不能被其后打开的其他库重定位。
- `RTLD_NODELETE`：`dlclose()` 不卸载库，在以后使用 `dlopen()` 重新加载库时不初始化库中的静态变量。
- `RTLD_NOLOAD`：不加载库，可用于测试库是否已加载，也可用于改变已加载库的 `flag`。
- `RTLD_DEEPBIND`：在搜索全局符号前先搜索库内的符号，避免同名符号的冲突。

连接动态库和动态打开动态库的结构如图 10-1 所示。

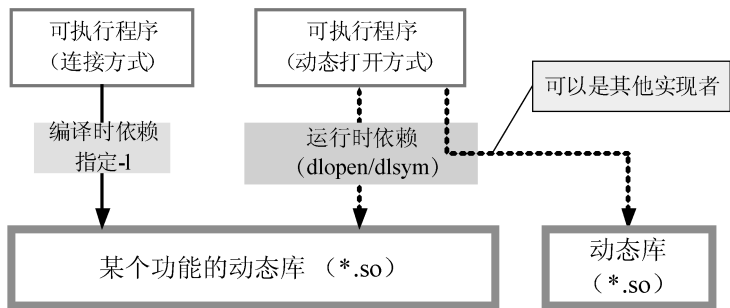


图 10-1 连接动态库和动态打开动态库

`dlopen` 机制主要有以下几个作用。

- 可以解除程序生成时的依赖：上层的程序需要在生成阶段使用其所依赖的库，只有在运行时才需要动态加载这种库，程序生成阶段（连接）的依赖关系由此解除。
- 可以选择多实现：当接口已经确定的时候，上层的程序可以选择使用多种下层实现库，这些库都不需要被连接，只需要在运行时调用 `dlopen()` 和 `dlsym()` 得到符号使用。
- 减少初始的载入时间：如果只用直接连接动态库的方式，库中符号将在可执行程序执行时就被首先加载；而如果使用 `dlopen()` 和 `dlsym()`，则可以在适当时候加载库，从而减少初始化时的载入时间。

- 回避许可问题：有一种说法是，如果一个动态库是 GPL 协议的，基于它运行的库，如果连接它也将是 GPL 的。如果不想按照 GPL 发布，可以通过 `dlopen()`和 `dlsym()` 的方式打开某库，而不指定这种库就是那个发布为 GPL 协议的库，这样新发布的库就可以不需要遵从 GPL 协议了，只需要在使用库的时候指定 `dlopen()`打开的库是那个 GPL 的库即可。这种方式能否规避 GPL 协议的传播是有争议的。

10.6.2 在 C 语言中使用 `dlopen`

在 C 语言中使用 `dlopen` 机制有以下几个要点：

- 不需要连接所使用的库，但需要使用 `-ldl`。
- 头文件可能依然需要，但是其中的函数和全部变量均不可用，只有数据结构可用。
- 对于函数类型的符号，应使用 `typedef` 定义类型。
- 可以在适当的时候进行 `dlopen()`和 `dlsym()`。
- `dlerror()`只能返回前一个错误。

调用 `dlsym()`取出的符号必须是库中导出的符号。

一段使用 `dlopen()`的程序如下所示：

```
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
typedef int (*init_t)(int );           // 定义一个参数，整型返回值的函数指针
typedef int (*hello_t)(int ,char** ); // 定义二个参数，整型返回值的函数指针
static init_t init_s = NULL;          // 静态变量保存函数指针
static hello_t hello_s = NULL;
void init_dl(){                        // 调用动态库的初始化函数
    void* handle = dlopen("./libtest_d.so", RTLD_LAZY); // 动态打开一个动态库
    if (NULL == handle) {
        printf("dlerror %s\n",dlerror());
        exit(0);
    }
    init_s = (init_t ) dlsym(handle, "init");           // 取出"init"函数指针
    hello_s = (hello_t ) dlsym(handle, "hello");        // 取出"hello"函数指针
    if (NULL == init_s || NULL == hello_s) {           // 判断两个符号取出成功
        printf("dlerror %s\n",dlerror());
        dlclose(handle);
        exit(0);
    }
}
void aftermain(void)
{
    printf("\n");
    printf("<<<< aftermain >>>>\n");
    printf(".....\n");
    return;
}
int main (int argc,char* argv[])
{
    init_dl();                                           // 调用动态库的初始化函数
    printf("==== main =====\n");
```

```

    init_s(1234);
    hello_s(argc,argv);           // 通过函数指针调用库中函数
    atexit(aftermain);           // 通过函数指针调用库中函数
    printf("..... exit main .....\\n");
    return 0;
}

```

该程序的编译过程如下所示：

```

$ <prefix>-gcc -Wall -g -c -o main.o
$ <prefix>-gcc -Wall -g main.o -ldl -o testbydl

```

在连接的时候，通过 `ldl` 连接了 `libdl.so` 库，而真正使用的 `libtest_d.so` 库则不需要连接。程序中的 `dlopen()` 函数直接打开了当前目录中的 `libtest_d.so` 动态库。

提示：调用 `dlopen()` 函数的时候，如果只有库的名称，程序将直接到默认的库路径中去寻找动态库。库的路径可以通过 `LD_LIBRARY_PATH` 进行设置。

在调用 `dlsym()` 函数取出符号的时候，“init”和“hello”等表示的都是函数，也可以是静态变量，其名称就是函数或者变量的名称字符串，返回值都是 `void` 类型，可以转换成实际的类型使用。对于函数类型的返回内容，通常可以使用 `typedef` 定义函数指针的类型。

使用 `dlopen` 的程序和不使用 `dlopen` 的区别主要在两个方面：一个方面是初始化流程取出符号；另一个方面是调用的时候调用的是函数指针（或者访问变量）。

第一处区别如下所示：

```

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#ifndef "hello.h"
#ifndef "init.h"
typedef int (*init_t)(int );
typedef int (*hello_t)(int ,char** );

```

使用 `dlopen` 的方式不需要包含 API 头文件，而是需要定义函数的原型。

第二处区别如下所示：

```

+static init_t init_s = NULL;
+static hello_t hello_s = NULL;
+
+void init_dl(){
+    void* handle = dlopen("./libtest_d.so", RTLD_LAZY);
+    if (NULL == handle) {
+        printf("dlerror %s\\n",dlerror());
+        exit(0);
+    }
+    init_s = (init_t ) dlsym(handle, "init");
+    hello_s = (hello_t ) dlsym(handle, "hello");
+    if (NULL == init_s || NULL == hello_s) {
+        dlclose(handle);
+        exit(0);
+    }
+}

```

动态打开和动态取出符号的过程需要在执行之前调用。符号的类型需要预先定义，并在取出的时候将 `void*` 转换。

第三处区别如下所示：

```
int main (int argc, char* argv[])
{
+   init_dl();
+   printf("==== main =====\n");

-   init(1234);
+   init_s(1234);

-   hello(argc, argv);
+   hello_s(argc, argv);
}
```

在初始化的时候需要先完成打开库和符号取出；调用的时候，所调用的是函数指针，而不是函数本身。

10.6.3 在 C++ 中使用 dlopen

1. C++ 库被 C 调用

如果 C++ 程序被 C 语言调用，无论是否使用 `dlopen`，情况都是类似的。C++ 的调用点需要导出一个成 C 语言的符号，而不是使用 C++ 的符号，导出符号的方法是使用 `extern "C"`。

```
extern "C" int foo(int a){
    return a;
}
```

在这种情况下，使用 `g++` 编译之后 C++ 程序仍然以 C 语言符号的形式存在，也就是直接成为 `foo` 等名称的符号。

C++ 程序被 C 语言以 `dlopen` 的形式调用，必须保证符号保持原始的名称，而不能是附加信息的 C++ 符号形式，否则就不能按照函数名称和变量名称找到符号。显然，除了名称问题之外，在 C++ 提供的接口中，还不能使用 C++ 的特定语法（例如：引用 `&` 等）。

2. C++ 库被 C++ 调用

在 C++ 对 C++ 的调用中也可以使用 `dlopen` 机制。在这种情况下只有接口的符号是 C 语言的形式，通过接口符号得到类的指针之后，调用的流程与 C++ 的直接调用完全相同。

因此，如果要想实现 C++ 对 C++ 的 `dlopen` 调用，通常需要让被调用的 C++ 程序导出一个 C 语言的符号，并以此符号表示某个类的实例。调用者只有获得一个实例，其他的调用才可以使用 C++ 函数的调用方式。

一个可以通过 `dlopen()` 函数打开并通过 `dlsym()` 函数取出符号的库包括下面几个文件。

- **Machine.h**: 一个 C++ 类的接口，作为调用主要的 API。
- **Factory.h**: 提供工厂功能，作为得到 **Machine** 的 API。
- **Factory.cpp**: **Factory** 的实现者，需要导出一个 C 语言的符号。
- **MachineImplement.h** 和 **MachineImplement.cpp**: **Machine** 的实现者。

Machine.h 当中定义了类 Machine，内容如下所示：

```
class Machine                                // 定义作为接口使用的类
{
public:
    virtual int add(int value) = 0;          // 纯虚函数：增加数值
    virtual int sub(int value) = 0;          // 纯虚函数：减少数值
    virtual int get_value() = 0;             // 纯虚函数：获取数值
    virtual void set_value(int value) = 0;    // 纯虚函数：设置数值
protected:
    Machine(){};
    virtual ~Machine(){};
};
```

Machine 类中提供的 API 均没有实现纯虚函数，因此这部分内容没有实现文件，而需要在 MachineImplement.* 当中实现。

MachineImplement.h 定义了 Machine 的继承者 MachineImplement，内容如下所示：

```
class MachineImplement : public Machine      // 继承实现 Machine 类
{
public:
    MachineImplement();
    virtual ~MachineImplement();
    virtual int add(int value);              // 定义 Machine 类中 4 个纯虚函数的实现
    virtual int sub(int value);
    virtual int get_value();
    virtual void set_value(int value);
    static MachineImplement* CreateMachineImplement();
    static void ReleaseMachineImplement(MachineImplement* machineimplement);
private:
    int m_value;                            // 保存内部的数值
};
```

MachineImplement 类本身要继承并实现 Machine 类中的各个纯虚函数，而静态函数 CreateMachineImplement() 用于产生 MachineImplement 的实例，另一个静态函数 ReleaseMachineImplement() 则用于删除 MachineImplement 的实例。

MachineImplement.cpp 的内容如下所示：

```
MachineImplement::MachineImplement() : m_value(0) // 实现的构造函数
{
    printf("MachineImplement Constructor\n");
}
MachineImplement::~MachineImplement()            // 实现的析构函数
{
    printf("MachineImplement Destructor\n");
}
int MachineImplement::add(int value)              // 接口函数的实现
{
    printf("MachineImplement::add\n");
    return (m_value += value);
}
int MachineImplement::sub(int value)              // 接口函数的实现
{
    printf("MachineImplement::sub\n");
    return (m_value -= value);
}
```

```
int MachineImplement::get_value()                // 接口函数的实现
{
    return m_value;
}
Void MachineImplement::set_value(int value)      // 接口函数的实现
{
    m_value = value;
}
MachineImplement* MachineImplement::CreateMachineImplement() // 静态创建函数
{
    return new MachineImplement();
}
void MachineImplement::ReleaseMachineImplement(MachineImplement* machineimplement) // 静态删除函数
{
    delete machineimplement;
}
```

Factory.h 定义了工厂功能 CreateMachine()和 ReleaseMachine(), 分别用于创建和释放 Machine 的实例, 这两个函数都需要以 C 符号形式导出, 供 dlsym()取出。

```
extern "C" {
    Machine* CreateMachine();                // 工厂接口: 创建
    void ReleaseMachine(Machine* machine);   // 工厂接口: 释放
}
```

Factory.cpp 当中的实现如下所示:

```
#include <stdio.h>
#include "MachineImplement.h"
#include "Factory.h"
Machine* CreateMachine()                // 创建接口的实现
{
    return (Machine*)MachineImplement::CreateMachineImplement();
}
void ReleaseMachine(Machine* machine)    // 释放接口的实现
{
    MachineImplement::ReleaseMachineImplement((MachineImplement*)machine);
}
```

调用 CreateMachineImplement()将创建 MachineImplement, 作为 Machine 类型返回。

根据以上定义的内容, 这个库的调用者需要包含 Factory.h 和 Machine.h 两个头文件, 其他文件中的内容不需要关心。本库接口的调用者不关心实现类 MachineImplement, 只需要使用接口类 Machine。

以上几个文件编成库的方式如下所示:

```
$ g++ -Wall -fPIC -shared -o libtools.so -I. MachineImplement.cpp Factory.cpp
```

此时将生成一个名为 libtools.so 的 C++库。

不使用 dlopen 的调用方式在 main1.cpp 文件中实现, 如下所示:

```
#include <stdio.h>
#include "Machine.h"
#include "Factory.h"                // 需要包含工厂接口的头文件
int main(int argc, char *argv[])
{
    Machine* machine;              // 定义类的句柄
    machine = CreateMachine();      // 调用工厂接口, 创建一个接口
}
```

```

    printf("value: %d \n", machine->get_value());
    machine->add(100);                      // 执行类中的设置操作和获取操作
    machine->sub(99);
    printf("value: %d \n", machine->get_value());
    machine->set_value(20);
    printf("value: %d \n", machine->get_value());
    ReleaseMachine(machine);               // 释放接口
    getchar();
    return 0;
}

```

生成可执行程序需要连接 libtools.so 库，如下所示：

```
$ g++ -o test1 main1.cpp -L. -ltools
```

程序的执行结果如下所示：

```

MachineImplement Constructor
value: 0
MachineImplement::add
MachineImplement::sub
value: 1
value: 20
MachineImplement Destructor

```

以上调用方式是典型的 C++ 类的调用，调用的可执行程序直接连接动态库。

使用 dlopen 的调用方式在 main2.cpp 文件中实现，如下所示：

```

#include "Machine.h"

typedef Machine* (*CreateMachine_t)(void);           // 定义创建函数指针类型
typedef void (*ReleaseMachine_t)(Machine* machine); // 定义释放函数指针类型

int main(int argc, char *argv[])
{
    void * handle;
    CreateMachine_t createmachine_fun;              // 工厂接口创建函数指针
    ReleaseMachine_t releasemachine_fun;            // 工厂接口释放函数指针

    handle = dlopen("./libtools.so", RTLD_LAZY );    // 动态打开动态库

    createmachine_fun = (CreateMachine_t)dlsym(handle, "CreateMachine");
    releasemachine_fun = (ReleaseMachine_t)dlsym(handle, "ReleaseMachine");

    Machine* machine;
    machine = createmachine_fun();
    printf("value: %d \n", machine->get_value());
    machine->add(100);
    machine->sub(99);
    printf("value: %d \n", machine->get_value());
    machine->set_value(20);
    printf("value: %d \n", machine->get_value());
    releasemachine_fun(machine);
    getchar();
    return 0;
}

```

生成可执行程序的过程如下所示：

```
$ g++ -o test2 main2.cpp -ldl
```

实际上,在 C++的 `dlopen` 方式中,只需要驱动"CreateMachine"和"ReleaseMachine"两个符号,然后的调用过程还是通过 `Machine` 类进行的。并且由于这两个符号是按照函数指针的形式取出的,因此甚至不需要包含 `Factory.h` 头文件。

在 C++对 C++的调用过程中,不使用 `dlopen` 和使用 `dlopen` 的区别主要在 C 形式工厂接口的使用上。

第一处区别如下所示:

```
#include <stdio.h>
#include <dlfcn.h>
#include "Machine.h"
#include "Factory.h"
+
+typedef Machine* (*CreateMachine_t)(void);
+typedef void (*ReleaseMachine_t)(Machine* machine);
```

在 `dlopen` 方式中,需要声明函数的原型,却不需要包含工厂接口的文件。二者对于最终调用类的头文件包含和使用都是相同的。

第二处区别如下所示:

```
int main(int argc, char *argv[])
{
+ void * handle;
+ CreateMachine_t createmachine_fun;
+ ReleaseMachine_t releasemachine_fun;
+
+ handle = dlopen("./libtools.so", RTLD_LAZY );
+
+ createmachine_fun = (CreateMachine_t)dlsym(handle,"CreateMachine");
+ releasemachine_fun = (ReleaseMachine_t)dlsym(handle,"ReleaseMachine");
+
+ Machine* machine;
```

`dlopen` 的符号需要提前取出,得到函数指针。

第三处区别如下所示:

```
// Create
- machine = CreateMachine();
+ machine = createmachine_fun();
printf("value: %d \n",machine->get_value());
```

第四处区别如下所示:

```
// Release
- ReleaseMachine(machine);
+ releasemachine_fun(machine);
```

第三处区别和第四处区别只是类建立和删除过程的区别。在不使用 `dlopen` 的时候,工厂函数是直接调用的;而在使用 `dlopen` 的时候,调用的是符号。所使用的类(本例的 `Machine`)的调用过程中,本身是没有区别的。

本例中的工厂接口只包括了建立和删除。如果程序复杂,工厂接口还可以包含更多功能,比如记录所建立类实例的数目。

第 11 章

Linux 用户空间的中间件

11.1 基于嵌入式 Linux 的系统与中间件

嵌入式 Linux 可以运行于各种硬件，基于嵌入式 Linux，可以构建各种用途的设备。在技术层面上，各种不同设备的相同点非常多。

中间件（见图 11-1）是嵌入式 Linux 用户开发的重要方面之一，起到承上启下的作用：

- 更快、更优地构建硬件平台。
- 更准确地实现驱动程序。
- 更好地复用中间件和开发环境。
- 更成熟地完成系统整合优化。

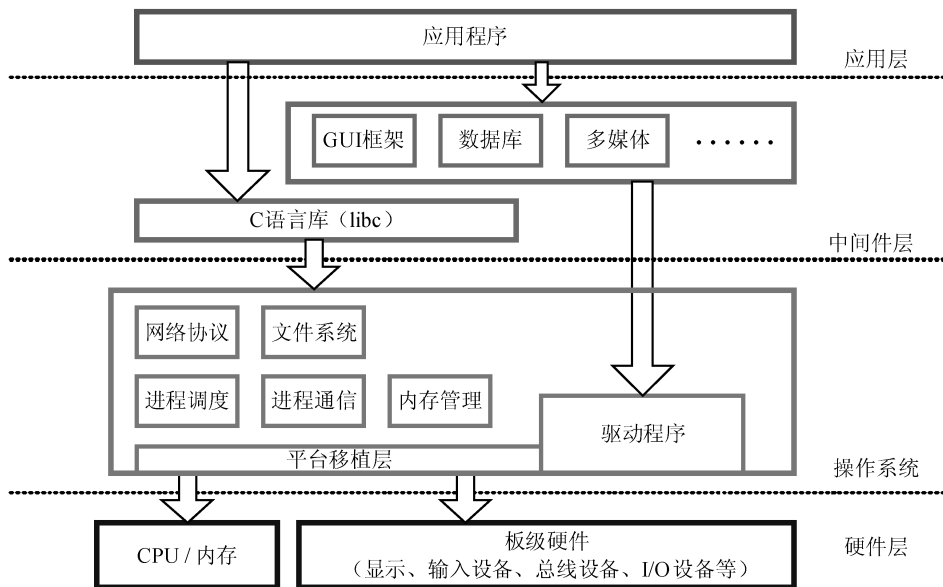


图 11-1 Linux 中间件的结构

11.2 网络协议相关

Linux 网络协议相关的编程以套接字为基础。套接字（socket）提供了传输层协议（TCP/UDP）到应用层的接口，开发者可以使用 socket 编写各种应用层的程序。

套接字（socket）是 Linux 内核空间到用户空间的一种主要接口。从编程的角度，套接字类似文件。套接字的优势在于可以使用 TCP 和 UDP 协议，因此开发者可以不用关心下层具体通信的方式和实现，它又可以利用接口的灵活性，实现各种应用层的程序。

套接字编程主要基于以下两种传输层协议：

- 面向流的 TCP（Transmission Control Protocol，传输控制协议）。
- 面向数据报的 UDP（User Datagram Protocol，用户数据报协议）。

网络协议和套接字的关系如图 11-2 所示。

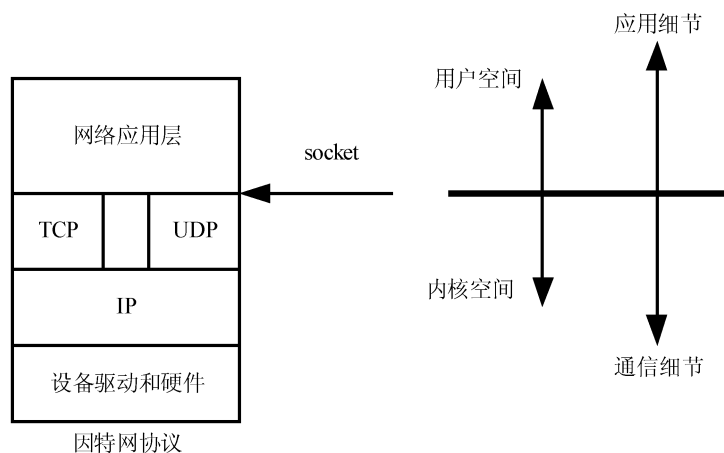


图 11-2 网络协议和套接字的关系

11.2.1 Linux 套接字编程的基础

Linux 网络编程需要经常使用一些与内存相关的函数，如下所示：

```
#include <strings.h>
void bzero(void *s, size_t n);
void bcopy(const void *src, void *dest, size_t n);
void *memset(void *s, int c, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
```

使用上面的函数在网络编程中可以将一个需要使用的结构体初始化为 0。

网络的地址在编程的过程中，需要从字符串到二进制方式的相互转换。在 Linux 中，提供了一些可以直接使用的函数。网址字符串和数字的转换关系如图 11-3 所示。

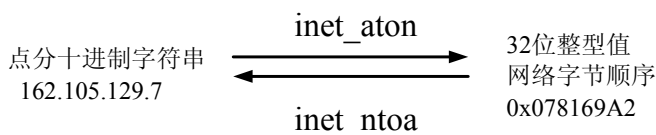


图 11-3 网址字符串和数字的转换

网址字符串和数字进行转换的函数如下所示：

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in);
```

`inet_aton()`用于将 32 位的数字点形式表示的字符串 IP 地址与 32 位的网络字节顺序的二进制形式的 IP 地址进行转换。`inet_ntoa()`用于将网络地址转换成“.”点隔的字符串格式。

`inet_pton()`用于将 IPv4 或 IPv6 字符串地址转换成网络主机地址。`inet_ntop()`用于将 IPv4 或 IPv6 网络主机地址转换成字符串。

套接字特有的操作在 `socket.h` 头文件中，主要的接口如下所示：

```
#include <sys/types.h>
#include <sys/socket.h>
struct sockaddr {
    sa_family_t sa_family;    // sa_family_t 的实际类型是 unsigned short
    char sa_data[14];
}
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
}
struct in_addr {
    __u32 s_addr;
};
int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int shutdown(int sockfd, int how);
```

`sockaddr` 结构是套接字通用的地址结构，由于 `sa_family_t` 占用两个字节，本结构的大小为 16 字节。`sockaddr` 结构可以进行扩展，只要保证前两个字节表示 `sa_family_t`（协议族）即可，后面的若干个字节可以自定义。`sockaddr_in` 结构就是 `sockaddr` 的扩展，其中的 `in` 表示 Internet，本结构为 IP 协议使用。

几个函数的功能如下所示：

- **socket()**函数通过协议族和套接字类型信息，返回一个整数类型的套接字描述符，基本等同于文件描述符。
- **bind()**函数用于绑定到套接字分配 IP 地址和端口号，必须和网络中的 IP 地址和端口一致。
- **listen()**函数使 **socket** 在协议地址上进行监听，并为该套接字建立一个连接队列，将到达的服务请求保存在此队列中，直到程序处理它们。
- **accept()**函数用于 TCP 服务器等待客户端的连接。使用 **listen()**建立好输入队列后，服务器可以调用 **accept()**函数，阻塞式地等待客户端的连接请求。
- **connect()**函数在 TCP 客户端使用，用于连接服务器。

在网络协议的函数中，参数 **domain** 表示协议的域，可取的几个值如下所示。

- **AF_INET (PF_INET)**: IPv4 网络协议。
- **AF_INET6 (PF_INET6)**: IPv6 网络协议。
- **AF_UNIX, AF_LOCAL**: 本地套接字。

参数 **type** 表示协议的类型，可取的几个值如下所示。

- **SOCK_STREAM**: 面向连接的稳定数据传输 (TCP)。
- **SOCK_DGRAM**: 不连续、不可靠的数据包连接 (UDP)。
- **SOCK_RAW**: 原始网络协议存取。

参数 **protocol** 用来指定套接字所使用的传输协议编号。这一参数通常不具体设置，一般设置为 0。

套接字在被建立之后等同于文件描述符，所以常常对其使用普通文件的操作，例如：**read()**、**write()**、**lseek()**、**poll()**、**close()**等用于文件操作函数，也可用于套接字；**recv()**、**send()**、**recvfrom()**和 **sendto()**则用于套接字发送和接收。

几个套接字的发送和接收函数如下所示：

```
#include <sys/types.h>
#include <sys/socket.h>
int recv( int sockfd,void *buf,int len,int flags );
int send( int sockfd,void *buf,int len,int flags );
int recvfrom( int s, void *buf, size_t len, int flags,
               struct sockaddr *from, socklen_t *fromlen );
int sendto( int s, const void *msg, size_t len, int flags,
             const struct sockaddr *to, socklen_t tolen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

在面向连接的 TCP 通信中，TCP 服务器首先需要建立 (**socket**) 并绑定 (**bind**) 端口，然后监听 (**listen**) 端口，之后阻塞等待客户端的连接 (**accept**)。TCP 的客户端连接 (**connect**) 到服务器上。连接的过程，服务器和客户端需要进行 3 次握手，这由协议的下层在内部实现。

在面向数据报的 UDP 协议中，方式简单很多，根据所建立的端口，UDP 的服务器和客户端需要使用 **recvfrom** 和 **sendto** 等函数完成数据的收发。

11.2.2 TCP 和 UDP 协议的流程

套接字编程通常可以基于面向连接的 TCP 协议和面向数据流的 UDP 协议。

TCP 协议是面向流（Stream）的，需要有连接建立，因此在 TCP 会话初期有“3 次握手”的过程，即对每次发送的数据量跟踪协商，使数据段的发送和接收同步，根据所接收到的数据量而确定数据发送、接收完毕后何时撤销联系，并建立虚连接。为了提供可靠的传送，TCP 在发送新的数据之前，以特定的顺序建立数据包的序号，并需要得到这些包传送给目标机之后的确认消息。从编程的角度，TCP 的服务器端调用 `bind()` 函数、`listen()` 函数之后，调用等待连接的 `accept()` 函数，此时 TCP 的客户端调用 `connect()` 函数则可以产生到服务器的连接。连接建立后，可以向读写普通文件一样在套接字中传送数据。

UDP 协议是面向数据报（Datagram）的，没有连接建立，UDP 数据报本身的头部由 4 个部分组成：源端口、目的端口、长度、检验组成，共 8 个字节。因此 UDP 服务器端和 UDP 客户端根据指定的地址和端口号，可以直接进行数据收发动作。

从编程的角度，TCP 协议和 UDP 协议的示例流程如图 11-4 所示。

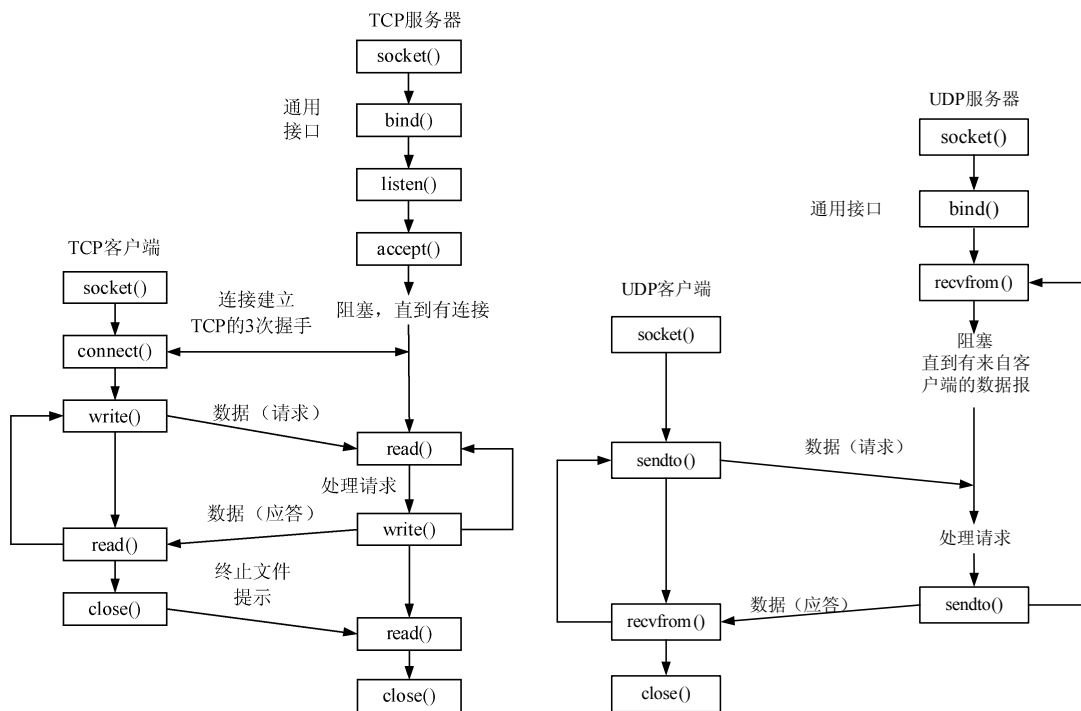


图 11-4 TCP 协议和 UDP 协议的示例流程

11.2.3 TCP 编程实例

本小节介绍一个基于 TCP 协议的客户端-服务器结构和网络程序。这是一个简单而完整的程序，它完成的功能是从客户端向服务器发送一个字符串，在服务器端将其转换成大写，送回客户端。程序非常简单，但是包含了 TCP 协议通信的基本内容和完整的流程。

1. TCP 服务器端程序

下面的内容是一个 TCP 服务器端的程序：等待客户端建立连接，将客户端发送的数据转换成大写后，发回给客户端。

TCP 服务器端程序如下所示：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAXLINE 80           // 字符串的最大长度
int port = 8000;             // 服务器提供的端口号
int main(void)
{
    struct sockaddr_in sin;    // 初始化服务器 socket 端口
    struct sockaddr_in pin;    // accept 后得到的新端口
    int listen_fd;             // accept 中服务器 socket 的描述符
    int conn_fd;               // accept 返回客户的套接字描述符
    int address_size;          // accept 返回地址的长度
    char buf[MAXLINE];         // 字符串缓冲区
    char str[INET_ADDRSTRLEN]; // 保存客户端的 IP 地址
    int i;
    int len;

    bzero(&sin, sizeof(sin)); // 在程序中，首先初始化服务器的 Socket 端口 sin
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port);

    listen_fd = socket(AF_INET, SOCK_STREAM, 0); // 监听套接字描述符
    bind(listen_fd, (struct sockaddr *) &sin, sizeof(sin)); // 绑定到端口

    listen(listen_fd, 20); // 绑定完成后，程序开始监听
    printf("Accepting connections ...\n");

    while (1) { // 程序下面进入循环，等待客户端的连入
        conn_fd = // 接受来自客户端的连接
            accept(listen_fd, (struct sockaddr *) &pin, &address_size);
        read(conn_fd, buf, MAXLINE);
        printf("received from client %s at port %d: %s\n",
            inet_ntop(AF_INET, &pin.sin_addr, str, sizeof(str)),
            ntohs(pin.sin_port), buf);

        read(conn_fd, buf, MAXLINE); // 可以用标准的函数 read 从中读取信息
        printf("received from client %s at port %d: %s\n",
            inet_ntop(AF_INET, &pin.sin_addr, str, sizeof(str)),
            ntohs(pin.sin_port), buf);
        len = strlen(buf);
        for (i = 0; i < len; i++) { // 转换字符到大写
            buf[i] = toupper(buf[i]);
        }
        write(conn_fd, buf, len + 1); // 使用标准写函数 write() 发送
        close(conn_fd);
    }
}
```

在程序中，需要先使用 `socket()` 函数建立一个套接字的描述符 (`listen_fd`)。建立的时候，使用 `SOCK_STREAM` 参数，表示流类型（对应 TCP 协议）的套接字，适用于 TCP 协议。建立之后，它和前面定义的端口 `sin` 还没有关系，需要使用 `bind()` 函数将它们绑定。

初始化端口 `sin` 中，首先使用 `bzero` 将端口数据结构完全置 0，然后将端口协议组置为 `AF_INET` (IPv4 协议)。将端口中的地址置为 `INADDR_ANY` (任何地址)，表示所有地址。将端口号 (`port`) 用宏 `htons` 处理后进行设置，`htons()` 用于将主机的 16 位（短整型）数据类型转换成网络字符顺序。

在调用 `bind()` 函数的时候，参数为数据结构 `sin`，其中地址值 (`sin.sin_addr.s_addr`) 可以为 `INADDR_ANY` (通配地址)，也可以为本地 IP 地址。端口 (`sin.sin_port`) 可以为一个非 0 值，也可以为 0。如果是 0，系统就会分配一个非 0 值。本地可以有多个 IP 地址，一旦绑定，则此套接字上发送的数据以此地址为 IP 源地址。`sin` 是 `struct sockaddr_in` 结构的数据类型，在 `bind()` 函数调用中，转换成 `struct sockaddr` 数据类型使用，这两个数据结构的大小是一样的。

`accept()` 函数用于接受来自客户端的连接。`accept()` 函数的返回值 `conn_fd` 为连入的套接字描述符。参数 `listen_fd` (输入) 为当前服务器的套接字描述符，`pin` (输出) 将返回服务器端新的端口，参数 `address_size` (输入/输出) 得到地址的长度。

`accept()` 函数的第一个参数 (`socket`)，当有连接的时候，`accept()` 会返回一个新的 `socket`，之后数据的传送与读取需使用新的 `socket` 完成，而原来参数 `s` 的 `socket` 能继续使用 `accept()` 来接受新的连接。由于 `accept()` 函数会一直等待连接阻塞程序的运行，因此该 TCP 服务器的 `while` 循环也是不会退出的。`listen_fd` 的值是不会改变的，每次调用 `accept()` 都将返回一个连接的描述符 (`conn_fd`)。然会使用这个描述符完成通信。

2. TCP 客户端程序

下面的内容是对应的 TCP 客户端程序：从命令行的参数中获得一个字符串，然后发给服务器，并接收服务器返回的字符串。如果命令行不带任何参数，就会发出默认的字符串："A default test sting"。

程序开始包含必要的头文件，并完成宏定义：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAXLINE 80
int port = 8000;
int main(int argc, char *argv[])
{
    struct sockaddr_in pin;                // 客户端的 socket 端口
    char buf[MAXLINE];                    // 字符串缓冲区
    int sock_fd;                          // 端口描述符
    char *str = "A default test sting";    // 默认字符串
    if (argc > 1) {                       // 通过命令行获得输入字符串
```

```

    str = argv[1];                // 如果命令行有输入, str 置为命令行字符
}
bzero(&pin, sizeof(pin));        // 初始化端口, 这个端口代表了远端服务器的信息
pin.sin_family = AF_INET;
inet_pton(AF_INET, "127.0.0.1", &pin.sin_addr);
pin.sin_port = htons(port);
sock_fd = socket(AF_INET, SOCK_STREAM, 0);    // 建立套接字, 并且连接到服务器
connect(sock_fd, (void *) &pin, sizeof(pin)); // 连接到服务器
write(sock_fd, str, strlen(str) + 1);        // 向套接字的文件描述符中写入字符串
read(sock_fd, buf, MAXLINE);                // 从服务器的描述符中读出服务器发回的字符串
printf("Response from server: %s\n", buf);
close(sock_fd);                            // 关闭套接字, 程序结束
return 0;
}

```

在 TCP 客户端的运行过程中, `connect()` 函数内部将通过 3 次握手完成建立服务器连接。由于参数 `pin` 的地址设置为本机 (127.0.0.1), 此时将连接到本机的服务器端。

提示: 客户端程序的 `connect` 过程之前, 服务器正在调用函数 `accept()`, 处于阻塞状态; 客户端程序 `connect` 的时候, 服务器的函数 `accept()` 将返回。

3. 程序说明

程序的使用过程如下: 在本机首先运行服务器端程序, 然后运行客户端程序, 并从命令行输入发送的字符串。程序运行的结果为: 服务器端将显示客户端的信息和输入字符串, 客户端将得到转换为大写以后的字符串。本程序的实质是在一台机器上通过 TCP 协议完成了一次通信过程。

服务器端程序的运行如下所示:

```

$ ./tcpserver
Accepting connections ...

```

客户端程序的运行如下所示:

```

$ ./tcpclient
Response from server: A DEFAULT TEST STING
$ ./tcpclient hello
Response from server: HELLO

```

客户端执行后, 服务器端继续打印出如下内容:

```

received from client 127.0.0.1 at port 57938: A default test sting
received from client 127.0.0.1 at port 57939: hello

```

这个例子中的网络程序是最简单的网络协议, 其功能简单, 而且不包括错误处理。系统调用不能保证每次都能成功。如果不成功, 则需要进行出错处理。这样一方面为了保证程序逻辑正常, 另一方面可以迅速得到故障信息。

对于服务器, 依次使用了以下的调用: `socket()`、`bind()`、`listen()`、`accept()`、`read()`、`write()`。

对于客户端, 依次使用了以下的调用: `socket()`、`connect()`、`read()`、`write()`。

可以增加简单错误处理, 例如 `connect()` 的处理:

```

n = connect( sock_fd, ( void * ) &pin, sizeof( pin ) );

```

```

if ( n == -1 ) {
    perror( "call to connect" ), exit( -1 );
}

```

对于 `read()` 和 `write()` 操作，应该考虑得更多一些。最常遇到的一个情况是，如果读操作和写操作还没有开始，就被信号中断，此时对 `read` 和 `write` 的调用也返回 -1，并且 `errno` 置为 `EINTR`。对于这种情况，应该再次调用 `read` 和 `write`，因为 `socket` 此时仍然是完好健康的。

一种典型的处理方法如下所示：

```

n = read( sock_fd, buf, MAXLINE );
if ( n == -1 ) {
    if ( errno == EINTR ) {
        goto readagain;
    } else {
        perror( "call to read" );
        exit( 1 );
    }
}
// 直接退出，也可以进行其他的错误处理

```

上面的方式是继续读的做法。此外，如果 `read` 的返回为 0，则表明已经读到文件末尾，对于网络 `socket`，则表明客户端已经主动地关闭了连接。

对于 `close` 调用，通常对其返回值不做检查，但最好还是进行一下检查。因为 `close` 也有可能不成功。如 `write()` 函数不返回错误往往也不代表确实写成功（这是因为为提高性能，`write` 可能使用“后写”技术），若写出错，调用 `close` 时，则会返回一个错误。如果不检查这个值，就会漏掉这种错误。

11.2.4 UDP 编程实例

与 TCP 编程相比，UDP 编程相对简单，至少不需要调用 `listen()` 和 `accept()` 函数。

1. UDP 服务器端程序

下面的内容是一个 UDP 服务器端程序：它可以将客户端发送的数据转换成大写后，发回给客户端。

程序开始包含头文件和宏定义：

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <errno.h>
#define MAXLINE 80 // 最大字符串长度
int port = 8000; // 端口号
int main(void)
{
    struct sockaddr_in sin; // 初始化服务器 socket 端口
    struct sockaddr_in rin; // 接收的 socket 端口

```

```

int sock_fd;                // socket 描述符
int address_size;           // 返回地址的长度
char buf[MAXLINE];         // 字符串缓冲区
char str[INET_ADDRSTRLEN]; // 保存客户端的 IP 地址
int i;
int n;
int len;
bzero(&sin, sizeof(sin)); // 初始化接收的 socket 端口
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(port);
sock_fd = socket(AF_INET, SOCK_DGRAM, 0); // 建立套接字
if (sock_fd == -1) {
    perror("call to socket");
    exit(1);
}
n = bind(sock_fd, (struct sockaddr *) &sin, sizeof(sin)); // 绑定到套接字
if (n == -1) {
    perror("call to bind");
    exit(1);
}
while (1) {                // 程序进入循环, 接收来自客户端的信息
    address_size = sizeof(rin);
    n = recvfrom(sock_fd, buf, MAXLINE, 0, // 从客户端接收数据
                  (struct sockaddr *) &rin, &address_size);
    if (n == -1) {
        perror("call to recvfrom.\n");
        exit(1);
    }
    printf("received from client %s at port %d: %s\n",
           inet_ntop(AF_INET, &rin.sin_addr, str, sizeof(str)),
           ntohs(rin.sin_port), buf);
    len = strlen(buf);
    for (i = 0; i < len; i++) // 转换成大写
        buf[i] = toupper(buf[i]);
    n = sendto(sock_fd, buf, len + 1, 0, (struct sockaddr *) &rin,
               address_size); // 发送给客户端
    if (n == -1) {
        perror("call to sendto.\n");
        exit(1);
    }
}
}

```

初始化服务器的 socket 端口的时候：协议组置为 AF_INET (IPv4 协议)。将端口中的地址置为 INADDR_ANY (任何地址)，表示所有地址。将端口号 (port) 用宏 htons 处理后设置，htons() 用于将主机的 16 位 (短整型) 数据类型转换成网络字符顺序。建立 socket 使用了参数 SOCK_DGRAM，表示数据报类型 (对应 UDP 协议)，此处判断套接字描述符是否合法，如果不合法将返回 -1，程序在此退出。

在 UDP 的程序中，不需要使用 accept() 建立连接，直接接收信息即可。recvfrom() 的返回值是获取的接收到的字符个数，参数 sock_fd (输入) 为服务器 Socket 描述符，rin (输出) 是来自客户端 (发送字符者) 的端口信息，address_size (输出) 为客户端地址长度。也就是说函数 recvfrom() 在返回数据的同时，也返回了客户端的信息。

提示：在 UDP 的程序中，不使用 `accept()` 建立连接；而直接使用 `recvfrom()` 函数接收数据。
`recvfrom()` 函数本身会阻塞程序，直到有数据输入。

在以上的程序中，一旦 `recvfrom()` 函数返回，即表示已经完成了一次 UDP 客户端到服务器的通信过程。在后面的程序中把接收到的字符转换成大写，发送回客户端：

从 UDP 服务器程序中可以看出，在 UDP 协议中，接收使用 `recvfrom()` 函数，需要使用套接字描述符并获取发送者的套接字地址；发送使用 `sendto()`，使用套接字描述符，目标用接收者的套接字地址表示。

2. UDP 客户端程序

下面的内容是对应 UDP 客户端的程序：在命令行中向服务器端发送数据，接收到服务器的数据后打印出。

程序的开始包含头文件和宏定义：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <stdlib.h>
#define MAXLINE 80          // 最大字符串长度
int port = 8000;           // 端口号
int main(int argc, char *argv[])
{
    struct sockaddr_in pin;    // socket 端口
    struct sockaddr_in rin;    // 接收 socket 端口
    char buf[MAXLINE];        // 字符串缓冲区
    int sock_fd;              // socket 描述符
    char str[MAXLINE];
    char sip[INET_ADDRSTRLEN];
    int n;
    int address_size;
    bzero(&pin, sizeof(pin)); // 初始化 socket 端口:
    pin.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &pin.sin_addr); // 地址设置为 127.0.0.1, 代表本机
    pin.sin_port = htons(port);
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0); // 建立套接字
    if (sock_fd == -1) {
        perror("call to socket");
        exit(1);
    }
    while (fgets(str, MAXLINE, stdin) != NULL) { // 从命令行接收字符
        sendto(sock_fd, str, strlen(str) + 1, 0, // 将字符发送到服务器端
            (struct sockaddr *) &pin, sizeof(pin));
        if (n == -1) {
            perror("call to sendto.\n");
            exit(1);
        }
        address_size = sizeof(rin);
        n = recvfrom(sock_fd, buf, MAXLINE, 0, (struct sockaddr *) &rin, &address_size);
```

```

        if (n == -1) {
            perror("call to recvfrom.\n");
            exit(1);
        } else {
            printf("Response from %s port %d: %s\n",
                inet_ntop(AF_INET, &rin.sin_addr,
                    sip, sizeof(sip)), ntohs(rin.sin_port), buf);
        }
    }
    close(sock_fd);          // 关闭描述符, 退出循环
    if (n == -1) {
        perror("call to close.\n");
        exit(1);
    }
    return 0;
}

```

UDP 客户端的程序也使用了 `sendto()` 函数和 `recvfrom()` 函数, 完成数据的收发。

3. 程序说明

在上述程序中, 首先建立服务器程序, 然后运行客户端程序, 在客户端程序运行的过程中, 可以从命令行输入字符, 然后客户端将字符发送给服务器, 服务器收到字符后将字符转换成大写后发回给客户端。

服务器运行后, 等待客户端的数据, 收发数据后的效果如下所示:

```

$ ./udpserver
received from client 127.0.0.1 at port 53556: udp
received from client 127.0.0.1 at port 53556: hello

```

客户端对应的运行情况如下所示:

```

$ ./udpclient
udp
Response from 127.0.0.1 port 8000: UDP

hello
Response from 127.0.0.1 port 8000: HELLO

```

在本节 UDP 客户端程序包含了一个循环, 通过这个循环可以从命令行不断获取字符, 并发送到服务器。在程序的使用上, UDP 和 TCP 的方式基本相同, 只是内部工作的机制不相同。

在 UDP 网络程序中, 客户端和服务器的分工并不明显, 二者基本是对等的, 没有建立连接的问题。在 UDP 编程中并没有连接的概念, 在发送和接收的过程使用 `recvfrom()` 函数和 `sendto()` 函数, 其中的参数包括了发送和接收的目标。

11.2.5 深入网络编程

在 TCP 网络服务器的实现中, `accept()` 函数本身可以实现阻塞, 使用其可以实现轮询的效果。在一个服务器对多客户端的结构中, `accept()` 可以结合 `fork` 机制来使用, 子进程中对一个连接进行处理, 父进程则在继续循环中等待下一个连接。

一个典型的程序结构如下所示：

```
while (1) {
    if((conn_fd = accept(listen_fd, (struct sockaddr *) &pin, &address_size))!=-1){
        continue;
    }
    if (0==fork()) {
        // .....省略代码：子进程中对一个连接的处理
        close(conn_fd);    // 在子进程中关闭连接
        exit(0);
    }
    close(conn_fd);        // 在主进程中关闭连接，在循环中等待连接
}
```

与基本的基于 TCP 的套接字操作相比，上面的程序仅仅对于 `accept()` 函数返回的结果调用了 `fork()`，由此会产生一个新的进程，在这个子进程当中处理连接的情况，而主进程继续等待新的连接。由此，处理请求的程序和等待连接的程序在运行时被分开，请求的处理不会阻塞连接。

网络服务器的实现还可以实现多路复用的模式。在此种模式下，可以避免创建过多子进程带来的开销。使用的原理是记录请求的多个文件描述符，在不阻塞的情况下，可以直接返回，然后进行处理，还可以设置超时。

Linux 编程通过 `select()` 函数实现多路复用（选择），主要的函数如下所示：

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

其中，`select()` 函数的几个参数如下所示。

- **ndfs**：监视的文件描述符数，视进程中打开的文件数而定，一般设为要监视各文件中的最大文件号加一。
- **readfds**：监视的可读文件描述符集合。
- **writefds**：监视的可写文件描述符集合。
- **exceptfds**：监视的异常文件描述符集合。
- **timeout**：本次 `select()` 的超时结束时间。

几个相关的辅助宏的含义如下所示。

- **FD_ZERO()**：清空 `fdset` 与所有文件描述符的联系。
- **FD_SET()**：建立 `fd` 与 `fdset` 的联系。
- **FD_CLR()**：清除 `fd` 与 `fdset` 的联系。
- **FD_ISSET()**：检查 `fdset` 联系的 `fd` 是否可读写。

11.2.6 用作 IPC 的 UNIX Socket

UNIX Socket 是在 Socket 的框架上发展而来的一种机制，它使用 Socket 的接口形式，但是与网络无关，UNIX Socket 主要用于 IPC（Inter Process Communication，进程间通信）。在 Linux 系统中 UNIX Socket 就是 Local Socket（本地套接字），Socket 本身的存在形式就是文件系统中的文件。UNIX Socket 也提供面向流和面向数据报两种 API 接口，类似于 TCP 和 UDP，操作的形式基于 Socket 通常的接口。

● UNIX Socket 和 IP 网络 Socket 的比较

基于 IP 网络的 Socket 通常也可用于同一台主机的进程间通信，通常可以使用本地地址 localhost（127.0.0.1）和某个自定义的端口。与之相比，UNIX Socket 用于 IPC 不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程即可。因此，UNIX Socket 相比基于 IP 的网络 Socket 更有效率。

● UNIX Socket 和 FIFO 的比较

FIFO 本质是管道，打开后可以作为一个先入先出文件操作。UNIX Socket 是全双工的，不仅能使用普通的文件接口，也有更加丰富的 Socket 接口。

使用 UNIX Socket 的过程与普通 Socket 类似，Socket 本身由 socket() 函数建立，返回套接字文件描述符，协议族（family）指定为 AF_UNIX，协议类型（type）可以选择 SOCK_DGRAM 或 SOCK_STREAM，protocol 参数如无特别含义可以指定为 0。

使用 Unix Socket 的服务器如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
// .....省略代码: 命令处理函数 command_handler()
int main (int argc, char* const argv[]) {
    const char* const socket_name = argv[1]; // 命令行第一个参数: UNIX Socket 文件名
    int socket_fd;
    struct sockaddr_un local_socket;
    int is_quit;
    if(!access(socket_name, F_OK)) {           // 查看 UNIX Socket 是否存在
        printf("UNIX socket [%s] is existed\n", socket_name);
        unlink(socket_name);
    }
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0); // 建立 UNIX Socket
    local_socket.sun_family = AF_LOCAL;           // 设定协议族
    strcpy (local_socket.sun_path, socket_name); // 设定套接字的名称
    bind(socket_fd, (const struct sockaddr*)&local_socket, SUN_LEN (&local_socket));
    listen (socket_fd, 5);
    while(1) {
        struct sockaddr_un client_name;
        socklen_t client_name_len;
        int client_socket_fd;
        client_socket_fd = accept (socket_fd, // 接收客户端的套接字
                                   (struct sockaddr*)&client_name, &client_name_len);
        is_quit = command_handler(client_socket_fd);
    }
}
```

```

        close (client_socket_fd);                // 关闭一个客户端的连接
    if (-1 == is_quit){
        printf("Break Command Loop\n");
        break;
    }
}
close (socket_fd);
unlink (socket_name);
return 0;
}

```

服务器在命令程序运行后,将以第一参数为名称建立一个套接字文件,以此等待客户端的连接并进行处理。在 Linux 系统中 PF_LOCAL 就是 PF_UNIX, AF_LOCAL 就是 AF_UNIX。然后需要设立一个 UNIX Socket 的名字,代表一个文件系统中的文件。其他 Socket 的处理过程和普通 Socket 基本相同。当通过调用 accept() 从客户端得到一个 Socket 连接之后,后续的操作由 command_handler() 函数完成,这实际上就是来自客户端命令的处理。

command_handler() 函数的实现如下所示:

```

#define LEN 128
int command_handler (int client_socket_fd) {
    char buffer[LEN];
    int i, size;
    size = read (client_socket_fd, buffer, LEN); // 读取套接字的文件
    printf ("[command] size=%d [%s]\n", size, buffer);
    if (!strcmp (buffer, "quit")){                // 表示退出的特殊处理
        sleep(2);
        write (client_socket_fd, "END", 4);      // 向客户端写回特定字符
        return -1;
    } else {                                     // 各个命令的处理
        for (i = 0; i < size-1; i++) {
            buffer[i] = toupper(buffer[i]);      // 小写转为大写,然后向客户端写回
        }
        write (client_socket_fd, buffer, size);
        return 0;
    }
}
}

```

本函数的实现只是简单的读写,实际上与 Socket 的操作并无特定关系。来自客户端的命令可以在此处进行解析,处理完成后用字符串返回给客户端。UNIX 套接字本身具有套接字的所有特性,例如,读写的先入先出、阻塞等。

提示: 服务器可以一面建立线程在后台运行,一面用字符串形式的接口与客户端进行交互。如果命令的形式比较简单,可以简化为单个字符,由此也可以用 switch case 的方式处理。

使用 UNIX Socket 的客户端如下所示:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#define LEN 128
void do_with_socket (int socket_fd, const char* text) { // 在 Socket 的操作

```

```

char buffer[LEN] = {};
int size;
write (socket_fd, text, strlen(text)+1); // 写入一个命令
printf ("[send] size=%d [%s]\n", (int)strlen(text)+1, text);
size = read (socket_fd, buffer, LEN);    // 读取来自服务器的返回结果
printf ("[receive] size=%d [%s]\n", size, buffer);
return;
}

int main (int argc, char* const argv[]) {
    const char* const socket_name = argv[1]; // 命令行第一个参数: UNIX Socket 文件名
    const char* const message = argv[2];    // 命令行第二个参数: 命令字符串
    int socket_fd;
    struct sockaddr_un local_socket;
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0); // 建立 UNIX Socket
    local_socket.sun_family = AF_LOCAL;           // 设定协议族
    strcpy (local_socket.sun_path, socket_name); // 设定套接字的名称
    connect (socket_fd, (const struct sockaddr *)&local_socket,
            SUN_LEN (&local_socket)); // 连接到套接字
    do_with_socket(socket_fd, message); // 进行一次的命令处理
    close (socket_fd);
    return 0;
}

```

客户端以命令程序运行后，参数是套接字的名称。本程序是一个单次运行的程序，在 `do_with_socket()` 函数中进行一次命令处理，先向服务器写入一个命令，然后等待服务器的返回。由于套接字文件描述符本身的特性，读写的过程都与另外一端配合，否则就会引起阻塞。

经过编译后让服务器生成 `ipcservice`，客户端生成 `ipcclient`。

服务器运行如下所示：

```

$ ./ipcservice ipc_file
UNIX socket [ipc_file] is existed

```

如果文件已经存在，将出现上述信息。服务器程序运行后，将会以命令行的第一个参数在当前目录中建立名为 `ipc_file` 的文件，此文件就是 Linux 其中的一个特殊套接字文件，代表字符为“s”。如果使用 `stat()` 函数查看信息，其 `st_mode` 的值为 `SOCK`（0x14）。

第一次和第二次运行客户端如下所示：

```

$ ./ipcclient ipc_file abc
[send] size=4 [abc]
[receive] size=4 [ABC]
$ ./ipcclient ipc_file hello
[send] size=6 [hello]
[receive] size=6 [HELLO]

```

此过程中，服务器的输出为：

```

[command] size=4 [abc]
[command] size=6 [hello]

```

前两次都是正常运行，客户端发送到服务器的命令以参数返回。

第三次运行客户端如下所示：

```

$ ./ipcclient ipc_file quit

```

```
[send] size=5 [quit]
[receive] size=4 [END]
```

此过程中，服务器的输出为：

```
[command] size=5 [quit]
quit Command Loop
```

注意：在第 3 次运行后，服务器延迟了 2 秒才返回，因此客户端的命令行也是在 2 秒之后才输出[receive]，表示 read()操作被阻塞了。运行完成后，服务器已经退出。

在服务器退出或者没有运行的情况下，继续执行命令如下所示：

```
$ ./ipccclient ipc_file null
[send] size=5 [null]
[receive] size=-1 []
```

发送之后，由于套接字本身没有服务器进行阻塞，因此 read()操作直接返回，返回的数目为-1。

11.3 GUI 应用开发

图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口）是指采用图形方式显示的计算机操作用户界面。在桌面系统中，如微软的 Windows 操作系统、Linux 下的 GTK+和 QT 都属于 GUI 系统的范畴。GUI 在嵌入式系统中也有了广泛的应用。相比桌面计算机系统，嵌入式 GUI 的功能相对简单，只是要提供给上层的应用程序绘制图形界面以及接收用户输入的能力。

从实现的方式上，GUI 既可以是一套库，也可以是和应用程序一起编译的源代码。嵌入式图形库的软件结构如图 11-5 所示。

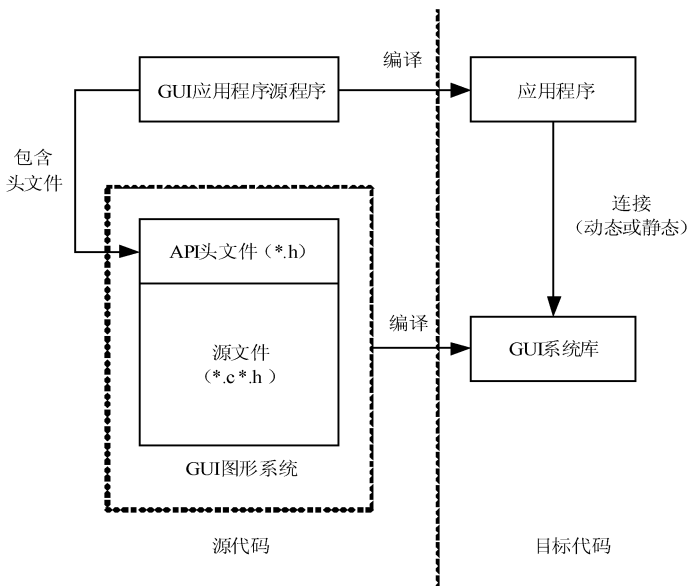


图 11-5 嵌入式图形库的软件结构

一般来说，嵌入式 GUI 系统分为 API 头文件和源文件两部分。API 头文件给应用程序使用，图形库源文件将被编译成为动态库或者静态库，应用程序编译后需要动态或者静态连接图形库。所谓静态连接，就是将图形库编入应用程序中（lib*.a）；动态连接是不将图形库编入应用程序中，而作为共享库（lib*.so）使用，动态库适用于多个应用程序同时使用一个 GUI 系统的情况，可以节省空间。

在嵌入式系统中，随着系统性能的提高和人机交互的要求，GUI 的应用越来越广泛。在嵌入式 GUI 开发中，包括 GUI 库的移植和应用程序开发两部分内容。GUI 库移植的重点是显示部分，对应 Linux 系统一般只需要通过 FrameBuffer 就可以完成；难点是输入部分（如：按键、触摸屏、键盘等）。在嵌入式 GUI 应用程序的开发中，一般都调用 GUI 图形库上层的接口，这可以屏蔽下层具体硬件和操作系统的信息。这使得嵌入式 GUI 的程序具有一定的可移植性，甚至可以将桌面的 GUI 程序移植到嵌入式系统中。

Linux 嵌入式系统的 GUI 系统结构如图 11-6 所示。

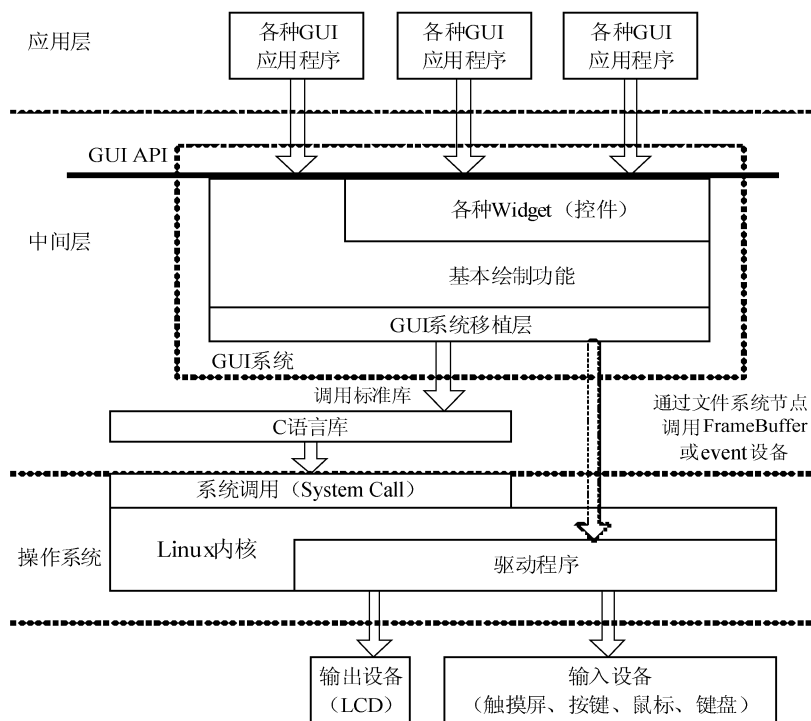


图 11-6 Linux 的 GUI 系统结构

Linux 的 GUI 系统通常分成了几个层次：GUI 系统移植层、GUI 系统核心、GUI 应用程序，其中 GUI 系统移植层和 GUI 系统核心属于中间件。在移植方面，GUI 系统通常需要考虑输出设备和输入设备两方面的情况。输出设备是系统运行的重点，负责显示图形界面，在 Linux 操作系统中，通常使用 FrameBuffer 作为驱动程序；输入设备具有多样性，包括了触摸屏、按键、鼠标、键盘等类型，在 Linux 操作系统中，Event 是比较标准的输入驱动框架。某些 GUI 系统为了屏蔽硬件之间的差异，其移植层通常考虑了驱动程序的差异。

GUI 系统的核心通常也有图形层和控件层两个层次。图形层提供基本的绘制功能，例如画点、画线、画圆等几何图形的绘制。控件层提供了 Widget 等功能，Widget 通常可以 and 用户进行交互，例如文本框、菜单、对话框等功能。控件层需要调用图形层完成绘制功能。在应用程序层，通常需要调用控件层实现各种功能，有的时候也可以直接调用图形层完成绘制。

11.3.1 Qt 系统

1. Qt 应用程序概述

Qt 是一个跨平台的 C++ 应用程序开发框架。其广泛用于开发 GUI 程序，也可用于开发非 GUI 程序。Qt 可以支持 Windows、Linux/X11、Mac OS X 等平台。Qt 的名称源于 1994 年成立的 Quasar Technologies 公司，该公司后更名为 TrollTech，中文名是“奇趣科技”。TrollTech 于 2008 年被 Nokia 公司收购，更名为 Qt 发展框架 (Qt Development Frameworks)。

Qt 工程的网站为 <http://qt-project.org/>。

Qt 开放源代码，并且提供自由软件的用户协议。这使得它可以被广泛地应用于各平台的开放源代码软件开发中。Qt 提供 3 种授权方式。3 种授权方式的功能、性能均没有区别，仅在于授权协议的不同。LGPL 和 GPL 是免费发布的，商业版则需收取授权费。

- Qt 商业版：Qt 商业授权适用于开发专属和/或商业软件。此版本适用于不希望与他人共享源代码，或者遵循 GNU 宽通用公共许可证 (LGPL) 2.1 版或 GNU GPL 3.0 版条款的开发人员。提供了技术支持服务。可以任意地修改 Qt 源代码，而不需要公开。
- GNU LGPL v2.1：Qt 4.5.0 及以后的版本开始遵循 GNU LGPL，允许连接到它的软件使用任意的许可证，可以被专属软件作为类库引用、发布和销售。可以购买支持服务。
- GNU GPL v3.0：如果你希望将 Qt 应用程序与受 GPL 3.0 版本条款限制的软件一同使用，或者希望 Qt 应用程序遵循该 GNU 许可证版本的条款，则此版本 Qt 适用于开发此类 Qt 应用程序。可以购买支持服务。

Qt 具有 Qt3、Qt4 和 Qt5 等版本，它们的编程方法基本相同，仅在某些类的使用上略有区别。Qt5 版本时，提供了 Qt Creator 工具作为 IDE 使用。

Qt 按照模块区分功能，其中包含的模块如下所示。

- Qt Core：核心的模块，与图形无关，可以被其他模块使用。
- Qt GUI：GUI 的基础 (Qt5 中包括 OpenGL)。
- Qt Multimedia：多媒体的核心类 (audio、video、radio 和 camera)。
- Qt Multimedia Widgets：多媒体的 GUI 的类。
- Qt Network：网络的编程类。
- Qt QML：用于 QML 和 JavaScript 语言编程。
- Qt Quick：使用客户用户接口的动态应用。
- Qt Quick Controls：可重用的基于 Qt Quick 的控件。
- Qt Quick Layouts：用于摆放 Qt Quick 项目。

- Qt SQL: 用于 SQL 数据库。
- Qt Test: 用于 Qt 应用和库单元测试。
- Qt WebKit: 用于浏览器开发 (Qt 4 和 Qt 5 有变化)。
- Qt Widgets: Qt GUI 的扩展。

除此之外, 还有 Qt Add-Ons 作为可插入的组件使用。例如: Qt Print、Qt D-Bus、Qt Print Support、Qt SVG 等。

QTE 是 Qt 的嵌入式版本, 它在原始 Qt 的基础上, 做了许多出色的调整以适合嵌入式环境。它提供了和 Qt 基本相同的 API。由此对上层的应用程序保持兼容性, 使用 QTE 可以方便地将桌面 Linux 上的 Qt 程序移植到嵌入式系统中。由于 QTE 不需要 X Server 或 Xlib, 而是采用 FrameBuffer 作为底层图形接口, 因此与 Qt/X11 相比, QTE 可以节省很多内存。QTE 的应用程序可以直接写内核帧缓冲, 因此它在嵌入式 Linux 系统上的应用非常广泛。

Qt/X11 与 QTE 的应用程序结构如图 11-7 所示。

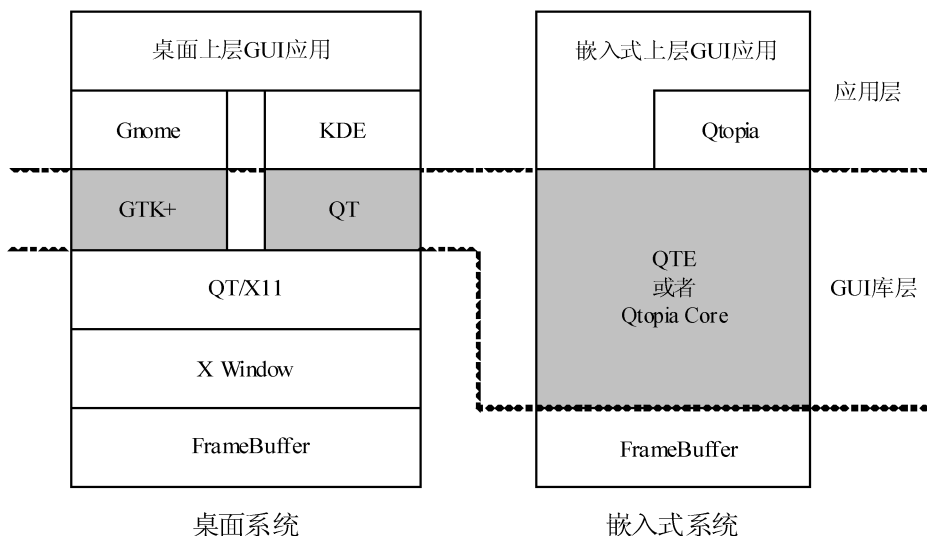


图 11-7 Qt/X11 与 QTE 的应用程序结构

Qttopia 是一个在 Qt 历史上著名的项目, 起源于 QPE (Qt Palmtop Environment, Qt 掌上电脑环境), 是构建于 Qt/Embedded 之上的一系列应用程序。从版本 4 开始, TrollTech 将 Qt/Embedded 并入了 Qttopia, 并推出了新的 Qttopia4。在 Qttopia4 中, 以前的 Qt/Embedded 被称为 Qttopia Core, 作为嵌入式版本的核心, 既可以与 Qttopia 配合, 也可以独立使用。以前的 Qttopia 也采用分层的结构: 底层为核心的应用框架和插件系统, 被称为 Qttopia Platform; 上层为应用程序, 按照不同的类型分成不同的包。

2. Qt 的编程机制

Qt 使用信号和槽用于对象间的通信。信号 (signal) 和槽 (slot) 机制是 Qt 的一个中心特征并且也许是 Qt 与其他工具包最不相同的部分。

标准的 C++ 对象模型为对象范例提供了十分有效的运行时支持。但是这种 C++ 对象模

型的静态性质在一定的领域是不够灵活的。图形用户界面编程就是一个同时需要运行时的效率和高水平的灵活性的领域。Qt 通过结合 C++ 的速度为这一领域提供了 Qt 对象模型的灵活性。

Qt 把下面这些特性添加到了 C++ 当中：

- 一种关于无缝对象通信机制，被称为信号和槽。
- 可查询和可设计的属性。
- 强大的事件和事件过滤器。
- 根据上下文进行国际化的字符串翻译。
- 完善的时间间隔驱动的计时器，使得在一个事件驱动的图形界面程序中很好地集成许多任务成为可能。
- 以一种自然的方式组织对象所有权的分层次和可查询的对象树。
- 被守护的指针，当参考对象被破坏时，可以自动地设置为无效。

使用元对象编译器（Meta Object Compiler）声明 Q_OBJECT 为对信号和槽的支持。除此之外，元对象编译器还实现对象属性。Q_PROPERTY 宏声明了一个对象属性，而 Q_ENUMS 声明在这个类中的属性系统中可用的枚举类型的一个列表。

在 Qt 中提供了各个基本类的树状结构（C++ 的继承），在应用程序中可以直接使用 Qt 基础类，或者派生成用户的类。

信号（signal）与槽（slot）机制是 Qt 最独特的特性，如图 11-8 所示。

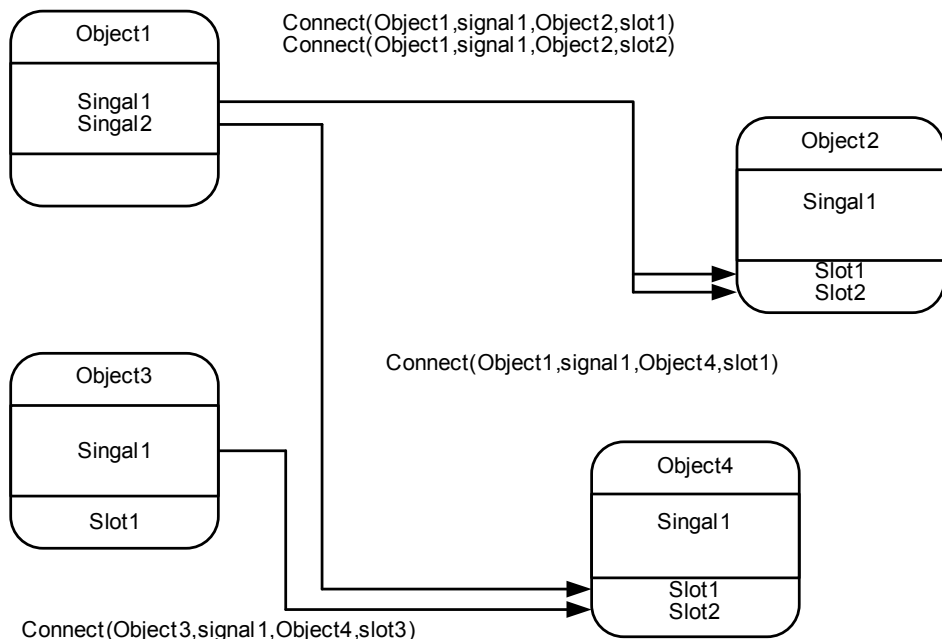


图 11-8 Qt 的信号与槽机制

使用信号与槽机制可以轻松完成系统两个部分之间的通信。在 GUI 系统中，可能同时存在很多窗口和控件，需要处理的某个控件发生事件后，另一个控件需要跟着做出响应。

对于此情况，使用信号与槽机制非常方便，只需要在一个控件中定义信号，在另一个控件中定义槽，然后将二者进行连接即可。在发生事件的时候，调用 `emit signal`，即可完成相应的调用。信号和槽可以是 Qt 基本类所提供的，也可以是用户自定义的。

Qt 编程使用 C++ 面向对象的所有机制，并且使用 Qt 自身一些基于 C++ 附加的功能、信号和槽以及相应的宏编译（moc）机制。QTE 的强大开发功能，为快速建立嵌入式 GUI 程序提供了很大的方便。

3. Qt 应用程序示例

本小节介绍一个基于 Qt 的简单应用程序，通过这个程序可以学习到 Qt 系统的基本程序开发。这个程序包含以下 3 个文件：`main.cpp`、`MainWindow.h` 和 `MainWindow.cpp`。

`main.cpp` 为程序入口源程序文件，如下所示：

```
#include <qapplication.h>
#include "MainWindow.h"
int main( int argc, char **argv )
{
    QApplication app( argc, argv );           // 建立 Qt 应用程序
    MainWindow win;                           // 建立窗口实例
    app.setMainWidget( &win );               // 设置窗口
    win.setGeometry(20,60,200,200);
    win.setCaption("Main Window");           // 设置窗口标题
    win.show();                               // 显示窗口
    return app.exec();                       // 进入 Qt 应用程序循环
}
```

`main.cpp` 文件提供程序的入口 `main()` 函数，此函数功能比较简单，但是包含了基于 Qt 的 GUI 程序的基本元素：一个 Qt 的应用程序和一个主窗口。`main()` 函数完成界面的建立，此处使用的界面控件类的实现就是 `MainWindow`。最后调用 `QApplication` 的 `exec()` 函数，进入应用程序的循环。

`MainWindow.h` 为类定义的头文件，如下所示：

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <qvbox.h>
class MainWindow : public QVBox               // 继承一个 QVBox
{
    Q_OBJECT
public:
    MainWindow( QWidget *parent=0, const char *name=0 );
public slots:
    void slotUpdate();
};
#endif
```

`MainWindow.h` 声明了类 `MainWindow`。作为一个 `Widget` 类的实现，`MainWindow` 需要继承 `QWidget`。`MainWindow` 类继承了 Qt 的类 `QVBox`。`QVBox` 本身是一个从 `QWidget` 继承的类。在类 `MainWindow` 中，声明了宏 `Q_OBJECT`，用以支持 Qt 的信号和槽特性，声明了公有的槽 `slotUpdate`（槽的本质是一个函数）。

`MainWindow.cpp` 为类的实现文件，如下所示：

```

#include "MainWindow.h"                // 包含定义类的头文件
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qapplication.h>
MainWindow::MainWindow( QWidget *parent, const char *name )
    : QVBox( parent, name )
{
    // 建立退出按钮
    QPushButton *quitButton = new QPushButton( "Quit", this, "quitButton" );
    quitButton->setFont( QFont( "Times", 18, QFont::Bold ) );
    // 连接退出按钮与退出
    connect( quitButton, SIGNAL(clicked()), qApp, SLOT(quit()) );
    // 建立滑动条
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 50 );
    slider->setValue( 0 );
    // 建立数字显示屏
    QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );
    // 连接滑动条和数字显示屏
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
    // 建立更改按钮
    QPushButton *changeButton = new QPushButton( "change", this, "resetButton" );
    changeButton->setFont( QFont( "Times", 18, QFont::Bold ) );
    // 连接更改按钮与自定义槽
    connect( changeButton, SIGNAL(clicked()), this, SLOT(slotUpdate()) );
}
void MainWindow::slotUpdate( void )
{
    setCaption( "Change Title" );      // 在槽的实现中设置标题
}

```

MainWindow.cpp 主要定义了自定义的类 MainWindow，它继承了 Qt 提供的基类 QVBox。注意，QApplication 中设置主窗口的函数，如 setMainWidget() 要求输入一个 QWidget 的指针，而在 main 函数中传递的参数是 MainWindow 的指针。

实际上，MainWindow 继承了 QVBox，QVBox 继承了 QHBox，QHBox 继承了 QFrame，QFrame 继承了 QWidget。因此，MainWindow 是 QWidget 间接的派生类，它的指针可以作为 QWidget 指针类型的参数。

在 MainWindow 中定义了以下对象：两个按钮（QPushButton）、一个滑动条（QSlider）、一个数字显示（QLCDNumber）。程序中使用了 Qt 的信号和槽机制完成对象间的通信。

在 MainWindow 的构造函数中，实现了如下功能。

（1）连接退出按钮和退出

退出按钮 quitButton 的单击信号连接到应用程序 qApp 的槽 quit（退出），从而单击按钮的时候，应用程序将退出。qApp 的 quit 槽是系统默认的，调用这个槽，QApplication 将退出。

（2）连接滑动条和 LCD 数字显示

滑动条 slider 的信号 valueChanged 连接到数字显示的 display，因此滑动条变化的时候，数字显示会相应变化。信号和槽都是系统默认的，但是带了一个附加的参数。

(3) 更改按钮 changeButton

更改按钮 `changeButton` 的单击信号连接到主窗口的槽 `slotUpdate` (退出), 因此单击时可以完成自定义的响应。`slotUpdate` 是自定义的槽, 本例提供的功能是更改标题。

在程序初始化的时候, 将初始化窗口上的几个元素: 窗口、退出按钮 (Quit)、滑动条、LCD 显示区域、变化按钮 (Change)。

此处函数实现的重点是表示连接的 `connect()` 函数, 它将按钮的单击事件 (`clicked()`) 和本程序定义的槽函数联系在一起, 因此单击发生之后, 槽函数就会被调用。

提示: 关于信号 (signal) 和槽 (slot) 的连接, 并非通过简单的 C++ 语法完成, 需要 `SIGNAL` 和 `SLOT` 两个宏在编译阶段进行处理。

Qt 的程序可以通过使用 `qmake` 等工具通过 `pro` 文件生成 `makefile`。上述程序的 `pro` 文件如下所示:

```
TEMPLATE = app
DESTDIR = ./bin
CONFIG += qtopia warn_on release
HEADERS += MainWindow.h
SOURCES += main.cpp MainWindow.cpp

INTERFACES =
TARGET = win
```

在 `pro` 文件中, `TEMPLATE` 表示当前目标类型, `app` 表示生成可执行程序, `lib` 表示生成库; `HEADERS` 为头文件 (*.h), `SOURCES` 为源文件 (*.cpp), `TARGET` 表示目标文件的名称。按照以上 `pro` 文件生成的 `makefile`, 最终目标文件为当前目录的 `bin` 目录下的可执行程序 `win`。

设置相应环境变量, 程序经过编译之后, 将生成 `main.o` 和 `MainWindow.o` 两个目标文件。由于 `MainWindow.h` 和 `MainWindow.cpp` 中含有槽 (slot), 因此还将生成它的元编译 (moc) 文件 `mocMainWindow.o`。连接时的过程将这 3 个目标文件组合成可执行应用程序, 并动态连接 Qt 库。

提示: 在一般的 C 或 C++ 程序中, 每个 C 语言或者 C++ 的源文件将生成一个目标文件 (*.o)。在 Qt 程序中, 除了源文件生成的目标文件外, 还将生成 `mocXXX.o` 目标文件 (moc 文件), 来支持具有信号和槽的源文件。

Qt 的初始化窗口、移动滑动条和更改标题的过程如图 11-9 所示。



图 11-9 Qt 的初始化窗口、移动滑动条和更改标题过程

一个 Qt 应用程序运行的流程如下所示。

(1) 建立 Qt 应用程序

定义一个 `QApplication` 的实例 `app`，这是所有 Qt 的 GUI 应用程序中都使用的类。每一个 `QApplication` 对应于一个 Linux 的进程。在它的构造函数中，将 `main` 函数的参数 `argc`（参数数目）和 `argv`（参数向量）传入。

(2) 建立和设置窗口

在 Qt 中每一个 `QApplication` 需要有一个主控件，因此定义一个 `MainWindow` 的实例 `win`，将其设置为 `app` 的主控件。`setMainWidget` 是 `QApplication` 的一个成员函数。这里的 `MainWindow` 是一个自定义的类，由于它间接继承了 `QWidget`，因此这个类实例的指针可以传入 `QWidget`。

(3) 进入 GUI 的循环中

窗口建立后，运行 `app`，进入它的循环中，直到其返回，`main` 函数随之退出。这个循环是 Qt 应用程序主要运行的时间，包括了接受用户的输入和界面上元素的更新、绘制。

当滑动滑动条的时候，将触发它的 `valueChanged(int)` 信号。由于这个信号被连接（`connect`）在 LCD 数字显示（类 `QLCDNumber` 的实例）的 `display(int)` 槽上，因此将传递 `int` 类型的值，由此更改 LCD 数字显示上的内容。

当单击 `change` 按钮的时候，将触发它的 `clicked()` 信号。由于这个信号（单击）连接在 `MainWindow` 类的自定义槽 `slotUpdate()` 上，这个动作将同时调用 `MainWindow` 的 `slotUpdate()` 函数。

当单击 `Quit` 的时候，将调用 Qt 应用程序的退出函数，退出循环 `app.exec()`。

11.3.2 MiniGUI 应用程序

MiniGUI 是由中国的北京飞漫软件技术有限公司创办的开源 Linux 图形用户界面支持系统。MiniGUI 的网址为 <http://www.minigui.com/>。

1. MiniGUI 应用程序概述

MiniGUI 使用 C 语言实现，接口也是 C 语言的，其应用程序的开发使用类似于 Win32 接口。MiniGUI 的程序结构如图 11-10 所示。

在 MiniGUI 编程中，实际上实现了使用 C 语言进行面向对象的编程。C 语言并不是面向对象的语言。实现面向对象的编程经常使用句柄（`handle`）、回调函数（`callback`），以及在图形用户系统常用的消息机制，用于程序与用户的交互。

● 句柄（`handle`）

句柄的本质是一个指针，在 GUI 系统中表示一个 GUI 的 item，如 MiniGUI 中的主窗口和对话框。MiniGUI 使用句柄标识对象的标识符。利用句柄，MiniGUI 将系统变量从应用项目中分离了出来。程序员使用句柄访问对象，因而就没有利用指针时可能发生的由于非法访问而导致的数据不一致问题。

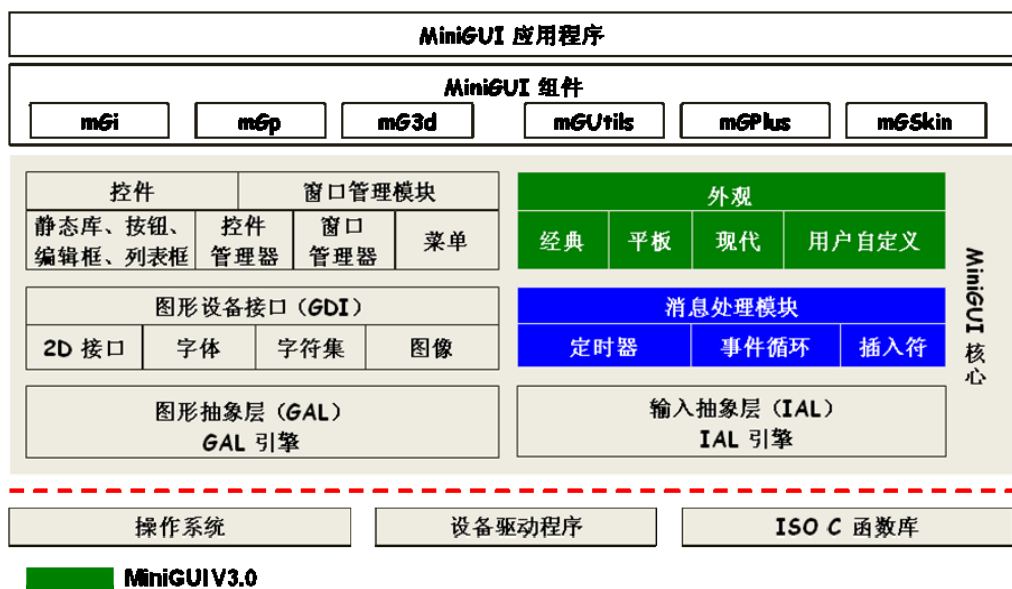


图 11-10 MiniGUI 的程序结构

● 回调函数 (callback)

回调函数的本质是一个函数，在程序中将其作为参数传递。往往是上层软件传给下层的函数指针，在某种事件发生时调用。在 GUI 系统中，回调函数用于定义各个窗口控件的行为。在 GUI 的开发中，往往窗口控件的外观是使用 GUI 系统定义的（用户有可能可以定制），而 GUI 的行为要由用户实现。回调函数代表的就是这种行为。

● 消息机制

MiniGUI 应用程序通过接收消息来和外界交互。消息由系统或应用程序产生，系统对输入事件产生消息，系统对应用程序的响应也会产生消息，应用程序可以通过产生消息来完成某个任务，或者与其他应用程序的窗口进行通信。总而言之，MiniGUI 是消息驱动的系统，一切运作都围绕着消息进行。

窗口是屏幕上的一个矩形区域。在传统的窗口系统模型中，应用程序的可视部分由一个或多个窗口构成。每一个窗口代表屏幕上的一块绘制区域，窗口系统控制该绘制区域到实际屏幕的映射，也就是控制窗口的位置、大小和可见区域。每个窗口被分配一个屏幕绘制区域来显示本窗口的部分或全部，也许根本没有分配到屏幕区域（该窗口完全被其他的重叠窗口所覆盖和隐藏）。

在一个复杂的 GUI 系统中，处理窗口之间的互相剪切是其首要解决的问题。这是由于多窗口系统首先要确保一个窗口中的绘制输出不会影响到另外一个窗口。因此，GUI 系统一般要利用 Z 序来管理窗口之间的互相剪切关系。根据窗口在 Z 序中所处的位置，GUI 系统要计算每个窗口受剪切的区域，即剪切域。通常，窗口的剪切域定义为互不相交的矩形集合。GUI 系统的底层图形引擎在进行输出时，要根据当前输出的剪切域进行输出的剪切操作，从而保证窗口的绘制输出不会互相影响。因为任何一个窗口的创建、销毁、隐藏、显示均有可能影响其他窗口的剪切域，所以首先要有一个高效的剪切域维护算法。MiniGUI

利用窗口 Z 序和窗口剪切算法实现窗口的绘制工作。

GUI 系统当中窗口区域的划分如图 11-11 所示。

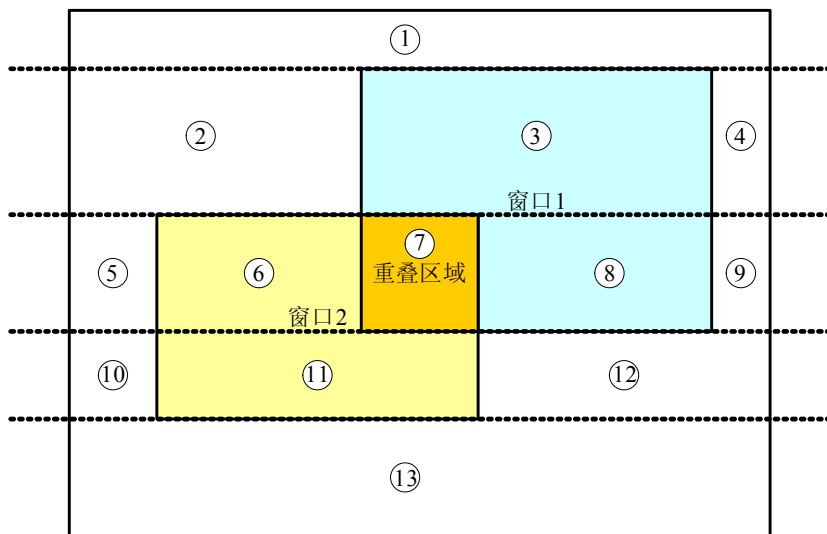


图 11-11 窗口区域的划分

例如：一个显示区域由若干个窗口组成，由于窗口的重叠情况，可能将整个屏幕划分为若干个区域。这样在屏幕的绘制中，就可以利用窗口的重叠特性，只绘制屏幕中的可见部分。当窗口 1 在窗口 2 的上方时，矩形区域 3、8、7 需要绘制窗口 1 的内容，矩形区域 6 和 11 需要绘制窗口 2 的内容。

一个 MiniGui 应用程序窗口一般包括如下部分：

- 一个可视的边界。
- 一个窗口 ID，客户程序使用该 ID 来操作窗口，MiniGUI 中称为“窗口句柄”。
- 一些其他特性：高、宽、背景色等。
- 可能有菜单和滚动条等附加窗口元素。

MiniGUI 的窗口分成下列 3 种基本类型。

- 主窗口：每一个 MiniGUI 应用程序一般都要创建一个主窗口，作为应用程序的主界面或开始界面。
- 对话框：主窗口通常包括一些子窗口，这些子窗口通常是控件窗口，也可以是自定义窗口类。应用程序还会创建其他类型的窗口，例如对话框和消息框。对话框本质上就是主窗口，应用程序一般通过对话框提示用户进行输入操作。消息框是用于给用户一些提示或警告的主窗口，属于内建的对话框类型。
- 控件：控件是窗口中的内容，例如文本框、按钮、列表框等。

在 MiniGUI 中，主窗口通常是一种比较特殊的窗口。因为主窗口代码的可重用性一般很低，如果按照通常的方式为每个主窗口注册一个窗口类的话，则会形成额外不必要的存储空间，所以 MiniGUI 没有在主窗口提供窗口类支持。主窗口中的所有子窗口，即控件，

均支持窗口类（也就是说控件类）的概念。MiniGUI 提供了常用的预定义控件类，包括按钮（button）、静态框（static）、列表框（listbox）、进度条（progressbar）、滑块（trackbar）、编辑框（edit）等。

在复杂的 MiniGUI 程序中，除了主窗口外，还需要使用对话框和各种控件。对话框将是一个在主窗口内的窗口，不对应独立的线程。控件是一个窗口或者对话框中的元素。MiniGUI 控件的数据结构如下：

```
typedef struct
{
    char*      class_name;           // 控件类
    DWORD      dwStyle;              // 控件风格
    int        x, y, w, h;           // 控件在对话框中的位置
    int        id;                   // 控件标示
    const char* caption;             // 控件标题
    DWORD      dwAddData;            // 附加数据
    DWORD      dwExStyle;            // 控件扩展风格
} CTRLDATA;
```

在 MiniGUI 中，程序也可以定制自己的控件类，注册后再创建对应的实例。采用控件类和控件实例的结构，可以提高代码的可重用性，还可以方便地对已有控件类进行扩展。例如，在需要建立一个只允许输入数字的编辑框时，就可以通过重载已有编辑框控件类而实现，而不需要重新编写一个新的控件类。在 MiniGUI 中，这种技术被称为子类化或窗口派生。

在 MiniGUI 中，使用了消息驱动作为应用程序的创建构架。在消息驱动的应用程序中，计算机外设发生的事件，例如键盘键的敲击、鼠标键的按击等，都由支持系统收集，将以事先的约定格式翻译成特定的消息。应用程序一般包含自己的消息队列，系统将消息发送到应用程序的消息队列中。

应用程序可以建立一个循环，在这个循环中读取消息并处理消息，一直处理到特定的消息传来为止。这样的循环就称为消息循环。一般来说，消息由代表消息的一个整数和消息的附加参数组成。例如，鼠标左键的按下消息，可以使用数字 133 来表示，其附加参数可能包含按下时的鼠标所在位置信息。

MiniGUI 的消息定义如下：

```
typedef struct
{
    HWND      hwnd;
    int        message;
    WPARAM    wParam;
    LPARAM    lParam;
    // .....省略部分内容
} MSG;
```

应用程序一般要提供一个处理消息的标准函数。在消息循环中，系统可以调用此函数，应用程序在此函数中处理相应的消息。在消息定义 MSG 中，包含了消息传递的窗口句柄（hwnd）、消息类型（message）、消息的两个参数 wParam 和 lParam。

消息驱动的应用程序的简单架构如图 11-12 所示。

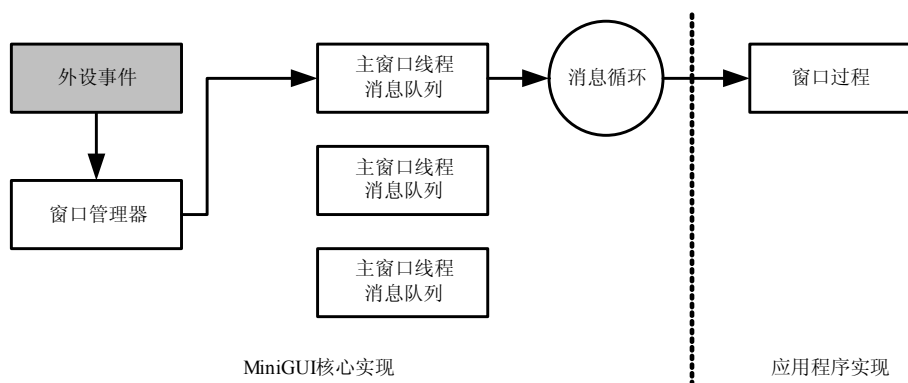


图 11-12 MiniGUI 的消息处理

在 MiniGUI 的程序架构中，每个主窗口线程对应一个消息队列，窗口管理器和消息循环由 MiniGUI 核心系统实现。外设事件被接收后，最终需要调用应用程序实现的窗口过程，窗口过程的本质就是一个回调函数，是应用程序为外设事件定制的行为。

在 MiniGUI 中，消息分为如下几种类型：

- 系统消息，为系统内部管理使用。
- 鼠标消息，鼠标的单击、移动等产生的消息。
- 键盘消息，键盘的按键消息。
- 窗口消息，窗口管理消息。
- 菜单消息，菜单管理消息。
- 命令消息等。

在 MiniGUI 的消息中，命令消息比较特殊。一般来说，其他几种消息都是由系统自动触发的，例如鼠标单击时产生鼠标消息，窗口建立时产生窗口建立消息。但是，命令一般是由应用程序自己产生的，即由应用程序的某个函数运行过程中发送一个消息，在另外的程序的其他地方接收该消息，并根据需要进行处理。

2. MiniGUI 应用程序示例

本小节是 MiniGUI 线程版的实例程序，它利用 MiniGUI 显示一个窗口，并在窗口中处理了一些事件。这只是一个很简单的 MiniGUI，但是通过其结构可以了解 MiniGUI 应用程序的基本框架和一些运行机制。

```

#include <stdio.h>
#include <string.h>
#include <time.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/mywindows.h>

static int HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{

```

```

        switch (message) {
            case MSG_CREATE:
// .....省略部分内容: 创建事件处理过程
                break;
            case MSG_LBUTTONDOWN:
// .....省略部分内容: 左按钮按下事件处理过程
                break;
            case MSG_LBUTTONUP:
// .....省略部分内容: 左按钮抬起事件处理过程
                break;
            case MSG_KEYUP:
// .....省略部分内容: 按键抬起事件处理过程
                return 0;
            case MSG_CLOSE:
                DestroyMainWindow (hWnd);
                PostQuitMessage (hWnd);
                return 0;
        }
        return DefaultMainWinProc(hWnd, message, wParam, lParam);
    }
}
void InitCreateInfo1(PMAINWINCREATE pCreateInfo)
{
    pCreateInfo->dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
    pCreateInfo->dwExStyle = WS_EX_NONE;
    pCreateInfo->spCaption = "Hello, world!";
    pCreateInfo->hMenu = 0;
    pCreateInfo->hCursor = GetSystemCursor(0);
    pCreateInfo->hIcon = 0;
    pCreateInfo->MainWindowProc = HelloWinProc;
    pCreateInfo->lx = 0;
    pCreateInfo->ty = 0;
    pCreateInfo->rx = 320;
    pCreateInfo->by = 240;
    pCreateInfo->iBkColor = PIXEL_red;
    pCreateInfo->dwAddData = 0;
    pCreateInfo->hHosting = HWND_DESKTOP;
}
void* callMainWindow(void* data)
{
    MSG Msg;
    HWND hMainWnd;
    MAINWINCREATE CreateInfo;
#ifdef _LITE_VERSION
    SetDesktopRect(0, 0, 1024, 768);
#endif
    mouse_calibrate ();
    InitCreateInfo1(&CreateInfo);
    hMainWnd = CreateMainWindow (&CreateInfo);
    printf ("The main window created.\n");
    if (hMainWnd == HWND_INVALID)
        return NULL;
    ShowWindow(hMainWnd, SW_SHOWNORMAL);
    while (GetMessage(&Msg, hMainWnd)) {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    MainWindowThreadCleanup (hMainWnd);
    return NULL;
}

```

// 初始化窗口信息

// 建立主窗口

// 显示主窗口

// 进入消息循环

// 删除主窗口

```
int MiniGUIMain(int args, const char* arg[])
{
    pthread_t thread1;
    CreateThreadForMainWindow(&thread1, NULL, callMainWindow, 0);
    return 0;
}
```

MiniGUIMain()是主函数，也就是基于 MiniGui 的程序中必须使用的入口点，相当于 main+MiniGUI 初始化。在该函数中，使用线程建立一个主窗口。

由于本应用是 MiniGUI 的线程版，需要使用 POSIX 的线程模型建立主窗口，因此程序中定义了一个 POSIX 线程 pthread。

随后，程序调用 CreateThreadForMainWindow()函数操作建立一个 MiniGUI 的窗口，线程作为参数传入。本操作将一个主窗口的建立函数赋给一个线程，这时系统将会启动一个线程来运行 callMainWindow 函数。

正如上面所述，在 MiniGUI 程序中，每一个主窗口对应一个线程，在线程中实现窗口的消息循环及相应的事件处理。如果需要建立多个主窗口，需要通过建立多个线程来实现。例如：

```
pthread_t thread1, thread2;
CreateThreadForMainWindow(&thread1, NULL, callMainWindow1, 0);
CreateThreadForMainWindow(&thread2, NULL, callMainWindow2, 0);
```

这时将会有两个线程来分别建立两个窗口。注意：赋给两个线程的窗口建立函数可以是同一个函数，这时将建立两个相同的窗口。

提示：在 MiniGUI 线程版中，多个主窗口线程运行在一个进程之内。这样，它们之间可以利用共享内存、函数调用等机制进行数据交换。这样的好处是不同主窗口之间的通信非常方便；缺点是不同线程之间缺乏保护，一个线程的崩溃将造成其他线程的崩溃。

callMainWindow()函数为建立主窗口的函数，将在线程中运行。

主窗口的建立有以下几个步骤。

(1) 建立主窗口信息

MAINWINCREATE 是 MiniGUI 专门为主窗口建立提供的数据结构。它包括了两个方面的重要信息：窗口的外形和窗口的行为。窗口的外形由 dwStyle、spCaption、iBkColor 等结构体成员指定，窗口的行为指定为 HelloWinProc（本质上是一个回调函数）。

在后面的程序中将根据 CreateInfo 建立窗口，然后显示窗口，并进入窗口的消息循环。

(2) 初始化主窗口信息

主窗口信息的初始化需要完成 MAINWINCREATE 结构的构建。建立窗口信息的函数包含了初始化窗口的外观和定制行为，其中定制过程将窗口过程赋值给窗口信息中的函数指针（MainWindowProc）。InitCreateInfo1()是本例中的一个私有函数，调用目的是将主窗口信息 CreateInfo（MAINWINCREATE 类型）的内容置成所需要的数据。

由于将 HelloWinProc 赋值给 pCreateInfo 的成员 MainWindowProc，因此这个函数将在窗口接收到消息的时候由 MiniGUI 系统调用，调用后实现应用程序需要的功能。

(3) 建立主窗口

主窗口信息初始化完成后, 就可以利用它建立主窗口。CreateMainWindow()函数负责建立 MiniGUI 主窗口, 它的输入是一个主窗口的信息, 返回值是一个主窗口的句柄。

实际上, 在 CreateMainWindow 的内部进行了内存分配操作 (malloc), 根据主窗口建立信息为主窗口分配内存 (在堆上), 并返回实际主窗口的指针 (即句柄)。

(4) 显示主窗口

主窗口建立完成后, 调用 ShowWindow()函数即可以利用主窗口的句柄实现对主窗口的各种操作。显示主窗口的第二个参数是显示的模式。

(5) 进入主窗口消息循环

在程序的运行过程中, 主窗口的建立过程是很短暂的, 主窗口的大部分时间都会运行在消息循环以内。在这个循环中, 主窗口将接收用户的输入, 并根据主窗口的处理过程 (即回调函数) 做出相关的响应。进入主窗口消息循环后要在 while()循环中完成, 循环中首先需要调用 GetMessage()函数获取程序中的消息。

GetMessage()函数用于获取当前主窗口的消息。当需要退出时将退出 while 循环, 主窗口的消息循环结束。当消息需要处理的时候, 进行转换 (TranslateMessage) 和派发 (DispatchMessage)。由注册在窗口内部的回调函数完成消息的处理。

HelloWinProc()函数为主窗口事件处理函数, 本质为窗口的回调函数, 用于处理窗口的各种行为, 如按键、创建窗口、关闭窗口等。本函数含有 4 个参数, 第一个参数是当前主窗口的指针, 后 3 个参数来自派发的消息。

实际上, 窗口回调函数的后 3 个参数, 即数据结构 MSG 中的主要成员。窗口过程根据消息 message 和消息的参数 wParam、lParam 分类处理事件。

一般来说, 参数 message 代表了消息的类型 (如: 按键按下、按键释放、窗口建立等), 在程序中使用 switch...case 来区分消息。

参数 wParam 和 lParam 提供消息的附加信息。

对于一些系统消息, 不是由应用程序来处理的, 而是由 MiniGUI 系统来处理的, 这时需要增加默认的消息处理函数。

DefaultMainWinProc()函数用于 MiniGUI 的默认消息处理。调用这个函数时, 将消息的几个参数进行了传递, 这时 MiniGUI 可以处理默认的消息。

以上程序可以实现一个 MiniGUI 基本窗口的绘制, 并响应用户的输入。在用户输入后, 由 MiniGUI 调用窗口的过程函数, 过程函数中根据不同消息的类型, 完成不同的功能, 由此实现不同的行为。

11.3.3 MicroWindows (Nano-X Window)

MicroWindows 是一个基于典型客户/服务器体系结构的 GUI 系统。底层是驱动程序; 中间层是硬件的抽象接口和窗口管理; 最高层提供兼容 X Window 和 Win32 子集的 API。在 2005 年 MicroWindows 项目被改为 Nano-X Window 项目。

MicroWindows (Nano-X Window) 的网址为 <http://www.microwindows.org/>。

MicroWindows (Nano-X Window) 的程序结构如图 11-13 所示。

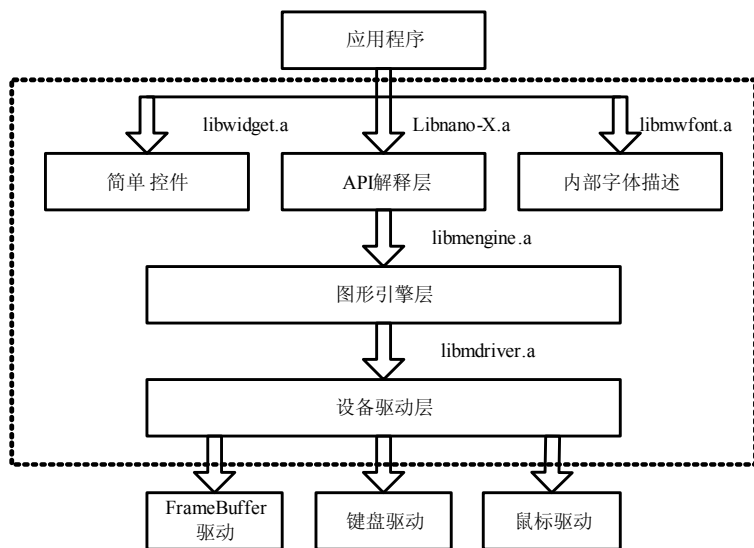


图 11-13 MicroWindows (Nano-X Window) 的程序结构

在 MicroWindows (Nano-X Window) 的程序结构中，底层是面向图形显示和键盘、鼠标或触摸屏的驱动程序；中间层提供底层硬件的抽象接口，并进行窗口管理；最高层分别提供兼容于 X Window 和 ECMA APIW (Win32 子集) 的 API。其中使用 Nano-X 接口的 AP 与 X 接口兼容，但是该接口没有提供窗口管理，如窗口移动和窗口剪切等高级功能。系统需要先启动 Nano-X 中 Server 程序的 nanox-server 和窗口管理程序 nanowm。用户程序连接 Nano-X 的 Server 获得自身的窗口绘制操作。

11.4 数据库

11.4.1 关于嵌入式数据库

嵌入式数据库是指运行在本机上、不用启动服务端的轻型数据库，它与应用程序紧密集成，被应用程序所启动，并伴随应用程序的退出而终止。

数据库和嵌入式数据库的结构分别如图 11-14 所示。

嵌入式数据库与非嵌入式数据库的差别在于运行模式方面。

嵌入式数据库市场主要有 SQLite、Berkeley DB、Firebird 嵌入服务器版，它们都是开源的软件，具体如下所示。

- SQLite，网址为：<http://www.sqlite.org>。
- Berkeley DB，网址为：<http://www.oracle.com/database/berkeley-db/index.html>。
- Firebird 嵌入服务器版，网址为：<http://www.firebirdsql.org>。

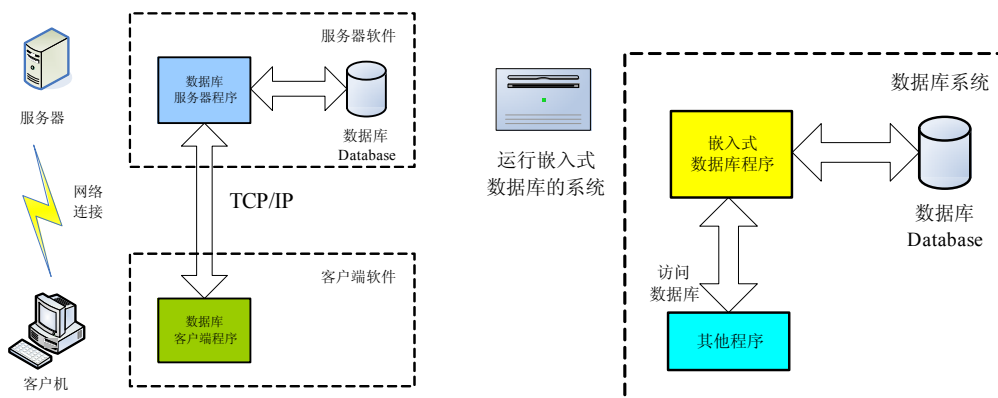


图 11-14 数据库和嵌入式数据库的结构

在容量方面，SQLite 可以支持 2TB，Berkeley DB 可以支持 256TB，Firebird 嵌入服务器版可以支持 64TB；在 SQL 语言方面，SQLite 支持大部分 SQL-92，Berkeley DB 不支持（仅支持 API），Firebird 嵌入服务器版支持完全 SQL-92 与大部分 SQL-99。

11.4.2 SQLite

SQLite 是一个基本实现了 SQL 语法的非常轻量级的关系型数据库，尤其适用于嵌入式系统。SQLite 没有服务器，解析 SQL 查询和访问数据库的各种函数是以静态库或共享库的形式与客户端连接在一起的，而数据库就是一个常规文件。

SQLite 在 2000 年由 D. Richard Hipp 开始开发，2001 年发布 2.0 版本，2004 年发布 3.0 版本，采用了不同的数据文件格式以及编程接口。

1. SQLite 的结构

SQLite 的结构如图 11-15 所示。

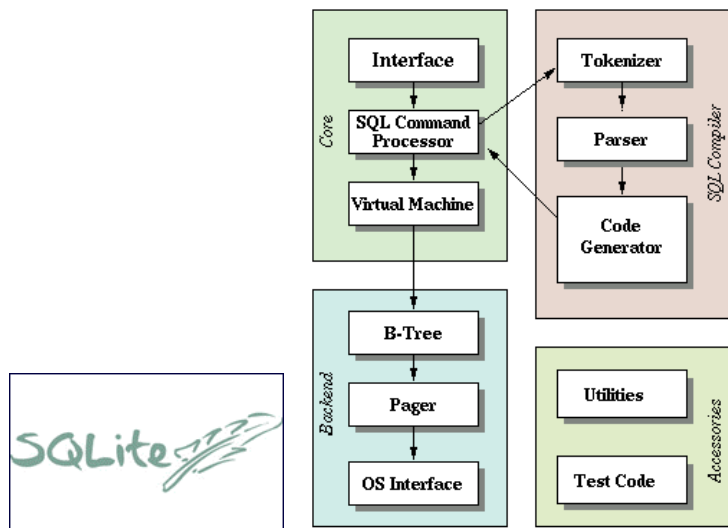


图 11-15 SQLite 的结构

SQLite 是开源的软件，具有如下的优势：

- 内存占用量小。
- 比 MySQL 快 2 倍，比 PostgreSQL 快 20 倍。
- ACID 兼容（原子性、一致性、独立性、可持久性），支持视图、子查询、触发器。
- 单个库文件中包含数据库引擎与接口，且其运行不依赖其他库。
- 可以将数据放进单个文件。
- 可以为 C/C++、Perl、PHP 等应用提供接口。
- 允许为 SQL 命令集动态添加自定义函数，而无须重编 SQLite 库。

相比同类的数据库软件，SQLite 也有一些缺点。

- 事务处理并发性：SQLite 通过数据库级上的独占性和共享锁来实现独立事务处理，这意味着多个进程或线程在同一时间可以从数据库读取数据，但是只能有一个可以同时写入，在写入之前，必须获得独占锁，其他的读操作不允许发生。
- 性能：在执行创建索引（CREATE INDEX）和删除表（DELTE TABLE）时明显比其他数据库慢。
- 用户管理/安全：数据库的访问是基于操作系统对文件的控制来控制的，不能通过用户来区分数据库中的不同数据库。例如：将数据库文件去掉写权限之后，然后再向其中插入或删除数据条目，将提示写失败。但是不能通过数据库本身来对权限进行设置。

SQLite 运行时的特点：基本操作就是对文件系统中的数据库文件的操作。SQLite 提供的 C 语言 API 也是基于 SQL 查询语言的，接口较少。

SQLite 的各个模块如下所示。

● 接口

SQLite 类库大部分的公共接口程序是由 main.c、legacy.c 和 vdbeapi.c 源文件中的功能执行的。但有些程序是分散在其他文件夹中的，因为在其他文件夹里它们可以访问有文件作用域的数据结构。sqlite3_get_table()在 table.c 中。sqlite3_mprintf()在 printf.c 中。sqlite3_complete()在 tokenize.c 中。Tcl 接口程序在 tclsqlite.c 中。

● Tokenizer

当执行一个包含 SQL 语句的字符串时，接口程序要把这个字符串传递给 Tokenizer。Tokenizer 的任务是把原有字符串分成一个个标识符，并把这些标识符传递给剖析器。Tokenizer 是在 C 文件 tokenize.c 中用手工编译的。

在这个设计中需要注意的一点是，Tokenizer 调用 Parser。熟悉 YACC 和 BISON 的人也许会习惯于用 Parser 调用 Tokenizer。SQLite 的作者已经尝试了这两种方法，并发现用 Tokenizer 调用 Parser 会使程序运行得更顺利。YACC 使程序更滞后一些。

● Parser

Parser 基于文件场景赋予 tokens。SQLite 的 Parser 是由 Lemon LALRparser generator 产生的。Lemon 和 YACC/BISON 做同样的工作，但是它使用不同的输入语句，这个输入语句是不易出错的。Lemon 的 Parser 是可重入的并且是线程安全的。Lemon 定

义了无终端解除程序的概念，所以当遇到语法错误的时候，它不会泄露内存。驱动 Lemon 的源文件在 `parse.y`。Lemon 的源代码是在 SQLite 分布区的 `tool` 子目录下的。Lemon 的文档分布在 `doc` 子目录下。

● 代码发生器

在剖析器收集完符号并把其转换成完全的 SQL 语句时，它调用代码产生器来产生虚拟的机器代码，这些机器代码将按照 SQL 语句的要求来工作。在代码产生器中有许多文件：`attach.c`、`auth.c`、`build.c`、`delete.c`、`expr.c`、`insert.c`、`pragma.c`、`select.c`、`trigger.c`、`update.c`、`vacuum.c` 和 `where.c`。这些文件中执行最具有重要意义的事情。`expr.c` 处理表达式代码的生成。`where.c` 处理 SELECT、UPDATE 和 DELETE 语句中 WHERE 子句代码的生成。文件 `attach.c`、`delete.c`、`insert.c`、`select.c`、`trigger.c`、`update.c` 和 `vacuum.c` 处理 SQL 语句中具有同名语句的代码生成。所有 SQL 的其他语句的代码是由 `build.c` 生成的。文件 `auth.c` 执行 `sqlite3_set_authorizer()` 的功能。

● 虚拟机

由代码生成器产生的程序由虚拟机来运行。总而言之，虚拟机主要用来执行一个为操作数据库而设计的抽象计算引擎。虚拟机有一个用来存储中间数据的存储栈。每个指令包含一个操作码和三个额外的操作数。

虚拟机本身被包含在一个单独的文件 `vdbe.c` 中。虚拟机也有它自己的头文件：`vdbe.h` 在虚拟机和其他的 SQLite 类库之间定义了一个接口程序，`vdbeInt.h` 定义了虚拟机的结构。文件 `vdbeaux.c` 包含了虚拟机所使用的实用程序和一些被其他类库用来建立虚拟机程序的接口程序模块。文件 `vdbeapi.c` 包含虚拟机的外部接口，比如 `sqlite3_bind_XXX` 类的函数。单独的值（字符串、整数、浮点数值、BLOBS）被存储在一个叫“Mem”的内部程序中，“Mem”在 `vdbeMem.c` 中。

SQLite 使用 C 语言程序执行 SQL 函数。即使内置的 SQL 函数也是用这种方法来执行的。大部分 SQL 内置函数在 `func.c` 文件中；日期和时间转换函数在 `date.c` 中。

● B-树 (B-tree)

SQLite 数据库在磁盘里维护，使用源文件 `btree.c` 中的 B-树执行。数据库中的每个表格和目录使用一个单独的 B-tree。所有的 B-tree 被存储在同样的磁盘文件里。文件格式的细节被记录在 `btree.c` 文件中。

● 页面高速缓存

B-tree 模块要求信息来源于磁盘上固定规模的程序块。默认程序块的大小是 1024 个字节，但是可以在 512~65 536 个字节间变化。页面高速缓存负责读、写和高速缓存这些程序块。页面高速缓存还提供重新运算和提交抽象命令，它还管理关闭数据库文件夹。B-tree 要使用页面高速缓存器中的特别页，当它想修改页或重新运行改变的时候，它会通报页面高速缓存。为了保证所有的需求被快速、安全和有效地处理，页面高速缓存处理所有的微小细节。

运行页面高速缓存的代码在专门的 C 源文件 `pager.c` 中。页面高速缓存的子系统的接口程序在头文件 `pager.h` 中定义。

● OS 接口

为了在 POSIX 和 Win32 之间提供一些可移植性，SQLite 操作系统的接口程序使用了一个提取层。OS 提取层的接口程序被定义在 `os.h` 中。每个支持的操作系统有它自己的实现文件：UNIX 使用 `os_unix.c`，Windows 使用 `os_win.c`。每个具体的操作系统具有它自己的头文件：`os_unix.h`、`os_win.h`，etc.

● 工具

内存分配和字符串比较程序位于 `util.c`。剖析器使用的表格符号被 `hash.c` 中的无用信息表格维护。源文件 `utf.c` 包含 UNICODE 转换子程序。SQLite 有它自己的执行文件 `printf()`（有一些扩展）。在 `printf.c` 中，还有它自己的随机数量产生器在 `random.c`。

● 测试代码

如果使用回归测试脚本，超过一半的 SQLite 代码数据库的代码将被测试。在主要代码文件中有许多 `assert()` 语句。另外，源文件 `test1.c` 通过 `test5.c` 和 `md5.c` 执行测试。`os_test.c` 程序用来模拟断电，来验证页面调度程序中的系统性事故恢复机制。

SQLite 分为库和基于库的可执行程序两个部分，`sqlite` 库提供了上层调用的 API，而自带的 `sqlite3` 是基于 `sqlite` 库的可执行程序，上层也可以基于 `sqlite` 库开发可执行程序。

SQLite 的编程结构如图 11-16 所示。

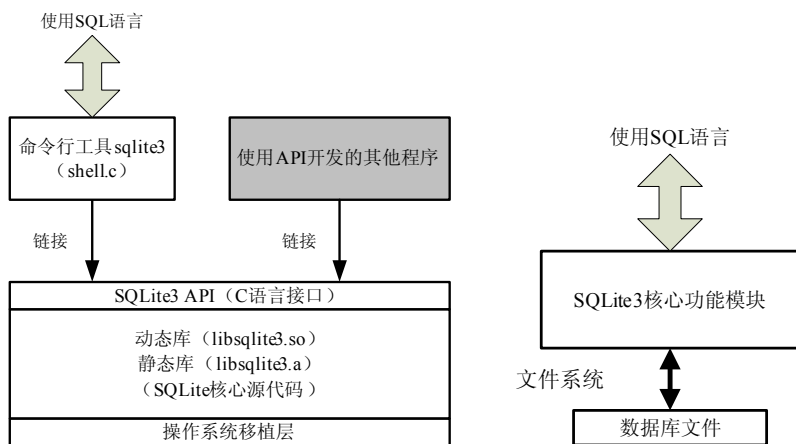


图 11-16 SQLite 的编程结构

SQLite 自带程序 `sqlite3` 是一个命令行工具，其中可以使用的命令包括两种。

● 系统命令：以“.”开头。

● SQL 命令：以“;”结束。

SQLite 的主要系统命令如下所示。

● `.databases`：列出数据库文件名。

● `.dump ?TABLE? ...`：dump 数据库到 SQL 文本格式。

● `.echo ON|OFF`：开关命令行显示。

● `.exit`：退出程序。

- `.explain ON|OFF`: 开关输出模式的 suitable for EXPLAIN。
- `.header(s) ON|OFF`: 开关显示头。
- `.tables ?PATTERN?`: 列出?PATTERN?匹配的表名。
- `.import FILE TABLE`: 将文件中的数据导入。
- `.dump ?TABLE?`: 生成形成数据库表的 SQL 脚本。
- `.output FILENAME`: 将输出导入到指定的文件中。
- `.output stdout`: 将输出打印到屏幕。
- `.help`: 帮助。
- `.mode MODE ?TABLE?`: 设置数据输出模式 (csv, html, tcl……)。
- `.nullvalue STRING`: 用指定的串代替输出的 NULL 串。
- `.read FILENAME`: 执行指定文件中的 SQL 语句。
- `.schema ?TABLE?`: 打印创建数据库表的 SQL 语句。
- `.separator STRING`: 用指定的字符串代替字段分隔符。
- `.show`: 打印所有 SQLite 环境变量的设置。
- `.quit`: 退出命令行接口。

SQL 的主要命令并不区分大小写，分为 DML 和 DDL。

其中，select、delete、update、insert 这 4 种 SQL 语句被称为 DML (Data Manipulation Language)，用来操作表格中的数据。

create、alter、drop 这几种 SQL 语句被称为 DDL (Data Definition Language)，用来创建、删除和修改对象的定义（这里的对象可以是表格、数据库、索引等）。

SQLite 支持的数据类型如下所示。

- NULL: 无类型。
- INTEGER: 整型。
- REAL: 实型（浮点数）。
- TEXT: 文本。
- BLOB: 二进制长对象。

2. SQLite 示例

- 建立数据库和表

使用命令行工具打开数据库文件的方式如下所示：

```
$ sqlite3 test.db
```

表示从当前目录下打开 test.db 数据库文件。如果该文件不存在，就将被新建。命令执行之后将会进入 sqlite3 的命令行界面（以 sqlite>为提示符）。

进入提示符 sqlite>后，可以调用 SQL 命令建立一个表：

```
sqlite> create table employee(id integer primary key, name text, gender text, age integer);
```

此命令将建立一个名为 employee 的表格，以 id 为主键。

建立表之后，可以通过内部命令查看数据库的情况：

```
sqlite> .tables
employee
sqlite> .schema employee
CREATE TABLE employee(id integer primary key, name text, gender text, age integer);
```

使用内部命令导入数据：

```
sqlite> .import data.txt employee
```

data.txt 的内容如下所示：

```
1|dq|male|24
2|jz|female|27
3|pp|male|26
4|cj|male|28
5|zc|male|25
```

由此，将把 data.txt 中的表格内容导入，形成数据库中的内容。这相当于在 SQLite 的程序中，建立一个表格型的数据结构（见图 11-17）。

id	name	gender	age
1	dq	male	24
2	jz	female	27
3	pp	male	26
4	cj	male	28
5	zc	male	25

图 11-17 表格型数据结构

执行内部的 dump 命令，查看数据库如下所示：

```
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE employee(id integer primary key, name text, gender text, age integer);
INSERT INTO "employee" VALUES(1,'dq','male',24);
INSERT INTO "employee" VALUES(2,'jz','female',27);
INSERT INTO "employee" VALUES(3,'pp','male',26);
INSERT INTO "employee" VALUES(4,'cj','male',28);
INSERT INTO "employee" VALUES(5,'zc','male',25);
COMMIT;
```

上面列出的主要内容实际上是前面的操作流程，也就是导入（.import）命令实际执行的命令序列。

● 查询数据

使用 SQL 命令 select，查询表中内容的方法如下所示：

```
sqlite> select * from employee;
1|dq|male|24
2|jz|female|27
3|pp|male|26
4|cj|male|28
5|zc|male|25
```

使用 SQL 命令 select，查看并指定顺序的方法如下所示：

```
sqlite>select * from employee where id>2 order by age desc;
4|cj|male|28
3|pp|male|26
5|zc|male|25
```

使用 SQL 命令 **select**，查看指定列的方法如下所示：

```
sqlite> select id>2, name, gender from employee;
0|dq|male
0|jz|female
1|pp|male
1|cj|male
1|zc|male
```

● 插入数据

使用 SQL 命令 **insert**，在表中插入数据的方法如下所示：

```
sqlite> insert into employee values (7,"hello","male",30);
```

使用 SQL 命令，查询结果如下所示：

```
sqlite> select * from employee;
1|dq|male|24
2|jz|female|27
3|pp|male|26
4|cj|male|28
5|zc|male|25
7|hello|male|30
```

使用另一种方式插入，如下所示：

```
sqlite> insert into employee (id, name, age) values (8, 'zh', 18);
```

使用 SQL 命令，查询结果如下所示：

```
sqlite> select * from employee;
1|dq|male|24
2|jz|female|27
3|pp|male|26
4|cj|male|28
5|zc|male|25
7|hello|male|30
8|zh||18
```

● 修改数据

使用 SQL 命令 **update**，在表中修改数据的方法如下所示：

```
sqlite> update employee set age=20 where id=2;
```

使用 SQL 命令，查询结果如下所示：

```
sqlite> select * from employee;
1|dq|male|24
2|jz|female|20
3|pp|male|26
4|cj|male|28
5|zc|male|25
7|hello|male|30
```

● 删除数据

使用 SQL 命令 `delete`，从表中删除数据的方法如下所示：

```
sqlite> delete from employee where age=28;
```

查看结果：

```
sqlite> select * from employee;
1|dq|male|24
2|jz|female|20
3|pp|male|26
5|zc|male|25
7|hello|male|30
```

● 表的删除

使用 SQL 命令 `drop`，删除整个表的方法如下所示：

```
sqlite> drop table employee;
```

表被删除后，可以再使用内部命令 `.table` 查看表的情况。

● SQLite 的图形化工具

为了方便地查询和操作数据库，SQLite 还有图形化的工具。这些图形化工具的操作目标是*.db 数据库文件，可以进行界面操作和 SQL 命令操作。

例如：SQLite Database Browsers 工具的界面如图 11-18 所示。

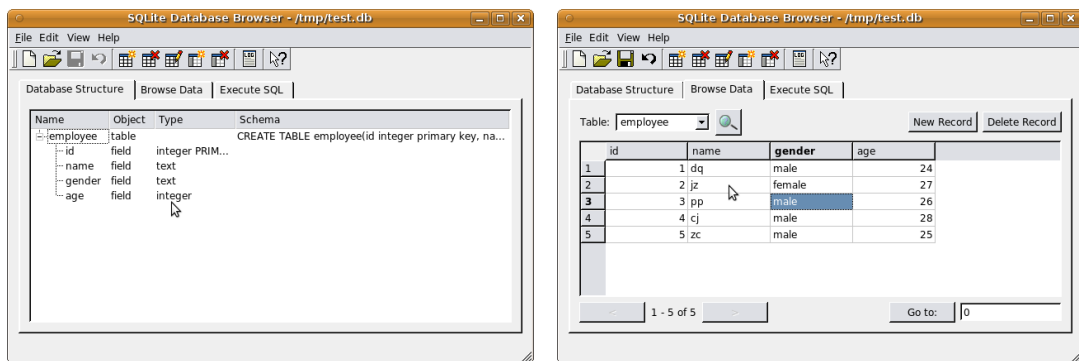


图 11-18 SQLite Database Browsers 工具（左：表的结构；右：表的内容）

SQLite 的图形化工具可以直接浏览数据库文件的结构，主要操作包括针对表的结构的操作和针对表的内容的操作，操作完成后，可以像保存普通文件一样保存数据库文件。

3. SQLite 的程序开发

SQLite 的源代码有两种形式：一种使用 `configure` 脚本方式配置，另一种使用 `Makefile` 直接编译。经过编译后形成名为 `libsqlite3.so` 的动态库，并以 `sqlite3.h` 为 API 的头文件。

SQLite (v3) 提供的 API 在 `sqlite3.h` 中，如下所示：

```
int sqlite3_open(const char *filename,sqlite3 **ppDb);
int sqlite3_close(sqlite3 *);
const char *sqlite3_errmsg(sqlite3*);
```

```
int sqlite3_exec(
    sqlite3*,
    const char *sql,
    int (*callback)(void*,int,char**,char**),
    void *,
    char **errmsg
);
```

sqlite3_open()函数和sqlite3_close()函数分别用于打开和关闭数据库,其中使用的sqlite3类型的指针为SQLite数据库的句柄。sqlite3_exec()函数是主要的接口,用于执行SQL命令,用字符串sql传入。如果执行的是查询命令,通常需要使用回调函数callback获得返回的内容。

回调函数callback四个参数的含义如下所示:

- 第一个参数类型为void*, 调用程序通过sqlite3_exec传入, 由调用者自定义。
- 第二个参数类型为int, 每次调用传入项目的个数。
- 第三个参数类型为char**, 表示传入项目的值。
- 第四个参数类型为char**, 表示其他的传入项目。

4. SQLite 的编程示例

一段基于SQLite的API编程示例程序如下所示:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
int rscallback(void *p, int argc, char **argv, char **argvv)
{
    int i;
    *(int *)p = 0;
    for(i=0; i<argc; i++) {
        printf("%s=%s ", argvv[i], argv[i]?argv[i]:"NULL");
    }
    putchar('\n');
    return 0;
}
int main(void)
{
    sqlite3 *db;
    char *err = 0;
    int ret = 0;
    int empty = 1;
    ret = sqlite3_open("./test.db", &db);
    if(ret != SQLITE_OK) {
        fputs(sqlite3_errmsg(db), stderr);
        fputs("\n", stderr);
        exit(1);
    }
    ret = sqlite3_exec(db, "select * from employee;", rscallback, &empty, &err);
    if(ret != SQLITE_OK) {
        fputs(err, stderr);
        fputs("\n", stderr);
        sqlite3_close(db);
        exit(1);
    }
    if(empty) {
```

```
        fputs("table employee is empty\n", stderr);
        exit(1);
    }
    sqlite3_close(db);
    return 0;
}
```

该程序打开了当前目录中的数据库文件 `test.db`，然后执行了一行 SQL 查询命令，查询的结果在自定义的回调函数 `rscallback` 中返回。

对程序进行编译，需要连接 `libsqlite3.so` 动态库。生成可执行程序后可以直接在命令行运行，执行的结果如下所示：

```
$ ./testdb
id=1 name=dq gender=male age=24
id=2 name=jz gender=female age=27
id=3 name=pp gender=male age=26
id=4 name=cj gender=male age=28
id=5 name=zc gender=male age=25
```

从返回结果中可见，回调函数 `rscallback()` 一共被调用了 5 次，也就是查询的 5 次返回结果。在每次返回的时候，`argc` 的值都为 4，数据返回了 4 个项目，这些内容就是 `argv[]` 数组中的某一个。

第 12 章

Linux 驱动基础

12.1 Linux 驱动概述

12.1.1 驱动的理念和结构

Linux 操作系统是一种方便移植的操作系统，设备驱动程序的处理分成了若干个层次。Linux 的驱动程序涉及的内容通常具有以下几个方面。

- 第一方面：内核所提供的复杂而完整的设备驱动程序架构。
- 第二方面：具体的驱动程序实现对某个硬件的操作。
- 第三方面：内核的通用部分调用设备驱动架构。
- 第四方面：用户空间的程序使用通用的系统调用，与驱动程序交互。

正如系统调用是应用程序和内核（kernel）之间的接口，而设备驱动程序是操作系统内核和机器硬件之间的接口。驱动程序部分通常运行于 Linux 的内核空间，并与硬件进行交互。驱动程序一方面需要将硬件的状况表示到 Linux 内核中，对其进行控制；另一方面也需要提供对上层软件（包括 Linux 内核和用户空间）的接口，供其他部分调用。

在以上几个方面中，真正与硬件相关的是第二部分，也是 Linux 系统中驱动程序实现的要点。对于第四方面，在用户空间使用驱动程序，通常是通过标准的系统调用完成的，但是针对某些驱动的特殊接口，可能使用不同的参数。对于第三部分，在内核中也可能调用驱动程序，但是此时调用的通常是驱动架构的标准接口，而不是具体驱动程序本身。因此，驱动程序架构是驱动程序的软件结构中承上启下的部分。

提示：Linux 内核源代码按照不同驱动的架构区分子目录，每个目录中包括各种具体驱动。

具体的驱动程序是内核的一部分，运行于内核空间，主要完成以下的功能：

- 对设备初始化和释放。
- 把数据从内核传送到硬件和从硬件读取数据。

- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
- 检测和处理设备出现的错误。

Linux 操作系统的驱动和上层软件结构如图 12-1 所示。

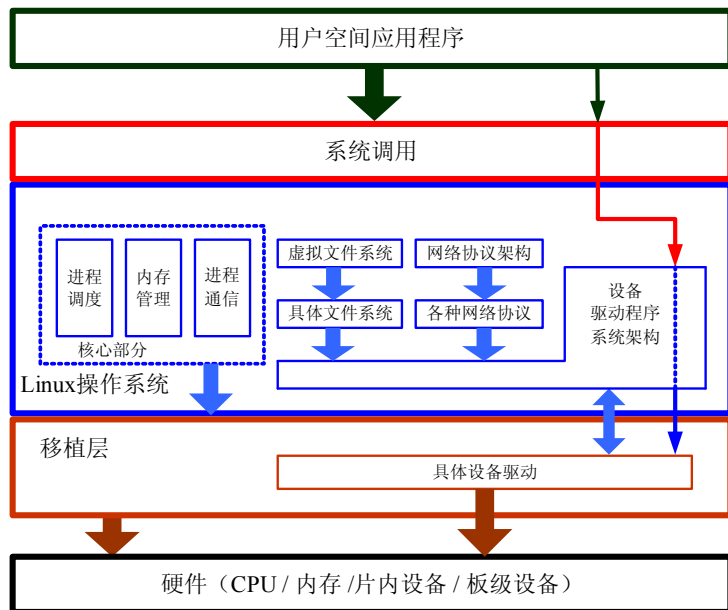


图 12-1 Linux 操作系统的驱动和上层软件结构

Linux 驱动程序的几点说明：

- 设备驱动程序架构的本身也分为多个层次。
设备驱动程序架构的主要作用是为可移植的部分，也就是具体驱动程序的实现，提供标准的接口，由具体驱动去实现。随着 Linux 驱动架构越来越复杂，设备的驱动架构本身也通常被划分成多个层次。例如：对于显示的驱动，字符设备架构是第一层架构，FrameBuffer 架构是第二层架构，FrameBuffer 架构基于字符设备架构，具体的 FrameBuffer 驱动再基于 FrameBuffer 架构去实现。
- 并非所有与硬件相关的部分都通过驱动程序架构来访问。
Linux 当中的与硬件相关的除了驱动程序之外，还有移植部分，这些移植部分也是和硬件相关的。例如：针对定时器、中断控制器的硬件的软件实现，包含在板级支持目录当中，而不是在驱动程序部分。
- 基于驱动程序的实现并非都与硬件相关。
驱动程序架构通常为硬件提供移植层，但是也有一些实现与硬件无关，例如：内存字符设备、内存盘块设备、回环网络设备都是驱动的实现，但是与具体的硬件无关。
- 并非所有驱动都有到用户空间的接口。
驱动程序通常有到用户空间的接口，但也有某些驱动程序实现后，其“子系统”模式提供给其他内核模块接口，这样的驱动就没有对用户空间的接口。

12.1.2 驱动程序对用户空间的接口

内核驱动在用户空间的接口主要为设备文件，但也有其他的形式。设备文件是文件类型的接口，除此之外，也有其他形式的文件接口，也有非文件形式的接口。

Linux 操作系统的驱动程序分成 3 种基本的类型设备：

- 字符设备（Char Device）。
- 块设备（Block Device）。
- 网络设备（Net Device）。

事实上，这种分类方式是按照驱动程序对用户空间的接口来区分的。字符设备和块设备是特殊的文件，在用户空间，通过设备文件访问字符设备和块设备，通过 socket 套接字访问网络设备。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，硬件设备通常只是一个设备文件（或者是类似文件的内容），应用程序可以像操作普通文件一样对硬件设备进行操作。

提示：Linux 驱动 3 种类型的设备是一个较老的概念，从目前的驱动角度来看，无论实际的驱动类型，还是对用户空间的接口，都不限于这 3 种类型。

12.2 设备文件和相关文件系统

12.2.1 设备文件

Linux 将外设看作一个文件来管理，用户使用外设就像使用普通文件一样，这就是 Linux 中的设备文件。设备文件的属性由三部分信息组成：

- 第一部分是文件的类型，值可能是 c/b（c 代表字符设备，b 代表块设备）。
- 第二部分是设备的“主设备号”。
- 第三部分是设备的“次设备号”。

设备类型和主设备号结合在一起通常确定了设备文件的驱动程序及其界面，而次设备号则说明目标设备是同类设备中的第几个。设备文件存放在/dev/目录下，它使用设备的主设备号和次设备号来区分指定的外设。

设备节点本身是否可以使用与其文件属性有关，而与其路径无关，可以放置于各个目录中，不仅限于/dev/目录。

查看/dev/目录当中一些字符设备的节点如下所示：

```
crw-rw-rw- 1 root root      1,  7 2010-12-13 18:28 /dev/full
crw-rw-rw- 1 root root      1,  3 2010-12-13 18:28 /dev/null
crw-rw-rw- 1 root root      1,  5 2010-12-13 18:28 /dev/zero
crw-rw-rw- 1 root tty       5,  0 2010-12-18 20:29 /dev/tty
crw-rw---- 1 root dialout   4, 64 2010-12-13 18:28 /dev/ttyS0
crw----- 1 root root      5,  1 2010-12-13 18:29 /dev/console
crw-rw---- 1 root video    29,  0 2010-12-13 18:28 /dev/fb0
```

例如：/dev/console 文件的信息以 c 开头，这是一个字符设备，主设备号是 5，次设备

号是 1，文件权限、用户等信息与普通文件相同。`/dev/full`、`/dev/null` 和 `/dev/zero` 几个设备都是主设备为 1 的字符设备，次设备号表示具体的设备。

查看 `/dev` 目录当中一些块设备的节点如下所示：

```
brw-rw---- 1 root disk      1,  0 2010-12-13 18:28 /dev/ram0
brw-rw---- 1 root disk      1,  1 2010-12-13 18:28 /dev/ram1
brw-rw---- 1 root disk      8,  0 2010-12-13 18:28 /dev/sda
brw-rw---- 1 root disk      8,  1 2010-12-13 18:28 /dev/sda1
brw-rw---- 1 root floppy    2,  0 2010-12-13 18:28 /dev/fd0
brw-rw---- 1 root disk      7,  0 2010-12-13 18:28 /dev/loop0
```

例如：`/dev/loop0` 文件的信息以 `c` 开头，这是一个块设备，主设备号是 7，次设备号是 0，`/dev/ram0`、`/dev/ram1` 几个设备都是主设备为 1 的块设备，次设备号表示具体的设备。

某些驱动程序在 `/dev` 的子目录中，例如，输入设备在 `/dev/input` 目录中，其中的一些内容如下所示：

```
crw-r----- 1 root root 13, 64 2010-05-30 16:28 event0
crw-r----- 1 root root 13, 65 2010-05-30 16:28 event1
crw-r----- 1 root root 13, 66 2010-05-30 16:28 event2
crw-r----- 1 root root 13, 63 2010-05-30 16:28 mice
crw-r----- 1 root root 13, 32 2010-05-30 16:28 mouse0
crw-r----- 1 root root 13, 33 2010-06-18 08:35 mouse1
```

`event0`、`mice`、`mouse0` 都是输入类型的设备，主设备号都是 13。

提示：设备节点的功能与类型、主次设备号相关，与设备节点的位置无关，在不同的系统中，设备节点的路径也可能不同。

12.2.2 sys 文件系统

`sys` 文件系统是 Linux 内核中设计较新的一种虚拟的基于内存的文件系统，它的作用与 `proc` 有些类似，但除了与 `proc` 相同的具有查看和设定内核参数功能之外，还有为 Linux 统一设备模型作为管理之用。

`sys` 文件系统当中的文件只有读和写两个接口，比较简单，因此不能支持复杂的操作。

- 显示信息：也就是读取，可以通过 `cat` 命令。
- 控制：也就是写入，可以通过 `cat` 或 `echo` 命令实现重定向。

提示：`sys` 文件系统靠 Linux 的设备模型来支持，核心的结构是 `kobject` 结构。

`sys` 文件系统的各个子目录如下所示。

- `/sys/block`：系统中当前所有的块设备。
- `/sys/bus`：这是内核设备按总线类型分层放置的目录结构，`devices` 中的所有设备都连接于某种总线之下。
- `/sys/class`：这是按照设备功能分类的设备模型。
- `/sys/devices`：内核对系统中所有设备的分层次表达模型。
- `/sys/dev`：字符设备和块设备的主次号，连接到真正的设备（`/sys/devices`）中。

- `/sys/fs`: 按照设计是用于描述系统中所有的文件系统。
- `/sys/kernel`: 内核所有可调整参数的位置。
- `/sys/module`: 系统中所有模块的信息。
- `/sys/power`: 系统中的电源选项。

`sys` 文件系统主要的功能是为用户空间提供信息，其中大量使用连接的方式，同样的内容可以在不同的目录中找到。这种情况就表示了按照不同方式的分类，例如对于设备，可以有按总线连接查看、按类型查看。

例如，查看系统所支持的各种休眠模式的命令如下所示：

```
$ cat /sys/power/state
standby mem disk
```

通过读取 `sys` 文件系统的文件，可以列出 `standby`、`mem`、`disk` 3 个字符串，它们表示系统支持的 3 种休眠方式。

控制进入休眠的命令如下所示：

```
$ echo standby > /sys/power/state
```

将 `standby` 字符串写入 `sys` 文件系统的文件，表示让系统实现休眠。

无论是读取，还是写入，这些内容都是通过内核提供的特殊机制提供的。

查看 `bus` 目录中的内容如下所示：

```
$ ls /sys/bus/
acpi bluetooth eisa isa MCA pci pci_express platform pnp scsi serio spi usb
```

`bus` 目录中的内容都是一些子目录，表示系统中支持总线类型，此处列出的内核驱动的软件系统中的总线，不一定和硬件总线相对应。例如，`platform` 表示平台总线。

查看 `platform` 目录，列出的部分内容如下所示：

```
$ ls /sys/bus/platform/ -l
total 0
drwxr-xr-x 2 root root 0 2010-10-10 13:18 devices
drwxr-xr-x 7 root root 0 2010-10-10 13:18 drivers
-rw-r--r-- 1 root root 4096 2010-10-10 13:18 drivers_autoprobe
--w----- 1 root root 4096 2010-10-10 13:18 drivers_probe
--w----- 1 root root 4096 2010-10-10 13:18 uevent
```

查看 `drivers` 目录，列出的部分内容如下所示：

```
$ ls -l /sys/bus/platform/drivers
total 0
drwxr-xr-x 2 root root 0 2010-10-10 13:19 dcdbas
drwxr-xr-x 2 root root 0 2010-10-10 13:19 i8042
drwxr-xr-x 2 root root 0 2010-10-10 13:19 parport_pc
drwxr-xr-x 2 root root 0 2010-10-10 13:19 pcspkr
drwxr-xr-x 2 root root 0 2010-10-10 13:19 serial8250
```

`/sys/bus/platform/devices/` 中的内容是一些软连接，连接到了 `/sys/devices/` 目录中。实际上，后者是设备信息的真正地方，前者的连接就相当于通过另外的一种分类形式，查看这些设备信息。

查看 `devices` 目录中的内容如下所示：

```
$ ls /sys/devices/
isa LNXSYSTM:00 pci0000:00 platform pnp0 pnp1 system virtual
```

这里列出的内容是几个目录，表示按照某种分类的设备集合。在不同的系统当中，由于硬件系统的差异，此处的目录差别会比较大。

通过 `class` 目录可以查看根据分类情况的设备，如下所示：

```
$ ls /sys/class/
atm          dma          graphics    input       net          ppdev        scsi_disk    sound
usb_endpoint video_output backlight  dmi         hidraw       mem          pci_bus
printer      scsi_generic spi_master  usb_host    vtconsole    bluetooth    firmware
hwmon        misc         power_supply scsi_device scsi_host    tty          vc
```

此处列出的目录当中，`input`（输入）、`graphics`（图形）、`tty`（终端）等表示的都是具体设备的类型，而 `mem` 表示内存设备，是一种与硬件无关的驱动。

查看 `mem` 目录当中的内容如下所示：

```
$ ls /sys/class/mem/ -l
total 0
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 full -> ../../devices/virtual/mem/full
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 kmsg -> ../../devices/virtual/mem/kmsg
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 mem -> ../../devices/virtual/mem/mem
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 null -> ../../devices/virtual/mem/null
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 oldmem -> ../../devices/virtual/mem/oldmem
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 port -> ../../devices/virtual/mem/port
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 random -> ../../devices/virtual/mem/random
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 urandom -> ../../devices/virtual/mem/urandom
lrwxrwxrwx 1 root root 0 2010-06-20 13:33 zero -> ../../devices/virtual/mem/zero
```

`mem` 目录中的子目录都是各个设备，从上面列出的情况可以得知，这些内容是到 `/sys/virtual/mem/` 目录当中的子目录的连接，此处的 `virtual` 表示虚拟设备类型。

12.2.3 proc 文件系统

`proc` 文件系统当中也包含了一些与驱动程序相关的内容，主要的内容如下所示。

- `/proc/interrupts`：中断信息，包括中断发生次数。
- `/proc/devices`：设备信息。
- `/proc/iomem`：I/O 的内存映射信息。
- `/proc/ioports`：I/O 端口信息（x86 有效）。
- `/proc/misc`：MISC 设备的信息。
- `/proc/irq/`：各个中断的目录，子目录为中断号。
- `/proc/bus/`：总线信息的目录。
- `/proc/tty/`：TTY 设备的相关信息。

查看 `devices` 文件可以得到设备节点的信息，列出的部分内容如下所示：

```
$ cat /proc/devices
Character devices:
1 mem
```

```

4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
6 lp
7 vcs
10 misc
13 input
14 sound

251 hidraw
252 usbmon
253 bsg
254 rtc

Block devices:
1 ramdisk
2 fd
259 blkext
7 loop
8 sd
9 md
11 sr
65 sd

252 device-mapper
253 pktcdvd
254 mdp

```

通过 `/proc/devices` 文件可以查看系统的各个设备，首先按照字符设备和块设备进行分类，然后分行列出，前面的数字就是各个设备的主设备号，后面的字符串是设备的名称。

通过 `/proc/interrupts` 文件可以查看系统中各个中断的发生次数。如果系统中具有多 CPU，它们的终端将分别列出，最右侧一列为中断号。

查看 `/proc/interrupts` 文件列出的一个情况如下所示：

```

$ cat /proc/interrupts
          CPU0       CPU1
0:         9284       1403   IO-APIC-edge   timer
1:           3         1   IO-APIC-edge   i8042
4:           1         1   IO-APIC-edge
6:           2         3   IO-APIC-edge   floppy
7:           0         0   IO-APIC-edge   parport0
8:           0         1   IO-APIC-edge   rtc0
9:           0         0   IO-APIC-fasteoi   acpi
12:          2         4   IO-APIC-edge   i8042
16:       346255       206   IO-APIC-fasteoi   uhci_hcd:usb3, HDA Intel
17:     7816900        27   IO-APIC-fasteoi   uhci_hcd:usb4, uhci_hcd:usb6
18:         11    3464936   IO-APIC-fasteoi   uhci_hcd:usb7
22:    51706526         1   IO-APIC-fasteoi   ehci_hcd:usb1
23:         181    8970258   IO-APIC-fasteoi   ehci_hcd:usb2, uhci_hcd:usb5
26:    57823593    10080       PCI-MSI-edge   ahci
27:          35    77017633   PCI-MSI-edge   eth0
28:           8    12386690   PCI-MSI-edge   i915
NMI:           0         0   Non-maskable interrupts
LOC: 1563524731 1568790144   Local timer interrupts

```

```

SPU:      0      0      Spurious interrupts
PMI:      0      0      Performance monitoring interrupts
PND:      0      0      Performance pending work
RES: 185097340 187120469 Rescheduling interrupts
CAL:      34372   49511   Function call interrupts
TLB:      6959142 6832708 TLB shootdowns
TRM:      0      0      Thermal event interrupts
THR:      0      0      Threshold APIC interrupts
MCE:      0      0      Machine check exceptions
MCP:      10592   10592   Machine check polls
ERR:      1
MIS:      0

```

以上列出的内容，来自一个 x86 体系结构双核处理器，因此有两个 CPU 被分别显示出，第一行的数字表示的就是中断号，CPU 下面的数字表示中断发生的次数。中断号和实际硬件系统的情况密切相关。一般嵌入式系统的中断比 x86 体系结构更多。

iomem 表示系统的内存映射情况，查看文件的内容如下所示：

```

$ cat /proc/iomem
00000000-00001fff : System RAM
00002000-00005fff : reserved
00006000-00009ebff : System RAM
00009ec00-00009ffff : RAM buffer
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000ffff : reserved
    000f0000-000ffff : System ROM
00100000-7d4ff7ff : System RAM
    00100000-00594466 : Kernel code
    00594467-007a6f07 : Kernel data
    00852000-008e1ed7 : Kernel bss
7d4ff800-7d553bff : ACPI Non-volatile Storage
7d553c00-7d555bff : ACPI Tables

fee00000-feeffff : reserved
    fee00000-fee00fff : Local APIC
ff970000-ff9707ff : 0000:00:1f.2
    ff970000-ff9707ff : ahci
ff980800-ff980bff : 0000:00:1d.7
    ff980800-ff980bff : ehci_hcd
ffb00000-fffff : reserved

```

前面的内容如 00000000-00001fff 是用 32 位的数字表示的地址范围，表示内存的地址空间，后面的内容表示此块内存区域的名称。

ioports 表示端口的信息，查看文件的内容如下所示：

```

$ cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
0064-0064 : keyboard
0070-007f : rtc0
0080-008f : dma page reg

ff40-ff5f : 0000:00:1d.2

```

```
ff40-ff5f : uhci_hcd
ff60-ff7f : 0000:00:1d.1
ff60-ff7f : uhci_hcd
ff80-ff9f : 0000:00:1d.0
ff80-ff9f : uhci_hcd
```

前面的内容如 0000-001f 是用 16 位表示的端口范围，后面是端口的名称。此处的信息，通常只有 x86 具有，在嵌入式系统的处理器中没有端口的概念，因此 `ioports` 文件也为空。

`/proc/irq/`目录主要用于提供中断平衡的信息，也就是中断的多处理器分配问题。SMP 的全称是 (Symmetrical Multi-Processing, 对称多处理)。系统的中断可以分配给不同的 CPU 进行处理，`/proc/irq/`目录中列出的内容是各个中断号的子目录，`/proc/irq/<irq_number>/smp_affinity` 文件的表示方式是位掩码，表示此中断交由哪一个 CPU 来处理。

查看两个中断号的信息如下所示：

```
$ sudo cat /proc/irq/8/smp_affinity
03
$ sudo cat /proc/irq/6/smp_affinity
02
```

03 就是二进制的 11b，表示两个 CPU 都可以进行处理；02 就是二进制的 10，表示只有第一个 CPU 处理，第 0 个不处理。此处的内容和 `/proc/interrupts` 文件结合起来，可以得知系统多中断的处理情况。

`/proc/bus/`目录提供一些总线上的信息，其中通常包括 `pci` 和 `input` 等目录。查看其中一个文件的内容如下所示：

```
$ cat /proc/bus/input/devices
I: Bus=0019 Vendor=0000 Product=0001 Version=0000
N: Name="Power Button"
P: Phys=PNP0C0C/button/input0
S: Sysfs=/devices/LNXSYSTM:00/LNXSYBUS:00/PNP0C0C:00/input/input0
U: Uniq=
H: Handlers=kbd event0
B: EV=3
B: KEY=100000 0 0 0
```

此处提供的是输入设备的一些信息。对于其他类型的内容，这个文件的格式并不相同。

第 13 章

Linux 的内核编程

13.1 Linux 内核编程概述

Linux 内核编程的工作将实现在内核态运行的代码。通常情况下，Linux 系统运行时，应用程序运行在用户空间，而内核以及其中的驱动程序运行在内核空间。

Linux 内核的代码是由纯 C 语言和少量体系结构相关的汇编程序写成的。因此内核编程这些语言，主要需要基于内核中的头文件，主要头文件在内核源代码的路径为：`include/linux`。

Linux 内核编程几方面的工作如下所示。

- Linux 内核的通用内容：各种硬件均通用。
- 特定体系结构的内容：ARM、x86 等不同的体系结构中特定的。
- 各种硬件设备的驱动程序：各种具体硬件相关的。
- 非硬件相关的内核模块：与硬件无关，但独立使用的部分。

Linux 内核编程的要点如下所示：

- 内核中没有标准的 C 库。
- 内核提供了基础和特殊的编程接口。
- 内核与用户空间的交互使用特殊手段。
- 可作为内核的一部分或者模块（*.ko）存在。

13.2 内核模块的编写

13.2.1 Linux 内核中的模块

Linux 模块（module）是内核中相对独立的单元。“模块”一词有着两方面的含义：一方面，如果一部分内容构建成不编译到内核映像中，只是作为一个*.ko 存在，从存在形式

的角度，这些 ko 文件被称为模块，可以动态插入和卸载；另一方面，即使一部分内容编入内核映像，但是它有着自身的初始化和卸载，从内核初始化的角度，这部分内容也被称为模块。

可加载的（动态的）内核模块（*.ko）是可在系统运行时动态地安装和拆卸的内核功能单元。利用可加载机制，可以根据需要，无须对整个内核重新编译连接，而将内核模块动态插入运行中的内核，成为内核的一个有机组成部分，也可以从内核卸载已安装的模块。

从 Linux 内核的角度，只有 CONFIG_MODULES 宏打开的时候，才能使用动态可加载的内核模块的功能。此选项在 Linux 内核的顶级配置菜单中选择，配置名称为：“Enable loadable module support”。这个定义来自于 init/Kconfig 文件，也只有这个选项打开后，内核的配置文件中才允许选择=m（模块），菜单配置中才会出现形如<M>的项目。

模块可以实现一部分功能，其本身可以是硬件相关或者硬件无关的。insmod 和 rmmod 用于插入和卸载模块，对应于 Linux 的 init_module 和 delete_module 系统调用。

Linux 模块的功能和加载如图 13-1 所示。

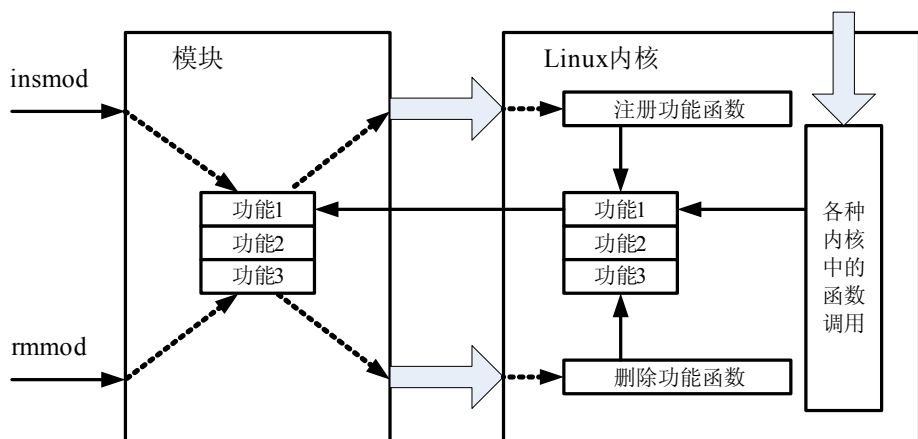


图 13-1 Linux 模块的功能和加载

一个模块被插入（加载）时的工作流程如下所示：

- 1) 打开要安装的模块，把它读到用户空间。
- 2) 必须把模块内涉及对外访问的符号（包括了其中的函数或变量）连接到内核，即把这些符号在内核映像中的地址填入该模块需要访问这些符号的指令及数据结构中。
- 3) 在内核创建一个模块的数据结构，申请资源。
- 4) 最后，把用户空间中完成了连接的模块映像装入内核空间，并在内核中注册本模块的有关数据结构，其中有指向执行相关操作函数的指针。

模块卸载的过程基本上和加载的过程相反。

提示：模块（module）一词常常是指生成的*.ko 文件。对于一部分内核中的代码，即便不编成*.ko 文件，有时也被称为模块。

13.2.2 内核模块的编译结构

模块可以编入内核，或者作为独立的*.ko 文件，无论哪种方式，都需要使用 Makefile 来组织。编译一个内核模块有两种方式：

- 直接使用内核中的 Kconfig 和 Makefile 结构。
- 使用外部的 Makefile。

文件进行配置编译的几个部分如下所示。

- **Kconfig 文件**：定义编译宏、相关的菜单、宏的默认值、选项的帮助，该文件需要加入整体编译流程，才能被使用。
- **Makefile 文件**：使用宏控制进行编译文件。
- **宏控制的代码**：在代码中使用宏条件编译。

内核结构中的 Kconfig 文件：

```
config SAMPLE_MODULE
    depends on OTHERS
    default y
    bool "Enable sample module ..."
    ---help---
    Sample Module.....
```

内核结构中的 Makefile 文件：

```
obj-$(CONFIG_SAMPLE_MODULE) += sample_module.o
```

注意：内容可以被编入内核或者编译成独立的 ko 文件，这通常与 Makefile 无关，而在于配置后选项的值是 y 还是 m。

使用外部 Makefile 的方式通常有如下的写法：

```
obj-m += sample_module.o
KDIR:=<Kernel_path>
PWD=$(shell pwd)
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    rm -rf *.o *.mod.o *.ko *.mod.c *.o.cmd *.mo.o.cmd *.ko.cmd .tmp_versions
```

编译方法是在当前目录下进行 make，如下所示：

```
$ make ARCH=arm CROSS_COMPILE=<prefix>
```

由于 Makefile 当中指定了\$(MAKE) -C，因此执行实际目录会转到内核目录。

obj-m 和 obj-y 分别表示编入内核或模块，在编译命令上也有差别。

内核源代码的头文件 linux/module.h 包括了对模块的操作。

两个主要的定义如下所示：

```
#include <linux/module.h>
module_init(module_init);
module_exit(module_cleanup);
```

module_init 和 module_exit 两个宏定义了模块被初始化和卸载的功能，其参数为一个函

数，分别在初始化和卸载的时候执行。

对于一个编成*.ko 的模块，当对模块执行 insmod（插入模块）和 rmmod（移除模块）操作的时候，将调用为模块编写的初始化和卸载函数。

模块可以使用如下宏导出符号到内核符号表：

```
EXPORT_SYMBOL("符号名");
EXPORT_SYMBOL_GPL("符号名")
```

导出的符号可以被其他模块使用，EXPORT_SYMBOL 表示直接导出，EXPORT_SYMBOL_GPL 适用于包含 GPL 许可权的模块。

在用户空间中，可以查找到内核符号表，方法是使用 proc 文件系统的/proc/kallsyms 文件。此文件列出的内容包括符号以及符号所在的内存地址。

为模块声明许可的方法如下：

```
MODULE_LICENSE("许可名称");
```

内核认识的特定许可包括"GPL"、"GPL v2"、"GPL and additional rights"、"Dual BSD/GPL"、"Dual MPL/GPL"、"Dual MPL/GPL"和"Proprietary"等。如果没有声明许可，就假定它是私有的。

模块相关的其他描述性定义：

```
MODULE_AUTHOR ("模块的作者");
MODULE_DESCRIPTION("模块描述声明");
MODULE_VERSION ("模块修订版本号");
MODULE_ALIAS ("模块的另一个名字");
MODULE_DEVICE_TABLE ("模块支持的设备");
MODULE_FIRMWARE("模块的固件的名称");
```

一个内核模块的源代码如下所示：

```
#include <linux/init.h>
#include <linux/module.h>
static int sample_module_init(void)
{
    printk(KERN_EMERG"sample_module register\n");
    return 0;
}
static void sample_module_cleanup(void)
{
    printk(KERN_EMERG"sample_module unregister\n");
    return;
}
module_init(sample_module_init);
module_exit(sample_module_cleanup);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("HanChao");
```

在以上的代码中，使用 module_init 和 module_exit 两个宏声明模块的初始化和卸载函数，它们将在 insmod 和 rmmod 的时候被调用。MODULE_LICENSE 等宏则用于声明辅助的模块信息。

13.3 内核编程接口

内核编程的基础一般由内核中的各个头文件提供，这些头文件通常在 Linux 内核的 `include/linux` 目录中，因此按照 `<linux/XXX.h>` 的方式对其进行包含。

13.3.1 Linux 编程风格

Linux 内核的编程风格也在 `Documentation/CodingStyle` 文件中有所说明，其中包括了缩进与换行（用 Tab、行尾不要加空格）、把长的行和字符串打散（每行 80 列）、大括号和空格的放置（前面是否用新行）、命名、`typedef`、注释等书写样式的建议，以及函数（`EXPORT*` 宏导出）、函数退出途径（`goto` 的用法）、数据结构、宏、枚举和 RTL、打印内核消息、分配内存、内联、函数返回值及命名等编程方式的建议。

内核的编程主要有以下几个方面的特点：

- 某些头文件可以在内核空间和用户空间同时使用，其中有差别的内容使用宏 `__KERNEL__` 进行区分。
- 某些头文件引用硬件相关的头文件 `<asm/XXX.h>`。
- 各种名称使用小写字母，每个词之间使用 `_` 分隔。
- `_t` 后缀表示不透明结构，`__` 前缀表示内部函数。
- 缩进使用 `tab`，函数的 `{}` 进行换行，注释使用 `C89/*...*/` 方式，不用 `C99`。

例如：在头文件 `fs.h` 当中一个公共结构的定义如下所示：

```
struct files_stat_struct {
    int nr_files;          /* read only */
    int nr_free_files;     /* read only */
    int max_files;         /* tunable */
};
```

`files_stat_struct` 结构就是一个公共的结构，这种结构通常可以被内核中的其他部分直接使用其中的成员。

在 `fs.h` 当中一个不透明结构的定义如下所示：

```
struct inodes_stat_t {
    int nr_inodes;
    int nr_unused;
    int dummy[5];         /* padding for sysctl ABI compatibility */
};
```

`inodes_stat_t` 结构以 “`_t`” 为后缀，这种写法在内核当中表示不透明结构，内核的其他部分可以使用这种结构，但不应当访问其中的成员。

头文件 `fs.h` 当中内部函数和外部函数的定义如下所示：

```
static inline int __mandatory_lock(struct inode *ino)
{
    return (ino->i_mode & (S_ISGID | S_IXGRP)) == S_ISGID;
}
static inline int mandatory_lock(struct inode *ino)
{
    return __mandatory_lock(ino);
}
```

```
return IS_MANDLOCK(ino) && __mandatory_lock(ino);
}
```

mandatory_lock()函数是一个接口函数，__mandatory_lock()函数是一个内部函数，内核的其他部分调用的时候，一般只调用接口函数，不调用内部函数。

提示：公共结构和不透明结构、接口函数和内部函数用于区分接口结构和模块内部的实现。它们的区别仅仅体现在文字的建议性说明上，并非语法的区别。

某些头文件当中使用__KERNEL__宏，具有如下的定义：

```
#ifdef __KERNEL__
#endif /* __KERNEL__ */
```

这用于头文件可能同时在内核中以及用户空间中使用的情况。这种头文件可以分别在内核和用户空间使用，其中的差别用__KERNEL__宏进行区分。

例如，在头文件 ioctl.h 当中，具有如下的定义：

```
#ifndef _LINUX_IOCTL_H
#define _LINUX_IOCTL_H
#include <asm/ioctl.h>
#endif /* _LINUX_IOCTL_H */
```

<asm/ioctl.h>表示引用体系结构的头文件，根据操作系统的不同引用不同的文件。

例如：如果在 ARM 系统中也就是 arch/arm/include/asm/ioctl.h 文件，此文件当中又有以下方式的引用：

```
#include <asm-generic/ioctl.h>
```

这里表示又引用了通用的头文件 include/asm-generic/ioctl.h。像这样就是直接引用了系统的通用（generic）目录中的同名头文件。如果某一个体系结构的头文件没有特殊的地方，直接包含通用头文件即可；如果有特殊的地方，还需要增加额外的内容。

13.3.2 Linux 编程主要接口

Linux 内核中的主要头文件如下所示。

- types.h: 使用 typedef 的基本类型定义。
- string.h: 内核中的字符串处理。
- list.h: 内核中对链表和哈希链表的支持。
- module.h: 内核模块相关的内容，包括动态增加、卸载和查找模块功能。
- kernel.h: 基本的宏，涉及打印、跟踪、控制台等功能。
- pid.h: 进程 id 的结构及获取、设置和转换等操作。
- sched.h: 内核进程和调度内容，定义了任务结构和相关操作。
- wait.h: 进程调度和控制，包括队列、等待和唤醒等。
- completion.h: 等待完成的原子操作。
- interrupt.h: 中断相关的设置、申请、取消以及 softirq 和 tasklet 相关功能。
- uaccess.h : 内核空间和用户空间的地址处理。

- **mm.h**: 内存管理相关结构、页的使用, 映射等功能。
- **slab.h**: 物理连续内存分配、释放、缓冲等功能。
- **vmalloc.h**: 虚拟内存的分配、释放、映射等功能。
- **mempool.h**: 内存缓冲区池的支持, 包括创建、销毁、分配、释放等功能。
- **timer.h**: 内核定时器相关的定义和功能函数。
- **spin_lock.h**: 自旋锁相关的功能。
- **semaphore.h**: 信号量相关的功能。
- **device.h**: 设备模型的核心部分, 包括设备的管理和设备文件系统等功能。

1. 类型

头文件 `types.h` 当中进行的类型定义如下所示:

```
/* bsd */
typedef unsigned char    u_char;
typedef unsigned short   u_short;
typedef unsigned int     u_int;
typedef unsigned long    u_long;
/* sysv */
typedef unsigned char    unchar;
typedef unsigned short   ushort;
typedef unsigned int     uint;
typedef unsigned long    ulong;
```

这些类型都是对 C 语言基本类型的简写。

`types.h` 当中的其他一些类型定义如下所示:

```
#ifndef __BIT_TYPES_DEFINED__
#define __BIT_TYPES_DEFINED__
typedef      __u8      u_int8_t;
typedef      __s8      int8_t;
typedef      __u16     u_int16_t;
typedef      __s16     int16_t;
typedef      __u32     u_int32_t;
typedef      __s32     int32_t;
#endif /* !(__BIT_TYPES_DEFINED__) */
```

`types.h` 当中的其他定义如下所示:

```
typedef      __u8      uint8_t;
typedef      __u16     uint16_t;
typedef      __u32     uint32_t;
#if defined(__GNUC__)
typedef      __u64     uint64_t;
typedef      __u64     u_int64_t;
typedef      __s64     int64_t;
#endif
```

其他一些 C 语言中的定义 `size_t` 和 `ssize_t` 如下所示:

```
#ifndef _SIZE_T
#define _SIZE_T
typedef __kernel_size_t    size_t;
#endif
```

```
#ifndef __SSIZE_T
#define __SSIZE_T
typedef __kernel_ssize_t    ssize_t;
#endif
```

`__kernel_size_t` 本身来自体系结构的定义，一般就是 32 位无符号整数。

2. 字符串处理

内核当中的字符串处理函数与用户空间的 C 库中的类似，由 `string.h` 头文件提供接口，其中的一些定义如下所示：

```
#include <asm/string.h>
#ifndef __HAVE_ARCH_STRCPY
extern char * strcpy(char *,const char *);
#endif
#ifndef __HAVE_ARCH_STRNCPY
extern char * strncpy(char *,const char *, __kernel_size_t);
#endif
```

几个用于内存处理的函数，也包含在 `string.h` 头文件中，如下所示：

```
#ifndef __HAVE_ARCH_MEMSET
extern void * memset(void *,int,__kernel_size_t);
#endif
#ifndef __HAVE_ARCH_MEMCPY
extern void * memcpy(void *,const void *,__kernel_size_t);
#endif
#ifndef __HAVE_ARCH_MEMMOVE
extern void * memmove(void *,const void *,__kernel_size_t);
#endif
```

有些函数在内核空间和用户空间中还有所区别，并不相同，如下所示：

```
#ifndef __HAVE_ARCH_STRLLEN
extern __kernel_size_t strlen(const char *);
#endif
#ifndef __HAVE_ARCH_STRNLEN
extern __kernel_size_t strnlen(const char *,__kernel_size_t);
#endif
```

这些函数虽然和用户空间中稍有区别，但使用方法是类似的。

3. 打印

由于不能使用 C 语言库，因此在内核中打印调试信息也不能使用 `printf()`。在打印调试信息的功能上，内核应该使用打印函数 `printk()`。`printk()` 的信息被内核存储到内核循环缓冲区，可以表现为用户空间的 `/proc/kmsg` 文件，通过 `dmesg` 命令可以读出，通过 `klogd` 和 `syslogd` 文件可以转存到 `/var/log/message` 等信息文件中。

```
#include<linux/slab.h>
int printk(const char *fmt, ...)
```

`printk` 的打印信息将会送到 Linux 的 `console`（控制台）上。在其函数内部，申请了一块静态的缓冲区，当 `console` 建立之后，可以将缓冲区的内容打印到终端上。与 `printf` 的最大不同点是 `printk` 不支持浮点数（`%f` 形式的参数格式）。

在 `printk` 中可以增加打印调试级别的选项。

```
#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 */
```

例如，打印的具体方式如下所示：

```
printk(KERN_DEBUG "priority = 7\n");
printk(KERN_INFO "priority = 6\n");
printk(KERN_NOTICE "priority = 5\n");
printk(KERN_WARNING "priority = 4\n");
printk(KERN_ERR "priority = 3\n");
printk(KERN_CRIT "priority = 2\n");
printk(KERN_ALERT "priority = 1\n");
printk(KERN_EMERG "priority = 0\n");
```

`KERN_DEBUG` 等宏附加在打印信息之前，表示打印的级别，本质上就是使用了两个字符串连接的方式。

4. 内存分配和访问

对于运行在内核空间的程序，内存分配不能使用 C 语言的库函数 `malloc()` 和 `free()`，而需使用特定函数。

`kmalloc()` 函数的第一个参数是要分配的块的大小，第二个参数为分配标志。返回值为分配的内存的虚拟地址，内存物理连续。

```
#include<linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(const void *);
void *kzalloc(size_t size, gfp_t flags);
void *kcalloc(size_t n, size_t size, gfp_t flags);
```

`kmalloc()` 函数在分配内存时所使用的标志（`flags`）如下所示。

- **GFP_KERNEL**：内核内存的正常分配，无内存分配时可能引起睡眠。通常用于执行一个系统调用。能够使当前进程在少内存的情况下进入睡眠来等待。使用 `GFP_KERNEL` 来分配内存的函数必须是可重入的，并且不能在原子上下文中运行。当前进程睡眠，内核采取正确的动作来定位一些空闲内存，或者通过刷新缓存到磁盘，或者交换出去一个用户进程的内存。
- **GFP_NOIO** 和 **GFP_NOFS**：类似 `GFP_KERNEL`，但增加了到内核能力的限制，以满足请求。
- **GFP_USER**：为用户空间分配内存，它可能睡眠。
- **GFP_HIGHUSER**：类似 `GFP_USER`，但从高端内存分配。
- **GFP_ATOMIC**：不可睡眠的分配，用来从中断处理和进程上下文之外的其他代码中分配内存，不会睡眠。在当前进程不应当被置为睡眠的时候使用，内核常试图保持

一些空闲页以便来满足原子的分配。分配甚至能够使用最后一个空闲页。当最后一个空闲页也不存在时，分配将失败。

内核中还有另一种内存分配方式，这种方式分配的是虚拟分配，物理上不一定连续。这些内容包含在 `vmalloc.h` 头文件中，如下所示：

```
#include<linux/vmalloc.h>
struct vm_struct {
    struct vm_struct *next;
    void *addr;
    unsigned long size;
    unsigned long flags;
    struct page **pages;
    unsigned int nr_pages;
    unsigned long phys_addr;
    void *caller;
}; void *vmalloc(unsigned long size);
void vfree(const void *addr);
void *vmmap(struct page **pages, unsigned int count,
            unsigned long flags, pgprot_t prot);
void vunmap(const void *addr);
```

`vm_struct` 结构用于描述虚拟内存，`vmalloc()` 函数可以分配大块内存，内存分配后得到一个连续的地址，但是由于在物理上不是连续的，因此这种内存不能使用 DMA（Direct Memory Access）进行内存访问。

5. 内存访问

C 语言函数的调用涉及参数和返回值的传递，除了直接传递的形参之外，还可以通过参数传递指针，指针指向的内存在调用者和被调用者之间传递。但内核空间 and 用户空间不可直接通过指针访问内存。因此。需要特别的函数进行处理。

`uaccess.h` 中的内容用于用户空间和内核空间进行内存交互处理，引用了体系结构相关实现：

```
int access_ok(int type,unsigned long addr,
              unsigned long size);
unsigned long copy_to_user(void *to, const void *from, unsigned long len);
unsigned long copy_from_user(void *to, const void *from, unsigned long len);
```

`copy_from_user()` 函数用于将用户空间的内存复制到内核空间，`copy_to_user()` 函数用于将内核空间的内存复制到用户空间。

6. 链表

`list.h` 提供了内核对双向链表容器的支持，链表的操作包括初始化添加、检查、合并、遍历等操作。

`list_head` 结构和其中的函数如下所示：

```
struct list_head {
    struct list_head *next, *prev;
};
void INIT_LIST_HEAD(struct list_head *list);
void list_add(struct list_head *new, struct list_head *head);
void list_del(struct list_head *entry);
void list_move(struct list_head *list, struct list_head *head);
```

```
void list_splice(const struct list_head *list, struct list_head *head);
#define list_entry(ptr, type, member)
#define list_for_each_prev(pos, head)
#define list_for_each_entry(pos, head, member)
#define list_prepare_entry(pos, head, member)
```

list_head 结构用于实现内核中链表的特殊方式，该结构的本意只是链表头。在内核需要使用链表的地方，需要由具体的结构包含 **list_head** 成员来使用链表，如下所示：

```
struct xxx_list {
    struct list_head entry;
    // 结构自己的成员
};
```

链表头 **list_head** 需要成为结构的首个成员，这样该结构的内存头就是链表头，这个结构也就可以转换成链表结构使用。首先需要使用 **INIT_LIST_HEAD()** 初始化这个结构，然后就可以使用各个链表操作函数进行链表操作。

第 14 章

Linux 的驱动核心架构

14.1 用户空间的接口

14.1.1 用户空间的驱动调用接口

Linux 内核中的驱动程序，通常是介于系统和硬件之间的部分。驱动程序被内核的公共部分调用，在大多数情况下，驱动程序需要提供从内核空间（Kernel Space）到用户空间（User Space）的接口。

驱动程序可以用几种不同的形式提供 Linux 内核空间到用户空间的接口：

- 系统调用（System Call）。
- 字符设备节点。
- 块设备节点。
- 网络设备。
- proc 文件系统。
- sys 文件系统。

除此之外，某些驱动可能只是给内核的其他部分所调用，没有直接到用户空间的接口。一般情况下，Linux 也不会通过增加系统调用提供驱动程序的接口。

14.1.2 系统调用

UNIX 系统调用的是内核空间提供给用户空间的统一接口。在各个不同的体系结构中，系统调用通常是相通的。Linux 内核中系统调用的头文件为 `unistd.h`，通常位于各个体系结构的头文件中：`arch/<arch>/include/asm/`。

内核中所使用的 `sys_XXX()` 形式的各个函数，是内核提供的系统调用的封装，它们的功能基于系统调用来实现。

例如，ARM 体系结构的内核中 `unistd.h` 文件的片断如下所示：

```
#define __NR_restart_syscall  (__NR_SYSCALL_BASE+ 0)
#define __NR_exit              (__NR_SYSCALL_BASE+ 1)
#define __NR_fork              (__NR_SYSCALL_BASE+ 2)
#define __NR_read              (__NR_SYSCALL_BASE+ 3)
#define __NR_write             (__NR_SYSCALL_BASE+ 4)
#define __NR_open              (__NR_SYSCALL_BASE+ 5)
#define __NR_close             (__NR_SYSCALL_BASE+ 6)
```

__NR_exit 等宏实际上就是系统调用号。系统调用号来自 UNIX 标准的定义，不同体系结构的系统调用号基本相同，只是在某些不常用系统调用的实现上有所区别。

14.1.3 驱动的主要调用函数

在 Linux 中，字符和块设备体现为设备节点的形式，对它们进行操作一般均基于 C 语言库中的文件操作函数。与普通文件访问的区别在于，对于设备文件的调用，每个调用的效果由驱动程序自行实现。

用于对文件打开、关闭、读、写、移动位置、同步的几个函数如下所示：

```
#include<sys/types.h>
#include<unistd.h>
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd,void * buf ,size_t count);
ssize_t write (int fd,const void * buf,size_t count);
off_t lseek(int fildes,off_t offset ,int whence);
int fsync(int fd);
```

以上函数可以完成基本的读、写功能。设备文件的读、写操作语义通常和普通文件相同，也可以根据驱动的实现自行定义。

负责端口控制的 ioctl() 函数如下所示：

```
#include<sys/ioctl.h>
int ioctl(int fd ,unsigned int cmd,...);
```

ioctl() 函数通常用于杂项控制，整型的 cmd 参数表示命令号，可以由驱动自行实现。ioctl() 函数是驱动程序实现自定义操作的主要手段。

负责查询阻塞的 poll() 函数如下所示：

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

poll() 函数的语义通常表现为阻塞的效果，当需要阻塞的时候函数不会返回，也可以设置超时。在驱动程序没有输入的时候，调用 poll() 函数可以实现无开销的等待。

负责内存映射的 mmap() 和 munmap() 函数如下所示：

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *start, size_t length);
```

mmap() 用于把一块内存映射到用户空间，完成映射后可以对这块内存实现读写，读写的效果体现在内核的存储空间上。

对于某些设备，以 `socket` 作为到用户空间的接口。例如：网络设备，实际上是某种协议族的实现，协议的内容也可以自定义。基于统一的接口函数，可以通过不同协议族、协议类型、协议来进行不同的调用，且 `sockaddr` 结构体可以根据实际需要进行扩展。

```
#include <sys/types.h>
#include <sys/socket.h>
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int shutdown(int sockfd, int how);
```

与普通的文件操作函数相同，套接字的操作函数也以文件描述符为操作对象，但是包括一些套接字的特殊接口。套接字也可以使用普通文件的读、写等接口。

提示：普通文件操作函数对应于内核的 `file_operations` 结构体。套接字函数对应于内核的 `proto_ops` 结构体。

14.2 字符设备和块设备的框架

14.2.1 文件操作 `file_operations`

文件是驱动程序的基本操作对象，它在内核空间和用户空间使用不同的结构表示。

- 文件系统中的文件：内核空间的 `inode` 结构 $\Leftrightarrow$ 用户空间的树状结构。
- 进程打开的文件：内核空间的 `file` 结构 $\Leftrightarrow$ 用户空间的文件描述符。

Linux 内核的 `include/linux/` 目录的 `fs.h` 文件中定义了 `file`、`file_operations` 几个结构体，它们分别表示进程打开的文件以及文件所支持的操作。

几个结构之间的关系如下所示：

- `file` 结构表示进程打开的文件，其中除了这个文件在磁盘中的相关信息之外，也包括一个 `file_operations` 结构体，表示这个被打开的文件所支持的操作。`file` 结构正对应了用户空间中的文件描述符。
- `file_operations` 结构主要由函数指针组成，各个函数指针基本和用户空间的文件操作函数相对应。各个函数指针的第一个成员一般都是 `file` 类型的指针（`open` 除外），表示进程打开文件的操作句柄。
- `inode` 中具有一个表示文件操作的 `file_operations` 结构。它通常在文件打开时，赋值给 `file` 中的对应结构。

14.2.2 字符设备的基本框架

字符设备特殊文件进行 I/O 操作不经过操作系统的缓冲区，进行 I/O 操作时每次只传输一个字符。典型的字符设备包括：鼠标、键盘、串口等。字符设备可以通过字符设备文件来访问。

字符驱动程序的框架如图 14-1 所示。

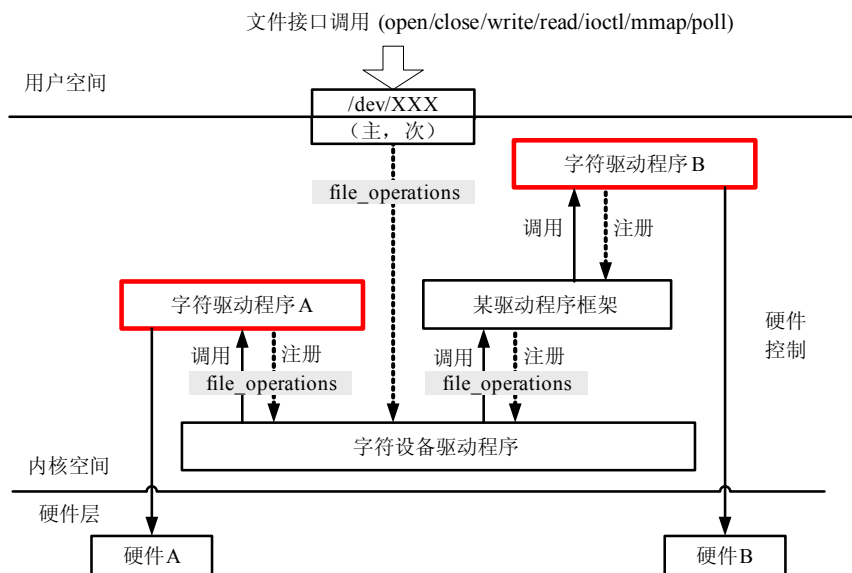


图 14-1 字符驱动程序的框架

字符设备的核心实现是 `file_operations` 结构体中的各个指针，它们大部分直接对应于用户空间的文件操作的系统调用（或者函数）。

字符设备的注册和注销，通常使用 `fs.h` 中的以下两个函数完成：

```
extern int register_chrdev(unsigned int, const char *,
                          const struct file_operations *);
extern void unregister_chrdev(unsigned int, const char *);
```

`register_chrdev()` 函数的第一个 `int` 类型的参数表示字符设备的设备号；第二个参数为设备名称；第三个参数类型为 `file_operations`，表示设备所对应的操作。

字符设备的实现理念就是：构建一组文件操作，将其赋值给一个字符设备的节点。在用户空间中，对文件的操作函数可以调用驱动所实现的一组文件操作。

图 14-1 中给出了字符设备驱动实现的两种形式。字符驱动程序 A 表示直接实现一个字符设备，也就是直接实现 `file_operations` 的各个函数指针。字符驱动程序 B 表示基于某个框架定义的字符设备，这时 `file_operations` 由这个类型的字符设备框架统一实现，驱动程序所实现的是字符设备框架所定义的结构。

根据对应设备的不同，驱动程序或多或少都需要进行进一步的实现。公共实现的 `file_operations` 可以替换成具体驱动程序实现的 `file_operations`。

14.2.3 块设备的框架

块设备使用随机的、无序的方式传输数据，且数据总是具有固定大小的块，而不是字节流。为了提高数据传输效率，块设备驱动程序内部采用块缓冲技术。典型的块设备包括光盘、硬盘、软盘、MMC/SD 卡的分区、Flash 的分区等。

块驱动程序的框架如图 14-2 所示。

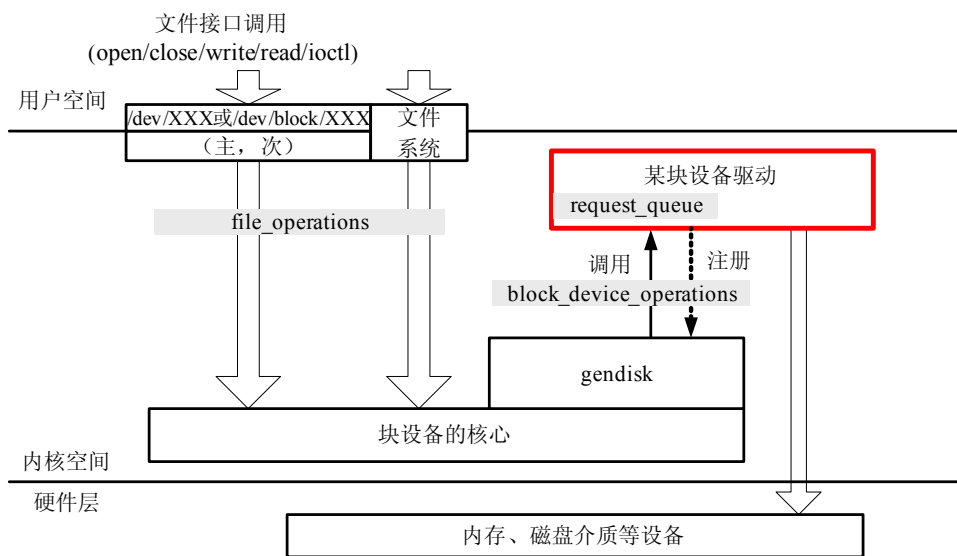


图 14-2 块驱动程序的框架

在用户空间通过块设备文件来访问块设备，访问方法是文件方面的系统调用（或函数）。块设备对用户空间的直接接口是块设备通用的 `file_operations`，其具体实现再由每个块设备的 `block_device_operations` 结构等完成。

块设备结构在 `fs.h` 头文件中定义，如下所示：

```

struct block_device {
    dev_t      bd_dev;
    struct inode * bd_inode;      // 表示设备对应的索引节点
    struct super_block * bd_super; // 表示设备对应的超级块
    int        bd_openers;
    struct mutex bd_mutex;
    struct list_head bd_inodes;
    void *      bd_claiming;
    void *      bd_holder;
    int        bd_holders;
    // ..... 省略部分内容
    struct gendisk * bd_disk;      // 表示设备对应的通用磁盘
    // ..... 省略部分内容
};
    
```

`block_device` 用于描述一个块设备，对块设备文件的操作，将会通过它完成。其中，`bd_inode` 表示其对应的 `inode`，具有一个 `gendisk` 结构的指针，表示通用的磁盘。

`gendisk` 结构实际上是块设备的核心结构，在 `genhd.h` 头文件中定义，如下所示：

```

struct gendisk {
    int major;                // 主设备号
    int first_minor;          // 第一个次设备号
    int minors;               // 次设备的数目
    char disk_name[DISK_NAME_LEN];
    char *(*devnode)(struct gendisk *gd, mode_t *mode);
    struct disk_part_tbl *part_tbl;
    struct hd_struct part0;
    const struct block_device_operations *fops;           // 块设备的操作
    struct request_queue *queue;
    // ..... 省略部分内容
};

```

gendisk 的含义为通用磁盘，在内核中表示一个磁盘，其中包含对应块设备的主、次设备号。块设备操作则使用 **block_device_operations** 结构表示，用于处理 I/O 请求的队列用 **request_queue** 结构表示。一个块设备通常用来表示一个多分区的磁盘，其中包含的 **gendisk** 用来描述这个磁盘。

块设备的操作在 **fs.h** 头文件中定义，如下所示：

```

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);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t,
                          void **, unsigned long *);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo) (struct block_device *, struct hd_geometry *);
    struct module *owner;
};

```

在块设备的实现核心中，对用户空间的文件调用，将调用到 **file_operations**，进而调用到某一个块设备的 **block_device_operations**。

对块设备的操作通常要经过请求（request）形成队列，这些操作最后一般都归结到 **block_device_operations** 的实现中。

块设备注册和注销函数如下所示：

```

extern int register_blkdev(unsigned int, const char *);
extern void unregister_blkdev(unsigned int, const char *);

```

这两个函数在块设备的层次完成了设备号的分配，第一个参数为设备号，第二个参数为设备名称。同一个广义磁盘具有一个主设备号，磁盘中不同的分区则通常对应一个次设备号。注册只是块设备设备号的标记，对于在一个真正的块设备的实现，则需要完成 **gendisk** 和 **block_device_operations** 结构的构建。

一个基于块设备框架的块设备驱动程序常使用如下的实现方式：

- 在初始化阶段注册块设备的主设备号。
- 构建队列结构 **request_queue** 和通用磁盘 **gendisk**。
- 在 **gendisk** 结构中的主设备号要与块设备的设备号一致，并定义所对应分区的次设

备号。

- gendisk 数据结构使用了块设备操作函数集合 block_device_operations 结构和请求序列 request_queue 结构。
- 在块设备操作函数集合中实现相应块设备的功能，实现过程中需要控制请求队列。

14.2.4 字符设备和块设备的默认 file_operations 实现

Linux 内核为字符设备、块设备提供了 file_operations 的默认实现。各个具体的驱动程序，则是建立于默认的 file_operations 之上的。

fs/char_dev.c 文件的 def_chr_fops 是默认的字符设备文件的 file_operations 实现。

fs/block_dev.c 文件的 def_blk_fops 是默认的块设备文件的 file_operations 实现。

1. 字符设备的 def_chr_fops

字符设备的默认 file_operations 的逻辑比较简单：在字符设备文件被打开的时候，从 inode 中获取 file_operations，设置为进程打开文件 file 的 file_operations，还要调用其中实现的打开函数 open。

fs/char_dev.c 文件中定义的 def_chr_fops 结构如下所示：

```
const struct file_operations def_chr_fops = {
    .open = chrdev_open,
};
```

def_chr_fops 中只实现了 file_operations 中的一个 open 函数指针。

chrdev_open()函数的内容如下所示：

```
static int chrdev_open(struct inode *inode, struct file *filp)
{
    struct cdev *p;                // 字符设备在内核的内部结构
    struct cdev *new = NULL;
    int ret = 0;
    spin_lock(&cdev_lock);         // 对设备操作所加的锁：cdev_lock 是全局的
    p = inode->i_cdev;              // 得到设备节点 inode 中的设备
    if (!p) {
        struct kobject *kobj;      // 内核的对象
        int idx;
        spin_unlock(&cdev_lock);
        kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
        if (!kobj) return -ENXIO;
        new = container_of(kobj, struct cdev, kobj); // 建立新的设备
        spin_lock(&cdev_lock);
        /* we dropped the lock. */
        p = inode->i_cdev;
        if (!p) {
            inode->i_cdev = p = new;
            list_add(&inode->i_devices, &p->list);
            new = NULL;
        } else if (!cdev_get(p))
            ret = -ENXIO;
    } else if (!cdev_get(p))        // 如果 inode 中没有设备，则表示出错
        ret = -ENXIO;
    spin_unlock(&cdev_lock);
}
```

```

cdev_put(new);
if (ret) return ret;
ret = -ENXIO;
filp->f_op = fops_get(p->ops); // 从字符设备中得到它实现的 file_operations
if (!filp->f_op) goto out_cdev_put;
if (filp->f_op->open) {
    ret = filp->f_op->open(inode, filp); // 调用文件的打开函数
    if (ret) goto out_cdev_put;
}
return 0;
// ..... 省略错误处理
}

```

chrdev_open()函数中除了内核必要的结构化处理之外，主要做了两项工作。第一项工作是：从设备节点中得到 file_operations，赋值给进程打开文件 file 的 file_operations。第二项工作是：如果设备中具有 open 函数指针，则对其调用。

由此，对于一个字符设备，从文件系统上的设备节点打开时，可以根据主次设备号对应到驱动程序上，从而可以从 inode 得到驱动程序实现的 file_operations。这个 file_operations 会被设置为进程打开文件的 file_operations，并且其中实现的 open 函数也会被调用。

2. 块设备的 def_blk_fops

由于块设备的驱动程序所实现的是 block_device 等内部结构，而在用户空间所调用的是文件接口，文件接口对应的就是块设备的 file_operations；因此，块设备的 file_operations 需要完成的核心工作就是让各种文件接口去调用块设备驱动程序所实现的内容。

fs/block_dev.c 文件中定义的 def_blk_fops 结构如下所示：

```

const struct file_operations def_blk_fops = {
    .open      = blkdev_open,           // 块设备的打开函数实现
    .release   = blkdev_close,         // 块设备的释放函数实现
    .llseek    = block_llseek,         // 块设备的移动位置实现
    .read      = do_sync_read,
    .write     = do_sync_write,
    .aio_read  = generic_file_aio_read,
    .aio_write = blkdev_aio_write,     // 块设备的异步写函数实现
    .mmap      = generic_file_mmap,
    .fsync     = blkdev_fsync,         // 块设备的文件同步函数实现
    .unlocked_ioctl = block_ioctl,     // 块设备的端口控制函数实现
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_blkdev_ioctl,
#endif
    .splice_read = generic_file_splice_read,
    .splice_write = generic_file_splice_write,
};

```

块设备默认的 file_operations 中，open、release、llseek、fsync 和 ioctl 等几个函数是具有特定实现的，其他的几个函数指针的实现则设置为系统通用的实现。

块设备 open 的实现为 blkdev_open()函数，如下所示：

```

static int blkdev_open(struct inode * inode, struct file * filp)
{
    struct block_device *whole = NULL;
    struct block_device *bdev;        // 块设备的结构

```

```
int res;
filp->f_flags |= O_LARGEFILE;           // 块设备文件打开参数的设置
if (filp->f_flags & O_NDELAY)            filp->f_mode |= FMODE_NDELAY;
if (filp->f_flags & O_EXCL)              filp->f_mode |= FMODE_EXCL;
if ((filp->f_flags & O_ACCMODE) == 3)    filp->f_mode |= FMODE_WRITE_IOCTL;
bdev = bd_acquire(inode);                // 从 inode 得到 block_device
if (bdev == NULL) return -ENOMEM;
if (filp->f_mode & FMODE_EXCL) {
    whole = bd_start_claiming(bdev, filp);
// ..... 省略错误处理
}
filp->f_mapping = bdev->bd_inode->i_mapping; // 设置块设备进程打开文件的内容
res = blkdev_get(bdev, filp->f_mode);      // 得到块设备的结构
if (whole) {
    if (res == 0)
        bd_finish_claiming(bdev, whole, filp); // 联系块设备和进程打开文件
    else
        bd_abort_claiming(whole, filp);
}
return res;
}
```

块设备的打开函数要从块设备的节点中得到 **block_device**，并和进程打开的文件联系在一起。**bd_start_claiming()**和**bd_finish_claiming()**函数则从 **block_device** 进一步得到 **gendisk**，进行驱动程序的操作。

bd_acquire()函数中调用了 **bdget()**函数，如下所示：

```
struct block_device *bdget(dev_t dev)
{
    struct block_device *bdev;           // 块设备结构
    struct inode *inode;                  // 块设备所对应的索引节点
    inode = iget5_locked(blockdev_superblock, hash(dev), bdev_test, bdev_set, &dev);
    if (!inode) return NULL;
    bdev = &BDEV_I(inode)->bdev;         // 得到块设备的内核结构
    if (inode->i_state & I_NEW) {
        bdev->bd_contains = NULL;
        bdev->bd_inode = inode;           // 设置 block_device 对应的 inode
        bdev->bd_block_size = (1 << inode->i_blkbits);
        bdev->bd_part_count = 0;
        bdev->bd_invalidated = 0;
        inode->i_mode = S_IFBLK;
        inode->i_rdev = dev;
        inode->i_bdev = bdev;
        inode->i_data.a_ops = &def_blk_aops; // 设置 address_space_operations
        mapping_set_gfp_mask(&inode->i_data, GFP_USER);
        inode->i_data.backing_dev_info = &default_backing_dev_info;
        spin_lock(&bdev_lock);
        list_add(&bdev->bd_list, &all_bdevs);
        spin_unlock(&bdev_lock);
        unlock_new_inode(inode);
    }
    return bdev;
}
```

bdget()函数调用之后，将根据设备节点的 **inode** 的信息，得到设备的 **block_device**，这个内容也就是后续块设备操作的基本结构。

ioctl 的实现函数如下所示：

```
static long block_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
    struct block_device *bdev = I_BDEV(file->f_mapping->host); // 得到块设备
    fmode_t mode = file->f_mode;
    if (file->f_flags & O_NDELAY) mode |= FMODE_NDELAY;
    else mode &= ~FMODE_NDELAY;
    return blkdev_ioctl(bdev, mode, cmd, arg); // 调用块设备的 ioctl 实现
}
```

blkdev_ioctl()函数在 block/ioctl.c 文件中实现，如下所示：

```
int blkdev_ioctl(struct block_device *bdev, fmode_t mode, unsigned cmd,
                unsigned long arg)
{
    struct gendisk *disk = bdev->bd_disk; // 得到通用的磁盘
    struct backing_dev_info *bdi;
    loff_t size;
    int ret, n;
    switch(cmd) {
        // ..... 省略部分内容
        case BLKROSET:
            ret = __blkdev_driver_ioctl(bdev, mode, cmd, arg);
            if (ret != -EINVAL && ret != -ENOTTY) return ret;
            if (!capable(CAP_SYS_ADMIN)) return -EACCES;
            if (get_user(n, (int __user *) (arg))) return -EFAULT;
            lock_kernel();
            set_device_ro(bdev, n);
            unlock_kernel();
            return 0;
        case BLKROGET:
            return put_int(arg, bdev_read_only(bdev) != 0);
        case BLKBSZGET: // 获得块设备块的大小
            return put_int(arg, block_size(bdev));
        case BLKSSZGET: // 获得块设备逻辑块的大小
            return put_int(arg, bdev_logical_block_size(bdev));
        case BLKPBSZGET: // 获得块设备物理块的大小
            return put_uint(arg, bdev_physical_block_size(bdev));
        case BLKIOMIN:
            return put_uint(arg, bdev_io_min(bdev));
        case BLKIOOPT:
            return put_uint(arg, bdev_io_opt(bdev));
        // ..... 省略部分内容
        default:
            ret = __blkdev_driver_ioctl(bdev, mode, cmd, arg);
            return ret;
    }
}
EXPORT_SYMBOL_GPL(blkdev_ioctl);
```

blkdev_ioctl()函数作为块设备 ioctl 的通用实现，先要实现块设备预定的 ioctl 命令，然后调用__blkdev_driver_ioctl()函数。

fs.h 中定义了块设备通用的 ioctl 命令号，如下所示：

```
#define BLKROGET _IO(0x12,94) /* get read-only status (0 = read_write) */
#define BLKRRPART _IO(0x12,95) /* re-read partition table */
#define BLKGETSIZE _IO(0x12,96) /* return device size /512 (long *arg) */
#define BLKFLSBUF _IO(0x12,97) /* flush buffer cache */
```

```
#define BLKRASET _IO(0x12,98) /* set read ahead for block device */
#define BLKRASET _IO(0x12,99) /* get current read ahead setting */
#define BLKFRASET _IO(0x12,100) /* set filesystem (mm/filemap.c) read-ahead */
#define BLKFRASET _IO(0x12,101) /* get filesystem (mm/filemap.c) read-ahead */
#define BLKSECTSET _IO(0x12,102) /* set max sectors per request (ll_rw_blk.c) */
#define BLKSECTGET _IO(0x12,103) /* get max sectors per request (ll_rw_blk.c) */
#define BLKSSZGET _IO(0x12,104) /* get block device sector size */
```

这些块设备的 `ioctl` 就是通过块设备核心，间接调用块设备驱动程序的实现得到的。

14.3 网络协议和网络设备的框架

网络系统是 Linux 中的一个子系统，其中包括了通用部分和每种协议、每种设备的特别实现部分。简单来说，网络系统分为下面几个层次：

- UNIX 统一的接口层（套接字）。
- 网络协议接口（包括通用和具体协议）。
- 网络设备层。

在 Linux 的网络系统中，核心部分可以使用各种不同的协议，每种协议可以单独实现。核心部分对外部的接口就是套接字（socket）。其中，最主要的一个协议就是基于 IP 的 Internet 协议。

网络设备（Net Device）是 Internet 协议中的一个具体设备的实现。网络设备是使用特殊的网络作为接口的设备，具有专门的结构负责数据的收发。与字符设备和块设备不同，在用户空间中并没有表示网络设备的设备文件。

网络系统和网络设备驱动程序的框架如图 14-3 所示。

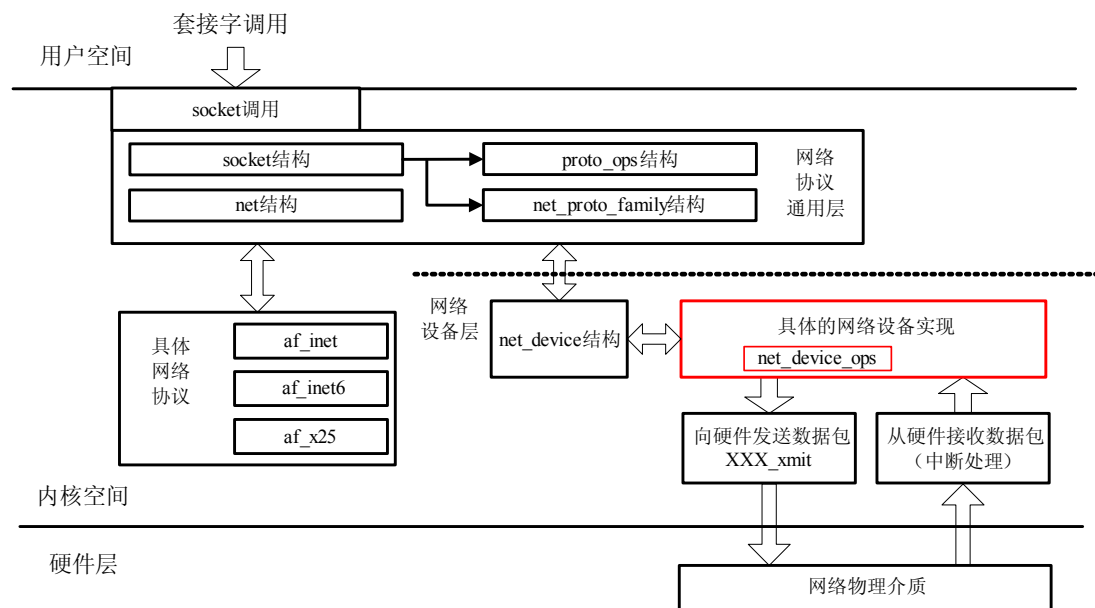


图 14-3 网络系统和网络设备驱动程序的框架

14.3.1 网络系统的核心

Linux 网络系统的核心部分主要包括了 `net.h` 和 `socket.h` 等头文件。

`net.h` 头文件定义了 `socket` 和 `net_proto_family` 结构，如下所示：

```
struct socket {
    socket_state      state;
    kmemcheck_bitfield_begin(type);
    short             type;
    kmemcheck_bitfield_end(type);
    unsigned long     flags;
    struct socket_wq   *wq;
    struct file        *file;           // 套接字对应的文件（一个被进程打开的文件）
    struct sock        *sk;
    const struct proto_ops *ops;       // 表示网络协议的操作
};

struct net_proto_family {
    int family;
    int (*create)(struct net *net, struct socket *sock, int protocol, int kern);
    struct module *owner;
};
```

`socket` 结构表示内核中的套接字，其中包含了一个 `proto_ops` 结构表示某种网络协议的操作。`net_proto_family` 结构表示网络协议族，可以在不同的协议中实现。

`proto_ops` 结构表示网络协议的操作，如下所示：

```
struct proto_ops {
    int family;
    struct module *owner;
    int (*release) (struct socket *sock);
    int (*bind) (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
    int (*connect) (struct socket *sock, struct sockaddr *vaddr,
        int sockaddr_len, int flags);
    int (*socketpair)(struct socket *sock1, struct socket *sock2);
    int (*accept) (struct socket *sock, struct socket *newssock, int flags);
    int (*getname) (struct socket *sock, struct sockaddr *addr,
        int *sockaddr_len, int peer);
    unsigned int (*poll) (struct file *file, struct socket *sock,
        struct poll_table_struct *wait);
    int (*ioctl) (struct socket *sock, unsigned int cmd, unsigned long arg);
    // ..... 省略部分内容
};
```

`proto_ops` 结构需要由具体的网络协议去实现，提供针对某种套接字的实现。

`socket.h` 头文件定义了各种协议族（Address Families），如下所示：

```
#define AF_UNSPEC 0
#define AF_UNIX    1    /* Unix domain sockets */
#define AF_LOCAL   1    /* POSIX name for AF_UNIX */
#define AF_INET    2    /* Internet IP Protocol */
#define AF_AX25    3    /* Amateur Radio AX.25 */
#define AF_IPX     4    /* Novell IPX */
#define AF_APPLETALK 5    /* AppleTalk DDP */
#define AF_NETROM   6    /* Amateur Radio NET/ROM */
#define AF_BRIDGE   7    /* Multiprotocol bridge */
#define AF_ATMPVC   8    /* ATM PVCs */
#define AF_X25     9    /* Reserved for X.25 project */
```

```
#define AF_INET6      10 /* IP version 6 */
#define AF_CAN        29 /* Controller Area Network */
#define AF_TIPC       30 /* TIPC sockets */
#define AF_BLUETOOTH  31 /* Bluetooth sockets */
```

net_proto_family 结构的第一成员 family 就是来自这些协议族的定义。例如：AF_INET 表示的就是 Internet 协议，AF_CAN 表示 CAN 总线的协议，AF_BLUETOOTH 表示蓝牙协议，AF_UNIX 和 AF_INET 表示本地套接字。各种 PF_开头的宏表示 Protocol Families，与 AF_开头的宏一一对应，含义相同。

socket.h 头文件中以 socket_为前缀的函数完成套接字相关的操作，如下所示：

```
enum {
    SOCK_WAKE_IO,
    SOCK_WAKE_WAITD,
    SOCK_WAKE_SPACE,
    SOCK_WAKE_URG,
};
int sock_register(const struct net_proto_family *fam);
void sock_unregister(int family);
int sock_create(int family, int type, int proto, struct socket **res);
void sock_release(struct socket *sock);
int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t len);
int sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t size, int flags);
int sock_map_fd(struct socket *sock, int flags);
```

对于某一种网络协议的实现者，需要使用 sock_register()函数将 net_proto_family 的实现注册。sock_release()函数用于实现释放某个协议。sock_sendmsg()和 sock_recvmsg()函数用于发送和接收。

提示：net 目录的各个子目录就是各种网络协议的实现，各个 af_*开头的文件当中就会实现 net_proto_family 结构，并进行注册。

以 kernel_为前缀的函数用于内核中网络相关的操作，各个函数如下所示：

```
int kernel_sendmsg(struct socket *sock, struct msghdr *msg,
                  struct kvec *vec, size_t num, size_t len);
int kernel_recvmsg(struct socket *sock, struct msghdr *msg,
                  struct kvec *vec, size_t num, size_t len, int flags);
int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen);
int kernel_listen(struct socket *sock, int backlog);
int kernel_accept(struct socket *sock, struct socket **newsock, int flags);
int kernel_connect(struct socket *sock, struct sockaddr *addr,
                  int addrlen, int flags);
int kernel_getsockname(struct socket *sock, struct sockaddr *addr, int *addrlen);
int kernel_getpeername(struct socket *sock, struct sockaddr *addr, int *addrlen);
int kernel_getsockopt(struct socket *sock, int level, int optname,
                  char *optval, int *optlen);
int kernel_setsockopt(struct socket *sock, int level, int optname,
                  char *optval, unsigned int optlen);
int kernel_sendpage(struct socket *sock, struct page *page, int offset,
                  size_t size, int flags);
int kernel_sock_ioctl(struct socket *sock, int cmd, unsigned long arg);
int kernel_sock_shutdown(struct socket *sock, enum sock_shutdown_cmd how);
```

这些函数可以对应到用户空间的各个 socket 操作中,例如 bind、listen、accept 和 connect 等,还有数据的收发。kernel_sendmsg()函数通过 sock_sendmsg()函数实现,kernel_recvmsg()函数通过 sock_recvmsg()函数实现。

14.3.2 网络协议的实现

网络协议的实现是 Linux 内核中网络模块之下的各个子模块。网络设备实现的就是 net_proto_family 和 proto_ops 结构中的函数指针。

具体网络协议的实现在内核代码 net 目录中的各个子目录中。几个典型的网络协议的实现如下所示。

- ax25/af_ax25.c: A25 网络协议的实现,协议族为 PF_AX25。
- can/af_can.c: CAN 网络协议的实现,协议族为 PF_CAN。
- bluetooth/sco.c: BlueTooth SCO 网络协议的实现,协议族为 PF_BLUETOOTH。
- ipv4/af_inet.c: IPv4 网络协议的实现,协议族为 PF_INET。
- ipv6/af_inet6.c: IPv6 网络协议的实现,协议族为 PF_INET6。
- ipx/af_ipx.c: IPX/SPX 网络协议的实现,协议族为 PF_IPX。
- netrom/af_netrom.c: NETROM 网络协议的实现,协议族为 PF_NETROM。
- rose/af_rose: Rose 网络协议的实现,协议族为 PF_ROSE。
- unix/af_unix.c: UNIX 文件套接字的实现,协议族为 PF_UNIX。

在各种网络协议中,CAN 总线协议是比较简单的一种总线协议。CAN 总线的全名是 Controller Area Network (控制器局域网)。CAN 是一种行总线,它提供高安全等级及有效率的实时控制,更具备了调试和优先权判别的机制,网络信息的传输变得更为可靠而有效率。CAN 总线多用于汽车电子。

net/can 目录中的 af_can.c 文件是核心实现;bcm.c 是 CAN 的广播管理(Broadcast Manager)实现;raw.c 是原始(Raw)实现,也称为“raw access with CAN-ID filtering”。

include/linux/can/core.h 头文件定义了 CAN 协议的原型 can_proto:

```
struct can_proto {
    int         type;
    int         protocol;    // 表示协议族
    struct proto_ops *ops;    // 表示协议的操作
    struct proto *prot;
};
```

include/linux/目录中的 can.h 头文件定义了 CAN 总线协议内部的几个定义:

```
#define CAN_RAW        1 /* RAW sockets */
#define CAN_BCM        2 /* Broadcast Manager */
#define CAN_TP16       3 /* VAG Transport Protocol v1.6 */
#define CAN_TP20       4 /* VAG Transport Protocol v2.0 */
#define CAN_MCNET      5 /* Bosch MCNet */
#define CAN_ISOTP      6 /* ISO 15765-2 Transport Protocol */
#define CAN_NPROTO     7
```

其中的基本定义如下所示:

```
static struct can_proto *proto_tab[CAN_NPROTO] __read_mostly;
static DEFINE_SPINLOCK(proto_tab_lock);
```

proto_tab 就是 CAN 协议的数组，类型为 can_proto。

af_can.c 中定义的 can_init() 函数如下所示：

```
static __init int can_init(void)
{
    printk(banner);
    memset(&can_rx_alldev_list, 0, sizeof(can_rx_alldev_list));
    rcv_cache = kmem_cache_create("can_receiver", sizeof(struct receiver),
                                0, 0, NULL);
    // ..... 省略部分内容
    can_init_proc();                // 初始化 CAN 实现的 proc 文件系统内容
    sock_register(&can_family_ops); // 注册 CAN 总线操作
    register_netdevice_notifier(&can_netdev_notifier);
    dev_add_pack(&can_packet);      // 增加一个 packet_type
    return 0;
}
```

模块的初始化过程中，调用 sock_register() 将 CAN 协议进行注册。

af_can.c 中定义的 net_proto_family 结构如下所示：

```
static const struct net_proto_family can_family_ops = {
    .family = PF_CAN,           // 表示协议族
    .create = can_create,       // 协议的创建函数
    .owner = THIS_MODULE,
};
```

所有网络协议的实现都需要定义一个 net_proto_family 结构。can_family_ops 结构当中协议的创建实现（create）被设置为了 can_create() 函数。

can_create() 函数如下所示：

```
static int can_create(struct net *net, struct socket *sock,
                    int protocol, int kern)
{
    struct sock *sk;
    struct can_proto *cp;
    int err = 0;
    sock->state = SS_UNCONNECTED;
    if (protocol < 0 || protocol >= CAN_NPROTO) return -EINVAL;
    if (!net_eq(net, &init_net)) return -EAFNOSUPPORT;
    // ..... 省略部分内容
    spin_lock(&proto_tab_lock);
    cp = proto_tab[protocol];
    if (cp && !try_module_get(cp->prot->owner)) cp = NULL;
    spin_unlock(&proto_tab_lock);
    if (!cp) return -EPROTONOSUPPORT;
    // ..... 省略错误处理
    sock->ops = cp->ops; // 设置套接字的操作为 CAN 网络的操作
    sk = sk_alloc(net, PF_CAN, GFP_KERNEL, cp->prot);
    // ..... 省略错误处理
    sock_init_data(sock, sk);
    sk->sk_destruct = can_sock_destruct;
    if (sk->sk_prot->init) err = sk->sk_prot->init(sk);
    // ..... 省略错误处理
    return err;
}
```

can_create()函数执行之后,表示建立了 CAN 总线的基本结构,主要的工作是要将表示套接字的标识协议操作 proto_ops 结构设置为 CAN 网络所实现的内容。此处的 cp->ops 就是 can_proto 结构中 proto_ops 类型的指针。其中的 ioctl 函数指针由 can_ioctl()函数实现。

14.3.3 网络设备的框架

在 Linux 系统中,Internet 协议是网络系统众多协议当中的一种。网络设备(Net Device)表示 Internet 协议所使用的设备。

头文件 netdevice.h 层中包括具体的网络设备接口,由具体网络设备的驱动程序去实现。网络驱动程序的主要实现方式就是如此。

net_device 结构的定义如下所示:

```
struct net_device {
    char            name[IFNAMSIZ];           // 表示网络设备的名称
    struct pm_qos_request_list *pm_qos_req;
    struct hlist_node name_hlist;
    char            *ifalias;
    // ..... 省略部分内容
    unsigned long    features;
    int              ifindex;
    int              iflink;
    struct net_device_stats stats;           // 用于保存网络设备的状态
    // ..... 省略部分内容
    const struct net_device_ops *netdev_ops; // 表示网络设备的操作
    const struct ethtool_ops *ethtool_ops;
    // ..... 省略部分内容
};
```

网络设备相关的几个函数如下所示:

```
extern struct net_device *alloc_netdev_mq(int sizeof_priv, const char *name,
                                           void (*setup)(struct net_device *),
                                           unsigned int queue_count);
#define alloc_netdev(sizeof_priv, name, setup) \
    alloc_netdev_mq(sizeof_priv, name, setup, 1)
extern int register_netdev(struct net_device *dev);
extern void unregister_netdev(struct net_device *dev);
```

alloc_netdev_mq()函数的参数 setup 是函数指针,用于让一个网络驱动实现在建立网络设备的过程,初始化 net_device 结构中的各个成员。register_netdev()和 unregister_netdev()函数完成设备的注册和注销。

net_device_ops 的定义如下所示:

```
struct net_device_ops {
    int (*ndo_init)(struct net_device *dev);
    void (*ndo_uninit)(struct net_device *dev);
    int (*ndo_open)(struct net_device *dev);
    int (*ndo_stop)(struct net_device *dev);
    netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
    u16 (*ndo_select_queue)(struct net_device *dev, struct sk_buff *skb);
    void (*ndo_change_rx_flags)(struct net_device *dev, int flags);
    void (*ndo_set_rx_mode)(struct net_device *dev);
    void (*ndo_set_multicast_list)(struct net_device *dev);
};
```

```
int      (*ndo_set_mac_address)(struct net_device *dev, void *addr);
int      (*ndo_validate_addr)(struct net_device *dev);
int      (*ndo_do_ioctl)(struct net_device *dev,
int      (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
int      (*ndo_change_mtu)(struct net_device *dev, int new_mtu);
int      (*ndo_neigh_setup)(struct net_device *dev, struct neigh_parms *);
void      (*ndo_tx_timeout)(struct net_device *dev);
struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
// ..... 省略部分内容
};
```

各个函数指针以 `ndo` 为开头，其全称就是 Net Device Operation。对于一个网络设备的实现者，可以重新实现 `net_device_ops` 结构。对不需要实现的函数指针，需要设置为 `NULL`。其中最重要的是 `ndo_start_xmit` 函数指针，用于网络数据的发送。参数类型为 `sk_buff`。网络操作结构在不同版本中的实现略有不同，核心内容都是发送和接收。

提示：`net_device_ops` 结构是较后期的 Linux 版本中（2.6.31）才引入的，在从前的 Linux 内核版本中，`net_device` 结构中有一个 `hard_start_xmit` 函数指针用于网络发送。

网络设备中与队列相关的发送函数如下所示：

```
static inline void netif_start_queue(struct net_device *dev){}
static inline void netif_tx_start_all_queues(struct net_device *dev){}
static inline void netif_wake_queue(struct net_device *dev){}
static inline void netif_tx_stop_queue(struct netdev_queue *dev_queue){}
```

网络设备的接收函数的几个相关定义如下所示：

```
extern int      netif_rx(struct sk_buff *skb);
extern int      netif_rx_ni(struct sk_buff *skb);
extern int      netif_receive_skb(struct sk_buff *skb);
extern gro_result_t dev_gro_receive(struct napi_struct *napi,
                                   struct sk_buff *skb);
extern gro_result_t napi_skb_finish(gro_result_t ret, struct sk_buff *skb);
extern gro_result_t napi_gro_receive(struct napi_struct *napi,
                                   struct sk_buff *skb);
extern void      napi_reuse_skb(struct napi_struct *napi, struct sk_buff *skb);
```

`netif_rx()` 函数通常在网络的数据包接收时，填充统一的 `sk_buff` 结构。`sk_buff` 的含义就是 Socket Buffer。

14.4 proc 文件系统的框架

`proc` 文件系统对应用户空间/`proc`/目录中的各个内容。其中的各个文件可以用于查看有关硬件、进程的状态。在驱动程序方面，`proc` 文件系统各个文件可以作为内核空间到用户空间的接口，它们既可以作为辅助的信息使用，也可以作为独立的驱动程序使用。

14.4.1 proc 文件系统的编程接口

`proc` 文件系统的节点常常提供的是读、写功能，分别用于显示信息和控制，也实现文件的其他操作。

头文件 `proc_fs.h` 定义创建 `proc` 文件系统的函数：

```
struct proc_dir_entry *create_proc_entry(const char *name,
                                         mode_t mode, struct proc_dir_entry *parent);
struct proc_dir_entry *proc_create_data(const char *name, mode_t mode,
                                         struct proc_dir_entry *parent,
                                         const struct file_operations *proc_fops, void *data);
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

`proc_dir_entry` 结构用于表示 `proc` 文件系统中的节点，可以表示目录和常规文件。`create_proc_entry()` 和 `proc_create_data()` 函数用于创建节点，`remove_proc_entry()` 函数用于删除节点。

`proc` 文件系统节点当中表示读、写操作的函数类型的定义如下所示：

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                          int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                           unsigned long count, void *data);
```

`proc_dir_entry` 结构如下所示：

```
struct proc_dir_entry {
    unsigned int low_ino;
    unsigned short namelen;
    const char *name;                // 表示这个节点的名字
    mode_t mode;
    nlink_t nlink;
    uid_t uid;                       // 文件的用户 id 和组 id
    gid_t gid;
    loff_t size;
    const struct inode_operations *proc_iops; // 表示索引节点的操作
    const struct file_operations *proc_fops;  // 表示 proc 文件系统的文件操作
    struct module *owner;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;          // 表示读的函数类型
    write_proc_t *write_proc;        // 表示写的函数类型
    atomic_t count;
    int pde_users;
    spinlock_t pde_unload_lock;
    struct completion *pde_unload_completion;
    struct list_head pde_openers;
};
```

`proc_dir_entry` 结构当中的 `mode`、`uid`、`gid` 几个结构对应文件系统的信息。`read_proc` 和 `write_proc` 函数指针用于定义一个 `proc` 文件系统中文件的读和写。如果需要进一步定义文件操作，可以实现 `file_operations` 结构。

在实际应用过程中，`proc` 文件系统的节点中常常使用的是读、写操作。可以支持对于文件的完整操作，但是并不常用。

提示：使用 `proc` 文件系统完全可以实现类似字符设备的驱动。但在通常的应用中，`proc` 文件系统中的文件通常只用于提供驱动程序辅助的功能。

14.4.2 proc 文件系统的实现

内核源代码 fs/proc 目录中的内容是核心的 proc 文件系统的实现,其中实现的内容主要包括 inode.c、root.c、base.c、generic.c、array.c 等文件,其他几个文件都是针对某个 proc 文件系统中节点的单独实现。

cmdline.c 文件实现了 proc/cmdline 节点,其实现如下所示。

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
static int cmdline_proc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "%s\n", saved_command_line); // 读取系统的命令行 saved_command_line
    return 0;
}
static int cmdline_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, cmdline_proc_show, NULL);
}
static const struct file_operations cmdline_proc_fops = {
    .open      = cmdline_proc_open,
    .read      = seq_read,          // 读取的文件操作
    .llseek    = seq_lseek,        // 移动位置操作
    .release   = single_release,
};
static int __init proc_cmdline_init(void)
{
    proc_create("cmdline", 0, NULL, &cmdline_proc_fops); // 设置一个节点
    return 0;
}
module_init(proc_cmdline_init);
```

cmdline 文件实现了 file_operations, seq_read()和 seq_lseek()两个函数是顺序文件的操作函数。在文件打开的过程中,调用 single_open()函数,并将 cmdline_proc_show()函数设置为其中的回调函数。saved_command_line 就是 char*类型的字符串,用于保存系统的命令行参数,cmdline_proc_show()函数将其取出拼凑成字符串,作为读的结果返回。

14.5 sys 文件系统的框架

sys 文件系统作为 Linux 内核支持的基于内存的虚拟文件系统,它的功能与 proc 文件系统类似。在驱动程序方面,sys 文件系统主要提供设备之间的连接关系,也可以作为特殊驱动程序对用户空间的接口。

14.5.1 sys 文件系统的编程接口

sys 文件系统文件通常有读和写两个接口,对于简单驱动的信息获取和控制已经足够;但是不能支持复杂的操作,例如:结构复杂的参数传递、大规模数据块操作等。

头文件 sysfs.h 与 sys 文件系统的使用相关,几个定义的内容如下所示:

```

#define __ATTR(_name, _mode, _show, _store) { \
    .attr = { .name = __stringify(_name), .mode = _mode }, \
    .show  = _show, \
    .store = _store, \
}
#define __ATTR_RO(_name) { \
    .attr = { .name = __stringify(_name), .mode = 0444 }, \
    .show  = _name##_show, \
}
#define __ATTR_NULL { .attr = { .name = NULL } }
#define attr_name(_attr) (_attr).attr.name
int __must_check sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);

```

sysfs_create_file()和 sysfs_remove_file()函数分别用于创建和删除 sys 文件系统中的文件节点。__ATTR 和 __ATTR_RO 用于属性（这里的属性就是 sys 文件系统中的文件）的定义，读写属性包括 show 和 store 两个操作，读属性只有 show 一个操作。

头文件 device.h 中的相关宏也和 sys 文件系统的操作相关。属性的定义如下所示：

```

#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
struct attribute {
    const char      *name;
    struct module    *owner;
    mode_t          mode;
    // ..... 省略部分内容
};
struct bin_attribute {
    struct attribute attr;
    size_t          size;
    void            *private;
    ssize_t (*read)(struct kobject *, struct bin_attribute *, char *, loff_t, size_t);
    ssize_t (*write)(struct kobject *, struct bin_attribute *, char *, loff_t, size_t);
    int (*mmap)(struct kobject *, struct bin_attribute *attr,
                struct vm_area_struct *vma);
};

```

bin_attribute 结构中包含 read、write 和 mmap 这 3 个函数指针，用以实现读、写和文件的映射。

表示 sys 文件系统操作的 sysfs_ops 结构如下所示：

```

struct sysfs_ops {
    ssize_t (*show)(struct kobject *, struct attribute *, char *);
    ssize_t (*store)(struct kobject *,
                    struct attribute *, const char *, size_t);
};

```

在一般的驱动程序代码中，通常只需要实现 show_XXX 和 store_XXX 形式的两个函数，将它们包含到一个 device_attribute 结构的数组中。

14.5.2 sys 文件系统的实现

内核源代码 fs/sys 目录中的内容是 sys 文件系统的实现，核心的部分是 inode.c、file.c、dir.c、symlink.c 几个函数。

一个典型的 sys 文件系统中的实现如下所示：

```
static struct device_attribute XXXX_attrs[] = {
    __ATTR(XXX, S_IRWXUGO, show_XXX, store_XXX),
    // ..... 省略: 其他__ATTR 的定义
};
static ssize_t show_XXX(struct device *dev,
    struct device_attribute *attr, char *buf)
{
    // ..... 省略实现的内容: 读操作的实现
}
static ssize_t store_XXX(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t count)
{
    // ..... 省略实现的内容: 写操作的实现
}
```

表示读的是 `show_XXX` 和表示写的是 `store_XXX`，然后对数组进行循环，使用 `device_create_file()` 函数进行注册。

sys 文件系统通常只有读、写两个操作，在用户空间中可以方便地访问这种文件，例如可以在命令行使用 `cat` 和 `echo` 等命令行工具访问。

提示：在目前的 Linux 内核中，一些只需要实现小数据量读、写简单操作的驱动程序，通常利用 sys 文件系统作为到用户空间的接口。

第 15 章

Linux 驱动的要害

15.1 驱动程序的核心实现

Linux 驱动程序一般是要构建预定义的数据结构，并将其注册，数据结构中主要的实现点是函数指针，用于完成所需要实现的动作，在特定情况下被内核或用户空间调用。

驱动程序中包括核心实现部分和辅助信息部分。例如：对于一个字符设备的驱动程序，构建 `file_operations` 结构，实现其中的函数指针，属于核心部分；而设备管理属于辅助部分。

Linux 驱动程序的本质是统一接口的特定实现。驱动程序所使用的数据结构注册的模式是内核中面向对象编程的方式。除了基本内容的填充之外，还可以实现类似继承的特性，通常有如下方式。

- 私有函数指针

内核的数据结构中常有形如 `void *private_data` 的成员，此种成员可由驱动程序自行定义和使用。内核只是替具体驱动保存此指针，而不会对指针指向的数据进行操作。

- 数据结构的包含

构建一个驱动私有的数据结构，以内核预定义数据结构为首成员，此形式表示了纯 C 语言中的继承。在使用预定义结构的指针时，传入私有结构。

一个简单的自定义字符设备驱动程序的实现如下所示：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/delay.h>
#define VIRTUAL_CHAR_MAJOR          240
#define VIRTUAL_CHAR_DEVICE_NAME    "VirtualCharDevice"
static int virtual_char_open(struct inode *inode, struct file *file)
{
    printk(KERN_DEBUG"++++++++++++++++++++\n");
    printk(KERN_DEBUG"virtual_char:device open\n");
}
```

```

    printk(KERN_DEBUG"pid:%d,comm:%s\n",current->pid,current->comm);
    printk("device <%d,%d>\n",MAJOR(inode->i_rdev),MINOR(inode->i_rdev));
    (file->private_data) = (void *)0x76543210;    // 设置私有的数据结构
    return 0;
}
static ssize_t virtual_char_read(struct file *file, char *buf, size_t count,
                                loff_t *offset)
{
    printk(KERN_DEBUG"virtual_char:device read:%d\n",count);
    if(count >= sizeof(unsigned int)){
        if(copy_to_user((void __user *)buf,
            (void *)(&file->private_data),sizeof(unsigned int))) // 处理私有的数据结构
            return -EFAULT;
    }
    return count;    // 表示读取的数目就是用户空间设置的数目（实际上只有 4 个字节）
}
static ssize_t virtual_char_write(struct file *file, const char *buf, size_t count,
                                loff_t *offset)
{
    printk(KERN_DEBUG"virtual_char:device write:%d\n",count);
    return count;    // 表示写入的数目就是用户空间设置的数目
}
static int virtual_char_ioctl(struct inode *inode, struct file *file,
                              unsigned int cmd, unsigned long arg)
{
    char argk[4];
    argk[0] = 0;   argk[1] = 1;   argk[2] = 2;   argk[3] = 3;
    printk(KERN_DEBUG"virtual_char:device ioctl:%x\n",cmd);
    switch(cmd) {
        case 0:    // 命令 0 的处理：输入用户空间传入的参数
            printk(KERN_DEBUG"ctl NO.0\n");
            if(copy_from_user(argk,(void __user *)arg,4))    return -EFAULT;
            printk("arg:%x,%x,%x,%x\n",argk[0],argk[1],argk[2],argk[3]);
            break;
        case 1:    // 命令 1 的处理：将参数输出给用户空间传入的参数
            printk(KERN_DEBUG"ctl NO.1\n");
            if(copy_to_user((void __user *)arg,argk,4))    return -EFAULT;
            break;
        default:
            break;
    }
    return 0;
}
static loff_t virtual_char_llseek(struct file *file,loff_t  offset,int whence)
{
    printk(KERN_DEBUG"virtual_char:device
    llseek: offset:%x whence:%x\n",(unsigned int)offset,whence);
    return 0;    // 表示本驱动读写的位置只能是 0
}
static int virtual_char_close(struct inode *inode, struct file *file)
{
    printk(KERN_DEBUG"virtual_char:device close\n");
    printk(KERN_DEBUG"-----\n");
    return 0;
}
static struct file_operations virtual_char_fops = {
    .llseek = virtual_char_llseek,
    .read   = virtual_char_read,
    .write  = virtual_char_write,

```

```

        .ioctl = virtual_char_ioctl,
        .open  = virtual_char_open,
        .release = virtual_char_close,
    };
static int virtual_char_init(void)
{
    int res;
    printk(KERN_DEBUG"virtual_char register\n");
    res = register_chrdev(VIRTUAL_CHAR_MAJOR,      // 注册字符设备驱动
        VIRTUAL_CHAR_DEVICE_NAME, &virtual_char_fops);
    if (res < 0) {
        printk(KERN_DEBUG"virtual_char register fails\n");
        return res;
    }
    return 0;
}
static void virtual_char_cleanup(void)
{
    printk(KERN_DEBUG"virtual_char unregister\n");
    unregister_chrdev(VIRTUAL_CHAR_MAJOR, VIRTUAL_CHAR_DEVICE_NAME);
    return;
}
module_init(virtual_char_init);
module_exit(virtual_char_cleanup);
MODULE_LICENSE("GPL");

```

将源代码经过编译，生成 `virtual_char.ko` 文件，这是 Linux 的模块文件，它可以动态被插入系统。在用户空间使用 `insmod`、`rmmod`、`lsmod` 对模块进行操作，如下所示：

```

$ insmod virtual_char.ko
$ rmmod virtual_char

```

使用 `mknod` 手动建立字符设备的设备节点：

```

$ mknod /dev/VirtualCharDevice c 240 0
$ ls /dev/VirtualCharDevice -l
crw-r--r-- 1 root root 240, 0 04-25 06:46 /dev/VirtualCharDevice

```

提示：在实际的 Linux 系统中，用户空间的设备节点通常使用 `udev` 等机制动态建立。驱动本身只需要一个设备文件，动态建立和手动建立文件对后续驱动的操作没有区别。

驱动的设备节点建立之后，就可以通过设备节点文件来进行操作了。驱动程序的使用，只和设备类型以及主、次设备号有关，与驱动程序设备节点的位置无关。

在用户空间中，操作设备节点文件的过程如下所示：

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#define DEF_FILE_NAME "/dev/virtualchar"
int main(int argc, char* argv[])
{
    int fd, size, i;
    char readbuf[8];
    char writebuf[8] = "writebuf";

```

```

char ioarg[4];
char* dev_file;
if(1 == argc){           // 如果命令行有第一个参数，就用这个参数为操作的对象
    dev_file = DEF_FILE_NAME;
}else{
    dev_file = argv[1];
}
printf("<<<< testfile name:%s >>>>\n",dev_file);
// 测试写的操作
printf("====write test====\n");
fd=open(dev_file,O_WRONLY);
size = write(fd,writebuf,sizeof(writebuf));    // 写入一块空内存
close(fd);
printf("write size:%d\n",size);
// 测试读的操作
printf("====read test====\n");
fd=open(dev_file,O_RDONLY);
size = read(fd,readbuf,sizeof(readbuf));        // 读取驱动中的数据
close(fd);
printf("read size:%d\n",size);
for(i=0;i<size;i++){
    printf("readbuf[%d]:%x\n",i,(unsigned char)readbuf[i]);
}
close(fd);
// 测试 ioctl 的操作
printf("====ioctl test====\n");
fd=open(dev_file,O_RDWR);
ioarg[0] = 0xf0; ioarg[1] = 0xf1; ioarg[2] = 0xf2; ioarg[3] = 0xf3;
printf("ioctl test 0\n");
ioctl(fd,0,ioarg);    // 0 号的 ioctl, 向驱动输入内容
printf("ioctl test 1\n");
ioctl(fd,1,ioarg);    // 1 号的 ioctl, 从驱动中获取内容
printf("arg:%x,%x,%x,%x\n",ioarg[0],ioarg[1],ioarg[2],ioarg[3]);
close(fd);
return 0;
}

```

本程序经过编译后执行，运行的结果如下所示：

```

# ./test /dev/VirtualCharDevice
<<<< testfile name:/dev/VirtualCharDevice >>>>
====write test====
write size:8
====read test====
read size:8
readbuf[0]:10
readbuf[1]:32
readbuf[2]:54
readbuf[3]:76
readbuf[4]:c9
readbuf[5]:87
readbuf[6]:4
readbuf[7]:8
====ioctl test====
ioctl test 0
ioctl test 1
arg:0,1,2,3

```

从输出结果可见：驱动显示写入的 8 个字节已经成功写入；读取的 8 个字节就是驱动中私有的数据；ioctl 的 0 号命令操作进行输入，1 号命令操作获取了内核中 4 个字节的数

据（从参数中返回）。

本驱动是一个与具体功能无关的驱动，没有直接执行操作。在驱动的运行过程中，还可以通过 `dmesg` 等工具看到驱动输出的信息。

15.2 设备、驱动和资源

Linux 2.6 之后的系统中，驱动程序的实现需要和系统资源结合起来，并且可以让同一个驱动在不同系统有差异地工作。驱动模型（`driver-model`）的几个相关概念如下所示。

- 驱动（`driver`）：每种硬件驱动程序的单独实现，一般由单独的源文件实现，其中可以使用资源。
- 设备（`device`）：在板级实现中定义，包含设备所对应的信息。
- 总线（`bus`）：总线是设备所处的连接树。
- 资源（`resource`）：包括内存、端口、中断和 DMA 等资源。

在内核的头文件 `device.h` 中，定义了以下几个数据结构。

- `bus_attribute`：总线的属性。
- `bus_type`：总线类型。
- `driver_attribute`：设备驱动的属性，属于某总线。
- `device_driver`：设备驱动。
- `device_type`：设备类型。
- `device_attribute`：设备的属性。
- `device`：设备，有自己的类型，这些设备都在某个总线上。

总线用于描述内核的连接方式，不一定与硬件对应。除了 `pic`、`usb`、`i2c` 等实际的总线之外，`platform` 也是总线的一种。

提示：在嵌入式系统中，大部分设备属于 `platform` 设备。

头文件 `platform_device.h` 中定义了平台设备（`struct platform_device`）和平台驱动（`struct platform_driver`），这两个结构分别是设备（`struct device`）和驱动（`struct driver`）两个结构的扩展。前者当中定义设备本身及其使用的资源，后者当中定义驱动探测、移除和电源管理相关的几个函数指针。

此处实现的总线结构与 `sys` 文件系统目录和文件信息有对应关系。例如，平台驱动和平台设备的情况如下所示。

- `/sys/devices/platform/`：子目录表示各个平台设备。
- `/sys/bus/platform/devices`：到 `/sys/devices/platform/` 的连接。
- `/sys/bus/platform/drivers/`：子目录表示各个平台驱动。

平台设备和平台驱动根据名称（`name`）进行匹配，通常情况下在板级实现中定义平台设备，在具体的驱动程序中定义平台驱动。

`platform_device` 表示平台设备，如下所示：

```
struct platform_device {
    const char * name;           // 表示设备的名称
    int id;
    struct device dev;
    u32 num_resources;           // 设备中资源的数目
    struct resource * resource;   // 设备中资源的指针
    const struct platform_device_id *id_entry;
    struct pdev_archdata archdata;
};
extern int platform_device_register(struct platform_device *);
extern void platform_device_unregister(struct platform_device *);
```

platform_device 中的 name 是设备的名称；resource 是设备用的资源，使用首指针和数目的方式来表示。其中包含的 device 结构中还有一个 platform_data 指针，可以保存设备中的私有数据。

platform_driver 表示平台驱动，如下所示：

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;    // 其中包括驱动的名称
    const struct platform_device_id *id_table;
};
extern int platform_driver_register(struct platform_driver *);
extern void platform_driver_unregister(struct platform_driver *);
```

platform_driver 的 driver 中具有表示名称的 name 成员。probe、remove、shutdown、suspend、resume 几个函数指针用于电源管理相关的操作。其中，probe 函数指针表示探测，一旦驱动和设备匹配，就会触发这个过程，常用于设备的初始化。

提示：如果一个驱动没有特殊的电源管理接口，在 probe 中实现它的初始化即可。

头文件 ioport.h 定义资源相关的内容。resource 结构的情况如下所示：

```
struct resource {
    resource_size_t start;        // 表示资源的起始数
    resource_size_t end;          // 表示资源的结束数
    const char *name;
    unsigned long flags;          // 表示资源的标志
    struct resource *parent, *sibling, *child;
};
#define IORESOURCE_BITS          0x000000ff    // 总线特定的位
#define IORESOURCE_TYPE_BITS    0x00000f00    // 资源类型
#define IORESOURCE_IO            0x00000100    // IO 类型的资源
#define IORESOURCE_MEM          0x00000200    // 内存类型的资源
#define IORESOURCE_IRQ          0x00000400    // 中断类型的资源
#define IORESOURCE_DMA          0x00000800    // DMA 类型的资源
```

几种资源的类型为：IO 资源表示端口地址，MEM 表示内存地址，IRQ 表示中断号，DMA 表示 DMA 号。

提示：IO 资源表示的端口是内存空间之外的内容，通常只有 x86 体系结构使用。MEM 表示内存空间的资源，嵌入式系统中常常用其表示特殊功能寄存器。

对于平台设备和平台驱动，一般在前者中声明设备，并定义资源和私有数据；在后者中将资源和私有数据取出并使用。

几个相关的操作函数如下所示：

```
struct resource ioport_resource;
struct resource iomem_resource;
struct resource *request_resource_conflict(struct resource *root,
                                           struct resource *new);
int request_resource(struct resource *root, struct resource *new);
int release_resource(struct resource *new);
int allocate_resource(struct resource *root, struct resource *new,
                     resource_size_t size, resource_size_t min,
                     resource_size_t max, resource_size_t align,
                     resource_size_t (*alignf)(void *,
                                                const struct resource *,
                                                resource_size_t,
                                                resource_size_t),
                     void *alignf_data);
resource_size_t resource_size(const struct resource *res);
unsigned long resource_type(const struct resource *res);
```

对于端口资源 `ioport_resource` 和内存资源 `iomem_resource`，几个宏可以完成如下操作。

- **request_region**：使用从 `start` 到 `end` 之间的端口，`name` 是设备名。在用户空间，分配的端口可以从 `/proc/ioports` 文件查看到。
- **release_region**：用于释放端口资源。
- **request_mem_region**：申请内存资源。也就是起始地址 `start` 到结束地址 `end` 之间的内存。在用户空间，分配的内存可以从 `/proc/iomem` 文件中查看到。
- **request_resource**：释放内存资源。

对于中断资源和 DMA 资源，可以用 `resource` 结构中的 `start` 到 `end` 之间的整数值来表示，大多数情况下一个资源的 `start` 和 `end` 值相等。

在实际的驱动程序中，设备及其资源的定义情况如下所示：

```
static struct resource xxx_resources[] = {
    {
        .start    = 0x07000000,
        .end      = 0x07000000 + 512 - 1,
        .flags     = IORESOURCE_MEM,          // 一个内存类型的资源
    },
    {
        .start    = 10,
        .end      = 10,
        .flags     = IORESOURCE_IRQ,         // 一个中断类型的资源
    },
};
static struct platform_device xxx_device = {
    .name        = "xxx",                   // 设备的名称
    .id          = -1,
    .num_resources = ARRAY_SIZE(xxx_resources),
```

```
.resource    = xxx_resources,    // 设备引用的资源
};
```

"xxx"表示的就是平台设备的名称。与之相对应，platform_driver 的定义如下所示：

```
static struct platform_driver xxx_driver = {
    .driver    = {
        .name    = "xxx",
        .owner    = THIS_MODULE,
    },
    .probe      = xxx_probe,
    .remove     = xxx_remove,
};
```

通常在 probe 函数中，可以获取资源和私有数据并使用。由此，可以让同一段驱动代码匹配不同的硬件，其中的差别就通过资源和私有数据表示。

15.3 中断的处理

中断是一个和系统运行异步的操作，中断发生后计算机转入中断处理函数。对于特定的硬件，中断可以替代轮询，实现快速的硬件响应。

中断是 CPU 实时地处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时，中断系统将迫使 CPU 暂停正在执行的程序，转而去进行中断事件的处理，中断处理完毕后，又返回被中断的程序处，继续执行下去。中断事件通常需要有一段单独的处理函数，有时也被称为中断服务程序（ISR，Interrupt Service Routines）。

在 Linux 系统中，对中断的处理属于系统核心部分。中断的处理如下所示：

- 在移植内核的时候根据系统硬件的状况，为每一个硬件设备分配中断号，当中断产生后，再根据硬件标志查找到产生中断的硬件，确定其中断号。在嵌入式系统的实现中，每个平台通常具有一个 irq.h 头文件，作为具体平台中断号的定义。irq.h 则是内核核心部分的头文件。
- 在驱动程序中，将相应硬件的中断服务程序（也就是中断处理函数）注册到其中断号上，由此实现对于这个中断号的处理操作。

中断处理函数和一般的 I/O 调用的区别在于：应用程序在调用驱动程序的 I/O 操作部分时，进程没有改变，只是进入了内核态。然而，当驱动程序的中断服务函数运行时，并不依赖任何一个进程。

头文件 interrupt.h 中定义的中断相关的类型如下所示：

```
enum irqreturn {
    IRQ_NONE,
    IRQ_HANDLED,
    IRQ_WAKE_THREAD,
};
typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x) ((x) != IRQ_NONE)
```

irqreturn_t 表示一个中断的返回值，本质上就是一个用整数表示的枚举值。

中断相关的处理函数如下所示：

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
int __must_check request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long flags, const char *name, void *dev);
void free_irq(unsigned int, void *);
extern void disable_irq(unsigned int irq);
extern void enable_irq(unsigned int irq);
```

`irq_handler_t` 是中断服务函数（ISR）的类型。设备驱动程序通过调用 `request_irq()` 函数来申请中断，通过 `free_irq()` 来释放中断。

提示：在旧版本的 Linux 内核中内核的驱动，`irq_handler_t` 函数类型以 `void` 为返回值，而不是通过整数返回。

一个在驱动程序中的中断处理函数如下所示：

```
static irqreturn_t xxx_irq_handler(int irq, void *dev_id)
{
    // 可以禁止中断，进入临界段
    disable_irq(IQR_XXX);
    // 中断处理部分
    // 退出临界段时，使能中断
    enable_irq(IQR_XXX);
    return 0;
}
```

`xxx_irq_handler()` 函数就是中断服务程序的实现。在驱动程序中需要使用 `request_irq()` 函数将其注册，作为某一个中断号的处理函数。

15.4 中断的下半部

在操作系统的概念中，中断处理流程通常分为两个阶段：上半部（Top Half）和下半部（Bottom Half）。对于一些紧迫的工作，例如：时间敏感任务，与硬件相关的任务，不能被中断的任务等，也就是所谓的上半部；对于一些相对次要的任务，应当推后执行，也就是所谓的下半部。

在 Linux 系统中，中断处理函数自身的执行属于上半部，而中断下半部则需要使用其他机制实现。

- 软中断：运行于中断上下文。
- 队列：运行于进程上下文。

对于软中断，在内核的编程中明确说明：在具体的代码中，绝大多数不应当直接使用软中断，而可以使用其衍生机制，主要是定时器和 tasklet。

15.4.1 软中断

`interrupt.h` 头文件中关于软中断的内容，如下所示：

```
enum
```

```

{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    // .....省略部分内容
};
struct softirq_action
{
    void    (*action)(struct softirq_action *);
};
asmlinkage void do_softirq(void);
extern void raise_softirq(unsigned int nr);

```

软中断的资源非常有限，仅有以上的几个。在内核当中的核心子系统的实现上，很少使用直接增加软中断的方式。定时器和 tasklet 是两种基于软中断的机制。

15.4.2 软中断之 tasklet

tasklet 是内核提供了一种调度机制，可以指定段代码在某个时间点之后运行 tasklet，其本身的实现是软中断当中的一种，通过 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 来完成。

在头文件 interrupt.h 中定义 tasklet 的相关内容：

```

struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);    // tasklet 的实现函数
    unsigned long data;
};
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
static inline void tasklet_schedule(struct tasklet_struct *t) {}
static inline void tasklet_hi_schedule_first(struct tasklet_struct *t){}
static inline void tasklet_disable_nosync(struct tasklet_struct *t){}
static inline void tasklet_disable(struct tasklet_struct *t){}
static inline void tasklet_enable(struct tasklet_struct *t){}
static inline void tasklet_hi_enable(struct tasklet_struct *t){}
extern void tasklet_kill(struct tasklet_struct *t);
extern void tasklet_kill_immediate(struct tasklet_struct *t, unsigned int cpu);
extern void tasklet_init(struct tasklet_struct *t,
    void (*func)(unsigned long), unsigned long data);

```

在驱动程序中，可以通过 DECLARE_TASKLET 宏声明一个 tasklet，第一个参数表示 tasklet 的名称，第二个参数表示 tasklet 的实现函数。一个 tasklet 建立之后，可以通过 tasklet_schedule() 等函数进行调度，表示执行这段内容。

15.4.3 软中断之定时器

定时器是操作系统提供的一种基本机制。在 Linux 系统中，内核定时器提供定时触发的核心功能。内核定时器可以配合中断机制来使用：中断中可以延时处理的内容，放到定时器中去处理。内核定时器也可以用于内核的一般代码。

Linux 内核定时器在 `timer.h` 中进行定义，如下所示：

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    struct tvec_base *base;
    void (*function)(unsigned long);    // 定时器的处理函数
    unsigned long data;
    int slack;
    // .....省略部分内容
};
#define init_timer(timer)
void add_timer(struct timer_list *timer);
```

一个定时器操作的例子如下所示：

```
static void* my_data;
struct timer_list my_timer;
{
    // ..... 增加定时器的内容
    init_timer(&my_timer);
    my_timer.data = (unsigned long)my_data;
    my_timer.function = my_handler;
    my_timer.expires = jiffies + 2 * HZ;
    add_timer(&my_timer);
}
void my_handler(unsigned long data) {
    // ..... 定时器处理的内容
    my_timer.expires = jiffies + 2 * HZ;
    // ..... 再次激活定时器
    add_timer(&my_timer);
}
```

Linux 内核定时器的使用方式为：初始化，设置一个超时时间；指定超时发生时执行的函数；激活定时器。内核定时器并不是周期运行的，而是只执行一次，它在超时后自动销毁。因此，如果要想实现周期执行的定时器，就需要在定时器执行函数返回前再次激活定时器。

`jiffies` 是 Linux 内核中的一个全局变量，用来记录自系统启动以来产生的节拍总数。启动时，内核将该变量初始化为 0，此后，每次时钟 Tick 处理程序都会增加该变量的值。在 32 位和 64 位的系统中 `jiffies` 的类型也不相同，内核代码使用的特殊方式，让其可以兼容。

`HZ` 是内核定义的宏，通常被定义为 1000，1 秒中 `jiffies` 会被增加 1000。因此 `jiffies + 2*HZ` 表示延迟 2 秒。

15.5 竞态处理

竞态指的是同时运行的两段代码（两个线程）同时访问一个资源的时候。由此带来的先后时间的不确定关系，则可能引起不同的行为。

Linux 驱动程序中的一些内容可以为用户空间调用，因此驱动中的代码可能有来自用户空间的并行调用，引起并发运行的情况，从而引发竞态。因此，必须对共享资源进行并发控制。

Linux 内核中解决并发的主要方法如下所示。

- 自旋锁（spin_lock）。
- 信号量（semaphore）。

15.5.1 自旋锁

自旋锁在等待的过程中，程序进入自旋循环。因此，自旋锁在竞态的等待中存在开销，但是可以在中断上下文中使用。

头文件 `spinlock.h` 中自旋锁的相关内容如下所示：

```
#define spin_lock_init(_lock)           // 初始化自旋锁
void spin_lock(spinlock_t *lock);       // 获取自旋锁
int spin_trylock(spinlock_t *lock);     // 试图获得自旋锁，可直接返回
int spin_trylock_irq(spinlock_t *lock); // 禁止中断，获取自旋锁
void spin_unlock(spinlock_t *lock);     // 释放自旋锁
int spin_is_locked(spinlock_t *lock);   // 查询自旋锁状态
int spin_can_lock(spinlock_t *lock);    // 查询自旋锁状态
```

`spinlock_t` 是一个内部使用的结构，调用者不需要了解其结构。`spin_lock()`用于获取一个锁，使用完成后通过 `spin_unlock()`释放一个锁。如果获取锁不成功，程序就会进入“自旋循环”。`spin_trylock()`则用于试图获得一个锁。即使不能获得，也不会进入自旋循环，而是向下执行。

15.5.2 信号量

信号量是一种睡眠锁，它会使得进程进入睡眠状态。如果获得信号量失败，进程则要进入睡眠状态。由于 Linux 中断的运行不在进程调度之内，因此信号量不能在中断中使用。

在头文件 `semaphore.h` 中定义信号量相关的内容如下所示：

```
struct semaphore {
    spinlock_t      lock;
    unsigned int     count;
    struct list_head wait_list;
};
// 信号量
void sema_init(struct semaphore *sem, int val) // 初始化信号量
#define init_MUTEX(sem)      sema_init(sem, 1) // 初始化信号量为 1
#define init_MUTEX_LOCKED(sem) sema_init(sem, 0) // 初始化信号量为 0
void down(struct semaphore *sem); // 获取信号量，如果不成功，则进入不可中断睡眠
void up(struct semaphore *sem); // 释放信号量，可以唤醒其他等待信号量的任务
```

MUTEX 在内核中表示互斥，down 表示获取，up 表示释放。

15.6 阻塞处理

阻塞可以构建一种无开销的等待。中断是硬件提供的机制，内核可以注册处理函数获得中断。但是，中断机制通常无法直接传送到用户空间。如果用户空间使用循环的方式查询，就将引起程序的开销。

因此在 Linux 系统中，阻塞和中断结合的方式是：在用户空间的操作中阻塞（等待）队列，在中断服务程序中唤醒队列。用户空间的进程操作可以进入睡眠，此时无额外的开销，这个睡眠可以由于中断唤醒，相当于无开销地将中断机制体现到了用户空间。

头文件 wait.h 中定义等待队列 wait_queue 的相关内容如下所示：

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait,
                                unsigned mode, int sync, void *key);
int default_wake_function(wait_queue_t *wait, unsigned mode, int flags, void *key);
#define __WAITQUEUE_INITIALIZER(name, tsk) { \
    .private = tsk, \
    .func = default_wake_function, \
    .task_list = { NULL, NULL } }
#define DECLARE_WAITQUEUE(name, tsk) \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
#define wake_up(x) __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_interruptible(x) __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wait_event(wq, condition)
#define wait_event_timeout(wq, condition, timeout)
#define wait_event_interruptible(wq, condition)
```

wait_event 用于等待一个队列，wake_up 用于唤醒一个队列。等待的时候就会实现阻塞；唤醒之后阻塞就会解除，继续向下执行。

阻塞机制和中断常常结合起来使用，其使用的典型情景如下所示。

在 read、write、ioctl 等文件操作的接口实现中通过调用 wait_event 等待一个队列，这样在用户空间的调用者调用到的时候就会无开销地阻塞。在驱动程序的中断处理函数中调用 wake_up 唤醒中断，原本阻塞的接口就可以解除阻塞，继续向下执行。其效果是在阻塞的时候没有开销，在中断到来的时候，用户空间可以立刻收到，常用于按键、总线收到数据等功能。

poll 调用也是与阻塞相关的调用，内核中的几个相关内容如下所示：

```
typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *,
                               struct poll_table_struct *);
typedef struct poll_table_struct {
    poll_queue_proc pproc;
    unsigned long key;
} poll_table;
```

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
```

`poll_wait` 本身并不实现阻塞。它的使用对应于在用户空间的 `poll()` 或者 `select()` 调用。`poll()` 是基本的调用函数, `select()` 可以通过多路选择的方式对多个设备实现阻塞。

15.7 异步操作

在 Linux 的驱动程序中, 可以利用异步操作实现类似用户空间的中断程序。主要方式为, 实现方法表示文件操作的 `file_operations` 中的 `fasync` 函数指针, 并在被调用的函数 (例如 `read/write`) 中, 调用 `kill_fasync()` 函数激发一个信号。

在用户空间调用 `fcntl()` 函数, 设置 `FASYNC` 文件标志时, 该文件所对应的设备驱动的 `fasync()` 接口将被调用。异步通知 `fasync` 应用于系统调用 `signal` 和 `sigaction` 函数, 也就是让一个信号与一个函数对应, 每当接收到这个信号就会调用相应的函数。

头文件 `fs.h` 中定义的内容如下所示:

```
struct fasync_struct {
    spinlock_t    fa_lock;
    int           magic;
    int           fa_fd;
    struct fasync_struct *fa_next;           // 链表
    struct file    *fa_file;
    struct rcu_head fa_rcu;
};
int fasync_helper(int fd, struct file *filp,
                  int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

`kill_fasync()` 函数用于发送一个信号, 其中的 `band` 参数可以使用 `POLL_IN` (表示设备可读) 和 `POLL_OUT` (表示设备可写)。`kill_fasync()` 函数可以在中断上下文中调用, 通过 `kill` 发送信号, 将硬件的中断机制直接传递到用户空间。

异步的调用方式在用户空间中, 通常可以用如下的方式:

```
int fd;                // 打开的文件描述符
int file_flag;         // 文件的标志
signal(SIGIO, xxx_sig_handler); // 注册一个信号处理器
fcntl(fd, F_SET_OWNER, getpid());
file_flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, file_flags | FASYNC);
```

调用之后, 在用户空间的信号处理器中, 处理的就是驱动程序中调用 `kill_fasync()` 所发送的信号。

第 16 章

几个典型的简单驱动

16.1 设备驱动概述

Linux 的字符设备和块设备一般以主设备号表示设备的类型，次设备号表示具体的设备。`major.h` 包括了预定义的主设备号，其片断如下所示：

```
#define UNNAMED_MAJOR 0      // 没有命名的主设备号
#define MEM_MAJOR 1         // 内存设备（主设备号 1 的字符设备）
#define RAMDISK_MAJOR 1     // 内存盘设备（主设备号 1 的块设备）
#define FLOPPY_MAJOR 2      // 软盘设备（主设备号 2 的块设备）
#define PTY_MASTER_MAJOR2   // 伪终端主设备（主设备号 2 的字符设备）
#define MISC_MAJOR 10       // MISC 主设备（主设备号 10 的字符设备）
```

`major.h` 中定义的内容是字符设备和块设备交错书写的，例如：`MEM_MAJOR` 和 `RAMDISK_MAJOR` 是两个相邻的定义，值都为 1，前者表示字符设备，后者表示块设备。

Linux 的文档目录 `Documentation` 中的 `devices.txt` 文件包括对字符设备和块设备的设备号分配和详细的描述。一般情况下，如果基于某些驱动框架实现设备，将自动生成某主设备号的设备；如果使用自定义的字符设备和块设备，可以使用 240~254 的主设备号。

对于一般字符设备和块设备的实现，`write`、`read`、`lseek` 等通用的调用遵循共同的行为。但是 `ioctl` 调用中还有一个命令号的概念，不同命令号还可以带有不同的参数。命令号本质就是一个 32 位的整数，可以自行定义。但很多时候，为了更清楚地表达此整数的含义，将使用一些宏自动生成。

`include/asm-generic/ioctl.h` 具有如下的定义：

```
#define _IOC_NRBITS 8
#define _IOC_TYPEBITS 8
#define _IOC_SIZEBITS 14
#define _IOC_DIRBITS 2
#define _IOC_NRMASK ((1 << _IOC_NRBITS)-1) // 命令号的位掩码
#define _IOC_TYPMASK ((1 << _IOC_TYPEBITS)-1)
#define _IOC_SIZEMASK ((1 << _IOC_SIZEBITS)-1)
#define _IOC_DIRMASK ((1 << _IOC_DIRBITS)-1)
```

```

#define _IOC_NRSHIFT 0 // 命令号的移位
#define _IOC_TYPESHIFT (_IOC_NRSHIFT+_IOC_NRBITS)
#define _IOC_SIZESHIFT (_IOC_TYPESHIFT+_IOC_TYPEBITS)
#define _IOC_DIRSHIFT (_IOC_SIZESHIFT+_IOC_SIZEBITS)
#define _IOC_NONE 0U
#define _IOC_WRITE 1U
#define _IOC_READ 2U
#define _IOC(dir,type,nr,size) \
    (((dir) << _IOC_DIRSHIFT) | ((type) << _IOC_TYPESHIFT) | \
    ((nr) << _IOC_NRSHIFT) | ((size) << _IOC_SIZESHIFT))

```

根据 `_IOC` 宏的定义，一个 `ioctl` 的命令实际上是方向（`dir`）、类型（`type`）、数字（`nr`）和大小（`size`）四个值相或的结果。

更直接使用的几个宏如下所示：

```

#define _IO(type,nr) _IOC(_IOC_NONE, (type), (nr), 0)
#define _IOR(type,nr,size) _IOC(_IOC_READ, (type), (nr), (_IOC_TYPECHECK(size)))
#define _IOW(type,nr,size) _IOC(_IOC_WRITE, (type), (nr), (_IOC_TYPECHECK(size)))
#define _IOWR(type,nr,size) \
    _IOC(_IOC_READ|_IOC_WRITE, (type), (nr), (_IOC_TYPECHECK(size)))
#define _IOR_BAD(type,nr,size) _IOC(_IOC_READ, (type), (nr), sizeof(size))
#define _IOW_BAD(type,nr,size) _IOC(_IOC_WRITE, (type), (nr), sizeof(size))
#define _IOWR_BAD(type,nr,size) \
    _IOC(_IOC_READ|_IOC_WRITE, (type), (nr), sizeof(size))

```

在一般情况下，`_IOC` 仅用于没有参数的情况；如果一个 `ioctl` 的命令为读性质的，就使用 `_IOR` 宏构建；如果一个 `ioctl` 的命令为写性质的，就使用 `_IOW` 宏构建；如果一个 `ioctl` 的命令为有读有写，就使用 `_IOWR` 宏构建。

在具体的驱动中，几个宏的使用方式常常如下所示：

```

#define RTC_AIE_ON _IO('p', 0x01) /* Alarm int. enable on */
#define RTC_ALM_SET _IOW('p', 0x07, struct rtc_time) /* Set alarm time */
#define RTC_ALM_READ _IOR('p', 0x08, struct rtc_time) /* Read alarm time */
#define RTC_IRQP_READ _IOR('p', 0x0b, unsigned long) /* Read IRQ rate */
#define RTC_IRQP_SET _IOW('p', 0x0c, unsigned long) /* Set IRQ rate */

```

第一个参数通常是一个字符，也就是一个 8 位的整数，表示一种驱动程序的类型。对于有后续参数的宏，第二个参数是命令的序号。第三个参数表示参数的类型，将转为它所占的字节数来使用。按照如此标准定义 `ioctl` 命令号，可以根据整数判断出该命令属于哪种驱动程序，以及命令所带的参数大小。

Linux 中驱动程序并非都用字符设备或块设备的设备节点作为到用户空间的接口，其他两种常见的情况是网络设备或使用 `sys` 文件系统的设备。

16.2 内存设备驱动

16.2.1 内存设备驱动的公共内容

Linux 中的内存设备（`mem`）包含一组字符设备，它们是最简单的字符设备驱动程序，不基于特别的硬件，而是用于表示内存，可支持基本输入/输出功能。

内存设备的设备节点的主设备号都是 1，次设备号代表了不同功能的设备。内存设备在内核的 `drivers/char/mem.c` 文件中实现。

一个 Linux 系统的几个字符设备节点如下所示：

```
# ls -l /dev/mem /dev/null /dev/zero /dev/full /dev/kmem
crw-rw-rw- 1 root root 1, 7 2009-01-30 /dev/full
crw-r----- 1 root kmem 1, 2 2009-01-30 /dev/kmem
crw-r----- 1 root kmem 1, 1 2009-01-30 /dev/mem
crw-rw-rw- 1 root root 1, 3 2009-01-30 /dev/null
crw-rw-rw- 1 root root 1, 5 2009-01-30 /dev/zero
```

内存设备的操作定义在字符设备注册中就已经注册，一般所有的系统中都有该设备。例如：`null` 设备常用于重定向的输出，让输出的内容都归于无有；`zero` 设备常用于使用 `dd` 命令时作为输入，可以产生一个指定大小的空文件。

在实现方式上是在打开设备文件节点时，根据打开文件的不同，将不同的 `file_operations` 赋值给各个次设备，

内存设备的核心定义如下所示：

```
static const struct file_operations memory_fops = {
    .open      = memory_open,
};
```

`memory_fops` 被注册成主设备号为 1 (`MEM_MAJOR`) 的设备，如下所示：

```
static struct class *mem_class;
static int __init chr_dev_init(void)
{
    int i;
    if (register_chrdev(MEM_MAJOR, "mem", &memory_fops)) // 注册字符设备
        printk("unable to get major %d for memory devs\n", MEM_MAJOR);
    mem_class = class_create(THIS_MODULE, "mem");
    for (i = 0; i < ARRAY_SIZE(devlist); i++)
        device_create(mem_class, NULL, // 在内核中创建 device
                      MKDEV(MEM_MAJOR, devlist[i].minor), devlist[i].name);
    return 0;
}
fs_initcall(chr_dev_init);
```

调用标准字符设备的创建函数 `register_chrdev()`，创建设备的名称就是“mem”，后面用一个循环创建各个次设备。

`devlist[]` 是一个表示设备信息的数组，定义如下所示：

```
static const struct {
    unsigned int  minor;           // 次设备号
    char          *name;           // 次设备名称
    umode_t       mode;           // 次设备的权限
    const struct file_operations *fops; // 次设备的文件操作
} devlist[] = {                  // 次设备列表
    {1, "mem",    S_IRUSR | S_IWUSR | S_IRGRP, &mem_fops},
    {2, "kmem",   S_IRUSR | S_IWUSR | S_IRGRP, &kmem_fops},
    {3, "null",   S_IRUGO | S_IWUGO,          &null_fops},
#ifdef CONFIG_DEVPORT
    {4, "port",   S_IRUSR | S_IWUSR | S_IRGRP, &port_fops},
#endif
};
```

```

    {5, "zero",      S_IRUGO | S_IWUGO,      &zero_fops},
    {7, "full",     S_IRUGO | S_IWUGO,      &full_fops},
    {8, "random",   S_IRUGO | S_IWUSR,      &random_fops},
    {9, "urandom",  S_IRUGO | S_IWUSR,      &urandom_fops},
    {11, "kmsg",    S_IRUGO | S_IWUSR,      &kmsg_fops},
#ifdef CONFIG_CRASH_DUMP
    {12, "oldmem",  S_IRUSR | S_IWUSR | S_IRGRP, &oldmem_fops},
#endif
};

```

内存设备的各个子设备（次设备号不同）的文件操作是不相同的，这也是内存设备驱动稍显特殊的地方。每个次设备都有一个自己的 `file_operations`。

每个内存设备次设备的 `file_operations` 在它们共同的打开函数 `memory_open()` 中做出了不同的设置，实现如下所示：

```

static int memory_open(struct inode *inode, struct file *filp)
{
    int minor;
    const struct memdev *dev;
    minor = iminor(inode);    // 设备节点打开的时候，根据节点得到其次设备号
    if (minor >= ARRAY_SIZE(devlist)) return -ENXIO;
    dev = &devlist[minor];    // 找到次设备号对应的设备
    if (!dev->fops)            return -ENXIO;
    filp->f_op = dev->fops;    // 设置文件的 file_operations
    if (dev->dev_info)
        filp->f_mapping->backing_dev_info = dev->dev_info;
    if (dev->fops->open)
        return dev->fops->open(inode, filp);
    return 0;
}

```

其中的逻辑就是，当在用户空间打开一个字符设备的节点时，此时内核中的打开函数将被调用，根据所打开设备节点的次设备号，对表示打开文件的结构 `struct file` 中的文件操作赋值。

提示：正如内存设备所实现的，一个字符设备默认的 `file_operations` 根据主设备号指定，在运行的过程中，文件操作也可以改变。

16.2.2 空设备

空设备就是 `null` 设备，空设备主要支持写操作。

`null` 设备的设备号为 3，其文件操作为 `null_fops`，如下所示：

```

static const struct file_operations null_fops = {
    .llseek      = null_llseek,
    .read        = read_null,
    .write       = write_null,
    .splice_write= splice_write_null,
};
static ssize_t read_null(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos)
{
    return 0;    // 读操作无数据，返回数目为 0
}

```

```

}
static ssize_t write_null(struct file *file, const char __user *buf,
                        size_t count, loff_t *ppos)
{
    return count;          // 写操作直接返回要写的数目，表示写入都成功
}
static loff_t null_lseek(struct file *file, loff_t offset, int orig)
{
    return file->f_pos = 0; // seek 操作，位置永远为 0
}

```

作为空设备的实现，读操作只能读到零数据，但无错误。

16.2.3 零设备

零设备就是 zero 设备，主要支持读操作、内存映射操作。

zero 设备的设备号为 5，其文件操作为 zero_fops，如下所示：

```

#define zero_lseek    null_lseek
#define write_zero    write_null
static const struct file_operations zero_fops = {
    .llseek    = zero_lseek,
    .read      = read_zero,
    .write     = write_zero,
    .mmap      = mmap_zero,
};

```

内存映射的 mmap_zero() 函数如下所示：

```

static int mmap_zero(struct file *file, struct vm_area_struct *vma)
{
    if (vma->vm_flags & VM_SHARED)
        return shmem_zero_setup(vma); // 调用共享内存模块完成映射
    return 0;
}

```

读操作 read_zero() 函数如下所示：

```

static ssize_t read_zero(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos)
{
    size_t written;
    if (!count)        return 0;
    if (!access_ok(VERIFY_WRITE, buf, count)) return -EFAULT; // 判断内存访问
    written = 0;
    while (count) {
        unsigned long unwritten;
        size_t chunk = count;
        if (chunk > PAGE_SIZE) chunk = PAGE_SIZE; // 只是为了延迟的处理
        unwritten = __clear_user(buf, chunk);    // 清空这块内存
        written += chunk - unwritten;
        if (unwritten) break;
        if (signal_pending(current))
            return written ? written : -ERESTARTSYS;
        buf += chunk;                          // 计算剩下的数目
        count -= chunk;
        cond_resched();
    }
}

```

```

    }
    return written ? written : -EFAULT;           // 返回实际的清零数目
}

```

zero 设备写操作、lseek 操作与 null 设备类似；读的实现是利用一个循环直接将来自用户空间的内存区设置为 0。内存映射的操作 mmap_zero() 函数则通过在内核中映射特殊的共享内存到用户空间。

16.2.4 满设备

满设备就是 full 设备，主要支持读操作、写操作（以磁盘满为失败结果）。

full 设备的次设备号为 7，其文件操作为 full_fops，如下所示：

```

#define full_lseek    null_lseek
#define read_full     read_zero
static const struct file_operations full_fops = {
    .llseek    = full_lseek,
    .read      = read_full,
    .write     = write_full,
};
static ssize_t write_full(struct file * file, const char __user * buf,
                          size_t count, loff_t * ppos)
{
    return -ENOSPC;
}

```

full 设备就是满设备，其读操作与零设备相同，lseek 操作与 null 操作相同，写操作将返回磁盘满的错误。

16.3 内存块设备驱动

内存块（RAMDisk）设备驱动程序是一个简单的块设备，建立于内存之上。RAMDisk 是将内存中的一部分空间模拟成一个磁盘设备，按照通常块设备的访问方式来访问这一片内存，如同真正磁盘的访问。内存块设备驱动的主设备号为 1；次设备号从 0 依次递增，可以指定个数和大小。

内存块设备在用户空间的设备节点为：/dev/ram<N>，<N>根据次设备号依次递增。

内存块设备驱动程序的代码路径为：drivers/block/brd.c。

内存块设备的辅助结构 brd_device 如下所示：

```

struct brd_device {
    int      brd_number;           // 表示内存块设备的序号
    int      brd_refcnt;          // 表示内存块设备的引用计数
    loff_t    brd_offset;
    loff_t    brd_sizelimit;
    unsigned brd_blocksize;
    struct request_queue *brd_queue; // 表示内存块设备的队列
    struct gendisk *brd_disk;       // 表示一个通用的磁盘
    struct list_head brd_list;
    // ..... 省略部分内容
};

```

`brd_device` 结构表示内存块设备中的一个，其中包括了块设备核心实现中的 `request_queue` 和 `gendisk` 结构。

内存块设备的初始化函数 `brd_init()` 的主要片断如下所示：

```
static int __init brd_init(void)
{
    int i, nr;
    unsigned long range;
    struct brd_device *brd, *next;
    part_shift = 0;
    if (max_part > 0) part_shift = fls(max_part);
    if (rd_nr > 1UL << (MINORBITS - part_shift)) return -EINVAL;
    if (rd_nr) {
        nr = rd_nr;
        range = rd_nr;
    } else {
        nr = CONFIG_BLK_DEV_RAM_COUNT;
        range = 1UL << (MINORBITS - part_shift);
    }
    if (register_blkdev(RAMDISK_MAJOR, "ramdisk")) return -EIO; // 注册块设备
    for (i = 0; i < nr; i++) {
        brd = brd_alloc(i);
        if (!brd) goto out_free;
        list_add_tail(&brd->brd_list, &brd_devices); // 将块设备加入链表
    }
    list_for_each_entry(brd, &brd_devices, brd_list)
        add_disk(brd->brd_disk);
    blk_register_region(MKDEV(RAMDISK_MAJOR, 0), range, // 注册块设备的节点
        THIS_MODULE, brd_probe, NULL, NULL);
    return 0;
    // ..... 省略部分内容
}
```

以上的初始化操作执行了各个内存块设备的建立和次设备号的分配。核心内容是分配 `brd_device` 结构的 `brd_alloc()` 函数。

`brd_alloc()` 函数的实现如下所示：

```
static struct brd_device *brd_alloc(int i)
{
    struct brd_device *brd;
    struct gendisk *disk;
    brd = kzalloc(sizeof(*brd), GFP_KERNEL); // 分配设备所需的空间
    if (!brd) goto out;
    brd->brd_number = i; // 设置内存块设备的序号为次设备号
    spin_lock_init(&brd->brd_lock);
    INIT_RADIX_TREE(&brd->brd_pages, GFP_ATOMIC);
    brd->brd_queue = blk_alloc_queue(GFP_KERNEL); // 分配其中的处理队列，并设置
    if (!brd->brd_queue) goto out_free_dev;
    blk_queue_make_request(brd->brd_queue, brd_make_request);
    blk_queue_ordered(brd->brd_queue, QUEUE_ORDERED_TAG, NULL);
    blk_queue_max_hw_sectors(brd->brd_queue, 1024);
    blk_queue_bounce_limit(brd->brd_queue, BLK_BOUNCE_ANY);
    brd->brd_queue->limits.discard_granularity = PAGE_SIZE;
    brd->brd_queue->limits.max_discard_sectors = UINT_MAX;
    brd->brd_queue->limits.discard_zeroes_data = 1;
    queue_flag_set_unlocked(QUEUE_FLAG_DISCARD, brd->brd_queue);
}
```

```

disk = brd->brd_disk = alloc_disk(1 << part_shift);
if (!disk) goto out_free_queue;
disk->major = RAMDISK_MAJOR; // 设置 gendisk 中的主设备号为 1
disk->first_minor = i << part_shift; // 设置 gendisk 中的首个次设备
disk->fops = &brd_fops; // 设置 gendisk 中的块设备操作
disk->private_data = brd;
disk->queue = brd->brd_queue; // 设置 gendisk 中的处理队列
disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
sprintf(disk->disk_name, "ram%d", i);
set_capacity(disk, rd_size * 2);
return brd;
// ..... 省略错误处理
}

```

brd_alloc()函数完成内存盘块设备中一个设备的分配，不同的内存块设备表示不同的分区，它们之间主要是次分区不同。**brd_device** 被存储成每个 **gendisk** 的私有数据，而 **brd_fops** 是它们共同的块设备操作。

brd_fops 是一个块设备的操作 **block_device_operations**，其定义的内容如下所示：

```

static const struct block_device_operations brd_fops = {
    .owner = THIS_MODULE,
    .locked_ioctl = brd_ioctl,
#ifdef CONFIG_BLK_DEV_XIP
    .direct_access = brd_direct_access,
#endif
};

```

brd_fops 的函数指针中主要定义了内存块设备的 **ioctl** 实现。

内存块设备中 **request_queue** 结构是核心的实现，所实现的 **brd_make_request** 函数主体如下所示：

```

static int brd_make_request(struct request_queue *q, struct bio *bio)
{
    struct block_device *bdev = bio->bi_bdev;
    struct brd_device *brd = bdev->bd_disk->private_data;
    int rw; struct bio_vec *bvec;
    sector_t sector;
    int i; int err = -EIO;
    sector = bio->bi_sector;
    // ..... 省略错误处理
    rw = bio_rw(bio); // 写入 BIO
    if (rw == READA) rw = READ;
    bio_for_each_segment(bvec, bio, i) { // 处理每一个 bio
        unsigned int len = bvec->bv_len;
        err = brd_do_bvec(brd, bvec->bv_page, len, bvec->bv_offset, rw, sector);
        if (err) break;
        sector += len >> SECTOR_SHIFT; // 增加所谓处理的扇区数目
    }
    // ..... 省略错误处理
}

```

此处使用了自己实现的函数来处理 **bio**，这部分将被块设备框架层调用。

16.4 回环块设备驱动

回环块设备驱动程序是一个简单的块设备。回环块设备驱动的主设备号为 7；次设备号从 0 依次递增，可以指定个数。

回环块设备的主要用途是用于挂接没有和块设备相联系的文件系统。例如：在执行 `mount -o` 挂接一个映像文件为文件系统的时候，就将使用一个回环设备。

回环块设备在用户空间的设备节点为：`/dev/loop<N>`，`<N>`根据设备号依次递增。

回环块设备驱动程序的头文件为：`include/linux/loop.h`，代码路径为：`drivers/block/loop.c`。

`loop.h` 中回环块设备特殊的控制命令如下所示：

```
#define LOOP_SET_FD      0x4C00      // 设置文件描述符
#define LOOP_CLR_FD      0x4C01      // 清除文件描述符
#define LOOP_SET_STATUS  0x4C02
#define LOOP_GET_STATUS  0x4C03
#define LOOP_SET_STATUS64 0x4C04
#define LOOP_GET_STATUS64 0x4C05
#define LOOP_CHANGE_FD   0x4C06
#define LOOP_SET_CAPACITY 0x4C07
```

回环块设备需要将设备和一个打开的文件结合起来，因此主要的 `ioctl` 控制命令包括文件描述符的设置和清除。这个被进程打开的文件就是要挂接的映像文件。

`loop.h` 定义了回环块设备的核心结构 `loop_device`，如下所示：

```
struct loop_device {
    int      lo_number;           // 表示内存块设备的序号
    int      lo_refcnt;
    loff_t   lo_offset;
    loff_t   lo_sizelimit;
    int      lo_flags;
    // ..... 省略部分内容
    struct file *lo_backing_file;
    struct block_device *lo_device; // 表示块设备的核心结构
    // ..... 省略部分内容
    struct request_queue *lo_queue; // 表示内存块设备的队列
    struct gendisk *lo_disk;        // 表示一个通用的磁盘
};
```

`loop_device` 结构包括本驱动实现所需要的上下文，也包括块设备必需的结构，如 `lo_device`、`gendisk` 和 `request_queue` 等。

回环块设备的初始化函数 `loop_init()` 如下所示：

```
static int __init loop_init(void)
{
    int i, nr;
    unsigned long range;
    struct loop_device *lo, *next;
    // ..... 省略部分内容
    if (register_blkdev(LOOP_MAJOR, "loop")) return -EIO; // 注册块设备
    for (i = 0; i < nr; i++) { // 处理各个回环块设备的次设备
        lo = loop_alloc(i);
        if (!lo) goto Enomem;
    }
}
```

```

        list_add_tail(&lo->lo_list, &loop_devices);
    }
    list_for_each_entry(lo, &loop_devices, lo_list) // 处理每一个设备
        add_disk(lo->lo_disk);
    blk_register_region(MKDEV(LOOP_MAJOR, 0), range, // 注册块设备的节点
        THIS_MODULE, loop_probe, NULL, NULL);

    return 0;
// ..... 省略错误处理
}

```

`loop_init()`函数在建立块设备的 `loop_probe()`函数中被调用，调用 `loop_init_one()`表示初始化一个回环块设备，其中又调用了回环块设备的分配函数 `loop_alloc()`。

`loop_alloc()`函数如下所示：

```

static struct loop_device *loop_alloc(int i)
{
    struct loop_device *lo;
    struct gendisk *disk;
    lo = kzalloc(sizeof(*lo), GFP_KERNEL);
// ..... 省略错误处理
    mutex_init(&lo->lo_ctl_mutex);
    lo->lo_number = i; // 设置次设备号
    lo->lo_thread = NULL;
    init_waitqueue_head(&lo->lo_event); // 初始化本驱动使用的一个队列
    spin_lock_init(&lo->lo_lock);
    disk->major = LOOP_MAJOR;
    disk->first_minor = i << part_shift;
    disk->fops = &lo_fops; // 设置块设备的操作
    disk->private_data = lo; // 将设置 loop_device 存在 gendisk 中
    disk->queue = lo->lo_queue; // 设置 gendisk 中的队列
    sprintf(disk->disk_name, "loop%d", i); // 获得设备名称
    return lo; // 返回一个 loop_device 类型的结构
// ..... 省略错误处理
}

```

`loop_alloc()`函数实现了一个块设备的构建流程，本驱动的上下文作为 `loop_device` 结构组织，然后将其设置为 `gendisk` 中的私有数据 (`private_data`)。

表示块设备结构 `block_device_operations` 的 `lo_fops` 如下所示：

```

static const struct block_device_operations lo_fops = {
    .owner = THIS_MODULE,
    .open = lo_open,
    .release = lo_release,
    .ioctl = lo_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = lo_compat_ioctl,
#endif
};

```

块设备操作 `lo_fops` 实现了 `open`、`release` 和 `ioctl` 几个函数指针。

负责块设备打开的函数 `lo_open()`如下所示：

```

static int lo_open(struct inode *inode, struct file *file)
{
    struct loop_device *lo
        = inode->i_bdev->bd_disk->private_data; // 从 inode 中获得块设备
}

```

```

mutex_lock(&lo->lo_ctl_mutex);
lo->lo_refcnt++;           // 设置引用计数
mutex_unlock(&lo->lo_ctl_mutex);
return 0;
};

```

回环块设备的结构 `loop_device` 保存在 `gendisk` 结构的 `private_data` 中，因此需要在设备打开的时候得到这个设备，并设置其引用计数加一。

回环块设备当中 `loop_make_request` 用于构建 `request_queue` 结构，如下所示：

```

static int loop_make_request(struct request_queue *q, struct bio *old_bio)
{
    struct loop_device *lo = q->queuedata;
    int rw = bio_rw(old_bio); // 写入 BIO
    if (rw == READA)         rw = READ;
    BUG_ON(!lo || (rw != READ && rw != WRITE));
    spin_lock_irq(&lo->lo_lock);
// ..... 省略错误处理
    loop_add_bio(lo, old_bio);           // 增加处理的 BIO
    wake_up(&lo->lo_event);              // 唤醒队列的处理
    spin_unlock_irq(&lo->lo_lock);
    return 0;
// ..... 省略错误处理
}

```

回环块设备的处理流程通常比较长，因此为每个处理建立了一个线程，线程的实现就是 `loop_thread()` 函数。在线程中将等待消息，`loop_make_request()` 函数唤醒消息。

回环块设备在用户空间支持特殊的 `ioctl` 命令，处理所用的 `lo_ioctl()` 函数如下所示：

```

static int lo_ioctl(struct block_device *bdev, fmode_t mode,
    unsigned int cmd, unsigned long arg)
{
    struct loop_device *lo = bdev->bd_disk->private_data;
    int err;
    mutex_lock_nested(&lo->lo_ctl_mutex, 1);
    switch (cmd) {
        case LOOP_SET_FD:           // 设置文件描述符
            err = loop_set_fd(lo, mode, bdev, arg);
            break;
        case LOOP_CHANGE_FD:       // 改变文件描述符
            err = loop_change_fd(lo, bdev, arg);
            break;
        case LOOP_CLR_FD:          // 清除文件描述符
            err = loop_clr_fd(lo, bdev);
            if (!err) goto out_unlocked;
            break;
// ..... 省略部分内容: LOOP_SET_STATUS、LOOP_GET_STATUS 等命令
        case LOOP_SET_CAPACITY:    // 设置能力
            err = -EPERM;
            if ((mode & FMODE_WRITE) || capable(CAP_SYS_ADMIN))
                err = loop_set_capacity(lo, bdev);
            break;
        default:
            err = lo->ioctl ? lo->ioctl(lo, cmd, arg) : -EINVAL; // 默认的控制
    }
    mutex_unlock(&lo->lo_ctl_mutex);
// ..... 省略错误内容
}

```

lo_ioctl()实现了回环块设备的各种特殊控制命令。LOOP_SET_FD 命令表示设置了在用户空间打开的映像文件，通过调用 loop_set_fd()函数完成。loop_set_fd()函数将通过 loop_thread()建立线程，此处用户空间传入的参数 arg 是整数类型的文件描述符，需要转化成 struct file 结构使用。LOOP_SET_CAPACITY 用于设置能力，能力表示的就是这种块设备的大小，传入的参数表示扇区（sector）的数目，乘以 512 之后（左移 9 位），作为块设备的大小使用。

16.5 回环网络设备驱动

回环网络设备是 Linux 中最为基本的网络驱动程序，它同样是 Linux 网络模块运行时必需的。回环网络设备的实现将把向其发送的数据发回。在具有网络支持的 Linux 系统中，回环网络设备一般都是存在的。

在 Linux 系统的命令行中输入 ifconfig 命令，可以得到系统网络设备的信息。在网络设备中，lo 即本机的回环设备，其 IP 地址为 127.0.0.1，子网掩码（mask）为 255.0.0.0。对网络地址发送数据包，就会通过回环网络设备传递给本机。

回环网络设备的实现源代码为：drivers/net/loopback.c。

回环网络设备的初始化函数 loopback_net_init()如下所示：

```
static __net_init int loopback_net_init(struct net *net)
{
    struct net_device *dev;          // 网络设备结构
    int err;
    err = -ENOMEM;
    dev = alloc_netdev(0, "lo", loopback_setup);    // 分配网络设备
    if (!dev)        goto out;
    dev_net_set(dev, net);
    err = register_netdev(dev);    // 注册网络设备
    if (err)
        goto out_free_netdev;
    net->loopback_dev = dev;
    return 0;
}
```

loopback_net_init()函数中分配了名称为"lo"的 net_device 结构，并将其进行了注册。

建立回环网络设备的函数 loopback_setup()如下所示：

```
static void loopback_setup(struct net_device *dev)
{
    dev->mtu      = (16 * 1024) + 20 + 20 + 12;
    dev->hard_header_len = ETH_HLEN;    // 硬件头的长度：值为 14
    dev->addr_len  = ETH_ALEN;    // 地址的长度：值为 6
    dev->tx_queue_len = 0;
    dev->type       = ARPHRD_LOOPBACK;
    dev->flags      = IFF_LOOPBACK;
    dev->priv_flags  &= ~IFF_XMIT_DST_RELEASE;
    dev->features    = NETIF_F_SG | NETIF_F_FRAGLIST
        | NETIF_F_TSO | NETIF_F_NO_CSUM | NETIF_F_HIGHDMA
        | NETIF_F_LLTX | NETIF_F_NETNS_LOCAL;
    dev->ethtool_ops = &loopback_ethtool_ops;
```

```

dev->header_ops = &eth_header_ops;
dev->netdev_ops = &loopback_ops;      // 进行网络设备操作的 net_device_ops
dev->destructor = loopback_dev_free;
}

```

loopback_setup()函数主要进行了 net_device 结构的填充, 主要的成员包括 net_device_ops 结构、ethtool_ops 结构等。

表示网络设备操作的 net_device_ops 结构的 loopback_ops:

```

static const struct net_device_ops loopback_ops = {
    .ndo_init      = loopback_dev_init,
    .ndo_start_xmit = loopback_xmit,
    .ndo_get_stats = loopback_get_stats,
};

```

loopback_ops 中包括 3 个函数指针: 其中, loopback_dev_init()函数用于设备的初始化, loopback_get_stats()函数返回表示网络状态的 net_device_stats, 主要是使用表示发送的 loopback_xmit()函数。

提示: Linux 较早期版本的 net_device 当中有一个 hard_start_xmit 函数指针, 功能同 net_device_ops 结构中的 ndo_start_xmit 函数指针, 同样用于网络数据的发送。

loopback_xmit()函数的实现如下所示:

```

static netdev_tx_t loopback_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct pcpu_lstats __percpu *pcpu_lstats;
    struct pcpu_lstats *lb_stats;
    int len;
    skb_orphan(skb);          // 孤立 sk_buff
    skb->protocol = eth_type_trans(skb, dev);
    pcpu_lstats = (void __percpu __force *)dev->ml_priv;
    lb_stats = this_cpu_ptr(pcpu_lstats);
    len = skb->len;            // 得到 sk_buff 的长度
    if (likely(netif_rx(skb) == NET_RX_SUCCESS)) {      // 进行网络数据的接收
        lb_stats->bytes += len;          // 设置字节的数目
        lb_stats->packets++;
    } else
        lb_stats->drops++;
    return NETDEV_TX_OK;
}

```

其中, 调用 skb_orphan()函数的含义是将 sk_buff 孤立, 也就是让它和发送 socket 的协议栈不再有任何联系, 相当于数据被发送, 并释放了 sk_buff。随后调用 netif_rx()函数, 表示发送数据的结果就是让本机接收一个数据。由此表示向回环网络设备发送的数据会被发送给自己。

第 17 章

几个典型的驱动框架和相应的驱动

17.1 Misc 驱动框架

Misc 驱动程序是 Linux 中的一个驱动程序框架，在这个框架下，可以实现多种驱动程序。Misc 驱动程序的主设备号为 10，次设备号由各个驱动程序的实现自行定义。

使用 Misc 驱动程序框架可以构建各种自定义的字符设备，这些设备的实现仍基于文件操作（file_operations）。各个 Misc 设备可以不占用独立的主设备号，更便于管理。

Misc 驱动程序的结构如图 17-1 所示。

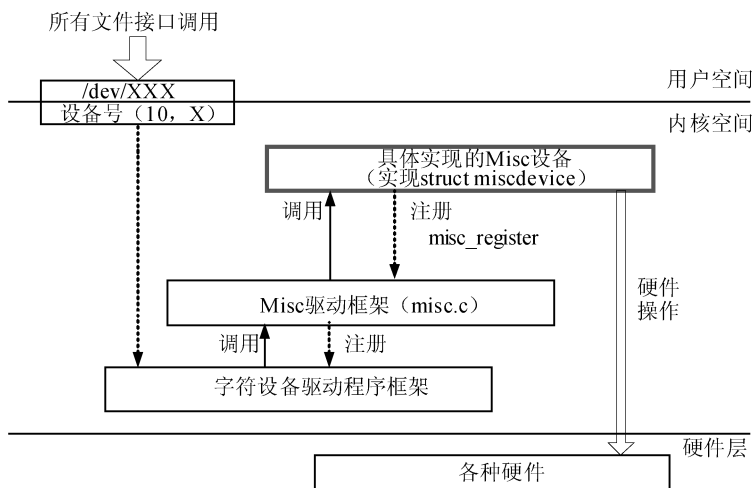


图 17-1 Misc 驱动程序的结构

Misc 驱动程序也预定义了如下一些设备号。

- 15: 一个触摸屏，设备节点为/dev/touchscreen/mk712 MK712。
- 130: /dev/watchdog，看门狗定时器端口。
- 135: /dev/rtc，实时时钟。

- 240~254: 用于自定义的设备。
- 255: 表示动态增加, 依次递增, 得到的是可以使用的最小次设备号。

因此, 用户在自定义一个 Misc 驱动程序的时候, 通常可以使用 240~254 的设备号; 或者使用 255, 则表示得到自动生成的设备号。

Misc 驱动程序的主要头文件为: `include/linux/miscdevice.h`。

Misc 驱动程序的核心文件为: `drivers/char/misc.c`。

`miscdevice.h` 文件中, Misc 设备的接口定义如下所示:

```
struct miscdevice {
    int minor;                // 表示次设备号
    const char *name;         // 表示次设备的名称
    const struct file_operations *fops; // 表示次设备号的文件操作
    struct list_head list;     // 表示设备的链表
    struct device *parent;
    struct device *this_device;
};
extern int misc_register(struct miscdevice * misc);
extern int misc_deregister(struct miscdevice *misc);
```

在自定义一个 Misc 驱动程序的时候, 需要构建 `file_operations` 结构体并实现其中的各个函数指针, 然后构建 `miscdevice` 结构, 之后将其注册为一个 Misc 设备。由此构建的设备和直接构建字符设备非常相似。

所有 Misc 设备的主设备号都是 10, 各个实现的具体设备的次设备号不同, 可以自行指定, 如果指定为 `MISC_DYNAMIC_MINOR` (255), 则表示次设备号依次自动生成。

提示: 即便有些 MISC 设备的次设备号已经有了定义, 但只要系统中没有实现用到它们的设备, 这些次设备号就依然可以使用于别的设备。

17.2 帧缓冲驱动框架和具体驱动

帧缓冲 (FrameBuffer) 是 Linux 用于显示设备的驱动程序。在桌面系统中用于显卡的驱动, 在嵌入式系统中常用于 LCD 控制器的驱动。帧缓冲驱动程序由驱动程序的框架和具体驱动程序的实现两部分组成。

17.2.1 帧缓冲驱动框架

Linux 帧缓冲的实现由 FrameBuffer 驱动框架和具体的驱动程序组成。FrameBuffer 驱动框架本身是一个构建于字符设备之上的框架, 所有 FrameBuffer 设备的主设备号都是 29, 每个 FrameBuffer 设备的次设备号递增生成 (一般由每个具体 FrameBuffer 程序的注册顺序决定)。FrameBuffer 驱动在用户空间的设备节点为 `/dev/fb<N>`, `<N>` 由 FrameBuffer 驱动的序号来决定。

FrameBuffer 驱动的结构如图 17-2 所示。

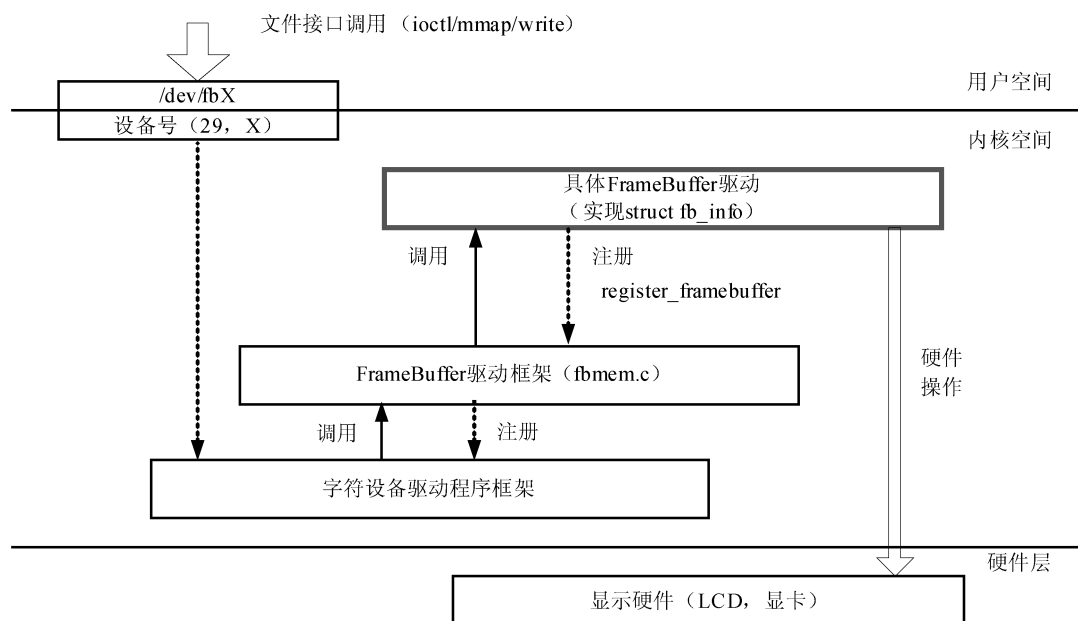


图 17-2 FrameBuffer 驱动的结构

FrameBuffer 驱动在用户空间大多使用文件操作对其进行控制，使用 `ioctl`、`mmap` 等文件系统的接口进行操作，`ioctl` 用于获得和设置信息，`mmap` 操作用于将 FrameBuffer 的内存映射到用户空间。此外，直接写 FrameBuffer 设备表示显示所输出的数据，直接读 FrameBuffer 设备表示读屏幕数据。对 FrameBuffer 的读写通常只是用于调试，在命令行就可以执行，但是它们与 FrameBuffer 的实现相关，在某些 FrameBuffer 驱动中不一定有效。

在内核的驱动程序实现中，具体 FrameBuffer 驱动程序实现的不是字符设备，而是 FrameBuffer 框架所定义的结构，主要包括屏幕的信息和 FrameBuffer 的操作等。

用户空间以文件接口进行调用的时候，将经过 FrameBuffer 驱动的框架，调用到具体的 FrameBuffer 设备驱动，由 FrameBuffer 设备驱动控制硬件。

FrameBuffer 驱动的主要头文件为：`include/linux/fb.h`。

FrameBuffer 驱动核心实现为：`drivers/video/fbmem.c`。

头文件 `fb.h` 中定义了 FrameBuffer 驱动需要的结构和函数。

几个控制命令的定义如下所示：

```
#define FBIOGET_VSCREENINFO    0x4600    // 获取变化屏幕信息
#define FBIOPUT_VSCREENINFO    0x4601    // 设置变化屏幕信息
#define FBIOGET_FSCREENINFO   0x4602    // 获取变化屏幕信息
#define FBIOGETCMAP           0x4604
#define FBIOPUTCMAP           0x4605
#define FBIOPAN_DISPLAY        0x4606    // 平移屏幕的显示位置
// .....省略部分操作
```

FrameBuffer 驱动的 `ioctl` 命令都由 FrameBuffer 驱动的框架实现了。具体的 FrameBuffer 驱动程序也可以重新实现某些命令，则还可以增加新的命令。

fb_info 为 FrameBuffer 信息，它是 FrameBuffer 驱动的核心数据结构。

```
struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var;    // 显示屏变量
    struct fb_fix_screeninfo fix;    // 显示屏固定量
    // .....省略部分操作
    struct fb_ops *fbops;           // FrameBuffer 的操作
    // .....省略部分操作
};
```

固定屏幕信息 (fb_fix_screeninfo) 定义了帧缓冲标识字符串、内存长度、每行字节数等信息，表示显示缓冲区的内存特性。变化屏幕信息 (fb_var_screeninfo) 定义了 FrameBuffer 的宽和高以及像素格式等信息，还可以支持虚拟显示屏。

FrameBuffer 屏幕信息的结构如图 17-3 所示。

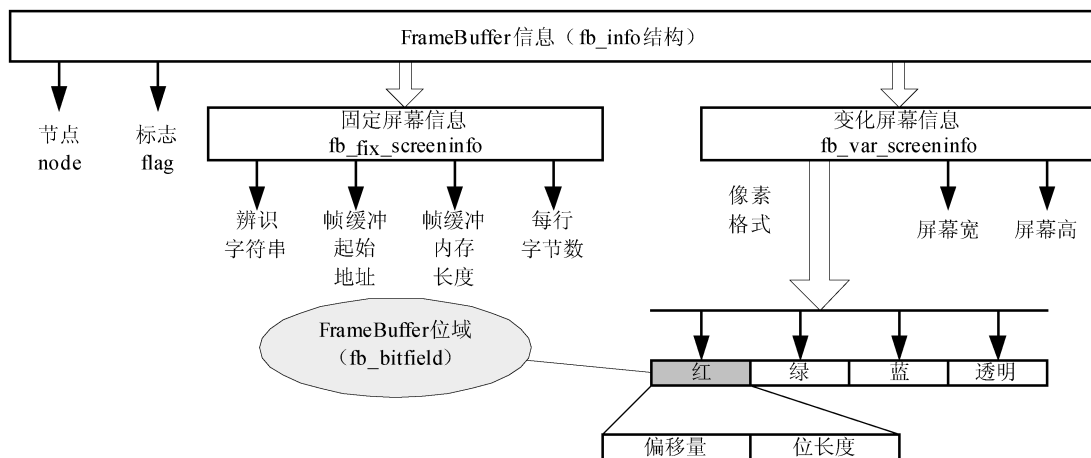


图 17-3 FrameBuffer 屏幕信息的结构

表示 FrameBuffer 操作数据结构 fb_ops 如下所示：

```
struct fb_ops {
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);
    ssize_t (*fb_read)(struct file *file, char __user *buf,
                      size_t count, loff_t *ppos);
    ssize_t (*fb_write)(struct file *file, const char __user *buf,
                       size_t count, loff_t *ppos);
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);
    int (*fb_set_par)(struct fb_info *info);
    // .....省略部分操作
    int (*fb_ioctl)(struct inode *inode, struct file *file,
                   unsigned int cmd, unsigned long arg, struct fb_info *info);
    // .....省略部分操作
    int (*fb_mmap)(struct fb_info *info, struct file *file,
                  struct vm_area_struct *vma);
};
```

fb_ops 结构中的各个函数都具有默认的实现，一般不需要具体的驱动实现。主要需要实现情况是：如果驱动中有特殊的控制命令，则需要实现 fb_ioctl；如果驱动中的读写需要特定的硬件操作，则需要重新实现 fb_write 和 fb_read。

FrameBuffer 的注册和注销函数的参数就是 fb_info 结构体。

```
extern int register_framebuffer(struct fb_info *fb_info);
extern int unregister_framebuffer(struct fb_info *fb_info);
```

FrameBuffer 驱动的核心文件 fbmem.c 已经注册了主设备号为 29 的字符设备，它内部注册了字符设备，实现了文件操作。

当注册具体的 FrameBuffer 设备驱动的时候，注册的是 fb_info 中的 fb_ops 类型结构的函数指针成员，它将被 FrameBuffer 驱动中 file_operations 的函数指针间接调用。

除此之外，FrameBuffer 框架中还提供一系列的辅助函数，帮助构建一个实际的 FrameBuffer 驱动程序。

17.2.2 虚拟帧缓冲驱动

虚拟帧缓冲（vfb, Virtual FrameBuffer）驱动程序是一个不基于硬件的驱动程序，它在内存中开辟虚拟显示设备的内存，这可以模拟对显示的操作。

从结构上，vfb 和一个真实的显示驱动程序基本相同，但是它实际上并不完成显示的功能，而只提供虚拟显示输出功能。在实际的系统中，vfb 通常仅仅用于测试，或者在没有支持 FrameBuffer 硬件的时候，作为默认的实现。

虚拟帧缓冲驱动的源代码为：drivers/video/vfb.c。

在内核配置的时候，如果选择了虚拟帧缓冲的支持（FB_VIRTUAL），将会自动选择帧缓冲驱动的几个配置宏（填充、区域复制、fb 的 sys 文件系统）。

vfb 变化屏幕信息（fb_var_screeninfo）的定义如下所示：

```
static struct fb_var_screeninfo vfb_default __initdata = {
    .xres      = 640,      .yres      = 480,      // 定义屏幕的大小
    .xres_virtual = 640,    .yres_virtual = 480,
    .bits_per_pixel = 8,
    .red       = { 0, 8, 0 }, // 定义 RGB888 的颜色空间
    .green     = { 0, 8, 0 },
    .blue      = { 0, 8, 0 },
    .activate  = FB_ACTIVATE_TEST,
    .height    = -1,       .width     = -1,
    .pixclock  = 20000,
    .left_margin = 64,      .right_margin = 64,
    .upper_margin = 32,     .lower_margin = 32,
    .hsync_len  = 64,      .vsync_len  = 2,
    .vmode     = FB_VMODE_NONINTERLACED,
};
```

vfb_default 结构的定义表示 vfb 的屏幕大小为 640×480，颜色空间为 RGB888，每个像素 24 个字节。

vfb 固定屏幕信息（fb_fix_screeninfo）的定义如下所示：

```
static struct fb_fix_screeninfo vfb_fix __devinitdata = {
```

```
.id = "Virtual FB",
.type = FB_TYPE_PACKED_PIXELS,
.visual = FB_VISUAL_PSEUDOCOLOR,
.xpanstep = 1, // 虚拟的屏幕可以支持任意的移位操作
.ypanstep = 1,
.ywrapstep = 1,
.accel = FB_ACCEL_NONE,
};
```

vfb 的 FrameBuffer 操作由 vfb_ops 结构表示，内容如下所示：

```
static struct fb_ops vfb_ops = {
    .fb_read = fb_sys_read,
    .fb_write = fb_sys_write,
    .fb_check_var = vfb_check_var, // vfb 的检查参数
    .fb_set_par = vfb_set_par, // vfb 的设置参数
    .fb_setcolreg = vfb_setcolreg, // vfb 的设置颜色寄存器
    .fb_pan_display = vfb_pan_display, // vfb 的移动操作
    .fb_fillrect = sys_fillrect,
    .fb_copyarea = sys_copyarea,
    .fb_imageblit = sys_imageblit,
    .fb_mmap = vfb_mmap, // vfb 的内存映射操作
};
```

在 vfb_ops 结构中的 sys_fillrect（填充矩形）、sys_copyarea（复制区域）和 sys_imageblit（绘制图像）是 FrameBuffer 驱动程序中公用的函数，而以 vfb_ 为前缀的几个函数是在 vfb 驱动中单独实现的。

vfb_pan_display()的实现如下所示：

```
static int vfb_pan_display(struct fb_var_screeninfo *var, struct fb_info *info)
{
    if (var->vmode & FB_VMODE_YWRAP) {
        if (var->yoffset < 0
            || var->yoffset >= info->var.yres_virtual || var->xoffset)
            return -EINVAL;
    } else {
        if (var->xoffset + var->xres > info->var.xres_virtual ||
            var->yoffset + var->yres > info->var.yres_virtual)
            return -EINVAL;
    }
    info->var.xoffset = var->xoffset; // 指定显示的位置
    info->var.yoffset = var->yoffset;
    if (var->vmode & FB_VMODE_YWRAP)
        info->var.vmode |= FB_VMODE_YWRAP;
    else
        info->var.vmode &= ~FB_VMODE_YWRAP;
    return 0;
}
```

vfb_pan_display()利用了虚拟缓冲完成显示区域调整的函数。在其中，并不需要完全重写显示区域，只需要将可以显示区域的地址赋值。

提示：在一个实际的硬件系统中，pan 功能的实现应当实际显示硬件的寄存器操作。

17.2.3 针对硬件实现的帧缓冲驱动

三星平台 S3C 系列处理器的 LCD 控制器驱动的实现结构比较类似，因此它们常常可以共用同一个 FrameBuffer 驱动程序，当中的某些配置可能不同，这些内容可以由平台数据来定义。三星平台 FrameBuffer 驱动程序的源代码为：drivers/video/s3c-fb.c。

三星平台 FrameBuffer 的平台驱动如下所示：

```
static struct platform_driver s3c_fb_driver = {
    .probe      = s3c_fb_probe,
    .remove     = __devexit_p(s3c_fb_remove),
    .suspend    = s3c_fb_suspend,
    .resume     = s3c_fb_resume,
    .driver     = {
        .name    = "s3c-fb",           // 设备的名称
        .owner   = THIS_MODULE,
    },
};
```

s3c_fb_driver 在初始化函数中，被注册成了平台驱动。因此，当平台设备匹配的时候，其中的 s3c_fb_probe() 函数将被调用。

s3c_fb_probe() 函数的实现如下所示：

```
static int __devinit s3c_fb_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct s3c_fb_platdata *pd;    // 驱动中的私有数据结构
    struct s3c_fb *sfb;
    struct resource *res;
    int win;
    int ret = 0;
    pd = pdev->dev.platform_data;    // 从平台设备中分配私有数据结构
    // ..... 省略错误处理
    sfb = kzalloc(sizeof(struct s3c_fb), GFP_KERNEL);
    // ..... 省略错误处理
    sfb->dev = dev;
    sfb->pdata = pd;
    sfb->bus_clk = clk_get(dev, "lcd");
    // ..... 省略错误处理
    clk_enable(sfb->bus_clk);
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0); // 获得私有数据
    // ..... 省略错误处理
    sfb->regs_res = request_mem_region(res->start, resource_size(res),
        dev_name(dev));
    // ..... 省略错误处理
    sfb->regs = ioremap(res->start, resource_size(res)); // 基地址
    // ..... 省略错误处理
    pd->setup_gpio();    // ..... 设置显示相关 GPIO
    writel(pd->vidcon1, sfb->regs + VIDCON1);
    for (win = 0; win < S3C_FB_MAX_WIN; win++)
        s3c_fb_clear_win(sfb, win);
    for (win = 0; win < S3C_FB_MAX_WIN; win++) {
        if (!pd->win[win]) continue;
        ret = s3c_fb_probe_win(sfb, win, &sfb->windows[win]); // 完成实际注册
    }
    // ..... 省略错误处理
}
```

```

platform_set_drvdata(pdev, sfb);
return 0;
// ..... 省略错误处理
}

```

此处的驱动实现取出了特定设备的寄存器基地址（IORESOURCE_MEM），其含义就是不同硬件（此处指 S3C 系列的各个处理器）中的 FrameBuffer 部分的功能相同，但是寄存器的基地址可能不同。类似的可变动值还有中断（IORESOURCE_IRQ）。一个可以针对多个硬件的驱动的处理方法都与之类似，其目的就是为了让一个驱动程序的实现可以为不同的硬件使用，而不需要更改其源代码，只需要更改针对不同硬件的平台设备即可。

s3c_fb_probe()调用 s3c_fb_probe_win()函数完成了实际的注册。由于在某个系统中可能有多个层的显示支持，因此调用是一个循环。

s3c_fb_probe_win()函数的主体如下所示：

```

static int __devinit s3c_fb_probe_win(struct s3c_fb *sfb, unsigned int win_no,
                                     struct s3c_fb_win **res)
{
    struct fb_var_screeninfo *var;
    struct fb_videomode *initmode;
    struct s3c_fb_pd_win *windata;
    struct s3c_fb_win *win;
    struct fb_info *fbinfo;
    int palette_size;
    int ret;
    palette_size = s3c_fb_win_pal_size(win_no);
    fbinfo = framebuffer_alloc(sizeof(struct s3c_fb_win) +
                              palette_size * sizeof(u32), sfb->dev); // 分配 fb_info 结构
// ..... 省略错误处理
    windata = sfb->pdata->win[win_no];
    initmode = &windata->win_mode;
// ..... 省略错误处理
    win = fbinfo->par;           // 把信息保存在 s3c_fb_pd_win 结构中
    var = &fbinfo->var;
    win->fbinfo = fbinfo;
    win->parent = sfb;
    win->windata = windata;
    win->index = win_no;
    win->palette_buffer = (u32 *) (win + 1);
    ret = s3c_fb_alloc_memory(sfb, win);           // ..... 分配内部数据结构
// ..... 省略错误处理
    s3c_fb_init_palette(win_no, &win->palette); // S3C 处理器特殊的 RGB 调色板
    fb_videomode_to_var(&fbinfo->var, initmode);
    fbinfo->fix.type = FB_TYPE_PACKED_PIXELS; // 为 fb_info 结构的成员赋值
    fbinfo->fix.accel = FB_ACCEL_NONE;
    fbinfo->var.activate = FB_ACTIVATE_NOW;
    fbinfo->var.vmode = FB_VMODE_NONINTERLACED;
    fbinfo->var.bits_per_pixel = windata->default_bpp;
    fbinfo->fbops = &s3c_fb_ops;           // 设置 fb_ops
    fbinfo->flags = FBINFO_FLAG_DEFAULT;
    fbinfo->pseudo_palette = &win->pseudo_palette;
    ret = s3c_fb_check_var(&fbinfo->var, fbinfo);
// ..... 省略错误处理
    ret = fb_alloc_cmap(&fbinfo->cmap, s3c_fb_win_pal_size(win_no), 1);
    if (ret == 0)
        fb_set_cmap(&fbinfo->cmap, fbinfo); // 设置内存映射
}

```

```

else
    dev_err(sfb->dev, "failed to allocate fb cmap\n");
s3c_fb_set_par(fbinfo);
ret = register_framebuffer(fbinfo);
// ..... 省略错误处理
*res = win;
return 0;
}

```

s3c_fb_ops 是 fb_ops 类型的结构，其中 fb_check_var、fb_set_par、fb_blank 和 fb_setcolreg 几个函数指针是 S3C 系统单独的实现。

s3c_fb_blank 是 fb_ops 结构中的 fb_blank 函数指针，其实现片段如下所示：

```

static int s3c_fb_blank(int blank_mode, struct fb_info *info)
{
    struct s3c_fb_win *win = info->par;          // 获取私有的数据
    struct s3c_fb *sfb = win->parent;
    unsigned int index = win->index;
    u32 wincon;
    wincon = readl(sfb->regs + WINCON(index));
    switch (blank_mode) {
    case FB_BLANK_POWERDOWN:
        wincon &= ~WINCONx_ENWIN;
        sfb->enabled &= ~(1 << index);
    case FB_BLANK_NORMAL:
        writel(WINxMAP_MAP | WINxMAP_MAP_COLOUR(0x0),
            sfb->regs + WINxMAP(index));          // 写寄存器控制硬件
        break;
    // ..... 省略其他情况
    }
    writel(wincon, sfb->regs + WINCON(index));    // 写寄存器控制硬件
    if (index == 0)
        s3c_fb_enable(sfb, blank_mode != FB_BLANK_POWERDOWN ? 1 : 0); // 进行使能
    return 0;
}

```

此处使用的 sfb->regs 就是根据平台设备得到的寄存器的基地址，用它加上具体寄存器的偏移量，可以完成对硬件的操作。

与驱动程序相配合的就是平台设备，需要根据处理器定义不同的资源。例如：一个平台设备的代码为 arch/arm/plat-samsung/dev-fb.c，主体的内容如下所示：

```

static struct resource s3c_fb_resource[] = {
    [0] = {                                          // 显示设备的寄存器基地址
        .start = S3C_PA_FB,
        .end   = S3C_PA_FB + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {                                          // 显示设备的垂直同步中断
        .start = IRQ_LCD_VSYNC,
        .end   = IRQ_LCD_VSYNC,
        .flags = IORESOURCE_IRQ,
    },
    [2] = {                                          // 显示设备的 FIFO 中断
        .start = IRQ_LCD_FIFO,
        .end   = IRQ_LCD_FIFO,
        .flags = IORESOURCE_IRQ,
    },
}

```

```

    },
    [3] = {
        .start = IRQ_LCD_SYSTEM,           // 写显示设备的 LCD 系统中断
        .end   = IRQ_LCD_SYSTEM,
        .flags = IORESOURCE_IRQ,
    },
};
struct platform_device s3c_device_fb = {
    .name       = "s3c-fb",                // 平台设备的名称
    .id         = -1,
    .num_resource = ARRAY_SIZE(s3c_fb_resource),
    .resource    = s3c_fb_resource,
    .dev.dma_mask = &s3c_device_fb.dev.coherent_dma_mask,
    .dev.coherent_dma_mask = 0xffffffffUL,
};

```

此处板级定义的平台设备的名称为"s3c-fb"，与驱动中平台驱动的名称相同，因此可以实现匹配，资源有四个：一个内存类型的地址和三个中断地址。在 S3C 系列的处理器中，这些资源的值可能不同。

17.3 输入-事件驱动框架

17.3.1 输入-事件驱动框架概述

输入（Input）驱动程序是 Linux 输入设备的驱动程序，分成游戏杆（joystick）、鼠标（mouse 和 mice）和事件设备（Event）3 种驱动程序。目前主要使用的是 Event 设备。实际的输入设备通常基于中断，因此输入设备驱动程序的框架通常需要将中断的异步信息转换成对用户空间的汇报。

输入驱动的主设备号是 13，次设备号的分配情况如下所示。

- 0~31: joystick 游戏杆。
- 33~62: mouse 鼠标。
- 63: mice 鼠标。
- 64~95: 事件设备（Event）。

每种类型的输入设备占用 5 位，因此每种设备的个数是 32 个。输入设备的设备节点通常在 /dev/input 目录中，其中可能有 by-id 和 by-path 两个子目录，它们表示按照不同方式的分类，其中的内容为到真实设备节点的连接。

Input 驱动框架的结构如图 17-4 所示。

输入设备驱动程序的头文件为：include/linux/input.h。

输入设备驱动框架和 Event 的代码分别为 drivers/input/ 目录当中的 input.c 和 evdev.c。

Event 设备是 Input 设备中的一种，可支持键盘、鼠标、触摸屏等多种输入设备。其主设备号为 13；次设备号递增生成，为 64~95。各个具体设备在 misc、touchscreen、keyboard 等目录中。

Event 设备在用户空间大多使用 read、ioctl、poll 等文件系统的接口进行操作。read 用

于读取输入信息。`ioctl` 用于获得和设置信息。`poll` 调用可以进行用户空间的阻塞。当内核有按键等中断时，通过在中断中唤醒 `poll` 的内核实现，这样在用户空间的 `poll` 调用也可以返回。

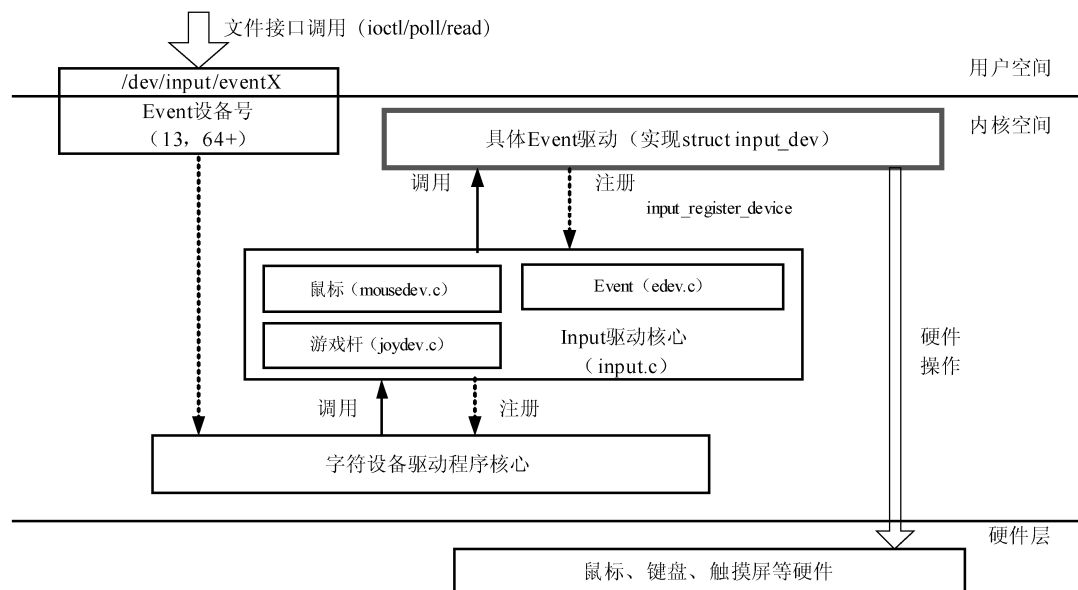


图 17-4 Input 驱动框架的结构

`input.h` 文件中定义的控制命令如下所示：

```
#define EVIOCGVERSION _IOR('E', 0x01, int) // 获得驱动的版本
#define EVIOCGID _IOR('E', 0x02, struct input_id) // 获得设备的 ID
#define EVIOCGREP _IOR('E', 0x03, unsigned int[2]) // 获得重复的设置
#define EVIOCSREP _IOW('E', 0x03, unsigned int[2]) // 设置重复的设置
#define EVIOCGKEYCODE _IOR('E', 0x04, unsigned int[2]) // 获得按键码
#define EVIOCSKEYCODE _IOW('E', 0x04, unsigned int[2]) // 设置按键码
// ..... 省略其他内容
```

这些定义的数值，就是对每个 Input 设备的控制命令，这些命令在不同 Input 设备中的实现有所不同。

`input_dev` 结构表示 Input 驱动程序的各种信息，结构内容如下所示：

```
input struct input_dev {
    const char *name;      const char *phys;
    const char *uniq;      struct input_id id;
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; // 通用的事件位
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; // 按键设备位
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; // 相对设备位
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; // 绝对设备位
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; // 杂项设备位
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; // LED 设备位
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; // 声音设备位
    unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)]; // 反馈设备 (force feedback) 位
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; // 开关 (switches) 设备位
    unsigned int keycodemax; unsigned int keycodesize; void *keycode;
```

```

int (*setkeycode)(struct input_dev *dev,          // 设置按键码
                  unsigned int scancode, unsigned int keycode);
int (*getkeycode)(struct input_dev *dev,          // 获得按键码
                  unsigned int scancode, unsigned int *keycode);
// ..... 省略其他内容
}
struct input_dev *input_allocate_device(void);
void input_free_device(struct input_dev *dev);
int __must_check input_register_device(struct input_dev *);
void input_unregister_device(struct input_dev *);

```

`input_dev` 实际上是输入设备信息的全集，例如对于事件设备，其中分为同步设备、按键设备、相对设备（例如：鼠标）、绝对设备（例如：触摸屏）等。

`input_handler` 结构表示输入设备的处理者，`input_handle` 结构表示一个连接到 `input_handler` 的设备句柄。

Event 设备是 Input 设备中的一种，其中实现了一个 `input_handler` 并注册。

而在 Input 设备的核心实现中，定义了一个名为 `input_devices_poll_wait` 的队列，用于阻塞和唤醒。如输入设备的文件操作 `file_operations` 中实现了 `poll` 函数指针，并通过调用 `poll_wait()` 函数实现阻塞，因此，输入设备一般都支持用户空间的 `poll` 调用。

`input_event()` 等函数为 Event 设备实现，用于在具体的 Event 驱动程序中上报事件。

17.3.2 针对硬件的事件驱动

具体硬件设备的 Event 驱动的种类很多，分别位于 `joystick`、`mouse`、`touchscreen` 等几个目录中。根据硬件特性的不同，有些 Event 驱动只是针对某些处理器当中的专用实现，有些 Event 驱动却是基于连接与 I2C、SPI 等总线的外部控件，其驱动的实现可以在不同的系统中使用。

基于 GPIO 的 Event 驱动程序的代码为：`drivers/input/keyboard/gpio_keys.c`，还有一个相关的头文件为：`include/linux/gpio_keys.h`。GPIO 按键驱动可以针对所有基于 GPIO 的矩阵键盘，在不同的系统中根据平台设备的不同来进行配置。

`gpio_keys.h` 中定义的数据结构如下所示：

```

struct gpio_keys_button {
    int code;          // Event 码
    int gpio;          // 与 GPIO 的对应内容
    int active_low;
    char *desc;
    int type;          // 输入事件的类型 (EV_KEY, EV_SW)
    int wakeup;        // 唤醒
    int debounce_interval; /* debounce ticks interval in msecs */
    bool can_disable;
};
struct gpio_keys_platform_data {
    struct gpio_keys_button *buttons;
    int nbuttons;
    unsigned int rep:1; // 使能自动重复功能
};

```

`gpio_keys_platform_data` 结构中的 `gpio_keys_button` 指针应为一个数组的地址，`nbuttons`

表示这个数组的数目。需要在板级支持的平台设备中声明，然后在 GPIO 的驱动中取出信息来配置某个特定驱动的实现。

GPIO 按键驱动平台设备的 `probe()` 函数主要内容如下所示：

```
static int __devinit gpio_keys_probe(struct platform_device *pdev)
{
    struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
    struct gpio_keys_drvdata *ddata;           // 表示输入设备数据的结构
    struct device *dev = &pdev->dev;
    struct input_dev *input;
    int i, error;
    int wakeup = 0;
    ddata = kzalloc(sizeof(struct gpio_keys_drvdata) +
                    pdata->nbuttons * sizeof(struct gpio_button_data), GFP_KERNEL);
    input = input_allocate_device();           // 分配一个输入设备
    // ..... 省略错误处理
    ddata->input = input;                     // 输入设备数据的构建
    ddata->n_buttons = pdata->nbuttons;
    mutex_init(&ddata->disable_lock);
    platform_set_drvdata(pdev, ddata);        // 保存平台数据
    input->name = pdev->name;                  // input_dev 数据结构的设置
    input->phys = "gpio-keys/input0";
    input->dev.parent = &pdev->dev;
    input->id.bustype = BUS_HOST;
    input->id.vendor = 0x0001;
    input->id.product = 0x0001;
    input->id.version = 0x0100;
    if (pdata->rep) __set_bit(EV_REP, input->evbit);
    for (i = 0; i < pdata->nbuttons; i++) {    // 各个按键的处理
        struct gpio_keys_button *button = &pdata->buttons[i];
        struct gpio_button_data *bdata = &ddata->data[i];
        unsigned int type = button->type ?: EV_KEY; // 默认类行为按键
        bdata->input = input;
        bdata->button = button;
        error = gpio_keys_setup_key(pdev, bdata, button); // 按键的实际构建
    }
    // ..... 省略错误处理
    if (button->wakeup) wakeup = 1;
    input_set_capability(input, type, button->code);
}
// ..... 省略部分内容
error = input_register_device(input);        // 注册输入设备
// ..... 省略错误处理
for (i = 0; i < pdata->nbuttons; i++)
    gpio_keys_report_event(&ddata->data[i]); // 根据按键报告事件
input_sync(input);
device_init_wakeup(&pdev->dev, wakeup);
return 0;
// ..... 省略错误处理
}
```

`probe()` 函数最核心的功能是在一个循环中完成了各个按键的注册，也就是根据 `nbuttons` 遍历 `gpio_keys_platform_data` 结构，对其中的每个按键进行事件的注册处理。

`gpio_keys_setup_key()` 是实际的 GPIO 的按键初始化函数，其实现的主体如下所示：

```
static int __devinit gpio_keys_setup_key(struct platform_device *pdev,
                                         struct gpio_button_data *bdata, struct gpio_keys_button *button)
```

```

{
    char *desc = button->desc ? button->desc : "gpio_keys";
    struct device *dev = &pdev->dev;
    unsigned long irqflags;
    int irq, error;
    setup_timer(&bdata->timer, gpio_keys_timer, (unsigned long)bdata);
    INIT_WORK(&bdata->work, gpio_keys_work_func); // 设置一个处理的队列
    error = gpio_request(button->gpio, desc);
    // ..... 省略错误处理
    error = gpio_direction_input(button->gpio); // 设置 GPIO 的方向为输入
    // ..... 省略错误处理
    irq = gpio_to_irq(button->gpio); // 找到 GPIO 对应的中断
    // ..... 省略错误处理
    irqflags = IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING;
    if (!button->can_disable) irqflags |= IRQF_SHARED;
    error = request_irq(irq, gpio_keys_isr, irqflags, desc, bdata); // 注册中断
    // ..... 省略错误处理
    return 0;
    // ..... 省略错误处理
}

```

函数执行的最后，会对不同的按键多次注册中断。每次注册使用同一个中断号，但是传给中断处理函数的参数不同，它表示按键数据（`gpio_button_data`）。

`gpio_keys_isr()`被注册为按键中断处理函数，其内容如下所示：

```

static irqreturn_t gpio_keys_isr(int irq, void *dev_id)
{
    struct gpio_button_data *bdata = dev_id;
    struct gpio_keys_button *button = bdata->button; // 得到按键数据
    BUG_ON(irq != gpio_to_irq(button->gpio));
    if (button->debounce_interval)
        mod_timer(&bdata->timer,
            jiffies + msecs_to_jiffies(button->debounce_interval));
    else
        schedule_work(&bdata->work); // 调度其中的队列
    return IRQ_HANDLED;
}

```

被调度的队列就是由 `gpio_keys_work_func()` 函数所实现的，作为处理按键事件的队列，其中又调用了 `gpio_keys_report_event()` 函数，此函数的内容如下所示：

```

static void gpio_keys_report_event(struct gpio_button_data *bdata)
{
    struct gpio_keys_button *button = bdata->button;
    struct input_dev *input = bdata->input;
    unsigned int type = button->type ?: EV_KEY;
    int state = (gpio_get_value(button->gpio) ? 1 : 0) ^ button->active_low;
    input_event(input, type, button->code, !!state); // 上报事件
    input_sync(input);
}

```

`gpio_keys_report_event()` 函数用于按键事件的上报，它将在按键的中断发生后被调用。其中逻辑就是获取到按键类型和具体的按键，调用 `input_event()` 函数进行上报，上报的按键码就来自那个按键。

17.4 GPIO 驱动框架和具体驱动

GPIO（General Purpose Input Output）的含义为通用输入/输出。GPIO 可以作为端口扩展器。从硬件上说，SOC 提供一些独立的引脚，某些用于输入，某些用于输出，利用高低电平表示 0 或者 1。可以通过 SOC 的寄存器进行读取和控制，也就是输入和输出。

17.4.1 GPIO 驱动框架

如果仅仅针对 GPIO 硬件构建驱动程序，实现一个字符驱动就可以完成，对用户空间的接口可以是 read、write 或者自定义的 ioctl。内核中却提供了 GPIO 框架，作为 GPIO 驱动的统一架构，这样做的好处主要有以下几点：

- 复用公共代码。
- 使用 sys 文件系统作为到用户空间的统一接口，方便操作。
- 同时提供内核中的 API，让内核的其他部分调用。

GPIO 驱动的结构如图 17-5 所示。

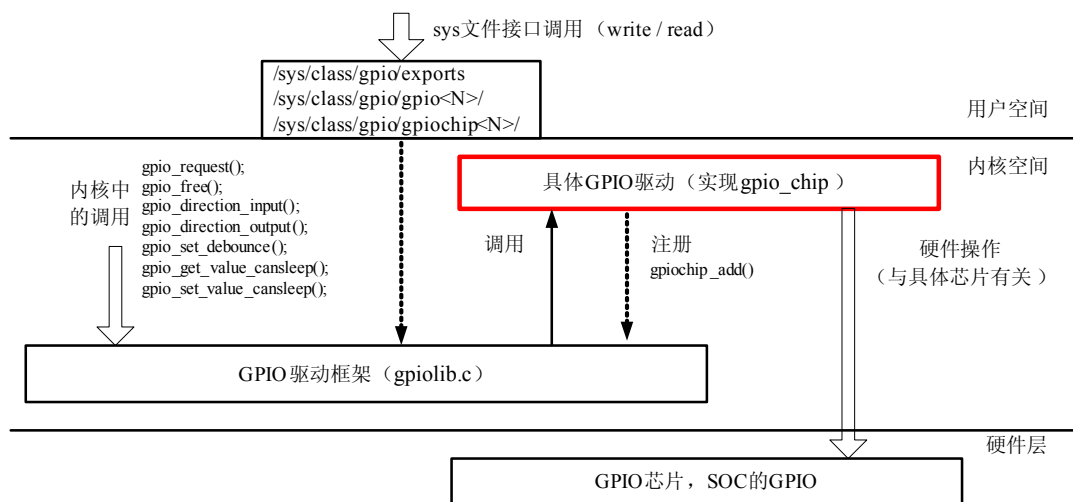


图 17-5 GPIO 驱动的结构

GPIO 驱动的头文件为：include/linux/gpio.h 和 include/asm-generic/gpio.h。

GPIO 驱动框架的实现代码为：drivers/gpio/gpio-lib.c。

GPIO 驱动框架不是设备节点，而是 sys 文件系统，其根目录为：/sys/class/gpio/。

/sys/class/gpio/目录中有一个用于写 export 的文件，向其中写入一个数字表示建立一个 GPIO 端口的子目录 gpio<N>，例如：

```
$ echo 19 > export
```

如下操作将在/sys/class/gpio/目录中建立一个 gpio19 子目录，这个子目录中就包含针对 19 号 GPIO 的操作。

/sys/class/gpio/gpio<N>目录当中包括了以下文件。

- **direction**: 可读写, 使用"in"和"out"两个值表示输入和输出。
- **value**: 可读写, 0 代表低电平, 1 (或其他非 0) 代表高电平。
- **edge**: 表示上升沿和下降沿, 可以为"none"、"rising"、"falling"。

/sys/class/gpio/gpiochip<N>/目录描述一个 GPIO 芯片创建信息, 其中包括如下几个文件。

- **base**: 表示首个 GPIO 从此数字开始。
- **ngpio**: 表示 GPIO 的个数。
- **label**: 提供诊断信息。

gpio.h 头文件中描述 GPIO 芯片的结构 gpio_chip 如下所示:

```
struct gpio_chip {
    const char      *label;           // 该文件的描述信息
    struct device    *dev;
    struct module    *owner;
    int              (*request)(struct gpio_chip *chip, unsigned offset);
    void             (*free)(struct gpio_chip *chip, unsigned offset);
    int              (*direction_input)(struct gpio_chip *chip, unsigned offset);
    int              (*get)(struct gpio_chip *chip, unsigned offset);
    int              (*direction_output)(struct gpio_chip *chip,
                                         unsigned offset, int value);
    int              (*set_debounce)(struct gpio_chip *chip,
                                     unsigned offset, unsigned debounce);
    void             (*set)(struct gpio_chip *chip, unsigned offset, int value);
    int              (*to_irq)(struct gpio_chip *chip, unsigned offset);
    void             (*dbg_show)(struct seq_file *s, struct gpio_chip *chip);
    int              base;            // 表示首个 GPIO 从此数字开始
    u16              ngpio;           // 表示 GPIO 的个数
    const char      *const *names;   // 表示其名称
    unsigned         can_sleep:1;
    unsigned         exported:1;
};
```

gpio_chip 用于构建 sys 文件系统 gpiochip<N>当中的每一个目录, 也是在构建具体的 GPIO 驱动时的实现单元。

提示: 各个 gpio_chip 实现的 GPIO 不应当重复。

完成一个 GPIO 芯片的注册和注销功能的函数如下所示:

```
extern int gpiochip_add(struct gpio_chip *chip);
extern int __must_check gpiochip_remove(struct gpio_chip *chip);
```

若干个函数用于控制 GPIO, 如下所示:

```
extern int gpio_request(unsigned gpio, const char *label);
extern void gpio_free(unsigned gpio);
extern int gpio_direction_input(unsigned gpio);
extern int gpio_direction_output(unsigned gpio, int value);
extern int gpio_set_debounce(unsigned gpio, unsigned debounce);
extern int gpio_get_value_cansleep(unsigned gpio);
extern void gpio_set_value_cansleep(unsigned gpio, int value);
```

gpio_request()等函数实际上是内核的调用接口，可以在各个驱动程序或者板级的支持中调用，完成 GPIO 的统一控制。

17.4.2 GPIO 具体硬件的驱动

drivers/gpio/目录中名为<gpio.c 的各个文件都是针对具体硬件的 GPIO 实现。具体的实现方式都是构建 gpio_chip 结构，并实现其中的各个函数指针，这些函数指针要通过针对具体硬件控制或读取来实现。

针对不同的硬件，GPIO 的实现有几个类型：

- 出自 SOC 芯片的 GPIO 控制器，其 GPIO 通常都由 SOC 统一控制，并无硬件 GPIO 芯片的概念，因此此时的 gpiochip 更多地表示为“一组 GPIO”。
- 独立的 GPIO 芯片：它们可以连接于外部总线，例如：I2C、SPI、PCI 等。
- 其他带有 GPIO 功能的芯片。

17.5 Power Supply 驱动框架和具体驱动

Power Supply 驱动程序用于让用户空间可以读取系统中的供电设备信息。供电设备可以是直流电源（AC）、USB 或者电池等。

17.5.1 Power Supply 驱动框架

Power Supply 驱动程序与用户空间的接口是 sys 文件系统，该类型驱动程序的目录为 /sys/class/power_supply/，其中的每个子目录表示一种供电设备的名称。

Power Supply 驱动头文件为：include/linux/power_supply.h；Power Supply 驱动框架的代码为：drivers/power/power_supply_core.c 和 drivers/power/power_supply_sysfs.c。

power_supply.h 文件中 Power Supply 的注册和注销函数如下所示：

```
int power_supply_register(struct device *parent, struct power_supply *psy);
void power_supply_unregister(struct power_supply *psy);
```

其中，power_supply 结构体为驱动程序需要实现的部分，其内容如下所示：

```
struct power_supply {
    const char *name; // 设备名称
    enum power_supply_type type; // 类型
    enum power_supply_property *properties; // 属性指针
    size_t num_properties; // 属性的数目
    char **supplied_to;
    size_t num_suppliants;
    int (*get_property)(struct power_supply *psy, // 获得属性
        enum power_supply_property psp, union power_supply_propval *val);
    void (*external_power_changed)(struct power_supply *psy);
    // ..... 省略部分内容
};
```

一个 Power Supply 驱动要实现 get_property 和 external_power_changed 这两个函数，而其名称对应于 /sys/class/power_supply/ 目录中所建立的子目录。power_supply_property 则是

一系列用枚举值表示的属性，也就是每个供电设备子目录当中的文件名。supplied_to 表示为哪个设备进行供电，通常指某个电池。

17.5.2 Power Supply 驱动

Power Supply 硬件系统中的供电设备各不相同，其驱动程序的实现方式也各异。有些 Power Supply 驱动只提供了简单的信息获取功能，有些 Power Supply 驱动却可以从中断中得到供电信息的变化。

drivers/power/目录当中的各个文件都是供电设备的驱动，其中 pda_power.c 是一个 PDA (Personal Digital Assistant, 指掌上电脑) 设备的通用供电驱动实现，包括 ac 和 usb 两个子设备。

其平台设备的探测函数如下所示：

```
static int pda_power_probe(struct platform_device *pdev)
{
    int ret = 0;
    dev = &pdev->dev;
    // ..... 省略错误处理
    pdata = pdev->dev.platform_data;
    // ..... 省略部分内容
    update_status();    // 更新状态信息：是否在线
    update_charger();   // 更新充电信息
    if (!pdata->wait_for_status)    pdata->wait_for_status = 500;
    if (!pdata->wait_for_charger)   pdata->wait_for_charger = 500;
    if (!pdata->polling_interval)   pdata->polling_interval = 2000;
    if (!pdata->ac_max_uA)          pdata->ac_max_uA = 500000;
    setup_timer(&charger_timer, charger_timer_func, 0);
    setup_timer(&supply_timer, supply_timer_func, 0);
    ac_irq = platform_get_resource_byname(pdev, IORESOURCE_IRQ, "ac");
    usb_irq = platform_get_resource_byname(pdev, IORESOURCE_IRQ, "usb");
    // ..... 省略部分内容
    ac_draw = regulator_get(dev, "ac_draw");
    // ..... 省略错误处理
    if (pdata->is_ac_online) { // AC 的注册
        ret = power_supply_register(&pdev->dev, &pda_psy_ac);
    // ..... 省略错误处理
        if (ac_irq) {
            ret = request_irq(ac_irq->start, power_changed_isr,
                              get_irq_flags(ac_irq), ac_irq->name, &pda_psy_ac);
        // ..... 省略错误处理
        } else {
            polling = 1;
        }
    }
    // ..... 省略部分内容
    if (pdata->is_usb_online) { // USB 的注册
        ret = power_supply_register(&pdev->dev, &pda_psy_usb);
    // ..... 省略错误处理
        if (usb_irq) {
            ret = request_irq(usb_irq->start, power_changed_isr,
                              get_irq_flags(usb_irq), usb_irq->name, &pda_psy_usb);
        // ..... 省略错误处理
        } else {
            polling = 1;
        }
    }
    if (polling) {
        setup_timer(&polling_timer, polling_timer_func, 0);
    }
}
```

```

        mod_timer(&polling_timer,
                jiffies + msecs_to_jiffies(pdata->polling_interval));
    }
    if (ac_irq || usb_irq)        device_init_wakeup(&pdev->dev, 1);
    return 0;
// ..... 省略错误处理
}

```

本驱动中实际上调用了其他硬件系统的函数，从而获取到供电信息。

ac 和 usb 两个供电设备的 power_supply 的定义如下所示：

```

static struct power_supply pda_psy_ac = {
    .name = "ac",
    .type = POWER_SUPPLY_TYPE_MAINS,
    .supplied_to = pda_power_supplied_to,
    .num_suppllicants = ARRAY_SIZE(pda_power_supplied_to),
    .properties = pda_power_props,
    .num_properties = ARRAY_SIZE(pda_power_props),
    .get_property = pda_power_get_property,
};
static struct power_supply pda_psy_usb = {
    .name = "usb",
    .type = POWER_SUPPLY_TYPE_USB,
    .supplied_to = pda_power_supplied_to,
    .num_suppllicants = ARRAY_SIZE(pda_power_supplied_to),
    .properties = pda_power_props,
    .num_properties = ARRAY_SIZE(pda_power_props),
    .get_property = pda_power_get_property,
};

```

此处的 pda_power_props 成员只有一个表示是否在线信息 (online) 的成员；supplied_to 有两个成员：主电池 ("main-battery") 和备用电池 ("backup-battery")。

17.6 TTY 驱动框架和驱动

TTY 是 TeleTYpe 的缩写。在 Linux 中，TTY 表示“终端”的驱动程序框架，这是一类驱动程序的统称。串口等驱动都通过 TTY 驱动实现。

17.6.1 TTY 驱动框架

TTY 驱动基于字符设备，并占用多个主设备号。TTY 驱动自身有基本的框架，并且还有伪终端、串口终端、虚拟终端等衍生结构。

TTY 驱动结构如图 17-6 所示。

在 Linux 中，TTY 设备也是字符设备，它将占用多个主设备号，如下所示。

- 2 (PTY_MASTER_MAJOR): Pseudo-TTY masters, 伪 TTY 主设备，设备节点的形式为 /dev/ptyp<N> 等。
- 3 (PTY_SLAVE_MAJOR): Pseudo-TTY slaves, 伪 TTY 从设备，设备节点的形式为 /dev/ttyp<N> 等。


```
int (*install)(struct tty_driver *driver, struct tty_struct *tty);
void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
int (*open)(struct tty_struct * tty, struct file * filp);
void (*close)(struct tty_struct * tty, struct file * filp);
void (*shutdown)(struct tty_struct *tty);
int (*write)(struct tty_struct * tty,
             const unsigned char *buf, int count);
int (*put_char)(struct tty_struct *tty, unsigned char ch);
void (*flush_chars)(struct tty_struct *tty);
int (*write_room)(struct tty_struct *tty);
int (*chars_in_buffer)(struct tty_struct *tty);
int (*ioctl)(struct tty_struct *tty, struct file * file,
             unsigned int cmd, unsigned long arg);
long (*compat_ioctl)(struct tty_struct *tty, struct file * file,
                    unsigned int cmd, unsigned long arg);
// .....省略部分操作
};
```

`tty_operations` 当中的各个函数指针主要表示了 TTY 设备的操作，其中的 `open`、`write`、`ioctl` 等函数指针的含义与普通文件操作类似，但是操作的句柄换成了 `tty_struct`。`tty_struct` 结构用于保存一个被打开的 TTY 设备的所有信息，也就是进程所打开的 TTY 设备上下文。

注册和注销函数在 `tty.h` 中定义：

```
extern int tty_register_driver(struct tty_driver *driver);
extern int tty_unregister_driver(struct tty_driver *driver);
extern struct device *tty_register_device(struct tty_driver *driver,
                                         unsigned index, struct device *dev);
extern void tty_unregister_device(struct tty_driver *driver, unsigned index);
```

在 TTY 的核心实现是 `tty_io.c` 文件，其中 `tty` 部分对用户空间的端口也实现了一系列的特殊 `ioctl` 控制命令，这些命令常以 `TIO` 为开头。

在 `tty_io.c` 中定义的初始化过程中，已经注册了 `/dev/tty` 设备和 `/dev/console` 设备，这是两个在内部实现的 TTY 设备，它们的主设备号为 5，次设备号分别为 0 和 1。

17.6.2 伪 TTY 驱动

伪终端（Pseudo Terminal）是成对的逻辑终端设备，它们不和物理设备相连，常用于虚拟的终端操作。伪终端分为 `master`（主）和 `slave`（从）设备，对 `master` 的操作会反映到 `slave` 上。例如：当使用 `telnet` 连接到一台机器的时候，它连接的实际就是一个伪终端，对伪终端的读、写和控制，就类似对一个串口的操作，也可以发送和接收字符。

伪终端的源代码文件为：`driver/tty/pty.c`。文件中构建了两个 `tty_driver`，名称分别为“`pty_master`”和“`pty_slave`”，设备名分别为“`pty`”和“`ttyp`”。其中使用的方法被称为 `Unix98`。

17.6.3 串口 TTY 和虚拟 TTY

串口 TTY 和虚拟 TTY 是两种实际常用的 TTY。串口 TTY 基于串口构建，设备节点的形式为 `/dev/ttyS<N>` 的形式，一般主设备号为 4，`<N>` 一般为 64~255。虚拟 TTY 用于给非 TTY 的设备提供 TTY 形式的接口，一般主设备号为 4，`<N>` 一般为 0~63。

串口 TTY 的代码为：`drivers/serial/serial_core.c`。

虚拟 TTY 的头文件为：include/linux/vc.h 和 console.h，代码为：drivers/char/vt.c。

虚拟 TTY 实际上是一个基于 TTY 框架之上的框架。

console.h 文件中定义了一个 consw 结构体，如下所示：

```
struct consw {
    struct module *owner;
    const char *(*con_startup)(void);
    void (*con_init)(struct vc_data *, int);
    void (*con_deinit)(struct vc_data *);
    void (*con_clear)(struct vc_data *, int, int, int, int, int);
    void (*con_putc)(struct vc_data *, int, int, int);
    void (*con_putcs)(struct vc_data *, const unsigned short *, int, int, int);
    void (*con_cursor)(struct vc_data *, int);
    int (*con_scroll)(struct vc_data *, int, int, int, int);
    void (*con_bmove)(struct vc_data *, int, int, int, int, int, int);
    int (*con_switch)(struct vc_data *);
    int (*con_blank)(struct vc_data *, int, int);
    int (*con_font_set)(struct vc_data *, struct console_font *, unsigned);
    int (*con_font_get)(struct vc_data *, struct console_font *);
    int (*con_font_default)(struct vc_data *, struct console_font *, char *);
    int (*con_font_copy)(struct vc_data *, int);
    int (*con_resize)(struct vc_data *, unsigned int, unsigned int, unsigned int);
    int (*con_set_palette)(struct vc_data *, unsigned char *);
    int (*con_scrolldelta)(struct vc_data *, int);
    int (*con_set_origin)(struct vc_data *);
    void (*con_save_screen)(struct vc_data *);
    u8 (*con_build_attr)(struct vc_data *, u8, u8, u8, u8, u8, u8);
    void (*con_invert_region)(struct vc_data *, u16 *, int);
    u16 *(*con_screen_pos)(struct vc_data *, int);
    unsigned long (*con_getxy)(struct vc_data *, unsigned long, int *, int *);
};
```

consw 结构当中的各个函数指针表示对控制台的操作，参数的类型为 vc_data，vc_data 表示的是虚拟终端中传输的数据。

还有一些函数完成 consw 结构的注册和注销等，如下所示：

```
int con_is_bound(const struct consw *csw);
int register_con_driver(const struct consw *csw, int first, int last);
int unregister_con_driver(const struct consw *csw);
int take_over_console(const struct consw *sw, int first, int last, int deflt);
void give_up_console(const struct consw *sw);
```

consw 结构表示一个终端操作，需要由虚拟终端的实现者来构建。由于虚拟终端当中所谓的终端操作具有不同的含义，因此需要自己实现每个终端操作的细节。例如：终端中输出字符的操作需要用 con_putc 实现，清屏的操作需要用 con_clear 实现。调用 register_con_driver() 函数注册一个虚拟终端之后，将会形成 tty<N> 设备。

用帧缓冲 (FrameBuffer) 驱动构建的虚拟终端是一种典型的情况，可以将终端的操作转化为屏幕上的显示，这种操作不同于串行的典型终端操作。帧缓冲终端功能的代码在 driver/video/console/ 目录中。主文件为 fbcon.c，实现的核心就是将终端的操作，转为调用 FrameBuffer 的接口实现屏幕的显示。为了显示字符，其中还有一些字符转位图的字库文件，例如 font_<name>.c 等。

第 18 章

MTD 系统和驱动

18.1 MTD 概述

MTD (Memory Technology Device) 被称为内存技术设备, 通常用于为 ROM、RAM、Flash 以及类似的存储设备构建驱动程序。MTD 不仅仅是一个驱动程序的框架, 也是 Linux 其中的一个介于硬件和文件系统之间的子系统。

MTD 的网站为: <http://www.linux-mtd.infradead.org/>。

MTD 可以主要用作闪存 (Flash 存储器) 的实现。闪存是一种非易失性存储器, 根据结构的不同可以将其分成 Nor Flash 和 Nand Flash 两种。

Nor Flash 和 Nand Flash 的情况及其区别包括下面几个方面:

- Nor Flash 连接于地址总线 and 数据总线。
- Nand Flash 使用 IO 接口, 地址和数据复用。
- Nor Flash 可以支持 XIP (eXecute In Place), 在芯片中直接运行。
- Nor Flash 分为扇区 (Sector), Nand Flash 分为块 (Block), 写入前要擦除。
- Nor Flash 的读速度比 Nand Flash 稍快一些。
- Nand Flash 的写入速度和擦除速度比 Nor Flash 快很多。
- Nand Flash 的集成度高、造价低。

Nand Flash 的几个相关概念如下所示。

- BBT: 坏块表 (Bad Block Table), 用于标识系统中的所有坏块。
- OOB: Nand 带外数据块 (Out Of Band), Nand Flash 当中的一块额外存储区域。
- ECC: 错误检查和纠正 (Error Correcting Code)。

MTD 架构可以支持带有 ECC 或者不带有 ECC 的 Nand Flash、Nor Flash, 还可以支持 OneNand Flash 等。

18.2 MTD 的核心

Linux 系统的 MTD 驱动的核心部分是 MTD 原始设备。具体的 Flash 等各种存储器基于 MTD 原始设备的接口来实现,然后 MTD 子系统将根据所实现的接口把设备集成到 Linux 系统中。

MTD 原始设备的实现,主要用于构建文件系统,也可以通过设备节点给用户空间提供接口。JFFS2、YAFFS 等用于 Flash 的文件系统,就是基于 MTD 设备来构建的。

MTD 驱动的结构如图 18-1 所示。

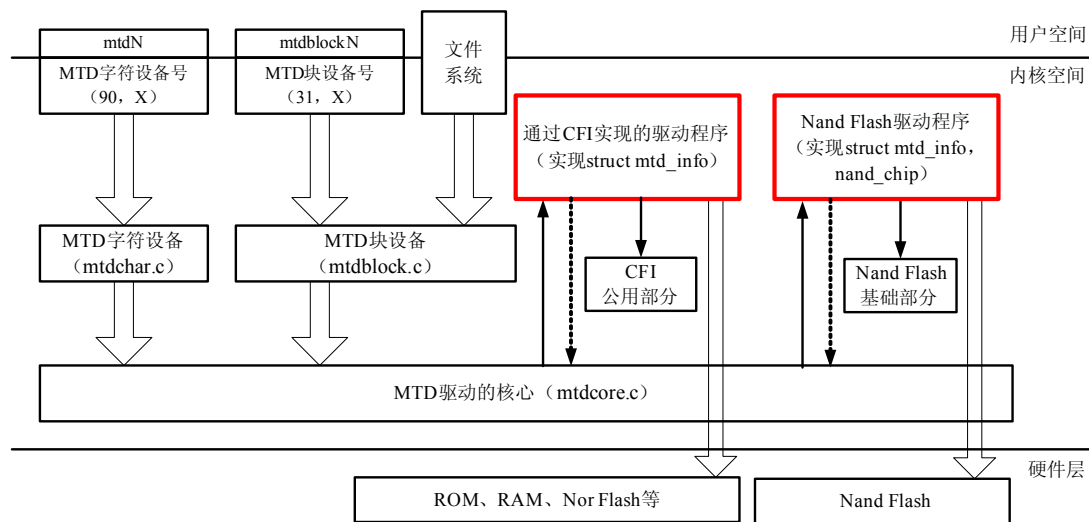


图 18-1 MTD 驱动的结构

在用户空间中,一个 MTD 原始设备可以同时提供字符设备和块设备的设备节点。MTD 字符设备的设备节点的主设备号为 90,次设备号依次排列;MTD 块设备的设备节点的主设备号为 31,次设备号依次排列。一般情况下,每一个 MTD 设备同时具有字符设备和块设备。

MTD 最主要的功能是构建文件系统,用户空间只需使用文件系统即可,驱动程序实现对用户空间是透明的。通过 proc 文件系统的/proc/mtd 可以查看 MTD 设备的信息。

MTD 的头文件:在 include/mtd/和 include/linux/mtd/中,包括 mtd.h、mtd-abi.h、partitions.h 等文件。

MTD 源代码的路径为 drivers/mtd/, 核心的部分包括了以下内容。

- mtdcore.c: MTD 核心,定义 MTD 原始设备。
- mtdsuper.c: MTD 超级块的支持。
- mtdpart.c: MTD 分区的支持(配置宏为 MTD_PARTITIONS)。
- mtdconcat.c: MTD 的联系层(配置宏为 MTD_CONCAT)。

MTD 用户空间的访问支持包括以下几个文件。

- mtdchar.c: MTD 字符设备(配置宏为 MTD_CHAR)。

- mtd_blkdevs: MTD 块设备公用（配置宏为 MTD_BLKDEVS）。
- mtdblock.c: MTD 块设备（配置宏为 MTD_BLOCK）。

18.2.1 MTD 的接口部分

mtd.h 是 MTD 系统的主要头文件，大部分核心数据结构的定义都在这个文件中。表示 MTD 原始设备的 mtd_info 是 MTD 架构的核心。

mtd_info 结构的定义如下所示：

```
struct mtd_info {
    u_char type;                // 类型
    uint32_t flags;             // 标志
    uint64_t size;              // 几种尺寸
    uint32_t erasesize;
    uint32_t writesize;
    uint32_t oobsize;           // 仅用于 Nand Flash 的内容
    uint32_t oobavail;
    // ..... 省略部分内容
    const char *name;
    int index;
    // ..... 省略部分内容
    int numeraseregions;
    struct mtd_erase_region_info *eraseregions;
    int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
    int (*point) (struct mtd_info *mtd, loff_t from, size_t len,
                 size_t *retlen, void **virt, resource_size_t *phys);
    void (*unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
    unsigned long (*get_unmapped_area) (struct mtd_info *mtd,
                                         unsigned long len, unsigned long offset, unsigned long flags);
    struct backing_dev_info *backing_dev_info;
    int (*read) (struct mtd_info *mtd, loff_t from, size_t len,
                size_t *retlen, u_char *buf);
    int (*write) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
                 const u_char *buf);
    int (*panic_write) (struct mtd_info *mtd, loff_t to,
                       size_t len, size_t *retlen, const u_char *buf);
    int (*read_oob) (struct mtd_info *mtd, loff_t from,
                    struct mtd_oob_ops *ops);
    int (*write_oob) (struct mtd_info *mtd, loff_t to,
                     struct mtd_oob_ops *ops);
    // ..... 省略部分内容
    int (*writev) (struct mtd_info *mtd, const struct kvec *vecs,
                  unsigned long count, loff_t to, size_t *retlen);
    void (*sync) (struct mtd_info *mtd);
    int (*lock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
    int (*unlock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
    int (*suspend) (struct mtd_info *mtd); // 电源管理的功能
    void (*resume) (struct mtd_info *mtd);
    int (*block_isbad) (struct mtd_info *mtd, loff_t ofs);
    int (*block_markbad) (struct mtd_info *mtd, loff_t ofs);
    // ..... 省略部分内容
    void *priv;
    // ..... 省略部分内容
    int (*get_device) (struct mtd_info *mtd); // 用于维护自身的引用计数
    void (*put_device) (struct mtd_info *mtd);
};
```

mtd_info 结构表示了 MTD 原始设备，是 MTD 子系统最核心的部分。mtd_info 结构中的各个函数指针都以此结构的指针为第一个参数，主要的操作包括了读、写和擦除等。

mtd_info 的首成员 type 表示存储器的类型，可选的值包括：MTD_RAM（内存存储器）、MTD_ROM（只读存储器）、MTD_NORFLASH（Nor Flash）、MTD_NANDFLASH（Nand Flash）等；flags 成员可选的值是 MTD_CAP_*等。

每一种存储器的驱动程序实现，就是要实现一个 mtd_info 原始设备。

提示：在 MTD 支持的各种硬件中，Nand Flash 的结构最复杂，因此 mtd.h 当中的公用部分也有一些内容仅仅用于 Nand Flash，如 OOB 的内容。

与 mtd_info 结构相关的几个函数如下所示：

```
extern int add_mtd_device(struct mtd_info *mtd);
extern int del_mtd_device (struct mtd_info *mtd);
extern struct mtd_info *get_mtd_device(struct mtd_info *mtd, int num);
```

mtd_notifier 结构和相关的函数如下：

```
struct mtd_notifier {
    void (*add)(struct mtd_info *mtd);    // 用于设备增加的时候
    void (*remove)(struct mtd_info *mtd); // 用于设备移除的时候
    struct list_head list;
};
extern void register_mtd_user (struct mtd_notifier *new);
extern int unregister_mtd_user (struct mtd_notifier *old);
```

mtd_notifier 表示 MTD 原始设备增加和删除时候的通知，可以从外部向 MTD 的核心部分进行注册，用于监听。

partitions.h 头文件中定义分区相关的内容，如下所示：

```
struct mtd_partition {
    char *name;                // 识别字符串
    uint64_t size;             // 分区大小
    uint64_t offset;           // 在 MTD 中的偏移
    uint32_t mask_flags;       // 主 MTD 的标志
    struct nand_ecclayout *ecclayout; // 分区的 OOB（仅 NAND）
};
#define MTDPART_OFS_NXTBLK (-2)
#define MTDPART_OFS_APPEND (-1)
#define MTDPART_SIZ_FULL (0)
int add_mtd_partitions(struct mtd_info *, const struct mtd_partition *, int);
int del_mtd_partitions(struct mtd_info *);
```

mtd_partition 表示一个 MTD 分区，MTD 分区就是加入 MTD 到原始设备（mtd_info）中的一个信息。由此表示，一个原始设备可以划分成若干个分区。

提示：在 MTD 驱动中，MTD 原始设备也被称为主（master），而一个 MTD 分区也被称为从（slave）。

18.2.2 MTD 的核心实现部分

MTD 核心部分提供的主要功能包括对原始设备的管理、对分区的增加和删除操作。主要在 `mtdcore.c` 和 `mtdpart.c` 文件中实现。虽然各种实现都是基于 `mtd_info` 的，但是核心部分并不关心 `mtd_info` 具体的实现是什么。

`add_mtd_device()` 表示将整个 MTD 设备加入系统，此部分内容在 `mtdcore.c` 文件中实现，将为加入的原始设备创建字符设备和块设备的节点，并将其加入到链表中。

`add_mtd_partition()` 表示在某个 MTD 设备中加入一个分区，此部分内容在 `mtdpart.c` 文件中实现。此函数表示分区的内容是动态建立的，其中也包含一个 `mtd_info`，其中的大部分成员来自于 MTD 原始设备（主分区），但大小、偏移量、名称以及进行操作的函数等成员不相同。在实现上，一个 MTD 分区当中的操作继承了主分区的操作，但是部分功能有所不同。增加分区的时候，也可能调用 `add_mtd_device()` 增加设备。

18.3 MTD 的设备层

MTD 对用户空间提供的接口是字符设备和块设备，一个 MTD 原始设备（也就是一个分区）可以对应于几个不同设备。

- `mtd<N>`：支持读写操作的字符设备（90）。
- `mtd<N>ro`：支持读操作的字符设备（90）。
- `mtdblock<N>`：块设备（31）。

字符设备可以用于控制，块设备则用于构建文件系统。同一个分区的两个字符设备的次设备号一般为相连的，分别为 $N \times 2$ 和 $N \times 2 + 1$ 。之所以有两个字符设备，主要是为了对应不同的权限。

提示：MTD 块设备和字符设备并非 MTD 系统核心，主要为了提供用户空间的接口。

18.3.1 MTD 字符设备

MTD 字符设备在 `mtdchar.c` 文件中实现。作为字符设备，主要支持 `read`、`write`、`lseek`、`ioctl` 等操作。

头文件 `mtd-abi.h` 中定义的一些 `ioctl` 命令如下所示：

```
#define MEMGETINFO      _IOR('M', 1, struct mtd_info_user)
#define MEMERASE        _IOW('M', 2, struct erase_info_user)
#define MEMWRITEOOB     _IOWR('M', 3, struct mtd_oob_buf)
#define MEMREADOOB     _IOWR('M', 4, struct mtd_oob_buf)
#define MEMLOCK         _IOW('M', 5, struct erase_info_user)
#define MEMUNLOCK       _IOW('M', 6, struct erase_info_user)
#define MEMGETREGIONCOUNT _IOR('M', 7, int)
#define MEMGETREGIONINFO _IOWR('M', 8, struct region_info_user)
#define MEMSETOOBSEL    _IOW('M', 9, struct nand_oobinfo)
#define MEMGETOOBSEL    _IOR('M', 10, struct nand_oobinfo)
// ..... 省略部分内容：其他的 ioctl
```

其中的 `mtd_info_user` 结构表示一个 MTD 设备，`erase_info_user` 结构表示擦除信息，`nand_oobinfo` 结构表示 Nand Flash 的 OOB 区域的信息，`mtd_oob_buf` 结构表示 Nand Flash 的 OOB 区域的操作，这些结构都用于用户空间。

本字符设备实现的典型特点是将分区的 `mtd_info` 作为所打开文件的私有数据 (`private_data`)，当需要信息和操作的时候，从其中取出 `mtd_info` 并使用。

用于实现 `ioctl` 的 `mtd_ioctl()` 函数如下所示：

```
static int mtd_ioctl(struct file *file, u_int cmd, u_long arg)
{
    struct mtd_file_info *mfi = file->private_data;
    struct mtd_info *mtd = mfi->mtd;
    void __user *argp = (void __user *)arg;
    int ret = 0;
    u_long size;
    struct mtd_info_user info;
    size = (cmd & IOCSIZE_MASK) >> IOCSIZE_SHIFT;
    // ..... 省略部分内容：访问检查
    switch (cmd) {
        case MEMGETREGIONCOUNT:
            if (copy_to_user(argp, &(mtd->numeraseregions), sizeof(int)))
                return -EFAULT;
            break;
        // ..... 省略部分内容：其他 ioctl 命令
        case MEMGETINFO:
            info.type = mtd->type;    // 从 mtd_info 结构取出信息放入 mtd_info_user
            info.flags = mtd->flags;
            info.size = mtd->size;
            info.erasesize = mtd->erasesize;
            info.writesize = mtd->writesize;
            info.oobsize = mtd->oobsize;
            info.ecctype = -1;
            info.eccsize = 0;
            if (copy_to_user(argp, &info, sizeof(struct mtd_info_user)))
                return -EFAULT;
            break;
        // ..... 省略部分内容：其他 ioctl 命令
    }
    return ret;
} /* memory_ioctl */
```

MTD 字符设备也支持读、写操作，其主体部分通过 `mtd_info` 当中的 `read` 指针和 `write` 指针实现。

MTD 字符设备的实现也就是对 `mtd_info` 的一层封装。

18.3.2 MTD 块设备

MTD 块设备有两层实现，由 `mtd_blkdevs.c`、`mtdblock.c` 等几个文件组成。

- `mtd_blkdevs.c` 是实际的块设备的实现，构建了 `block_device_operations` 结构。头文件 `blktrans.h` 中定义的 `mtd_blktrans_dev` 和 `mtd_blktrans_ops` 结构，前者表示一个转换的设备层，后者表示这个设备的操作。
- `mtdblock_ro.c` 提供只读的块设备的实现。

- mtdblock.c 提供一个 MTD 块设备的真正实现，实现了 mtd_blktrans_ops 中的函数指针，通过 mtd_info 完成实际的功能。

提示：MTD 块设备没有提供块擦除操作，其并不适用于 Nand Flash。

mtdblock.c 当中实现的读、写两个操作如下所示：

```
static int mtdblock_readsect(struct mtd_blktrans_dev *dev,
                           unsigned long block, char *buf)
{
    struct mtdblk_dev *mtdblk = container_of(dev, struct mtdblk_dev, mbd);
    return do_cached_read(mtdblk, block<<9, 512, buf);
}

static int mtdblock_writesect(struct mtd_blktrans_dev *dev,
                             unsigned long block, char *buf)
{
    struct mtdblk_dev *mtdblk = container_of(dev, struct mtdblk_dev, mbd);
    if (unlikely(!mtdblk->cache_data && mtdblk->cache_size)) {
        mtdblk->cache_data = vmalloc(mtdblk->mbd.mtd->erasesize);
        if (!mtdblk->cache_data) return -EINTR;
    }
    return do_cached_write(mtdblk, block<<9, 512, buf);
}
```

mtdblock_readsect()和 mtdblock_writesect()两个函数是 mtd_blktrans_ops 当中的读、写指针。do_cached_read()和 do_cached_write()提供真正的实现，核心部分通过调用 mtd_info 当中的读、写指针完成。相比字符设备，块设备的读写操作增加了缓冲区的处理。

18.4 CFI 硬件实现层

CFI 是 Common Flash Interface 的缩写，含义为通用 Flash 接口，其中包括了 RAM、ROM 和 Nor Flash 的实现。

18.4.1 公用部分

CFI 的相关实现在 driver/mtd/chips/目录中，chipreg.c 和 cfi_probe.c 是公用文件，主要的头文件是 map.h。

map.h 文件中定义的 mtd_chip_driver 如下所示：

```
struct mtd_chip_driver {
    struct mtd_info *(*probe)(struct map_info *map);
    void (*destroy)(struct mtd_info *);
    struct module *module;
    char *name;
    struct list_head list;
};

void register_mtd_chip_driver(struct mtd_chip_driver *);
void unregister_mtd_chip_driver(struct mtd_chip_driver *);
```

mtd_chip_driver 结构完成各种实现的注册，其中 probe 函数指针的实现需要返回一个 mtd_info 结构的指针，表明提供一个 mtd_info 的实现。

18.4.2 ROM 的 MTD 实现

map_rom.c 文件是针对 ROM (Read Only Memory, 只读存储器) 的实现。

其中实现的部分如下所示:

```
static struct mtd_chip_driver maprom_chipdrv = {
    .probe    = map_rom_probe,
    .name     = "map_rom",
    .module   = THIS_MODULE
};
static struct mtd_info *map_rom_probe(struct map_info *map)
{
    struct mtd_info *mtd;
    mtd = kzalloc(sizeof(*mtd), GFP_KERNEL); // 分配一个 map_info 结构
    if (!mtd)
        return NULL;
    map->fldrv = &maprom_chipdrv;
    mtd->priv = map;
    mtd->name = map->name;
    mtd->type = MTD_ROM;                // 设置 MTD 原始设备的类型
    mtd->size = map->size;
    mtd->get_unmapped_area = maprom_unmapped_area;
    mtd->read = maprom_read;            // 设置 MTD 原始设备的各种操作函数指针
    mtd->write = maprom_write;
    mtd->sync = maprom_nop;
    mtd->erase = maprom_erase;
    mtd->flags = MTD_CAP_ROM;
    mtd->erasesize = map->size;        // 擦除大小等于内存大小
    mtd->writesize = 1;
    __module_get(THIS_MODULE);
    return mtd;
}
```

用于读取操作的 maprom_read() 函数如下所示:

```
static int maprom_read (struct mtd_info *mtd, loff_t from,
                        size_t len, size_t *retlen, u_char *buf)
{
    struct map_info *map = mtd->priv;
    map_copy_from(map, buf, from, len); // 进行简单的复制操作
    *retlen = len;
    return 0;
}
```

由于是只读存储器, 因此表示写的 maprom_write() 函数返回-EIO 错误 (输入/输出错误), 表示擦除的 maprom_erase() 函数返回-EROFS (只读文件系统错误)。

18.4.3 RAM 的 MTD 实现

map_ram.c 文件是针对 RAM (Random Access Memory, 随机存储器) 的实现。其中所实现的大部分内容都和对 ROM 的实现类似。mtd_chip_driver 驱动的名称为 “map_ram”。其中的读函数与 ROM 实现相同, 写实现也通过类似的方式实现。

表示擦除的 mapram_erase() 函数如下所示:

```
static int mapram_erase (struct mtd_info *mtd, struct erase_info *instr)
{

```

```

    struct map_info *map = mtd->priv;
    map_word allff;
    unsigned long i;
    allff = map_word_ff(map);
    for (i=0; i<instr->len; i += map_bankwidth(map)) // 根据长度进行处理
        map_write(map, allff, instr->addr + i);
    instr->state = MTD_ERASE_DONE;                // 标志擦除操作完成
    mtd_erase_callback(instr);                    // 进行通知
    return 0;
}

```

mapram_erase ()函数的实现利用写入内存模拟了“擦除”的操作，完成操作之后，还需要调用回调函数进行对上层的通知。

18.4.4 Nor Flash 的 MTD 实现

对于不同 Nor Flash 总线带宽（8 位、16 位和 32 位）存在差异，driver/mtd/chips 目录当中的几个文件表示 Nor Flash 存储器的实现，几个文件如下所示。

- cfi_cmdset_0001.c: Intel/Sharp 的 Flash 实现。
- cfi_cmdset_0002.c: AMD/Fujitsu/Spansion 的 Flash 实现。
- cfi_cmdset_0020.c: ST Advanced Architecture 的 Flash 实现。

这几个文件没有通过 cfi_cmdset_0001 的 probe 函数指针实现，而是在公用的 gen_probe.c 文件中调用几个具体芯片的初始化函数。其中都包括了 Nor Flash 在擦除和写入时的特定命令序列。

18.5 Nand Flash 的硬件实现层

Nand Flash 的实现要复杂很多，因为需要考虑复杂的擦除和写入的问题。虽然 Nand Flash 具有共性的部分已经有了实现，但是每种 Nand 芯片需要处理的硬件相关的特性依然很多。Nand Flash 本身也分为 Bare NAND、SmartMediaCards、DiskOnChip 等几种子类型，它们的实现也有区别。

提示：一个基于Nand Flash的系统通常由文件系统(如 JIFFS)、MTD 核心、通用 Nand Flash 部分及硬件相关 Nand Flash 部分组成。

18.5.1 公用部分

Nand Flash 公共的头文件为 include/linux/mtd/nand.h。

Nand Flash 公共的实现部分在 driver/mtd/nand 目录中，nand_base.c、nand_btt.c、nand_ecc.c 等几个文件是公用的实现。

nand.h 定义了一些公共函数和内容如下所示：

```

extern int nand_scan (struct mtd_info *mtd, int max_chips);
extern int nand_scan_ident(struct mtd_info *mtd, int max_chips,
                        struct nand_flash_dev *table);
extern int nand_scan_tail(struct mtd_info *mtd);

```

```
extern void nand_release (struct mtd_info *mtd);
extern void nand_wait_ready(struct mtd_info *mtd);
extern int nand_lock(struct mtd_info *mtd, loff_t ofs, uint64_t len);
extern int nand_unlock(struct mtd_info *mtd, loff_t ofs, uint64_t len);
// 各种命令的定义
#define NAND_CMD_READ0          0
#define NAND_CMD_READ1         1
#define NAND_CMD_RNDOUT        5
#define NAND_CMD_PAGEPROG      0x10
#define NAND_CMD_READOOB       0x50
#define NAND_CMD_ERASE1        0x60
#define NAND_CMD_STATUS        0x70
```

这部分内容主要在 `nand_base.c` 中实现，供各实际芯片驱动调用。

`nand_chip` 结构如下所示：

```
struct nand_chip {
    void __iomem *IO_ADDR_R;    // 芯片的读地址
    void __iomem *IO_ADDR_W;    // 芯片的写地址
    uint8_t (*read_byte)(struct mtd_info *mtd);
    u16 (*read_word)(struct mtd_info *mtd);
    void (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
    void (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
    int (*verify_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
    void (*select_chip)(struct mtd_info *mtd, int chip);
    int (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
    int (*block_markbad)(struct mtd_info *mtd, loff_t ofs);
    // ..... 省略部分内容
}
```

`nand_chip` 结构的前两个成员表示经过内存映射之后的读地址和写地址，在 Flash 驱动的代码中可以直接对其进行读写。`nand_chip` 结构也包含了众多函数指针。在 Nand Flash 的实现中，`nand_chip` 结构常常保存成 `mtd_info` 结构的私有数据（private），针对各个芯片的实现可以对 `nand_chip` 结构当中的指针进行替换，表示本芯片特殊的实现。

Nand Flash 驱动程序主要需要处理 IO 端口读写的基地址，读写、延迟等工作，在初始化的阶段也需要处理不同的扫描、分区等逻辑。

`nand_ecc.c` 文件提供了软件 ECC 的算法，主要就是通过软件的方式完成了大小为 256 的数据块当中的 1 位纠错。其中提供的函数如下所示：

```
void __nand_calculate_ecc(const u_char *dat, unsigned int eccsize,
                        u_char *ecc_code);
int nand_calculate_ecc(struct mtd_info *mtd, const u_char *dat, u_char *ecc_code);
int __nand_correct_data(u_char *dat, u_char *read_ecc, u_char *calc_ecc,
                      unsigned int eccsize);
int nand_correct_data(struct mtd_info *mtd, u_char *dat,
                    u_char *read_ecc, u_char *calc_ecc)
```

这些函数也是 `nand_chip` 结构中 `nand_ecc_ctrl` 指针的实现，某些 Nand Flash 芯片需要利用此处的软件 ECC 功能。

18.5.2 GPIO 的 Nand Flash 实现

在某个硬件平台上，Nand Flash 芯片通常可以直接连接在处理器的输入/输出引脚

(GPIO) 上。driver/mtd/nand 目录中的 gpio.c 文件提供了 Nand Flash 相对通用的实现。

GPIO Nand Flash 实现的私有数据结构 gpiomtd 如下所示:

```
struct gpiomtd {
    void __iomem      *io_sync;
    struct mtd_info    mtd_info;    // MTD 原始设备
    struct nand_chip    nand_chip;   // 保存 Nand Flash 的特殊数据结构
    struct gpio_nand_platdata plat;  // 表示 GPIO 的相关引脚
};
```

其中的 gpio_nand_platdata 结构包含了 gpio_nce、gpio_nwp、gpio_cle 等成员, 它们表示了 GPIO 的相关引脚。

gpio.c 当中实现了名称为"gpio-nand"的平台驱动, 其探测函数如下所示:

```
static int __devinit gpio_nand_probe(struct platform_device *dev)
{
    struct gpiomtd *gpiomtd;    // GPIO Nand Flash 实现的私有结构
    struct nand_chip *this;
    struct resource *res0, *res1;
    int ret;
    if (!dev->dev.platform_data) return -EINVAL;
    res0 = platform_get_resource(dev, IORESOURCE_MEM, 0); // 获得平台资源
    if (!res0) return -EINVAL;
    gpiomtd = kzalloc(sizeof(*gpiomtd), GFP_KERNEL); // 分配 gpiomtd 结构
    // ..... 省略错误处理
    this = &gpiomtd->nand_chip;
    this->IO_ADDR_R = request_and_remap(res0, 2, "NAND", &ret); // 获得内存地址
    // ..... 省略错误处理
    res1 = platform_get_resource(dev, IORESOURCE_MEM, 1);
    // ..... 省略错误处理
    memcpy(&gpiomtd->plat, dev->dev.platform_data, sizeof(gpiomtd->plat));
    ret = gpio_request(gpiomtd->plat.gpio_nce, "NAND NCE"); // 调用 GPIO 系统
    if (ret) goto err_nce;
    gpio_direction_output(gpiomtd->plat.gpio_nce, 1); // 设置 GPIO 方向
    if (gpio_is_valid(gpiomtd->plat.gpio_nwp)) {
        ret = gpio_request(gpiomtd->plat.gpio_nwp, "NAND NWP");
        if (ret) goto err_nwp;
        gpio_direction_output(gpiomtd->plat.gpio_nwp, 1);
    }
    // ..... 省略部分内容: 通过 GPIO 设置 Nand Flash 的引脚
    this->IO_ADDR_W = this->IO_ADDR_R; // 设置 nand_chip 结构
    this->ecc.mode = NAND_ECC_SOFT;
    this->options = gpiomtd->plat.options;
    this->chip_delay = gpiomtd->plat.chip_delay;
    this->cmd_ctrl = gpio_nand_cmd_ctrl;
    this->dev_ready = gpio_nand_devready;
    if (this->options & NAND_BUSWIDTH_16) { // 16 位总线的处理
        this->read_buf = gpio_nand_readbuf16;
        this->write_buf = gpio_nand_writebuf16;
        this->verify_buf = gpio_nand_verifybuf16;
    } else { // 8 位总线的处理
        this->read_buf = gpio_nand_readbuf;
        this->write_buf = gpio_nand_writebuf;
    }
}
```

```

        this->verify_buf = gpio_nand_verifybuf;
    }
    gpiomtd->mtd_info.priv = this;          // 设置 mtd_info 当中的 nand_chip
    gpiomtd->mtd_info.owner = THIS_MODULE;
    // ..... 省略错误处理
    if (gpiomtd->plat.adjust_parts)
        gpiomtd->plat.adjust_parts(&gpiomtd->plat, gpiomtd->mtd_info.size);
    add_mtd_partitions(&gpiomtd->mtd_info, gpiomtd->plat.parts,
        gpiomtd->plat.num_parts);    // 增加一个 MTD 的分区
    platform_set_drvdata(dev, gpiomtd);
    return 0;
    // ..... 省略错误处理
}

```

如果与平台设备相匹配，`gpio_nand_probe()`函数将会执行。首先要复制平台设备当中定义的 `dev->dev.platform_data`，获得连接 Nand Flash 的引脚，然后初始化驱动所需要的 `mtd_info` 和 `nand_chip` 结构。

`nand_chip` 的读、写函数指针的实现如下所示：

```

static void gpio_nand_writebuf(struct mtd_info *mtd, const u_char *buf, int len)
{
    struct nand_chip *this = mtd->priv;
    writesb(this->IO_ADDR_W, buf, len);    // 直接在地址写入
}
static void gpio_nand_readbuf(struct mtd_info *mtd, u_char *buf, int len)
{
    struct nand_chip *this = mtd->priv;
    readsb(this->IO_ADDR_R, buf, len);    // 直接从地址读出
}

```

Nand Flash 基本的读、写操作的实现都要从 `nand_chip` 结构当中得到地址信息，然后对这个地址进行操作。

`nand_chip` 的 `cmd_ctrl` 函数指针的实现如下所示：

```

static void gpio_nand_cmd_ctrl(struct mtd_info *mtd, int cmd, unsigned int ctrl)
{
    struct gpiomtd *gpiomtd = gpio_nand_getpriv(mtd);
    gpio_nand_dosync(gpiomtd);
    if (ctrl & NAND_CTRL_CHANGE) {        // 通过 GPIO 引脚进行控制
        gpio_set_value(gpiomtd->plat.gpio_nce, !(ctrl & NAND_NCE));
        gpio_set_value(gpiomtd->plat.gpio_cle, !(ctrl & NAND_CLE));
        gpio_set_value(gpiomtd->plat.gpio_ale, !(ctrl & NAND_ALE));
        gpio_nand_dosync(gpiomtd);
    }
    if (cmd == NAND_CMD_NONE) return;
    writeb(cmd, gpiomtd->nand_chip.IO_ADDR_W);    // 此处表示直接写一个内存地址
    gpio_nand_dosync(gpiomtd);
}

```

在此，`gpio_nand_cmd_ctrl()`函数利用了核心部分 `nand_chip` 当中的功能，也利用了 GPIO 系统在内核中的 API，例如 `gpio_set_value()`等。

18.5.3 处理器芯片上的 Nand Flash 实现

很多嵌入式处理器都具有 Nand Flash 的控制器，这些控制器可以辅助处理器更容易地连接和使用 Nand Flash。因此，`driver/mtd/nand` 目录中的其他各个*_nand.c 文件提供了针对特定芯片的 Nand Flash 实现。

在针对具体芯片的实现中，需要有针对性地完成处理器的初始化工作，`nand_chip` 结构的读、写函数指针的实现一般都基于 `IO_ADDR_R` 和 `IO_ADDR_W` 完成。另外，还有针对各个不同硬件的 ECC（错误检查和纠正）实现。

第 19 章

USB 系统和驱动

19.1 USB 概述

19.1.1 USB 规范

USB (Universal Serial Bus, 通用串行总线) 是连接计算机系统与外部设备的一个串口总线标准, 也是一种输入/输出接口的技术规范。USB 最初是由英特尔公司与微软公司倡导发起的, 支持热插拔和即插即用。USB 的设计为非对称式的, 它由一个主机控制器和若干通过集线器 (hub) 设备以树形连接的设备组成。一个控制器下最多可以有 5 级 hub, 包括 hub 在内最多可连接 128 个设备 (7 位寻址)。

USB 的网站为: <http://www.usb.org>。

各种 USB 标准的标志如图 19-1 所示。



图 19-1 USB 标准的标志

USB 分为不同版本的不同标准, 如下所示。

- USB 1.0 和 USB 1.1: 其中, USB 1.1 于 1996 年 1 月发布, 最大传输带宽为 12Mb/s。

- USB 2.0: 于 1998 年 9 月发布, 修正 1.0 版已发现的问题, 大部分是关于 USB 集线器 (hub) 的。
 - USB 2.0: 于 2000 年 4 月发布, 增加到更高的数据传输速率: 480Mb/s。
 - USB OTG (On-The-Go); USB 2.0 规格的补充标准, 支持让 USB 设备作为主机。
 - USB 3.0: 于 2008 年 11 月发布, 速率增加到 5Gb/s, 支持全双工, 支持光纤传输。
- USB 不同标准的主要差别体现在速率特性上, 差别如表 19-1 所示。

表 19-1 USB 不同标准的速率特性

USB版本	速率称号	带宽	速率
USB 3.0	超高速 (SuperSpeed)	5Gb/s	约 500MB/s(5000Mb/s)
USB 2.0	高速 (Hi-Speed)	480Mb/s	约 60MB/s(60 000KB/s)
USB 1.1	全速 (Full Speed)	12Mb/s	约 1.5MB/s(1500KB/s)
USB 1.0	低速 (Low Speed)	1.5Mb/s	187.5B/s

USB 的连线有 4 四根, 它们的功能如表 19-2 所示。

表 19-2 USB 连线的功能

触点	功能 (主机)	功能 (设备)
1	V _{BUS} (4.75—5.25 V)	V _{BUS} (4.4—5.25 V)
2	D-	D-
3	D+	D+
4	接地	接地

USB 的 D+和 D-信号, 各自使用半双工的差分信号并协同工作。

USB 在物理上也有不同的插头形式, 如图 19-2 所示。

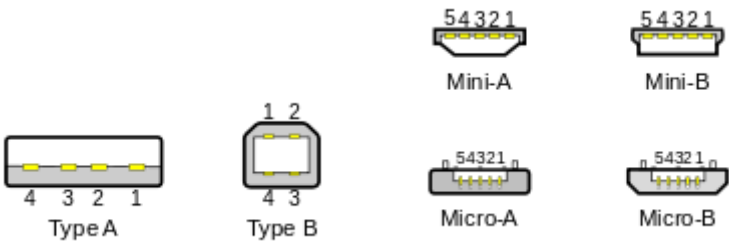


图 19-2 USB 的插头形式

一个 USB 主机通过 hub 链可以连接多个设备。一个物理设备可以承担多种功能, USB 的术语中设备 (device) 指的是功能 (functions)。集线器 (hub) 由于作用特殊, 按照正式的观点集线器并不是一种功能。直接连接到主机上的是根集线器。

依附在总线上的设备可以是需要特定驱动程序的完全定制的设备, 也可能属于某个设备类别。这些类别定义了某种设备的行为和接口描述符, 这样一个驱动程序可能用于所有

此种类别的设备。

USB 设备分类由 USB 设计论坛设备工作组决定，主要的 USB 设备类型如下所示。

- 0x00：保留。
- 0x01：音频设备，例如声卡。
- 0x02：USB 通信控制设备，例如网卡、调制解调器、串行接口。
- 0x03：人机界面设备（HID），例如键盘、鼠标 0x05。
- 0x04：显示器设备。
- 0x05：物理界面设备，例如控制杆。
- 0x06：静止图像捕捉设备，例如扫描仪。
- 0x07：打印，例如打印机。
- 0x08：大容量访问设备，例如 Flash 存储器设备、移动硬盘等。
- 0x09：集线器。
- 0x0A：通信设备，例如调制解调器、网络配置卡、ISDN、传真 0x0B：智能卡设备，例如读卡器。
- 0x0E：图像设备，例如摄像头。
- 0xE0：无线传输设备，例如蓝牙。
- 0xFE：特殊的应用。
- 0xFF：厂商自定义的设备。

19.1.2 USB 的软件系统

USB 在硬件上由 USB 主机和 USB 设备组成，因此 USB 的软件也分为主机端和设备端两个部分。

USB 的软件架构如图 19-3 所示。

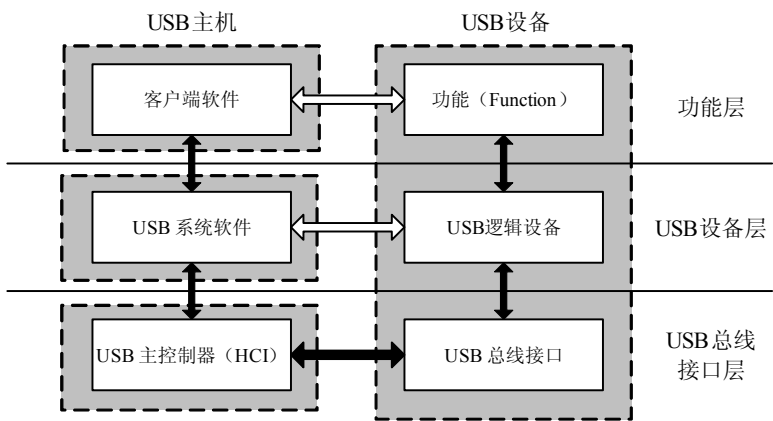


图 19-3 USB 的软件架构

理论上一个 USB 的物理设备可以承担多种功能，设备-功能与管道（pipe）联系在一起，管道把主机控制器和被称为端点（endpoint）的逻辑实体连接起来。一个端点只能单向

传输数据，自然管道也是单向的。每个 USB 设备至少有两个端点 / 管道：它们分别是进出方向的，编号为 0，用于控制总线上的设备。

所有 USB 设备都需要实现一个默认的控制方法。这种方法将端点 0 作为输入端点，同时也将端点 0 作为输出端点。USB 系统用这个默认方法进行初始化，按照一般方式使用逻辑设备。默认控制通道支持了对控制的传送。

设备可以有除端点 0 以外的其他端点，端点的数目取决于这些设备的实现。低速设备在 0 号输入及输出端点外，只能有两个额外的可选端点。而高速设备可具有的额外端点数仅受限于协议的定义，USB 协议中规定，最多 15 个额外的输入端点和最多 15 个额外的输出端点。

按照各自的传输类型，管道被分为如下 4 类。

- 控制传输 (Control)：一般用于短的、简单的对设备的命令和状态反馈，例如用于总线控制的 0 号管道。
- 同步传输 (Isochronous)：按照有保障的速度（可能，但不必然是尽快地）传输，可能有数据丢失，例如实时的音频、视频。
- 中断传输 (Interrupt)：用于必须保证尽快反应的设备（有限延迟），例如鼠标、键盘。
- 批量传输 (Bulk)：使用余下的带宽大量地（但是没有对于延迟、连续性、带宽和速度的保证）传输数据，例如普通的文件传输。

USB 的端点在每个方向上按照 0~15 编号，因此一个设备最多有 32 个活动管道，16 个输入，16 个输出。两个方向的端点 0 总是留给总线管理，占用了 32 个端点中的 2 个。在管道中，数据使用不同长度的包传递，端点可以传递的包长度上限一般是 2^n 字节，所以 USB 包经常包含的数据量依次有 8、16、32、64、128、256、512 或者 1024 字节。

Linux 系统对 USB 的实现包括如下两个部分。

- 主机部分 (Host-side USB)：用于 Linux 系统支持 USB 主机的情况。
- 设备部分 (USB Gadget)：用于 Linux 系统支持 USB 设备的情况。

19.2 Linux 的 USB 主机端支持

19.2.1 USB 主机端的软件结构

Linux 的 USB 主机端的支持是指以 Linux 操作系统运行的机器为 USB 主机实现的场景。自下而上，软件分为 3 层结构：USB 主控制器的支持、USB 的核心、USB 的驱动，它们分别对应于 USB 软件协议栈的 3 层结构。

- USB 部分主要的头文件：include/linux/usb.h。
- USB 部分主要的其他头文件：include/linux/usb/。
- USB 主机核心代码：drivers/usb/core/。
- USB 主控制器代码：drivers/usb/host/。
- USB 的驱动代码，串口实现的代码为 drivers/usb/serial，大容量存储器的代码为 drivers/usb/storage，其他各种 USB 的代码 drivers/usb/class 和 drivers/usb/misc。

USB 主机部分的所有功能编译宏（USB）控制。此宏打开后，core 目录中的大部分内容将会被编译，其中包括了 USB 的核心和文件系统部分。

USB 核心部分的使能依赖至少一个 HCD，HCD 就是 USB 核心部分的底层实现。对于实际的功能，还需要某个具体的 USB 驱动来实现。

USB 本身确实是一个总线，因此 sys 文件系统表现了 USB 系统的重要信息，主要的目录有以下几个。

- /sys/bus/usb/devices/*/

目录中的内容表示为各个 USB 设备，格式通常为：<HUB 号>-<HUB 端口>:<配置>.<接口>，这些格式实际上描述了连接在 USB 主机上的各种设备。

- /sys/bus/usb/drivers/*

各个 USB 驱动，也就是各种功能在 USB 主机端的软件支持。

19.2.2 USB 主机端的核心部分

usb.h 是 Linux 的 USB 系统在主机端的核心头文件，其中定义了大量 USB 相关的结构体和相关的函数。

usb.h 当中的几个主要数据结构如下所示。

- usb_host_endpoint: 主机端点的描述和队列。
- usb_host_interface: 主机对一个接口的封装。
- usb_interface: 设备驱动进行交互的接口。
- usb_host_config: 表示一个设备的配置。
- usb_bus: 作为一种总线的封装结构。
- usb_class_driver: 作为一种主类型，注册到一个主设备号上。
- usb_device: 表示一个 USB 的设备。
- usb_driver: 表示 USB 驱动，是 usb_interface 在 USB 核心的表示。
- usb_device_driver: 表示 USB 驱动，是 usb_device 在 USB 核心的表示。

usb_device 结构如下所示：

```
struct usb_device {
    int      devnum;                // 在 USB 总线之上的设备号
    char     devpath[16];           // 设备的路径
    u32      route;
    enum usb_device_state state;     // 设备的状态
    enum usb_device_speed speed;    // 设备的速度
    struct usb_tt *tt;
    int      ttport;
    unsigned int toggle[2];
    struct usb_device *parent;       // 表示这个设备连接的 hub
    struct usb_bus *bus;             // 表示这个设备连接的总线
    struct usb_host_endpoint ep0;    // 表示这个设备的端点 0 在主机端的表示
    struct device dev;              // 内核中的设备表示
    struct usb_device_descriptor descriptor; // USB 的设备描述符号
    struct usb_host_config *config;  // 这个 USB 设备所有的配置
    struct usb_host_config *actconfig; // 这个 USB 设备激活的配置
    struct usb_host_endpoint *ep_in[16]; // 这个 USB 设备的 16 个输入端点
```

```

    struct usb_host_endpoint *ep_out[16];           // 这个 USB 设备的 16 个输出端点
// ..... 省略部分内容
    int maxchild;                                   // 作为 hub 的时候, 连接的信息
    struct usb_device *children[USB_MAXCHILDREN];
};

```

`usb_device` 是一个连接于 USB 总线的 USB 设备在内核中的表示。几个重要的成员为：`devnum` 表示设备在 USB 总线的地址，`devpath` 表示设备的 ID 字符串，`speed` 表示设备的速度，`ep_in` 和 `ep_out` 分别是设备输入/输出端点的数组。

`usb_driver` 结构如下所示：

```

struct usb_driver {
    const char *name;
    int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
    void (*disconnect) (struct usb_interface *intf);
    int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    int (*resume) (struct usb_interface *intf);
    int (*reset_resume) (struct usb_interface *intf);
    int (*pre_reset) (struct usb_interface *intf);
    int (*post_reset) (struct usb_interface *intf);
    const struct usb_device_id *id_table;
    struct usb_dynids dynids;
    struct usbdrv_wrap drvwrap;
    unsigned int no_dynamic_id:1;
    unsigned int supports_autosuspend:1;
    unsigned int soft_unbind:1;
};

```

每个 `usb_driver` 的操作对象就是一个 `usb_interface`，它表示了一个 USB 驱动操作 USB 设备的句柄。

`usb_driver` 的注册和注销函数如下所示：

```

int usb_register_driver(struct usb_driver *, struct module *, const char *);
void usb_deregister(struct usb_driver *);

```

`usb_driver` 结构用于联系 USB 核心和 USB 具体的实现。

USB 的某一种实际功能就是一个 USB 驱动的实现，USB 驱动通过与核心的交互来实现。所有注册的 USB 驱动体现在 `sys` 文件系统的 `/sys/bus/usb/drivers/` 当中，其中的各个子目录对应于一个 `usb_driver` 的实现。

URB 的含义为 USB Request Block (USB 请求块)，`urb` 结构是 USB 当中最为复杂的一个数据结构。通常不对 `urb` 结构本身的成员进行处理，而是通过工具函数进行处理。

一些 `urb` 的操作函数如下所示。

```

extern void usb_init_urb(struct urb *urb);
extern struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
extern void usb_free_urb(struct urb *urb);
#define usb_put_urb usb_free_urb
extern struct urb *usb_get_urb(struct urb *urb);
extern int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
extern int usb_unlink_urb(struct urb *urb);
extern void usb_kill_urb(struct urb *urb);
extern void usb_poison_urb(struct urb *urb);

```

```
extern void usb_unpoison_urb(struct urb *urb);
extern void usb_kill_anchored_urbs(struct usb_anchor *anchor);
extern void usb_poison_anchored_urbs(struct usb_anchor *anchor);
extern void usb_unpoison_anchored_urbs(struct usb_anchor *anchor);
extern void usb_unlink_anchored_urbs(struct usb_anchor *anchor);
extern void usb_anchor_urb(struct urb *urb, struct usb_anchor *anchor);
extern void usb_unanchor_urb(struct urb *urb);
extern int usb_wait_anchor_empty_timeout(struct usb_anchor *anchor,
                                         unsigned int timeout);
extern struct urb *usb_get_from_anchor(struct usb_anchor *anchor);
extern void usb_scuttle_anchored_urbs(struct usb_anchor *anchor);
extern int usb_anchor_empty(struct usb_anchor *anchor);
```

URB 本身描述 USB 的传输请求，一个 URB 必须使用 `usb_alloc_urb()` 进行分配，使用 `usb_free_urb()` 进行释放，`usb_fill_*_urb()` 用于初始化 URB，`usb_submit_urb()` 函数则用于提交 URB，`usb_unlink_urb()` 和 `usb_kill_urb()` 函数用于取消一个 URB。

USB 的核心部分可以提供 URB 接口完成基于 USB 协议的传输，在各个 USB 驱动中完成具体的操作。

19.2.3 USB 驱动的实现

1. USB 驱动的功能

USB 驱动的含义为，支持连接到 USB 主机控制器上的某一个功能。一个 USB 设备连接到主机上之后体现成何种设备，就是由 USB 驱动决定的。

著名的 USB 驱动包括了大容量存储（U 盘）、USB 串口、HID（人机设备接口）等。USB 驱动实现后可以向其他子系统注册各种设备，例如：可以产生各种设备节点、sys 文件系统接口等。例如一个 USB 鼠标可以产生一个 input 设备的设备节点。

提示：由于 USB 在硬件上是标准接口，各个 USB 设备在不同硬件平台的连接没有区别，因此各个平台不需要针对实现。

2. USB 的骨架实现

`usb-skeleton.c` 是一个简单 USB 驱动的骨架实现，它提供了一个向用户空间的字符设备的节点，也就是把 USB 的底层功能封装成了用户空间设备节点的形式。

`usb-skeleton.c` 文件中实现的 `usb_driver` 的定义如下所示：

```
static struct usb_driver skel_driver = {
    .name = "skeleton", // 驱动名称
    .probe = skel_probe,
    .disconnect = skel_disconnect,
    .suspend = skel_suspend,
    .resume = skel_resume,
    .pre_reset = skel_pre_reset,
    .post_reset = skel_post_reset,
    .id_table = skel_table,
    .supports_autosuspend = 1,
};
```

`skel_driver` 是一个名称为 "skeleton" 的 `usb_driver`，由此表示一个 USB 主机端描述连接

设备的功能。

skel_driver 的探测函数 skel_probe()如下所示:

```
static int skel_probe(struct usb_interface *interface,
                     const struct usb_device_id *id)
{
    struct usb_skel *dev;                // 私有的数据结构
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    size_t buffer_size;
    int i;
    int retval = -ENOMEM;
    dev = kzalloc(sizeof(*dev), GFP_KERNEL); // 分配私有的数据结构 usb_skel
    // ..... 省略错误处理
    kref_init(&dev->kref);
    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
    mutex_init(&dev->io_mutex);
    spin_lock_init(&dev->err_lock);
    init_usb_anchor(&dev->submitted);      // 初始化 usb_anchor 结构
    init_completion(&dev->bulk_in_completion);
    dev->udev = usb_get_dev(interface_to_usbdev(interface)); // 获取 usb_device
    dev->interface = interface;             // 获取 usb_interface
    iface_desc = interface->cur_altsetting; // 设置端点, 使用块输入/输出端点
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;
        if (!dev->bulk_in_endpointAddr && // 设置块输入端点
            usb_endpoint_is_bulk_in(endpoint)) {
            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
            dev->bulk_in_size = buffer_size;
            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        }
        // ..... 省略错误处理
        dev->bulk_in_urb = usb_alloc_urb(0, GFP_KERNEL);
        // ..... 省略错误处理
    }
    if (!dev->bulk_out_endpointAddr && // 设置块输出端点
        usb_endpoint_is_bulk_out(endpoint)) {
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
    // ..... 省略错误处理
    usb_set_intfdata(interface, dev); // 设置私有指针
    retval = usb_register_dev(interface, &skel_class); // 设置 usb_class_driver
    // ..... 省略错误处理
    return 0;
    // ..... 省略错误处理
}
```

usb_skel 是本驱动定义的私有数据结构, 其首个成员为 usb_device, 下一个成员为 usb_interface, 其他成员主要为 USB 块操作的相关内容。

其执行的核心流程就是通过从 usb_interface 获得了 usb_device, 设置其端点, 然后将私有的结构 usb_skel 设置到 usb_interface 当中。通过 usb_interface 得到一个主机端的操作的句柄, 也就是句柄 usb_host_interface 结构, 然后从中取出端点, 对端点进行操作。

最后设置的 skel_class 是 usb_class_driver 结构, 其名称是"skel%d", 用于从 USB 核心

得到一个 USB 的次序号，其中包含了一个 `file_operations` 结构，它将被 USB 核心注册后产生相应的设备节点。

`skel_read()` 和 `skel_write()` 函数分别是其中的读、写实现，它们的功能都通过 USB 核心的 URB 操作完成。例如：`skel_read()` 调用的 `skel_do_read_io()` 函数如下所示：

```
static int skel_do_read_io(struct usb_skel *dev, size_t count)
{
    int rv;
    usb_fill_bulk_urb(dev->bulk_in_urb, dev->udev,           // 填充块传输的 URB
        usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
        dev->bulk_in_buffer, min(dev->bulk_in_size, count),
        skel_read_bulk_callback, dev);
    spin_lock_irq(&dev->err_lock);                          // 读的时候要加锁
    dev->ongoing_read = 1;
    spin_unlock_irq(&dev->err_lock);
    rv = usb_submit_urb(dev->bulk_in_urb, GFP_KERNEL); // 提交 URB 操作
    // ..... 省略错误处理: rv < 0 的情况也需要设置读已经完成
    return rv;
}
```

通过设备节点的读操作，将被转化成通过 USB 核心的 URB 块传输方式来完成。由此，各个 USB 设备可以直接在用户空间来通过设备节点访问其数据。

3. USB 大容量存储器的实现

USB 大容量存储器的实现是针对 USB 存储器的实现，也就是 U 盘在 Linux 主机端的支持。USB 大容量存储器的实现内容在 `drivers/usb/storage` 目录中，主要包括了 `csiglu.c`、`protocol.c`、`transport.c`、`usb.c`、`initializers.c` 等几个文件。

`usb.c` 文件中实现了名称为 "usb-storage" 的 `usb_driver`，作为与 USB 系统联系的核心，`scsiglu.c` 文件负责与 SCSI 系统产生联系，也就是利用 SCSI 系统的结构，提供用户空间中 `sda`、`sda<N>` 等形式的设备节点。

4. USB 串口的实现

为了适应很多桌面电脑已经没有串口的情况，在需要使用串口的时候可以用 USB 转串口硬件支持。USB 串口的实现提供了 USB 转串口硬件在主机端的支持，并需要将其表现成串口的形式。

USB 串口的实现内容在 `drivers/usb/serial/` 目录中，主要包括了 `usb-serial.c`、`generic.c`、`bus.c` 等几个文件。

`usb-serial.c` 文件中实现了名称为 "usb_serial_driver" 的 `usb_driver`，又实现了一个 `tty_operations`，向 `tty` 系统进行了注册，将 USB 串口表现成了 TTY 驱动的形式。

19.2.4 HCI 的实现

在 Linux 系统当中，HCD (Host Controller Driver, 主控制器驱动) 是 HCI (Host Controller Interface, 主控制器接口) 的软件支持。HCI 指 USB 主机端的控制器，HCD 是 USB 的主机端对于真正不同硬件的实现软件。

HCI 有几种不同形式的实现，如下所示。

- 开放主机控制器接口（OHCI）：康柏、微软使用，对 USB 1.1 的支持。
- 通用主机控制器接口（UHCI）：Intel 使用，对 USB 1.1 的支持。
- 增强型主机控制器接口（EHCI）：对 USB 2.0 的支持。
- 扩展型主机控制器接口（XHCI）：Intel 使用，对 USB 3.0 的支持。

Linux 的 HCD 实现包括 UHCI、OHCI 和 EHCI 等几种不同的方式，它们的代码都在 `driver/usb/host` 目录中。几种 USB 主机控制接口以 `<*>-hcd.c` 作为文件名，通用的实现部分为 `ohci-hcd.c`、`uhci-hcd.c` 和 `ehci-hcd.c` 等名字，与之相对应，`ohci.h`、`uhci.h` 和 `ehci.h` 分别为它们的头文件。

提示：Linux 内核中 USB 部分的 EHCI 应用于 USB 2.0；为了使用 USB 1.1 标准，还需要结合 UHCI 或者 OHCI 来使用。

一个具体的硬件 HCD 驱动与通用的 HCI 驱动部分结合之后，才能构成真正的 USB 主机端的驱动。凡是名称为 `ohci-<*>.c`、`ehci-<*>.c` 的文件都是针对某个硬件 HCI 的实现。

19.3 Linux 的 USB 设备端支持

19.3.1 USB 设备端的软件结构

运行 Linux 的系统也可以作为 USB 的设备端。在 Linux 系统的软件方面，USB Gadget 用于实现 USB 设备端的软件功能。

USB Gadget 主要头文件为：`include/usb/gadget.h`。

USB Gadget 主要代码的路径为：`drivers/usb/gadget/`。

USB Gadget 实现的结构相对比较简单，软件从端点的角度支持，实际上只是一个将功能向 USB 进行过渡的端口，具体的最终实现，还取决于本设备实际具有的功能。一般情况下，一个设备只能实现一个功能，例如：U 盘、USB 网卡、USB 音频等。通过 Gadget 驱动程序的实现，可以让这个 USB 设备插入某个 USB 主端口后具有某一个功能。

USB Gadget 分成 3 个层次，下层是针对硬件的 UDC 驱动，中间层是 USB 设备端的核心布局，上层则是具体的 Gadget 驱动（由各种功能组成所谓的 Gadget）。Gadget 的下层实际上是在不同硬件的实现层，上层是各个并列的功能。

19.3.2 Gadget 的核心部分

头文件 `gadget.h` 中包含了 USB Gadget 公共的内容，主要具有以下几方面的内容。

- `usb_request`：表示 USB 的请求。
- `usb_ep`：表示 USB 的端点（endpoint）。
- `usb_ep_ops`：表示 USB 端点的操作。
- `usb_gadget_ops`：表示 Gadget 的操作。

- **usb_gadget**: 表示 Gadget。
- **usb_gadget_driver**: 表示一个 Gadget 驱动。

表示端点的 **usb_ep** 结构如下所示:

```
struct usb_ep {
    void          *driver_data;           // 驱动中的私有数据
    const char     *name;                 // 端点的名称
    const struct usb_ep_ops *ops;         // 表示端点的操作
    struct list_head ep_list;
    unsigned       maxpacket:16;
};
```

usb_ep 是一个 USB 端点在设备端的表示, 其中包含了一个链表, 用于联系这个设备的所有端点。**usb_ep_ops** 结构表示了一个端点特定的硬件操作。

用于端点操作的函数族如下所示:

```
static inline int usb_ep_enable(struct usb_ep *ep,
                                const struct usb_endpoint_descriptor *desc){}
static inline int usb_ep_disable(struct usb_ep *ep){}
static inline struct usb_request *usb_ep_alloc_request(struct usb_ep *ep,
                                                       gfp_t gfp_flags){}
static inline void usb_ep_free_request(struct usb_ep *ep,
                                       struct usb_request *req){}
static inline int usb_ep_queue(struct usb_ep *ep,
                               struct usb_request *req, gfp_t gfp_flags){}
static inline int usb_ep_dequeue(struct usb_ep *ep, struct usb_request *req){}
static inline int usb_ep_set_halt(struct usb_ep *ep){}
static inline int usb_ep_clear_halt(struct usb_ep *ep){}
static inline int usb_ep_set_wedge(struct usb_ep *ep){}
static inline int usb_ep_fifo_status(struct usb_ep *ep){}
static inline void usb_ep_fifo_flush(struct usb_ep *ep){}
```

这些形式为 **usb_ep_***() 的函数族是针对 USB 端点的操作, 在每一个 USB Gadget 的实现中, 调用这些函数完成配置和运行时的功能。

表示操作的 **usb_gadget_ops** 结构如下所示:

```
struct usb_gadget_ops {
    int (*get_frame)(struct usb_gadget *);
    int (*wakeupt)(struct usb_gadget *);
    int (*set_selfpowered)(struct usb_gadget *, int is_selfpowered);
    int (*vbus_session)(struct usb_gadget *, int is_active);
    int (*vbus_draw)(struct usb_gadget *, unsigned mA);
    int (*pullup)(struct usb_gadget *, int is_on);
    int (*ioctl)(struct usb_gadget *, unsigned code, unsigned long param);
};
```

usb_gadget_ops 用于表示 USB 硬件的操作, 通常需要由 UDC 驱动实现。

用于描述一个 USB 从设备的 **usb_gadget** 结构如下所示:

```
struct usb_gadget {
    const struct usb_gadget_ops *ops; // 表示 usb_gadget 的操作
    struct usb_ep *ep0;              // 表示这个设备的 0 端点
    struct list_head ep_list;        // 表示这个设备的端点列表
    enum usb_device_speed speed;
    unsigned is_dualspeed:1;         unsigned is_otg:1;
```

```
unsigned is_a_peripheral:1; unsigned b_hnp_enable:1;
unsigned a_hnp_support:1; unsigned a_alt_hnp_support:1;
const char *name;
struct device dev;
};
```

在 `usb_gadget` 结构当中，关键的内容就是表示 Gadget 操作的 `ops` 成员（`usb_gadget_ops` 类型的指针）和表示 0 号端点的 `ep0` 成员（`usb_ep` 类型的指针）。

另有一些名称为 `usb_gadget_*`() 的函数，用于对 `usb_gadget` 进行操作。

表示 USB 从设备的 `usb_gadget_driver` 结构如下所示：

```
struct usb_gadget_driver {
    char *function;           // 用于描述这个 gadget 的功能
    enum usb_device_speed speed; // 描述设备的速度
    int (*bind)(struct usb_gadget *);
    void (*unbind)(struct usb_gadget *);
    int (*setup)(struct usb_gadget *, const struct usb_ctrlrequest *);
    void (*disconnect)(struct usb_gadget *);
    void (*suspend)(struct usb_gadget *);
    void (*resume)(struct usb_gadget *);
    // ..... 省略部分内容
};
```

`usb_gadget_driver` 结构当中的 `setup` 函数指针将通过 `usb_gadget` 结构的 0 号端点完成建立工作。

一个 USB Gadget 实现应当是构建 `usb_gadget_driver` 并进行注册，经过 `usb_gadget` 对 `usb_ep` 进行操作。

19.3.3 Gadget 驱动

1. Gadget 的功能

Gadget 驱动的一个基本原则是，一个系统当中只有一个 Gadget 设备，但是一个设备可以具有多个功能，实际的功能由通用部分和硬件相关的部分进行组合。

在 Gadget 的源代码目录中，凡是以 `f_` 为前缀的源文件实际上是具体功能的实现，这些文件没有被加入 `Makefile`，而是要被其他的源文件直接包含。

几个典型的 USB Gadget 功能如下所示。

- `f_audio.c`：音频支持，类似 USB 音箱设备。
- `f_hid.c`：人机交互设备，类似 USB 鼠标、键盘。
- `f_serial.c`：USB 串口的支持。
- `f_rndis.c`：RNDIS（Remote Network Driver Interface Specification）的含义为远程网络驱动接口规范，用于在 USB 上面运行 TCP/IP。

经过编译过程的组合之后，各个源代码文件会生成以 `g_` 为前缀的目标文件，则是通过包含源文件的形式，构建一个 Gadget。

提示：Gadget 组织的特点是，编译得到的目标文件名和源代码的文件名经常不同。

由于一个系统只能同时具有一个 Gadget，因此，如果需要事先多个 Gadget 的替换，则可以通过在用户空间重新加载模块（ko）的方式来完成。

2. 功能组合

composite.c 文件实现了将 Gadget 组合起来的功能，而 linux/usb/composite.h 是它提供的头文件。其中定义的结构 usb_composite_driver 和 usb_composite_dev 分别表示组合的驱动和组合的设备。

usb_composite_driver 结构如下所示：

```
struct usb_composite_driver {
    const char          *name;
    const struct usb_device_descriptor *dev;
    struct usb_gadget_strings **strings;
    int (*bind)(struct usb_composite_dev *);
    int (*unbind)(struct usb_composite_dev *);
    void (*suspend)(struct usb_composite_dev *);
    void (*resume)(struct usb_composite_dev *);
};
extern int usb_composite_register(struct usb_composite_driver *);
extern void usb_composite_unregister(struct usb_composite_driver *);
```

usb_composite_driver 结构中的绑定 bind 和解除绑定 unbind 两个函数指针表示设备的连接情况，unbind 和 suspend 函数指针用于电源管理。这些函数指针都是具体的 Gadget 驱动需要实现的。

composite.c 文件当中对注册函数 usb_composite_register()的实现如下所示：

```
int usb_composite_register(struct usb_composite_driver *driver)
{
    if (!driver || !driver->dev || !driver->bind || composite) return -EINVAL;
    if (!driver->name) driver->name = "composite";
    composite_driver.function = (char *) driver->name;
    composite_driver.driver.name = driver->name;
    composite = driver;
    return usb_gadget_register_driver(&composite_driver); // 在 UDC 中实现
}
```

各个具体的 Gadget 驱动就要通过调用 usb_composite_register()函数，将功能进行注册。其中调用 usb_gadget_register_driver()函数的作用其实相当于一个钩子，需要在不同的平台 USB 设备端驱动（UDC）中实现，用于适配不同的平台。

在不同的 Gadget 实现中，如果需要多个 Gadget 功能的组合，则会在 C 语言的源代码中直接包含 composite.c 文件和表示各个具体功能的 f_*.c 文件。

3. 零实现的 Gadget

zero.c 是 USB 零实现，zero.c 文件经过编译后，将生成 g_zero.o 目标文件。此处实现的内容被称为 Gadget Zero，是一个同时包括了输出（sink）和输入（source）的功能，使用块方式进行传输。本功能可以在大多数双向的 USB 控制器实现，可以作为测试。

zero.c 文件的头部如下所示：

```
#include "composite.c"
```

```
#include "usbstring.c"
#include "config.c"
#include "epautoconf.c"
#include "f_sourcesink.c"
#include "f_loopback.c"
```

f_sourcesink.c 和 f_loopback.c 两个文件是 USB 具体功能的实现,前者负责配置 USB 的输入和输出,后者负责连接输出和输入,形成一个环。二者都实现了 USB 的配置(usb_configuration),实际的功能通过对端点的操作完成。

zero.c 文件的入口部分如下所示:

```
static struct usb_composite_driver zero_driver = {
    .name      = "zero",
    .dev       = &device_desc,    // usb_device_descriptor 类型
    .strings   = dev_strings,     // usb_gadget_strings 类型的数组
    .bind      = zero_bind,
    .unbind    = zero_unbind,
    .suspend   = zero_suspend,
    .resume    = zero_resume,
};
static int __init init(void)
{
    return usb_composite_register(&zero_driver);
}
```

此处定义了一个名称为"zero"的 usb_composite_driver,并调用 usb_composite_register()函数进行了注册。

device_desc 结构就是 usb_device_descriptor,表示为 USB 设备的描述符。

4. 大容量存储器的 Gadget

mass_storage.c 是 USB 大容量存储器的实现。mass_storage.c 文件经过编译后,将生成 g_mass_storage.o 目标文件。

mass_storage.c 文件的头部如下所示:

```
#include "composite.c"
#include "usbstring.c"
#include "config.c"
#include "epautoconf.c"
#include "f_mass_storage.c"           // 大容量存储器的功能实现
```

文件中采用了直接包含 C 文件的形式,表示了通过 composite 实现组合功能。本驱动程序自身主要的功能在 f_mass_storage.c 文件中实现,其中又包含了 storage_common.c 文件。

usb_composite_driver 类型的 msg_driver 结构如下所示:

```
static struct usb_composite_driver msg_driver = {
    .name      = "g_mass_storage",
    .dev       = &msg_device_desc, // usb_device_descriptor 类型
    .strings   = dev_strings,     // usb_gadget_strings 类型的数组
    .bind      = msg_bind,
};
```

此处定义了一个名称为"g_mass_storage"的 usb_composite_driver,表明实现了一个

Gadget 功能。

5. 人机交互设备 Gadget

hid.c 是 USB 人机交互设备的实现。hid.c 文件经过编译后，将生成 g_hid.o 目标文件。hid.c 文件的头部如下所示：

```
#include "composite.c"
#include "usbstring.c"
#include "config.c"
#include "epautoconf.c"
#include "f_hid.c"                // HID 的功能实现
```

本驱动程序自身主要的功能在 f_hid.c 文件中实现。hid.c 定义了一个名称为 "g_hid" 的 usb_composite_driver，表明实现了一个 Gadget 功能。f_hid.c 文件通过 USB 的功能实现，并实现了一个 file_operations。

6. Audio Gadget

audio.c 是 USB 音频的实现。audio.c 文件经过编译后，将生成 g_audio.o 目标文件。audio.c 文件的头部如下所示：

```
#include "composite.c"
#include "usbstring.c"
#include "config.c"
#include "epautoconf.c"
#include "u_audio.c"              // 实现对音频系统（ALSA）的调用
#include "f_audio.c"              // 音频的功能实现
```

audio.c 文件当中定义了名称为 "g_audio" 的 usb_composite_driver。

Audio 的具体功能实际上在 f_audio.c 和 u_audio.c 文件中完成，前者通过调用 Linux 内核当中的声音系统完成音频的操作，包括了音频的输出和输入，后者是 Audio 的 USB Gadget 功能。

19.3.4 UDC 驱动的实现

由特定硬件平台的 UDC（USB Device Controller）驱动程序来实现，UDC 的驱动也被称为 Gadget Driver，表示一个具体平台的设备端的硬件驱动。

gadget.h 头文件中定义了两个函数，如下所示：

```
int usb_gadget_register_driver(struct usb_gadget_driver *driver);
int usb_gadget_unregister_driver(struct usb_gadget_driver *driver);
```

这两个函数都将在组合功能的初始化当中被调用，通过调用 usb_gadget_driver 的 bind() 函数，将硬件的功能绑定到 USB 设备端的架构当中。在实现的过程中，又以表示端点操作的 usb_gadget_ops 的实现最为关键。

drivers/usb/gadget/ 目录当中各个名称为 *_udc.c 的文件就是不同平台的 USB 设备端的硬件实现。其中有一些 USB 设备端的控制器是用于桌面计算机的，基于 PCI 总线。

大多数系统的 Gadget 实现都是 UDC 驱动加组合文件（composite.c）。各种 UDC 的实

现都要实现表示端点操作的 `usb_ep_ops` 结构，还要实现 `usb_gadget_register_driver()` 函数，完成这个 UDC 到 Gadget 系统的功能注册。

三星系列处理器的 UDC 实现的文件为 `s3c2410_udc.c`，头文件为 `s3c2410_udc.h`。本文件能支持的是 S3C24xx 系列处理器，所支持的是 USB 1.1 的全速设备端。

`s3c2410_udc.c` 中的 `usb_gadget_register_driver()` 函数如下所示：

```
int usb_gadget_register_driver(struct usb_gadget_driver *driver)
{
    struct s3c2410_udc *udc = the_controller; // s3c2410_udc 包括 usb_gadget 成员
    int retval;
    // ..... 省略错误处理: bind 和 drsetup 不存在或非全速情况, 错误发生参数
    // ..... 省略部分内容: 作为模块的处理
    udc->driver = driver; // 保存 usb_gadget_driver 的句柄
    udc->gadget.dev.driver = &driver->driver;
    // 增加 usb_gadget 中的 driver
    if ((retval = device_add(&udc->gadget.dev)) != 0) {
        printk(KERN_ERR "Error in device_add() : %d\n", retval);
        goto register_error;
    }
    if ((retval = driver->bind (&udc->gadget)) != 0) { // 调用绑定接口
        device_del(&udc->gadget.dev);
        goto register_error;
    }
    s3c2410_udc_enable(udc); // 调用功能函数, 使能 UDC
    return 0;
    // ..... 省略错误处理
}
```

`s3c2410_udc` 是本驱动自定义的结构，实际上就是对 `usb_gadget` 和 `usb_ep` 类型结构的封装和扩展。其中的 `gadget` 成员就是 `usb_gadget`。

`s3c2410_udc_enable()` 等函数都调用了几个核心的读、写函数来实现，如下所示：

```
static inline u32 udc_read(u32 reg)
{
    return readb(base_addr + reg); // 地址采用基地址+寄存器地址的方式
}
static inline void udc_write(u32 value, u32 reg)
{
    writeb(value, base_addr + reg); // 地址采用基地址+寄存器地址的方式
}
static inline void udc_writew(void __iomem *base, u32 value, u32 reg)
{
    writew(value, base + reg); // 地址采用基地址+寄存器地址的方式
}
```

作为对嵌入式处理器的实现，S3C 平台使用的都是对特殊功能寄存器的直接读写，因此 `udc_read()` 和 `udc_write()` 等函数中 `u32` 类型的 `reg` 就是寄存器的地址。基地址 `base` 是为了处理在系列处理器中的差异。

`s3c2410_udc_enable()` 等函数实现的核心内容都是得到端点 `usb_ep`，从中得出各种信息，调用 `udc_write()` 和 `udc_read()` 等函数完成对 USB 设备端硬件的操作。

本驱动定义的 `usb_gadget` 当中的 `usb_gadget_ops` 的定义如下所示：

```
static const struct usb_gadget_ops s3c2410_ops = {
    .get_frame      = s3c2410_udc_get_frame,
    .wakeup         = s3c2410_udc_wakeup,
    .set_selfpowered = s3c2410_udc_set_selfpowered,
    .pullup         = s3c2410_udc_pullup,
    .vbus_session   = s3c2410_udc_vbus_session,
    .vbus_draw      = s3c2410_vbus_draw,
};
```

s3c2410_ops，表示本 USB 设备端的操作实现。其中包括了 usb_gadget_ops 中 ioctl 以外所有的函数指针。各个函数指针的实现也是基于 udc_write()和 udc_read()完成的。

```
static int s3c2410_udc_get_frame(struct usb_gadget *_gadget)
{
    int tmp;
    tmp = udc_read(S3C2410_UDC_FRAME_NUM2_REG) << 8; // 读高 8 位寄存器
    tmp |= udc_read(S3C2410_UDC_FRAME_NUM1_REG);      // 读低 8 位寄存器，并组成 16 位
    return tmp;
}
```

s3c2410_udc_get_frame()函数通过读 S3C2400 系列处理器的 USB 设备寄存器，获得数据，并将其返回。

第 20 章

SPI 总线和驱动

20.1 SPI 概述

SPI (Serial Peripheral Interface) 含义为串行外围接口，是摩托罗拉公司首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要应用在 EEPROM、Flash、实时时钟、AD 转换器，还用于数字信号处理器和数字信号解码器之间。

SPI 接口用于 CPU 和外围低速器件之间进行同步串行数据传输，在主器件的移位脉冲下，数据按位传输，高位在前，低位在后，为全双工通信，数据传输速度总体来说比 I2C 总线要快。

SPI 的单主单从连接和 SPI 的数据寄存器如图 20-1 所示。

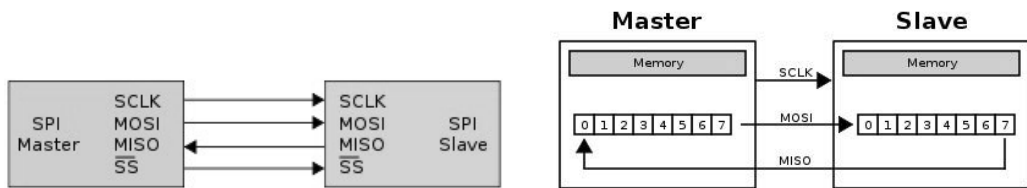


图 20-1 SPI 的单主单从连接（左）和 SPI 的数据寄存器（右）

SPI 四根线的名称和功能如下所示。

- MOSI: 主器件数据输出，从器件数据输入，也称为 SDO。
- MISO: 主器件数据输入，从器件数据输出，也称为 SDI。
- SCLK: 时钟信号，由主器件产生，最大为 $f_{PCLK}/2$ ，从模式频率最大为 $f_{CPU}/2$ 。
- SS: 从器件使能信号，由主器件进行控制，有的芯片中称为 CS (Chip Select)。

SPI 的主控制器和从设备连接之后，通常是将移位寄存器各位输出到引脚上，按照时钟频率发送。

SPI 的基本时序如图 20-2 所示。

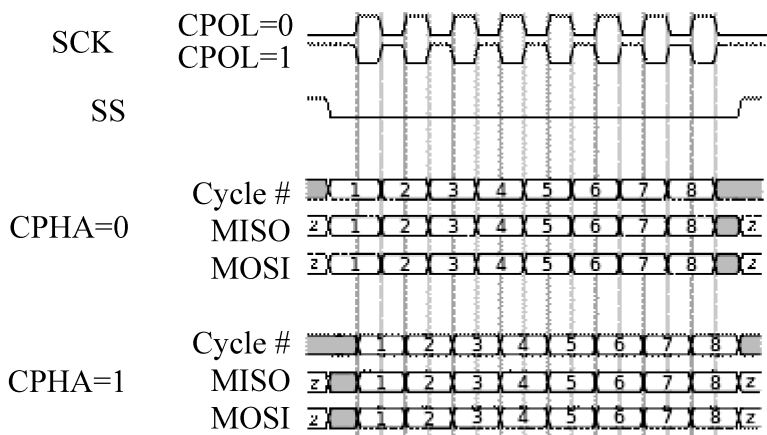


图 20-2 SPI 的基本时序

SPI 的工作模式是同步的，数据引脚 MOSI 和 MISO 仅用于传送数据，格式非常简单，同步功能则由时钟线 SCLK 提供。

在嵌入式系统中，通常的嵌入式处理器当中一般具有 SPI 主控制器，通过片选信号连接不同 SPI 从设备，然后就可以通过不同地址的选择与不同的 SPI 从设备进行交互。同一个时间只有一个 SPI 从设备有效。

典型 SPI 连接和 Daisy-chained 的 SPI 连接如图 20-3 所示。

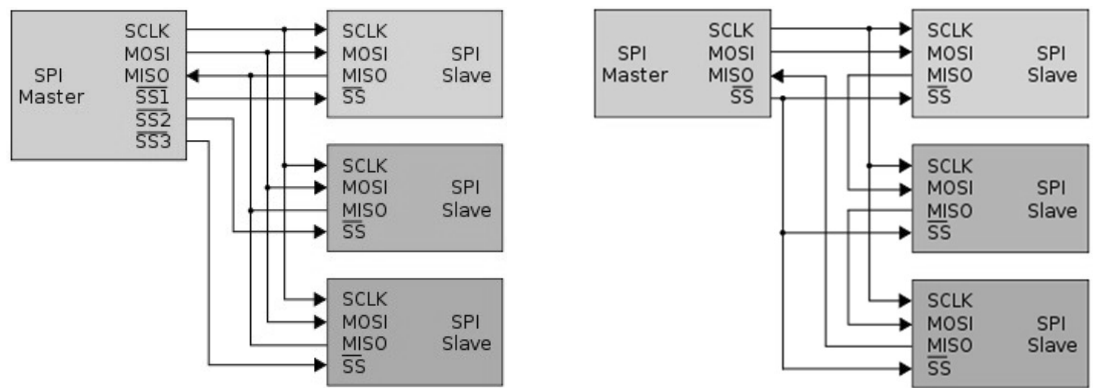


图 20-3 典型 SPI 连接（左）和 Daisy-chained 的 SPI 连接（右）

Daisy-chained 的含义是菊花链、顺序链，在这种模式中将从设备的芯片逐级相连，通过软件控制传输数据，可以节省片选引脚。

20.2 SPI 总线驱动的框架

Linux 的 SPI 框架是一个总线型的框架，SPI 的驱动架构可以分为 3 个层次：SPI 核心

spi_message 结构如下所示:

```
struct spi_message {
    struct list_head transfers;
    struct spi_device *spi;           // 这个消息联系到的 SPI 从设备
    unsigned    is_dma_mapped:1;
    void        (*complete)(void *context); // 完成时刻调用的回调函数
    void        *context;
    unsigned    actual_length;
    int         status;
    // .....省略部分内容: 由 SPI 驱动所有 spi_message
};
```

spi_message 就是在 SPI 设备上发送的消息,用于执行一个数据传输队列中的原子操作。SPI 总线之后等待一个 spi_message 传输完成后,才能传输下一个。spi_message 结构中的 complete 函数指针负责通知传输完成。一个发送给 spi_device 的 spi_message 通常以 FIFO 的形式执行。

SPI 消息的几个相关函数如下所示:

```
static inline void spi_message_init(struct spi_message *m){}
static inline void spi_message_add_tail(struct spi_transfer *t,
                                       struct spi_message *m){}
static inline void spi_transfer_del(struct spi_transfer *t){}
static inline struct spi_message *spi_message_alloc(unsigned ntrans,
                                                    gfp_t flags) {}
static inline void spi_message_free(struct spi_message *m) {}
```

spi_message_init()等几个函数完成 spi_message 结构的构建,对消息的实际操作,还需要由其他的函数完成。

SPI 设备的操作函数如下所示:

```
extern int spi_setup(struct spi_device *spi);
extern int spi_async(struct spi_device *spi, struct spi_message *message);
extern int spi_sync(struct spi_device *spi, struct spi_message *message);
static inline int spi_write(struct spi_device *spi, const u8 *buf, size_t len);
static inline int spi_read(struct spi_device *spi, u8 *buf, size_t len);
extern int spi_write_then_read(struct spi_device *spi,
                               const u8 *txbuf, unsigned n_tx, u8 *rxbuf, unsigned n_rx);
static inline ssize_t spi_w8r8(struct spi_device *spi, u8 cmd);
static inline ssize_t spi_w8r16(struct spi_device *spi, u8 cmd);
```

以上函数表示了对一个 SPI 设备 spi_device 的操作。这些读、写函数最终都通过 spi_async() 函数完成,操作的基础是通过 SPI 主控制器 spi_master 完成的, spi_message 是其中的主要参数。

表示 SPI 主控制的 spi_master 结构如下所示:

```
struct spi_master {
    struct device dev;           // 表示主控制器的设备
    s16 bus_num;                // 总线序号
    u16 num_chipselect;          // 片选号
    u16 dma_alignment;           // SPI 控制器的 DMA 内存信息
    u16 mode_bits;               // 对应于 spi_device.mode
    u16 flags;                   // 其他标志
#define SPI_MASTER_HALF_DUPLEX BIT(0) // 全双工 (Duplex)
```

```
#define SPI_MASTER_NO_RX BIT(1) // 读
#define SPI_MASTER_NO_TX BIT(2) // 写
int (*setup)(struct spi_device *spi);
int (*transfer)(struct spi_device *spi, struct spi_message *mesg); // 传输
void (*cleanup)(struct spi_device *spi);
};
```

bus_num 为该控制器对应的 SPI 总线号。num_chipselect 为控制器支持的片选数量，即能支持多少个 spi 设备。setup 函数指针要设置 SPI 总线的模式、时钟等的初始化函数。针对设备设置 SPI 的工作时钟及数据传输模式等，在 spi_add_device 函数中对其进行调用。transfer 函数指针是实现 SPI 总线的实现核心，实现数据的双向传输，可能会睡眠。

SPI 主控制器的操作函数如下所示：

```
extern struct spi_master * spi_alloc_master(struct device *host, unsigned size);
extern int spi_register_master(struct spi_master *master);
extern void spi_unregister_master(struct spi_master *master);
extern struct spi_master *spi_busnum_to_master(u16 busnum);
```

一个主控制器的实现者将 spi_master 结构分配之后，会注册到系统中。

SPI 设备和 SPI 驱动用于 SPI 从设备的功能，SPI 从设备通过 SPI 主控制器工作。

spi_device 设备的结构如下所示：

```
struct spi_device {
    struct device dev;
    struct spi_master *master; // SPI 从设备连接的主控制器
    u32 max_speed_hz; // 最大的速度
    u8 chip_select; // 片选
    u8 mode; // 由 SPI_CPHA 等宏表示
    u8 bits_per_word; // 每个字的位数
    int irq; // 中断号
    void *controller_state;
    void *controller_data;
    char modalias[SPI_NAME_SIZE]; // 别名
};
```

spi_device 表示 SPI 从设备在主设备中的代理（Proxy）。程序本身只能控制 SPI 主控制器，因此所有针对 SPI 设备的操作都要转化成 SPI 主控制器来完成。

spi_driver 结构如下所示：

```
struct spi_driver {
    const struct spi_device_id *id_table; // 该驱动支持的设备
    int (*probe)(struct spi_device *spi);
    int (*remove)(struct spi_device *spi);
    void (*shutdown)(struct spi_device *spi);
    int (*suspend)(struct spi_device *spi, pm_message_t mesg);
    int (*resume)(struct spi_device *spi);
    struct device_driver driver;
};
```

spi_driver 表示主设备端的驱动。spi_driver 为 spi_device 服务。spi_driver 注册时会扫描 SPI bus 上的设备，进行驱动和设备的绑定。probe 函数用于驱动和设备匹配时被调用。

spi_board_info 结构如下所示：

```

struct spi_board_info {
    char        modalias[SPI_NAME_SIZE];    // 别名
    const void   *platform_data;
    void        *controller_data;
    int         irq;                        // 中断号
    u32         max_speed_hz;              // 最大的速度
    u16         bus_num;                   // 总线号
    u8          chip_select;               // 片选
    u8          mode;                      // 由 SPI_CPHA 等宏表示
};

```

`spi_board_info` 用于在一个系统的板级定义中表示所有的 SPI 从设备。定义的目的就是为了让系统软件知道 SPI 硬件的连接情况。

提示： SPI 的主控制器对连接设备没有动态检测能力，因此系统中连接于 SPI 总线的设备都要通过板级定义的方式来确定。

20.3 简单字符设备 spidev

SPI 有一个通用的简单字符设备，被称为 `spidev`，其主设备号为 153，次设备号依次递增。SPI 简单字符设备为 SPI 封装了在用户空间的简单接口，可以用于简单的半双工或者全双工操作。在用户空间中的设备节点通常是 `/dev/spidev<N>.<CS>`，`<N>` 表示 SPI 主控制器的序号，`<CS>` 表示 SPI 从设备的片选地址。

`spidev` 实际上就是一个 SPI 的“万能驱动”，可以用于所有的 SPI 从设备，它不关心设备特有的功能，只是能通过原始数据操作的方式操作 SPI 从设备。

`spidev` 的头文件为：`include/linux/spi/spidev.h`。

`spidev` 的源代码为：`drivers/spi/spidev.c`。

`spidev` 对用户空间提供了读、写和 `ioctl` 等接口，并实现了名为“`spidev`”的 `spi_driver`，因此它就是 SPI 从设备驱动的一个具体实现。从功能的角度来说，该实现只是 SPI 主控制器到用户空间的封装。它也会创建 `sys` 文件系统的 `/sys/class/spidev/` 目录。

`spidev.h` 文件中定义了 `ioctl` 的命令号，如下所示：

```

#define SPI_IOC_RD_MODE          _IOR(SPI_IOC_MAGIC, 1, __u8)
#define SPI_IOC_WR_MODE          _IOW(SPI_IOC_MAGIC, 1, __u8)
#define SPI_IOC_RD_LSB_FIRST     _IOR(SPI_IOC_MAGIC, 2, __u8)
#define SPI_IOC_WR_LSB_FIRST     _IOW(SPI_IOC_MAGIC, 2, __u8)
#define SPI_IOC_RD_BITS_PER_WORD _IOR(SPI_IOC_MAGIC, 3, __u8)
#define SPI_IOC_WR_BITS_PER_WORD _IOW(SPI_IOC_MAGIC, 3, __u8)
#define SPI_IOC_RD_MAX_SPEED_HZ  _IOR(SPI_IOC_MAGIC, 4, __u32)
#define SPI_IOC_WR_MAX_SPEED_HZ  _IOW(SPI_IOC_MAGIC, 4, __u32)

```

主要的控制功能包括了读写、位测试、最大速度的读和写等。

`spi_ioc_transfer` 和消息处理如下所示：

```

struct spi_ioc_transfer {
    __u64    tx_buf;        // 发送缓冲
    __u64    rx_buf;        // 接收缓冲
};

```

```

__u32    len;                // 数据的长度
__u32    speed_hz;
__u16    delay_usecs;
__u8     bits_per_word;
__u8     cs_change;
__u32    pad;
};
#define SPI_MSGSIZE(N) \
    (((N)*(sizeof (struct spi_ioc_transfer))) < (1 << _IOC_SIZEBITS)) \
    ? ((N)*(sizeof (struct spi_ioc_transfer))) : 0)
#define SPI_IOC_MESSAGE(N) _IOW(SPI_IOC_MAGIC, 0, char[SPI_MSGSIZE(N)])

```

`spi_ioc_transfer` 用于描述一个单独的数据传输，`ioc` 的意思就是 `ioctl`。`SPI_MSGSIZE` 宏用于构建一个 `spi_ioc_transfer` 结构的数组，`SPI_IOC_MESSAGE` 是以 `spi_ioc_transfer` 结构为参数类型的 `ioctl` 号。

`spidev.c` 文件当中实现了 `file_operations` 结构 `spidev_fops`，并注册为主设备号是 `SPIDEV_MAJOR` (153) 的字符设备。其中实现的读写函数，最终都调用了 `spi_async()` 函数，也就是调用了主控制器来进行实现。当具有一个 `spi_device`，并向系统注册之后，就会根据其主控制器的总线号 (`bus_num`) 和片选号 (`chip_select`) 形成一个设备节点。

内核中给了一段用户空间调用 `spidev` 的参考程序 `spidev_test.c`，如下所示：

```

int main(int argc, char *argv[])
{
    int ret = 0;
    int fd;
    parse_opts(argc, argv);           // 解析命令行的参数
    fd = open(device, O_RDWR);        // 打开"/dev/spidev1.1"文件
    if (fd < 0) pabort("can't open device");
    ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);    // 设置 SPI 的模式
    if (ret == -1) pabort("can't set spi mode");
    ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);    // 获取 SPI 的模式
    if (ret == -1) pabort("can't get spi mode");
    ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);    // 设置每个字节位数
    if (ret == -1) pabort("can't set bits per word");
    ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);    // 获取每个字节位数
    if (ret == -1) pabort("can't get bits per word");
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);    // 设置最大速度
    if (ret == -1) pabort("can't set max speed hz");
    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);    // 获取最大速度
    if (ret == -1) pabort("can't get max speed hz");
    printf("spi mode: %d\n", mode);
    printf("bits per word: %d\n", bits);
    printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);
    transfer(fd);           // 对文件进行真正的操作
    close(fd);
    return ret;
}

```

`parse_opts()`只用于解析命令行参数，用于选择执行不同的功能，以上的 `ioctl` 控制用于完成设备的初始化。`transfer()`函数用于交互。

```

static void transfer(int fd)
{
    int ret;

```

```

uint8_t tx[] = { // 表示发送的数据
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
    0xF0, 0x0D,
};
uint8_t rx[ARRAY_SIZE(tx)] = {0, }; // 接收数据的大小和发送数据相同
struct spi_ioc_transfer tr = { // 构建用于发送 spi_ioc_transfer 结构
    .tx_buf = (unsigned long)tx,
    .rx_buf = (unsigned long)rx,
    .len = ARRAY_SIZE(tx),
    .delay_usecs = delay,
    .speed_hz = speed,
    .bits_per_word = bits,
};
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr); // 进行一次交互
if (ret < 1)    perror("can't send spi message");
for (ret = 0; ret < ARRAY_SIZE(tx); ret++) { // 打印得到的结果
    if (!(ret % 6))    puts("");
    printf("%.2X ", rx[ret]);
}
puts("");
}

```

spidev 实际上是通用而简单的设备，只是为每个注册的设备提供了一个用户空间的节点，对这些设备的操作形式为原始的接口。

20.4 SPI 主控制器的实现

SPI 主控制器是 SPI 总线在处理器端的驱动程序。很多嵌入式处理器之中都具有 SPI 控制器，可以通过它去控制各个 SPI 总线上面的设备。由于 SPI 信号比较简单，因此通过 GPIO 模拟 SPI 也是常见的形式。

20.4.1 GPIO 实现的 SPI 主控制器

Bit-banging 是一种用软件替代专职硬件的串行通信技术。软件直接对微处理器引脚的状态进行设置和采样，其功能包括：时钟、电平、同步等所有参数。由于 SPI 的引脚信号比较简单，因此即使系统中没有 SPI 主控制器，用若干 GPIO 引脚模拟 MOSI、MISO、SCK 和片选信号也可以实现 SPI 主控制器。由于实际硬件系统的限制，GPIO 模拟 SPI 的速度不可能达到很高。

内核中提供了 GPIO 模拟 SPI 主控器的实现 (spi_gpio)，它基于 GPIO 在内核中的接口实现了一个 SPI 主控制器。为了同样的一个驱动可以在不同硬件中使用，GPIO 的地址可以在板级进行配置。

spi_gpio 内核源代码为 driver 目录中的 spi_gpio.c 和 spi_bitbang.c。

spi_gpio.c 文件中实现了一个名称为 "spi_gpio" 的平台驱动，其探测函数如下所示：

```

static int __init spi_gpio_probe(struct platform_device *pdev)
{
    int                status;

```

```

    struct spi_master      *master;        // 主控制器结构
    struct spi_gpio        *spi_gpio;      // 本驱动定义的数据结构
    struct spi_gpio_platform_data *pdata;
    pdata = pdev->dev.platform_data;
// ..... 省略部分内容
    status = spi_gpio_request(pdata, dev_name(&pdev->dev));
// ..... 省略错误处理
    master = spi_alloc_master(&pdev->dev, sizeof *spi_gpio); // 分配 SPI 主控制器
// ..... 省略错误处理
    spi_gpio = spi_master_get_devdata(master); // 获取平台数据
    platform_set_drvdata(pdev, spi_gpio);      // 设置 spi_gpio 结构为平台数据
    spi_gpio->pdev = pdev;
    if (pdata) spi_gpio->pdata = *pdata;
    master->bus_num = pdev->id;                  // SPI 主控制器的基本信息和实现
    master->num_chipselect = SPI_N_CHIPSEL;
    master->setup = spi_gpio_setup;
    master->cleanup = spi_gpio_cleanup;
// ..... 省略: spi_gpio 结构信息的结构
    status = spi_bitbang_start(&spi_gpio->bitbang); // 开始模拟 SPI 主控制器
// ..... 省略错误处理
    return status;
// ..... 省略错误处理
}

```

使用 GPIO 模拟 SPI 主控制器的过程比较直接，就是引脚时序的模拟。但是到了具体平台中，使用哪个引脚需要配置，因此在驱动中需要获取 GPIO 引脚的配置信息。

在 spi_gpio.h 当中包含了如下结构：

```

struct spi_gpio_platform_data {
    unsigned sck;
    unsigned mosi;
    unsigned miso;
    u16      num_chipselect;
};

```

spi_gpio_platform_data 结构的四个成员表明了模拟 SPI 四根引脚的 GPIO 号。在板级的定义中，需要定义一个 spi_gpio_platform_data 结构，并且作为名称为"spi_gpio"的平台设备（platform_device）之中的平台数据（platform_data）。由此，表示了不同平台对于模拟 SPI 的 GPIO 号的分配。

20.4.2 S3C64xx 的 SPI 主控制器

三星的 S3C 系列处理器都具有 SPI 控制器。各个处理器的 SPI 控制器的寄存器地址和功能大致相同，但在数目和细节上略有区别。

S3C64xx 系列处理器的 SPI 主控制器的实现源代码为 driver/spi/spi_s3c64xx.c。spi_s3c64xx.c 文件中实现了名为"s3c64xx-spi"的平台驱动。

其探测函数 s3c64xx_spi_probe()如下所示：

```

static int __init s3c64xx_spi_probe(struct platform_device *pdev)
{
    struct resource      *mem_res, *dmatx_res, *dmarx_res;
    struct s3c64xx_spi_driver_data *sdd; // 本驱动程序的私有数据
    struct s3c64xx_spi_info *sci;

```

```

    struct spi_master *master;
    int ret;
// ..... 省略错误处理
    dmatx_res = platform_get_resource(pdev, IORESOURCE_DMA, 0); // 发送的 DMA 资源
// ..... 省略错误处理
    dmarx_res = platform_get_resource(pdev, IORESOURCE_DMA, 1); // 接收的 DMA 资源
// ..... 省略错误处理
    mem_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
// ..... 省略错误处理
    master = spi_alloc_master(&pdev->dev, sizeof(struct s3c64xx_spi_driver_data));
// ..... 省略错误处理
    sci = pdev->dev.platform_data;
    platform_set_drvdata(pdev, master);
    sdd = spi_master_get_devdata(master);
    sdd->master = master; // 设置 s3c64xx_spi_driver_data 结构
    sdd->cntrlr_info = sci;
    sdd->pdev = pdev;
    sdd->sfr_start = mem_res->start;
    sdd->tx_dmach = dmatx_res->start;
    sdd->rx_dmach = dmarx_res->start;
    sdd->cur_bpw = 8;
    master->bus_num = pdev->id; // 设置 spi_master 结构
    master->setup = s3c64xx_spi_setup;
    master->transfer = s3c64xx_spi_transfer;
    master->num_chipselect = sci->num_cs;
    master->dma_alignment = 8;
    master->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
// ..... 省略错误处理
    sdd->regs = ioremap(mem_res->start, resource_size(mem_res)); // 寄存器地址
// ..... 省略错误处理
    sdd->clk = clk_get(&pdev->dev, "spi");
// ..... 省略错误处理
    sdd->src_clk = clk_get(&pdev->dev, sci->src_clk_name);
// ..... 省略错误处理
    sdd->workqueue = create_singlethread_workqueue( // 创建 workqueue
        dev_name(master->dev.parent));
// ..... 省略错误处理
    s3c64xx_spi_hwinit(sdd, pdev->id); // 进行硬件的初始化工作：包括硬件寄存器操作
    spin_lock_init(&sdd->lock); // 进行线程的初始化
    init_completion(&sdd->xfer_completion);
    INIT_WORK(&sdd->work, s3c64xx_spi_work);
    INIT_LIST_HEAD(&sdd->queue);
// ..... 省略错误处理
    return 0;
// ..... 省略错误处理
}

```

用于传输的 `s3c64xx_spi_transfer()` 函数中使用队列来处理，表示队列的 `s3c64xx_spi_work()` 函数中调用了 `handle_msg()`，其中包括了硬件寄存器的操作。在配置方面，S3C64xx 系列处理器的 SPI 主控制器需要使用不同的寄存器基地址和 DMA 资源等，这些内容要从平台数据中得出。

例如在板级定义的 `arch/arm/mach-s3c64xx/dev-spi.c` 文件中，具有如下内容：

```

struct platform_device s3c64xx_device_spi0 = {
    .name      = "s3c64xx-spi",
    .id        = 0,

```

```

        .num_resources      = ARRAY_SIZE(s3c64xx_spi0_resource),
        .resource           = s3c64xx_spi0_resource,
        .dev = {
            .dma_mask        = &spi_dmamask,
            .coherent_dma_mask = DMA_BIT_MASK(32),
            .platform_data   = &s3c64xx_spi0_pdata,
        },
    };
EXPORT_SYMBOL(s3c64xx_device_spi0);

```

s3c64xx_device_spi0 就是一个同名平台驱动使用的平台设备，其中包括了特定的寄存器地址、中断资源和 DMA 资源。

20.5 SPI 从设备的驱动

从系统角度来说，SPI 从设备必须通过 SPI 主控制器来操作。从硬件的角度来说，SPI 从设备要通过 SPI 总线的接口进行控制，而这种总线接口又是由 SPI 主控制器所实现的。从用户空间的角度来说，软件关心的是 SPI 从设备各种各样的功能，而不是 SPI 总线本身。因此，需要针对特定 SPI 驱动通过调用 SPI 总线来实现，并提供成各种类型的接口，例如：字符设备、sys 文件系统等。

SPI 从设备的驱动实现的是 spi_driver，实现的方式仅仅和各种各样的 SPI 从设备相关，而与 SPI 总线接口无关，并不直接调用 SPI 主控器的实现。一种 SPI 驱动一般只能服务于一种 SPI 从设备，它们之间的联系就是 SPI 从设备在板级定义，让 SPI 核心知道当前系统连接哪些 SPI 从设备，SPI 从设备驱动按照名称实现服务于同名设备。

AT25 有一种 SPI 接口的 EEPROM，在内核的源代码中，它作为从设备工作，要实现的 spi_driver，其代码路径为 driver/misc/eeprom/at25.c。at25.c 当中实现了 spi_driver，并提供了 sys 文件系统作为到用户空间的接口。at25 的 SPI 设备驱动程序通过了一个层次封装，可以利用 sys 文件系统访问连接于 SPI 总线上的 EEPROM。

spi_driver 文件中定义的 spi_driver 如下所示：

```

static struct spi_driver at25_driver = {
    .driver = {
        .name      = "at25",
        .owner      = THIS_MODULE,
    },
    .probe         = at25_probe,
    .remove        = __devexit_p(at25_remove),
};

```

at25_driver 结构的类型为 spi_driver，SPI 驱动程序的名称为"at25"。此处就作为驱动程序的入口，其探测函数 at25_probe()如下所示：

```

static int at25_probe(struct spi_device *spi)
{
    struct at25_data *at25 = NULL;    // 本驱动程序定义的私有结构
    const struct spi_eeprom *chip;
    int err;

```

```

int          sr;
int          addrlen;
chip = spi->dev.platform_data;    // 获得设备的平台数据
// ..... 省略错误处理, 省略: 设置 8、16 和 24 位的地址
sr = spi_w8r8(spi, AT25_RDSR);    // 通过主控制器, 读一个范围的状态寄存器
// ..... 省略错误处理
if (!(at25 = kzalloc(sizeof *at25, GFP_KERNEL))) {
    err = -ENOMEM;
    goto fail;
}
mutex_init(&at25->lock);
at25->chip = *chip;                // 设置私有结构 at25_data
at25->spi = spi_dev_get(spi);      // 获得地址等信息
dev_set_drvdata(&spi->dev, at25);
at25->addrlen = addrlen;
sysfs_bin_attr_init(&at25->bin);  // at25->bin 类型为 bin_attribute
at25->bin.attr.name = "eeprom";    // sys 文件系统的文件名称
at25->bin.attr.mode = S_IRUSR;
at25->bin.read = at25_bin_read;    // 设置读的功能
at25->mem.read = at25_mem_read;
at25->bin.size = at25->chip.byte_len;
if (!(chip->flags & EE_READONLY)) { // 设置写的功能
    at25->bin.write = at25_bin_write;
    at25->bin.attr.mode |= S_IWUSR;
    at25->mem.write = at25_mem_write;
}
// ..... 省略错误处理
err = sysfs_create_bin_file(&spi->dev.kobj, &at25->bin); // 创建 sys 文件系统
if (err) goto fail;
if (chip->setup)
    chip->setup(&at25->mem, chip->context);
return 0;
// ..... 省略错误处理
}

```

本驱动程序中通过调用 `sysfs_create_bin_file()` 函数, 在 `sys` 文件系统中建立了二进制文件, 名为 "eeprom", 作为 EEPROM 对用户空间的接口。

`at25_bin_read()` 函数通过调用 `at25_ee_read()` 函数实现, 其内容如下所示:

```

at25_ee_read(struct at25_data *at25, char *buf, unsigned offset, size_t count)
{
    u8  command[EE_MAXADDRLLEN + 1]; // 表示发送的命令
    u8  *cp;
    ssize_t          status;
    struct spi_transfer t[2]; // 调用 SPI 核心的传输机制
    struct spi_message m;    // 用于传递参数信息的 SPI 消息
// ..... 省略部分内容
    spi_message_init(&m);
    memset(t, 0, sizeof t);
    t[0].tx_buf = command; // 初始化发送的缓冲区
    t[0].len = at25->addrlen + 1;
    spi_message_add_tail(&t[0], &m);
    t[1].rx_buf = buf;     // 初始化接收的缓冲区
    t[1].len = count;
}

```

```
spi_message_add_tail(&t[1], &m);  
mutex_lock(&at25->lock);  
status = spi_sync(at25->spi, &m);    // 调用 SPI 系统的接口进行交互  
mutex_unlock(&at25->lock);  
return status ? status : count;  
}
```

由此，at25 驱动程序的操作通过 spi_sync() 等函数向 SPI 总线发送 spi_message 完成。并且进行了一个层次封装，因此可以利用 sys 文件系统访问连接与 SPI 总线上的 EEPROM。

第 21 章

I2C 总线和驱动

21.1 I2C 概述

21.1.1 基本概念

I2C (Inter-Integrated Circuit, 又称 I-squared-C、I-two-C、IIC) 总线是一种由飞利浦公司 (目前为 NXP) 开发的两线式串行总线, 最主要的优点是其简单性和有效性。I2C 总线支持多主控 (multimastering), 其中任何能够进行发送和接收的设备都可以成为主总线。一个主控能够控制信号的传输和时钟频率。在任何时间点, I2C 总线都只能有一个主控。

I2C 的最高传输速率为 100kb/s。每个电路和模块都有唯一的地址, 在信息的传输过程中, I2C 总线上并接的每一模块电路既是主控器或被控器, 又是发送器或接收器。

I2C 的官方文档为: http://www.nxp.com/documents/user_manual/UM10204.pdf。

I2C 标志和连接结构如图 21-1 所示。

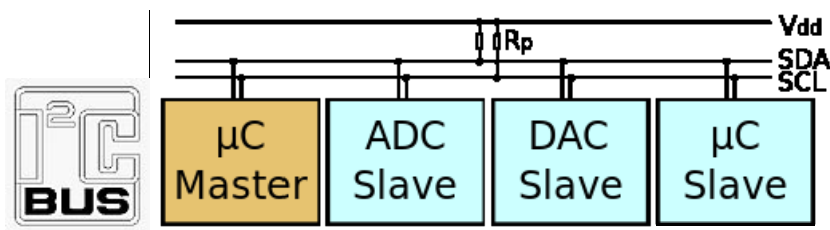


图 21-1 I2C 标志 (左) 和 I2C 连接结构 (右)

在硬件上, I2C 总线是由数据线 (SDA) 和时钟 (SCL) 构成的串行总线, 可发送和接收数据。SDA 是双向数据线, SCL 是时钟线。在 I2C 总线上传送数据, 首先送最高位, 由主机发出启动信号, SDA 在 SCL 高电平期间由高电平跳变为低电平, 然后由主机发送一个字节的的数据。数据传送完毕, 由主机发出停止信号, SDA 在 SCL 高电平期间由低电平跳变为高电平。

I2C 的数据传输如图 21-2 所示。

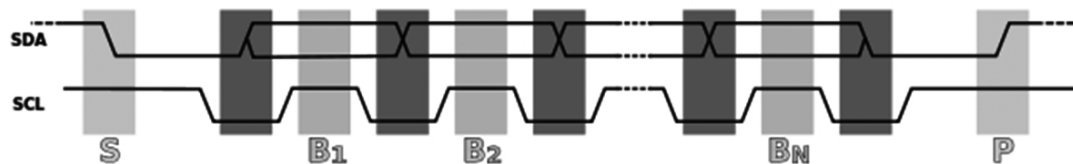


图 21-2 I2C 的数据传输

I2C 主控制器发出的控制信号分为地址码和控制量两部分：地址码用来选址，即接通需要控制的电路，确定控制的种类；控制量决定该调整类别及需要调整的量。这样，各控制电路虽然挂在同一条总线上，却彼此独立，互不相关。

提示：在嵌入式系统中，I2C 总线通常用于连接嵌入式处理器和外围设备。嵌入式处理器具有多个 I2C 控制器，可以用于连接 E2PROM、触摸屏、传感器等 I2C 设备。

21.1.2 SMBus

SMBus（System Management Bus）的含义为系统管理总线，1995 年由 Intel 公司提出，应用于移动 PC 和桌面 PC 系统中的低速率通信。它主要是希望通过一条廉价并且功能强大的总线（由两条线组成），来控制主板上的设备并收集相应的信息。

SMBus 大部分基于 I2C 总线规范，是一种二线制串行总线。与 I2C 一样，SMBus 无须增加额外引脚。创建该总线主要是为了增加新的功能特性，但只工作在 100kHz 以内，且专门面向智能电池管理应用。它工作在主-从模式：主器件提供时钟，在其发起一次传输时提供一个起始位，在其终止一次传输时提供一个停止位；从器件拥有一个唯一的 7 或 10 位从器件地址。

SMBus 与 I2C 总线之间在时序特性上存在一些差别。首先，SMBus 需要一定数据保持时间，而 I2C 总线则是从内部延长数据保持时间。SMBus 具有超时功能，因此当 SCL 太低而超过 35ms 时，从器件将复位正在进行的通信。相反，I2C 采用硬件复位。SMBus 具有一种警报响应地址（ARA），因此当从器件产生一个中断时，它不会马上清除中断，而是一直保持到其收到一个由主器件发送的含有其地址的 ARA 为止。SMBus 只工作在 10kHz 到最高 100kHz。最低工作频率 10kHz 是由 SMBus 超时功能决定的。

提示：SMBus 和普通 I2C 的基本传输规范相同，只是数据格式略有不同。

21.2 I2C 总线驱动的框架

21.2.1 I2C 核心框架

I2C 驱动框架是一个总线框架，驱动核心功能就是将一个 I2C 控制器适配到 Linux 的

- `drivers/i2c/i2c-boardinfo.c`: 用于 I2C 从设备的注册管理。
- `drivers/i2c/algos/`: I2C 的相关算法, 也就是 I2C 的通信方法。
- `drivers/i2c/busses/`: I2C 主控制器设备的实现。

提示: I2C 总线本身可能就是一个平台设备 (嵌入式系统), 也可能是 PCI 设备 (桌面计算机)。

当 I2C 主控制器设备实现, 并且各个从设备正确表示之后, 它们的信息可以体现在 `sys` 文件系统中。`sys` 文件系统中 I2C 总线相关的内容包括以下几个目录。

- `/sys/bus/i2c/devices/`: 表示 I2C 总线上的设备, 包括 `i2c-<N>`, `<N>-<addr>` 等目录。
- `/sys/bus/i2c/drivers/`: 表示 I2C 总线上的驱动。
- `/sys/class/i2c-adapter/`: 表示 I2C 的总线, 格式为 `i2c-<N>`, 如: `i2c-1`、`i2c-2` 等。
- `/sys/class/i2c-dev/`: 表示 I2C 的设备。

其中最为关键的目录是 `/sys/bus/i2c/devices/i2c-<N>/`, 在这个目录中包括了名为 `<N>-<addr>` 的各个子目录, `<N>` 表示 I2C 主控制器的序号, `<addr>` 表示从设备的地址。

`i2c.h` 当中的几个主要结构含义如下所示。

- `i2c_adapter` 结构: 表示一个 I2C 主控制器 (总线)。
- `i2c_client` 结构: 表示一个 I2C 从设备。
- `i2c_msg` 结构: 表示一个在 I2C 上发送的消息。

`i2c.h` 当中的核心发送和接收函数如下所示:

```
extern int i2c_master_send(struct i2c_client *client, const char *buf,
                           int count);
extern int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

`i2c_master_send()` 用于从总线向一个设备发送数据, `i2c_master_recv()` 用于从总线向一个设备接收数据。

I2C 总线传输数据位包括以下内容。

- S 位 (1 位): 开始位。
- P 位 (1 位): 停止位。
- Rd/Wr (1 位): 读写位, 读为 1, 写为 0。
- A, NA (1 位): 接受或者反接受位。
- Addr (7 位): 7 位的地址, 也可以被扩展到 10 位。
- Comm (8 位): 命令字节。
- Data (8 位): 数据字节。
- Count (8 位): 数目长度。

`i2c_master_send()` 的时序为:

S Addr Wr [A] Data [A] Data [A] ... [A] Data [A] P

`i2c_master_recv()` 的时序为:

S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P

以上内容在[]之内的表示来自 I2C 设备，否则表示来自 I2C 主控制器。

核心的传输函数如下所示：

```
extern int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
```

i2c_transfer()函数表示向主控制器传输若干个消息，i2c_msg 结构由从设备地址、标志和数据组成。i2c_master_send()和 i2c_master_recv()两个函数都基于 i2c_transfer()函数实现。表示主控制器的 i2c_adapter 结构如下所示：

```
struct i2c_adapter {
    struct module *owner;
    unsigned int id;
    unsigned int class;                // 表示允许探测 (probe) 的类型
    const struct i2c_algorithm *algo;  // 本总线的算法和数据，也就是数据传输方法
    void *algo_data;
    struct rt_mutex bus_lock;
    int timeout;                       // 超时，使用 jiffies 表示
    int retries;
    struct device dev;                 // 表示该总线在内核中的设备结构
    int nr;
    char name[48];
    struct completion dev_released;
    struct list_head userspace_clients; // 表示用户空间调用者的链表
};
```

在内核驱动中，i2c_adapter 指 I2C 控制器，也就是 I2C 总线本身。一个 I2C 总线当中的核心成员为 i2c_algorithm。

表示从设备的 i2c_client 结构如下所示：

```
struct i2c_client {
    unsigned short flags;              // 表示 I2C 客户端的标志
    unsigned short addr;               // I2C 芯片 7 位的设备地址
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter;       // 连接的主控制器
    struct i2c_driver *driver;         // 使用到的 i2C 驱动
    struct device dev;                 // 表示从设备在内核中的设备结构
    int irq;                           // 表示从设备的中断号
    struct list_head detected;         // 表示从设备的链表
};
```

正如 I2C 总线的硬件连接结构那样，每一个 I2C 设备都要联系到一个 I2C 主控制器上。因此，i2c_client 结构中包含 i2c_adapter 指针类型的成员，指向主控制器。

某些 I2C 总线可以实现单独的传输协议，也就是算法 (algorithm)，对于普通的 I2C 控制器和 SMBus 可以单独实现。

表示消息的 i2c_msg 结构如下所示：

```
struct i2c_msg {
    __u16 addr;    // 从设备的地址
    __u16 flags;   // 表示读写命令，I2C_M_TEN 等宏表示
    __u16 len;     // 消息内存指针 buf 和长度 len
    __u8 *buf;
};
```

i2c_msg 结构表示 I2C 开始传输后的一个传输片断，关键的信息是一个从设备的地址，

这个地址可以是 7 位或者 10 位，后面的指针表示了要传输的数据。

i2c_smbus_data 结构如下所示：

```
#define I2C_SMBUS_BLOCK_MAX
union i2c_smbus_data {
    __u8 byte;
    __u16 word;
    __u8 block[I2C_SMBUS_BLOCK_MAX + 2];    // 表示 SMBus 传输的数据
};
```

i2c_smbus_data 表示专门用于 SMBus 的联合体，其中的成员可以是字节、双字节或者多字节。按照 SMBus 的标准，在多字节表示的情况下，block[0]表示字节的长度。

提示：i2c_msg 和 i2c_smbus_data 结构都用于与用户空间的通信。

表示算法的 i2c_algorithm 结构如下所示：

```
struct i2c_algorithm {
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
        unsigned short flags, char read_write,
        u8 command, int size, union i2c_smbus_data *data);
    u32 (*functionality)(struct i2c_adapter *); // 确定主控制器支持的功能
};
```

当一个 i2c_adapter 具有 i2c_algorithm 结构的时候，可以有选择地实现 master_xfer 或者 smbus_xfer 两个函数指针，分别用于通常主控制器的传输和 SMBus 的传输。一个 I2C 主控制器可能实现二者之一，functionality 函数指针的返回结果表示实现哪种功能。

i2c.h 当中定义的 i2c_smbus_xfer()函数就是对 SMBus 情况的实现，函数如下所示：

```
extern s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
    unsigned short flags, char read_write, u8 command,
    int size, union i2c_smbus_data *data);
```

i2c_smbus_xfer()函数的功能类似于 i2c_transfer()函数，二者的应用场合分别为普通的 I2C 操作和 SMBus 操作。

提示：在实现中，i2c_transfer()函数通过 i2c_algorithm 中的 master_xfer 函数指针实现，i2c_smbus_xfer()函数通过 smbus_xfer 函数指针实现。它们都可用于读或写，需要通过 I2C 主控制器完成，I2C 主控制器与具体的硬件及其实现相关。

另外几个用于 SMBus 读写的函数如下所示：

```
extern s32 i2c_smbus_read_byte(struct i2c_client *client);
extern s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
extern s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
extern s32 i2c_smbus_write_byte_data(struct i2c_client *client,
    u8 command, u8 value);
extern s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
extern s32 i2c_smbus_write_word_data(struct i2c_client *client,
    u8 command, u16 value);
extern s32 i2c_smbus_read_block_data(struct i2c_client *client,
    u8 command, u8 *values);
```

```
extern s32 i2c_smbus_write_block_data(struct i2c_client *client,
                                     u8 command, u8 length, const u8 *values);
extern s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
                                     u8 command, u8 length, u8 *values);
extern s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,
                                     u8 command, u8 length, const u8 *values);
```

i2c_smbus_read_XXX()和 i2c_smbus_write_XXX()等系列函数可以用于 SMBus 的读和写，这都是 SMBus 所支持的数据操作，也都是对 i2c_smbus_xfer 调用并进行封装的结果。

例如：i2c_smbus_read_byte()的时序如下所示：

S Addr Rd [A] [Data] NA P

例如：i2c_smbus_write_byte()的时序如下所示：

S Addr Wr [A] Data [A] P

21.2.2 I2C 总线接口

每一个 I2C 主控制器设备都将在用户空间有一个字符设备的节点。设备节点的路径一般为/dev/i2c-<N>，主设备号是 89，次设备号来自 I2C 总线的编号。

include/linux/i2c-dev.h 文件中定义的 ioctl 命令如下所示：

```
#define I2C_SLAVE      0x0703    // 使用从设备地址
#define I2C_SLAVE_FORCE 0x0706    // 强制使用从设备的地址
#define I2C_TENBIT      0x0704    // 是否使用 10 位地址，如果不是则为 7 位
#define I2C_FUNCS       0x0705    // 获得 I2C 主控制器的功能掩码
#define I2C_RDWR        0x0707    // 读写
#define I2C_PEC         0x0708    // 在 SMBus 中使用 PEC
#define I2C_SMBUS       0x0720    // 进行 SMBus 传输
```

这些 ioctl 命令是用户空间操作 I2C 总线的主要接口，用户空间通过操作 I2C 总线进而对 I2C 总线上连接的设备进行操作。

两个分别表示 SMBus 和基本 I2C 主控制器的操作的数据结构如下所示：

```
struct i2c_smbus_ioctl_data {
    __u8 read_write;           // 读写标志
    __u8 command;             // 命令
    __u32 size;               // SMBus 的数据
    union i2c_smbus_data __user *data;
};
struct i2c_rdwr_ioctl_data {
    struct i2c_msg __user *msgs; // i2c_msg 结构的指针
    __u32 nmsgs;               // i2c_msg 结构的数目
};
```

i2c_smbus_ioctl_data 和 i2c_rdwr_ioctl_data 两个数据结构都用于 ioctl 调用时的参数，在用户空间和内核空间的格式相同。

在 drivers/i2c/i2c-dev.c 文件中，实现了 file_operations 结构，其中支持读、写和 ioctl 操作，并注册为字符设备。当具有一个实现的 i2c_adapter 注册之后，就会出现一个 I2C 设备的设备节点。

在用户空间的编程中，调用 I2C 主控制器，需要包括用户空间的 i2c-dev.h 文件。

在用户空间中，对 I2C 控制器的打开操作如下所示：

```
int file;
int adapter_nr = 2;           // I2C 的设备号
char filename[20];
snprintf(filename, 19, "/dev/i2c-%d", adapter_nr);
file = open(filename, O_RDWR);
if (file < 0) {
    // 错误处理代码
    exit(1);
}
```

在用户空间控制 I2C 主控制器的代码如下所示：

```
int addr = 0x40;              // I2C 从设备的地址
if (ioctl(file, I2C_SLAVE, addr) < 0) {
    // 错误处理代码
    exit(1);
}
```

随后就可以在 I2C 主控制器进行数据传输了，普通 I2C 和 SMBus 操作类似。

21.2.3 I2C 设备和驱动

对于一个 I2C 从设备，可以挂在 I2C 总线上的不同地址处，因此可以将驱动统一实现，可以在不同平台中使用。根据 Linux 驱动程序的通常理念，一个 I2C 从设备需要在板级定义资源，在驱动中使用该信息。

i2c_driver 表示其结构如下所示：

```
struct i2c_driver {
    unsigned int class;
    int (*attach_adapter)(struct i2c_adapter *); // 连接到主控制器
    int (*detach_adapter)(struct i2c_adapter *); // 解除连接到主控制器
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);
    void (*shutdown)(struct i2c_client *);
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *); // 以上为驱动模型的通用接口
    void (*alert)(struct i2c_client *, unsigned int data); // 警报消息
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
    struct device_driver driver;
    const struct i2c_device_id *id_table; // I2C 设备的 id
    int (*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list; // 可以探测的地址
    struct list_head clients;
};
```

i2c_driver 表示 I2C 从设备所使用的驱动，也就是某种类型的 I2C 从设备的软件，I2C 驱动的实现可以复用。一个 i2c_driver 的实现将针对一类 I2C 从设备，当地址信息匹配的时候，某个从设备就使用这个驱动。

一个常见的 i2c_driver 定义方式如下所示：

```
static struct i2c_device_id foo_idtable[] = { // 设备的 id 表
    { "foo", my_id_for_foo },
```

```

    { "bar", my_id_for_bar },
    { }
};
MODULE_DEVICE_TABLE(i2c, foo_idtable);
static struct i2c_driver foo_driver = {
    .driver = { .name = "foo", },
    .id_table   = foo_ids,
    .probe      = foo_probe,
    .remove     = foo_remove,
    .class      = I2C_CLASS_SOMETHING,
    .detect     = foo_detect,
    .address_list = normal_i2c,
    .shutdown   = foo_shutdown,    // 可选
    .suspend    = foo_suspend,     // 可选
    .resume     = foo_resume,      // 可选
    .command    = foo_command,     // 可选
}

```

对于 I2C 设备而言，它们连接于 I2C 总线完全是硬件的引脚连接。I2C 总线控制器不会去遍历检查连接其上的设备。因此，系统连接了哪些 I2C 设备，通常需要在板级支持的代码中表示。

i2c.h 当中的 `di2c_board_info` 结构如下所示：

```

struct i2c_board_info {
    char    type[I2C_NAME_SIZE];    // 设备的名称
    unsigned short  flags;
    unsigned short  addr;           // 设备的地址
    void    *platform_data;
    struct dev_archdata  *archdata;
#ifdef CONFIG_OF
    struct device_node *of_node;
#endif
    int      irq;                  // 设备的中断号
};
#define I2C_BOARD_INFO(dev_type, dev_addr) \
    .type = dev_type, .addr = (dev_addr)

```

`i2c_board_info` 结构表示连接于 I2C 总线的 I2C 设备的板级信息，用于根据设备去找驱动。`i2c_board_info` 中的主要成员是 I2C 从设备的名称、地址和中断号。宏 `I2C_BOARD_INFO` 则用于构建一个 `i2c_board_info` 结构。

几个用于 `i2c_board_info` 注册的函数如下所示：

```

extern int i2c_register_board_info(int busnum,
    struct i2c_board_info const *info, unsigned n);
extern struct i2c_client * i2c_new_device(struct i2c_adapter *adap,
    struct i2c_board_info const *info);
extern struct i2c_client * i2c_new_probed_device(struct i2c_adapter *adap,
    struct i2c_board_info *info, unsigned short const *addr_list);

```

在目录 `/sys/bus/i2c/devices` 下的设备就是这个 `i2c_board_info` 结构体里所描述的 I2C 设备，设备名字就是根据 `i2c_board_info` 结构体中定义的 I2C 地址来命名的。

21.3 具体的 I2C 主控制器

一个具体的 I2C 实现也就是 I2C 主控制器的实现。在嵌入式系统中，I2C 主控制器通常都是嵌入式处理器的一个片内部件，这种片内部件常具有多个 I2C 主控制器。

三星 S3C 系列处理器的 I2C 主控制器硬件结构和使用方式类似，因此驱动程序也可以复用，其驱动程序的源代码为 `drivers/i2c/busses/i2c-s3c2410.c`。

`i2c-s3c2410.c` 当中有平台驱动的定义如下所示：

```
static struct platform_device_id s3c24xx_driver_ids[] = {
    { .name = "s3c2410-i2c", .driver_data = TYPE_S3C2410, },
    { .name = "s3c2440-i2c", .driver_data = TYPE_S3C2440, },
    { },
};
MODULE_DEVICE_TABLE(platform, s3c24xx_driver_ids);
static struct platform_driver s3c24xx_i2c_driver = { // 平台驱动
    .probe      = s3c24xx_i2c_probe,
    .remove     = s3c24xx_i2c_remove,
    .id_table   = s3c24xx_driver_ids,
    .driver     = {
        .owner   = THIS_MODULE,
        .name    = "s3c-i2c",
        .pm      = S3C24XX_DEV_PM_OPS,
    },
};
```

I2C 总线控制器的驱动程序本身是一个平台驱动，此处 I2C 总线控制器平台的名称为 "s3c-i2c"。id_table 的两个成员表示有两种可能，要在平台设备中实现匹配。

用于探测的 `s3c24xx_i2c_probe()` 函数如下所示：

```
static int s3c24xx_i2c_probe(struct platform_device *pdev)
{
    struct s3c24xx_i2c *i2c;
    struct s3c2410_platform_i2c *pdata;
    struct resource *res;
    int ret;
    pdata = pdev->dev.platform_data;
    // ..... 省略错误处理
    i2c = kzalloc(sizeof(struct s3c24xx_i2c), GFP_KERNEL);
    // ..... 省略错误处理
    strlcpy(i2c->adap.name, "s3c2410-i2c", sizeof(i2c->adap.name));
    i2c->adap.owner   = THIS_MODULE;
    i2c->adap.algo    = &s3c24xx_i2c_algorithm; // 此处实现的 I2C 算法
    i2c->adap.retries = 2;
    i2c->adap.class   = I2C_CLASS_HWMON | I2C_CLASS_SPD;
    i2c->tx_setup     = 50;
    spin_lock_init(&i2c->lock);
    init_waitqueue_head(&i2c->wait);
    i2c->dev = &pdev->dev;
    i2c->clk = clk_get(&pdev->dev, "i2c");
    // ..... 省略部分内容
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    // ..... 省略部分内容
    i2c->adap.algo_data = i2c;
    i2c->adap.dev.parent = &pdev->dev;
```

```
// ..... 初始化 I2C 控制器
ret = s3c24xx_i2c_init(i2c);
if (ret != 0) goto err_iomap;
i2c->irq = ret = platform_get_irq(pdev, 0);
// ..... 省略错误处理
ret = request_irq(i2c->irq, s3c24xx_i2c_irq, IRQF_DISABLED,
    dev_name(&pdev->dev), i2c);
// ..... 省略错误处理
ret = s3c24xx_i2c_register_cpufreq(i2c);
// ..... 省略错误处理
i2c->adap.nr = pdata->bus_num;
ret = i2c_add_numbered_adapter(&i2c->adap);
// ..... 省略错误处理
platform_set_drvdata(pdev, i2c);
dev_info(&pdev->dev, "%s: S3C I2C adapter\n", dev_name(&i2c->adap.dev));
return 0;
// ..... 省略错误处理
}
```

struct s3c24xx_i2c 是此处驱动中定义的一个结构。其中包括了 i2c_adapter 类型的成员，其他的成员主要用于传递平台数据，方便匹配同系列的不同处理器。

用于实现 I2C 算法的 i2c_algorithm 结构如下所示：

```
static const struct i2c_algorithm s3c24xx_i2c_algorithm = {
    .master_xfer      = s3c24xx_i2c_xfer,
    .functionality    = s3c24xx_i2c_func,
};
```

此处支持的是普通的 I2C，而非 SMBus，因此 i2c_algorithm 结构中实现的就是 master_xfer，smbus_xfer 函数指针没有实现。

s3c24xx_i2c_func() 函数的实现如下所示：

```
static u32 s3c24xx_i2c_func(struct i2c_adapter *adap)
{
    return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL | I2C_FUNC_PROTOCOL_MANGLING;
}
```

I2C_FUNC_I2C 等宏都是在 i2c.h 中定义的，用于描述一个所实现的 I2C 控制器的功能。

s3c24xx_i2c_xfer() 函数的实现如下所示：

```
static int s3c24xx_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    struct s3c24xx_i2c *i2c = (struct s3c24xx_i2c *)adap->algo_data;
    int retry;
    int ret;
    for (retry = 0; retry < adap->retries; retry++) {
        ret = s3c24xx_i2c_doxfer(i2c, msgs, num); // 真正的传输处理
        if (ret != -EAGAIN) return ret;
        dev_dbg(i2c->dev, "Retrying transmission (%d)\n", retry);
        udelay(100);
    }
    return -EREMOTEIO;
}
```

s3c24xx_i2c_doxfer() 函数用于真正的 I2C 传输，其中进行了真正的硬件寄存器操作，

根据所使用的 I2C 控制器的序号不同，寄存器的偏移量也不同。

在三星处理器板级实现的 arch/arm/plat-samsung/dev-i2c1.c 文件中，具有如下资源的定义：

```
static struct resource s3c_i2c_resource[] = {
    [0] = {
        .start = S3C_PA_IIC1,
        .end   = S3C_PA_IIC1 + SZ_4K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_IIC1,
        .end   = IRQ_IIC1,
        .flags = IORESOURCE_IRQ,
    },
};
struct platform_device s3c_device_i2c1 = { // 平台设备的定义
    .name       = "s3c2410-i2c",
    .id         = 1,
    .num_resources = ARRAY_SIZE(s3c_i2c_resource),
    .resource    = s3c_i2c_resource,
};
```

如果此处的平台设备与 I2C 控制器平台驱动相匹配，则会把 I2C 总线作为平台设备的实现。

在板级文件 arch/arm/mach-s3c64xx/mach-smdk6400.c 中还有如下内容：

```
static struct i2c_board_info i2c_devs[] __initdata = {
    { I2C_BOARD_INFO("wm8753", 0x1A), },
    { I2C_BOARD_INFO("24c08", 0x50), },
};
static void __init smdk6400_machine_init(void)
{
    i2c_register_board_info(0, i2c_devs, ARRAY_SIZE(i2c_devs)); // 设备的注册
    platform_add_devices(smdk6400_devices, ARRAY_SIZE(smdk6400_devices));
}
```

wm8753 是一个音频 Codec 芯片，24c08 是一个 EEPROM，在板级信息中表示了它们挂在 I2C 的第 0 个总线上。

21.4 I2C 从设备的驱动

I2C 从设备的驱动实现的理念是利用 I2C 主控制器，实现对某些特定的 I2C 设备的操作，并向用户空间提供某种形式的接口。一个 I2C 从设备的驱动实现之后，可以用于不同的 I2C 总线。

I2C 从设备驱动程序实现的核心就是 i2c_driver 结构。I2C 从设备驱动程序能使用的条件就是当前平台具有合适的 I2C 总线控制器，且在硬件连接上有一个从设备确实连接到相应的地址。

QT2160 是 Atmel 的 AT42QT2160 触控芯片，其接口就是 I2C 总线。QT2160 的 I2C 驱

动实现后提供给用户空间 input 设备。对于这个 I2C 从设备，在用户空间不会通过 I2C 总线控制器去控制，而是作为标准的 input 设备使用。

QT2160 驱动的源代码为：input/keyboard/qt2160.c。

qt2160.c 文件中的核心定义如下所示：

```
static const struct i2c_device_id qt2160_idtable[] = {
    { "qt2160", 0, }, { }
};
MODULE_DEVICE_TABLE(i2c, qt2160_idtable); // 定义 I2C 从设备
static struct i2c_driver qt2160_driver = { // 声明一个 i2c_driver
    .driver = {
        .name      = "qt2160",
        .owner     = THIS_MODULE,
    },
    .id_table     = qt2160_idtable,
    .probe       = qt2160_probe,
    .remove      = __devexit_p(qt2160_remove),
};
static int __init qt2160_init(void)
{
    return i2c_add_driver(&qt2160_driver); // 在初始化过程中注册 i2c_driver
}
```

i2c_driver 驱动在注册之后，如果与板级定义的数据匹配，其探测函数将会被调用。此处用于探测的函数 qt2160_probe() 内容如下所示：

```
static int __devinit qt2160_probe(struct i2c_client *client,
                                const struct i2c_device_id *id)
{
    struct qt2160_data *qt2160; // qt2160_data 为驱动的私有结构
    struct input_dev *input;     // 准备进行注册的输入设备
    int i;
    int error;
    error = i2c_check_functionality(client->adapter,
                                    I2C_FUNC_SMBUS_BYTE); // 检查功能，支持 SMBus 的字节操作
    // ..... 省略错误处理
    if (!qt2160_identify(client)) return -ENODEV; // 辨认设备
    qt2160 = kzalloc(sizeof(struct qt2160_data), GFP_KERNEL); // 分配设备结构
    input = input_allocate_device(); // 注册一个 input 设备
    // ..... 省略错误处理
    qt2160->client = client;
    qt2160->input = input;
    INIT_DELAYED_WORK(&qt2160->dwork, qt2160_worker); // 初始化队列
    spin_lock_init(&qt2160->lock);
    input->name = "AT42QT2160 Touch Sense Keyboard"; // 输入设备的名称
    input->id.bustype = BUS_I2C;
    // ..... 省略：输入设备的 keycode 处理、校准
    if (client->irq) {
        error = request_irq(client->irq, qt2160_irq, // 进行中断的注册
                            IRQF_TRIGGER_FALLING, "qt2160", qt2160);
    }
    // ..... 省略错误处理
    error = input_register_device(qt2160->input);
    // ..... 省略错误处理
    i2c_set_clientdata(client, qt2160); // 本驱动特定数据的设置
    qt2160_schedule_read(qt2160); // 启动队列的调度
}
```

```
    return 0;
    // ..... 省略错误处理
}
```

QT2160 要使用 SMBus 进行操作，因此在探测的时候需要检测 I2C 主控制器是否有 SMBus 功能，`client->adapter` 表示的就是本 I2C 客户端连接到的 `i2c_adapter`(I2C 主控制器)。

QT2160 当中最为底层的实现在 `qt2160_write()`和 `qt2160_read()`，两个函数如下所示：

```
static int __devinit qt2160_read(struct i2c_client *client, u8 reg)
{
    int ret;
    ret = i2c_smbus_write_byte(client, reg);    // 调用 SMBus 设置地址
    // ..... 省略错误处理
    ret = i2c_smbus_read_byte(client);
    // ..... 省略错误处理
    return ret;
}
static int __devinit qt2160_write(struct i2c_client *client, u8 reg, u8 data)
{
    int error;
    error = i2c_smbus_write_byte(client, reg); // 调用 SMBus 接口设置寄存器地址
    // ..... 省略错误处理
    error = i2c_smbus_write_byte(client, data); // 调用 SMBus 接口数据
    // ..... 省略错误处理
    return error;
}
```

在读写之前，操作的功能就是先通过 SMBus 的写，来设置总线上的寄存器地址，然后再进行读或者写。

根据 `qt2160_probe()`中注册的 `input` 设备，在执行的过程中需要通过 `qt2160_worker()`线程运行，它又调用了 `qt2160_get_key_matrix()`，其中通过 I2C 总线读取了数据，然后调用输入设备的 `input_report_key()`函数上报 `input` 驱动的事件。

第 22 章

PCI 总线和驱动

22.1 PCI 概述

PCI 的全称是 Peripheral Component Interconnect（外围部件互联标准），是一种连接电子计算机主板和外部设备的总线标准。

Intel 公司于 1990 年前后发展了 PCI 标准，并联合 IBM、Compaq、AST、HP、DEC 等 100 多家公司成立了 PCI 集团。1992 年，第一个 PCI 标准发布。1993 年，PCI-SIG（Peripheral Component Interconnect Special Interest Group，外围部件互联专业组）发表了 PCI 2.0 标准，这个标准第一次建立了连接器与主板插槽间的标准。PCI 总线常见于现代的个人计算机中，并已取代了 ISA 和 VESA 局部总线，成为标准扩展总线。PCI 总线亦常见于其他电子计算机类型中。PCI 的下一代标准是总线 PCI Express。

22.1.1 PCI 的基本结构

PCI 总线是一个处理器系统的局部总线，可以用来连接外部设备。PCI 设备具有独立的地址空间，即 PCI 总线地址空间，该空间与存储器地址空间隔离。PCI 总线的各个设备工作在各自的时钟频率中，彼此互不干扰。处理器需要通过桥访问 PCI 设备。

PCI 设备都有独立的配置空间，其中包括了这个 PCI 设备在 PCI 总线中使用的基地址，其可以由软件动态配置，也就是说 PCI 设备使用的地址可以根据需要由软件动态分配。PCI 设备可以通过 4 根中断请求信号向处理器提交中断请求，PCI 总线上的设备可以共享这些中断请求信号。在数据交换方面，处理器可以访问 PCI 设备的地址空间，PCI 设备可以通过 DMA 机制访问主内存。

提示：PCI 总线主要用于 x86 处理器的计算机系统，一些 ARM、PowerPC、MIPS 嵌入式处理器也具有 PCI 总线。很多网卡、声卡等设备使用 PCI 接口。

PCI 总线的基本规格如下：

- 支持 10 台外设。
- 总线频率为 33.33 MHz 或者 66.7MHz。
- 在 32 位的带宽中，最大传输 133 MB/s。
- 32 或 64 位的内存位址。
- 32 位的 I/O 端口空间。
- 32 位的使用 5V 电压，64 位的使用 3.3V 电压。
- 256 字节的配置空间。
- 有 4 条中断线。

PCI 总线在个人计算机中使用得较多，一般 PCI 的控制部分都集成在主板的芯片中（传统的方式是在南桥中），PCI 设备以插卡的方式进入。

个人计算机上的 32 位和 64 位 PCI 插槽如图 22-1 所示。

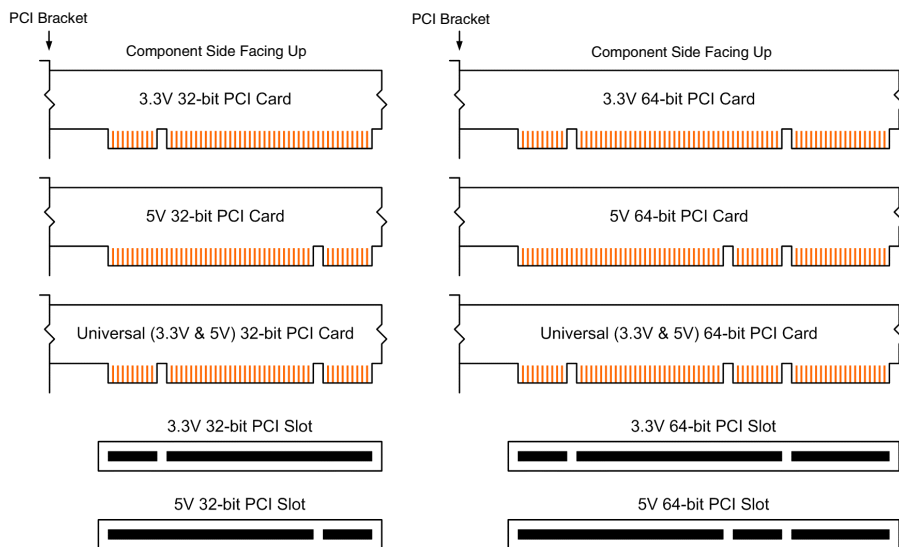


图 22-1 32 位和 64 位 PCI 插槽

个人计算机典型结构中的 PCI 总线如图 22-2 所示。

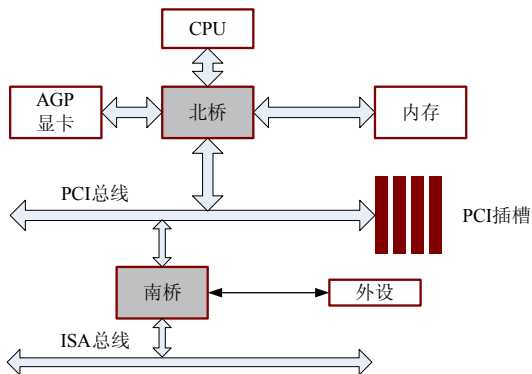


图 22-2 个人计算机典型结构中的 PCI 总线

PCI 总线具有下面的特点：

- 具有地址数据多路复用的高性能 32 位或 64 位同步总线，总线引脚数目和部件数量少，降低了成本及布线的复杂度。
- PCI 总线支持线性突发传输模式，确保了总线不断满载数据进行高速传输。
- PCI 总线的设计是独立于处理器的，它具有严格的总线规范和良好的兼容性。PCI 扩展卡可以插入任何符合 PCI 规范的微机和工作站系统中，可以方便地进行硬件移植。目前 PCI 已成为嵌入式系统的局部总线之一。
- 隐蔽的总线仲裁，减小了仲裁开销。
- 极小的存取延迟，采用总线主控和异步数据转移操作。
- PCI 提供数据和地址奇偶校验功能，保证了数据的完整性和准确性。
- PCI 总线与 CPU 的时钟频率无关，它能支持多个外设，设备间通过局部总线可以完成数据的快速传递，有效解决了数据传输的瓶颈问题。
- PCI 对扩展卡和元件能够进行自动配置，实现设备的即插即用。由于使用方便、灵活，产品寿命长，所以 PCI 总线产品与其他总线标准相比具有巨大的优越性和更为广阔的应用前景。
- 对于个人计算机的 x86 架构，由主 CPU 发起读操作访问 PCI 目标设备时，不能进行突发读操作，这是由于个人计算机启动时 BIOS 将 PCI 设备映射到非 Cache 存储器中，会出现读操作阻塞。对于突发写操作，也存在同样的问题。因此，个人计算机环境下开发基于 PCI 的产品，不能支持突发传输。为了获得高的数据传输率，就必须使用 PCI 主桥设计 PCI 卡，并在 DMA 模式下操作。

在 PCI 应用系统中，如果一个设备取得了总线控制权，则它就被称为“主设备”，而被主设备选中进行通信的设备被称为“从设备”或“目标设备”。

22.1.2 PCI 的总线信号

PCI 主设备的必备信号为 49 条，从设备的必备信号为 47 条。可选信号为 51 条，主要用于 64 位 PCI 的扩展。

PCI 的总线信号主要有以下部分。

- CLK (clock) (IN)：PCI 系统时钟信号，输入。时钟信号的频率范围是 0~33MHz，或 33.33~66.66MHz。
- RST (reset) (IN)：复位信号，输入，低电平有效。
- AD31~AD0 (双向三态输入/输出)：AD31~AD0 是一组双向、三态信号，为地址和数据分时复用。
- C/BE[3:0] (双向三态输入/输出)：命令或字节使能信号，双向、三态，由主设备驱动，表示各种 PCI 总线的命令（共 16 种）。
- PAR (双向三态输入输出)：奇偶校验信号，分别被主设备和从设备驱动。
- FRAME (持续低电平有效三态)：周期帧信号。
- TRDY (持续低电平有效三态)：目标设备准备好信号。

- IRDY（持续低电平有效三态）：启动方准备好信号。
- STOP（持续低电平有效三态）：目标设备放弃或重试数据传送信号。
- IDSEL（持续低电平有效三态）：初始化设备选择信号。
- LOCK（持续低电平有效三态）：锁定寻址目标信号。
- DEVSEL（持续低电平有效三态）：设备选择信号。
- REQ（双向三态输入/输出）：总线请求信号。
- GNT（双向三态输入/输出）：总线允许信号。
- INTA、INTB、INTC 和 INTD（漏极开路）：4 个中断请求信号，低电平触发有效。
- PERR（持续低电平有效三向信号）：校验错误信号。
- SERR（持续低电平有效三向信号）：系统错误信号。

PCI 总线还具有 64 位扩展信号和 JTAG 信号、电源管理信号、时钟控制信号。

16 种 PCI 总线的命令分别是：0000（Interrupt Acknowledge）、0001（Special Cycle）、0010（I/O Read）、0011（I/O Write）、0100（Reserved）、0101（Reserved）、0110（Memory Read）、0111（Memory Read）、1000（Reserved）、1001（Reserved）、1010（Configuration Read）、1011（Configuration Write）、1100（Memory Read Multiple）、1101（Dual Address Cycle）、1110（Memory Read Line）、1111（Memory Write and Invalidate）。

PCI 总线规定单功能设备只能使用 INTA 中断信号，而多功能设备才能使用 INTB、INTC、INTD 中断信号。

22.1.3 PCI 的总线操作

突发传送是 PCI 总线操作的一个主要特点，几乎所有 PCI 支持的数据交换都可以由突发传送来实现。突发传送是一种包含一个地址段，后面跟着两个或两个以上数据项的数据传送方式。

对于总线拥有者来说，总线主设备只需进行一次仲裁。在突发周期内，主设备在地址段发出起始地址和操作类型，总线上所有的设备都锁定地址和操作类型，并将其译码，以确定谁是从设备。从设备将起始地址锁存在地址计数器中，并且随着一个个数据项的传送递增地址。

在主设备和从设备都没有插入等待状态的情况下，数据项（双字或四字）在每个 PCI 时钟周期的上升沿传送。66MHz 的 PCI 总线采用 32 或 64 位数据传送时，可以达到 264MB/s 或 528MB/s 的传送速率。

每个 PCI 突发传送都有两个参与者：启动方和从设备。启动方或总线主设备是启动传送的设备。在 PCI 规范中，总线主设备和启动方两个术语意义相同。从设备是指启动方为实现数据传送目的而正在寻址的设备。通常，PCI 启动方和从设备被称为与 PCI 兼容的代理。

PCI 总线上的所有活动都是和 PCI 时钟 CLK 同步的。

- 地址段：PCI 操作都是以第一个 PCI 时钟周期内的地址段开始的。在地址段，主设备通过驱动地址总线来鉴别从设备，通过驱动 PCI 命令/字节使能总线来确定操作类

型（也称为命令类型）。在地址段，主设备使 FRAME 信号有效，表示总线的起始地址和命令类型有效。每个 PCI 从设备必须在时钟的下一个上升沿锁存地址和命令，以便顺序译码。

- 声明一次操作：当 PCI 从设备确定自己是操作的从设备以后，通过有效的 DEVSEL（设备选择）声明本次操作有效。
- 数据项传输：主设备和从设备之间传输数据的阶段。通常从一次操作的第二个 PCI 时钟周期开始，数据项传输的数据字节数由命令/字节使能信号决定，这些信号在数据项中是由主设备驱动的。
- 操作过程：PCI 操作中所有的数据传送都可以是突发传送。
- 传送结束和总线空闲：主设备通过无效 FRAME 和有效 IDDY 表明，突发传送的最后一个数据在传送过程中，当最后的数据传送完成时，主设备通过使 IDDY 无效，将 PCI 总线返回空闲状态，TRDY 和 C/BE 自然也是无效状态。如果另一个主设备已经被 PCI 总线仲裁器授权拥有总线，并正在等待当前主设备放弃总线，此时，它在 PCI 时钟的同一个上升沿检测到无效的 FRAME 和 IDDY，则可知总线已经返回空闲状态。

22.1.4 PCI 的总线配置

PCI 总线支持设备的自动监测和配置。PCI 设备具有即插即用功能，每个 PCI 功能都必须根据规范，定义基本配置的寄存器。

PCI 配置共有 256 个字节，前面 64 个字节为 PCI 描述符。

PCI 描述符的分配如图 22-3 所示。

Device ID（设备ID）		Vendor ID（厂商ID）		00h
Status（状态）		Command（命令）		04h
Class Code（分类码）			Revision ID	08h
BIST（自测）	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Register（基地址寄存器）				10h
Base Address Register（基地址寄存器）				14h
Base Address Register（基地址寄存器）				18h
Base Address Register（基地址寄存器）				1Ch
Base Address Register（基地址寄存器）				20h
Base Address Register（基地址寄存器）				24h
Cardbus CIS Pointer（卡总线的CIF指针）				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address（扩展ROM基地址）				30h
Reverved（保留）			Cap. Pointer	34h
Reverved（保留）				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

图 22-3 PCI 描述符的分配

PCI 总线配置空间的 256 字节，具有特定记录结构，是一个地址空间。该空间分为头标区和设备相关区两部分。在每个区中，设备只设置必需设置的和与之相关的寄存器。一个设备的配置空间不仅在系统自举时可以访问，在其他时间也可以访问。

头标区的长度为 64 字节，偏移地址从 00H 到 3FH，该区中的各个域用来识别各自不同的设备，并使设备能以一般的方法控制。每个设备都必须按照该区的格式和规定进行设置。而配置空间的其余 192 字节可以因设备而异。

各个寄存器的含义如下所示。

- Device ID 和 Vendor ID 寄存器

这两个寄存器的值由 PCISIG 分配，只读。其中，Vendor ID 代表 PCI 设备的生产厂商，而 Device ID 代表这个厂商所生产的具体设备。如 Intel 公司的基于 82571EB 芯片的系列网卡，其 Vendor ID 为 0x8086，而 Device ID 为 0x105E。

- Revision ID 和 Class Code 寄存器

只读。其中，Revision ID 寄存器记载 PCI 设备的版本号。该寄存器可以被认为是 Device ID 寄存器的扩展。

- Header Type 寄存器

只读，由 8 位组成。第 7 位为 1 表示当前 PCI 设备是多功能设备，为 0 表示为单功能设备。第 6~10 位表示当前配置空间的类型。系统软件需要使用该寄存器区分不同类型的 PCI 配置空间。该寄存器的初始化必须与 PCI 设备的实际情况对应，而且必须为一个合法值。

- Cache Line Size 寄存器

该寄存器记录 HOST 处理器使用的 Cache 行长度。在 PCI 总线中，和 Cache 相关的总线事务需要使用这个寄存器，如：存储器写并无效，Cache 多行读等总线事务。如果 PCI 设备不支持与 Cache 相关的总线事务，系统软件可以不设置该寄存器，此时该寄存器为初始值 0x00。

- Subsystem ID 和 Subsystem Vendor ID 寄存器

与 Device ID 和 Vendor ID 类似，也是记录 PCI 设备的生产厂商和设备名称。

- Expansion ROM Base address 寄存器

本寄存器记录 ROM 程序的基地址。有些 PCI 设备在处理器还没有运行操作系统之前，就需要完成基本的初始化设置，这些 PCI 设备需要提供一段 ROM 程序，而处理器在初始化过程中将运行这段 ROM 程序，初始化这些 PCI 设备。

- Capabilities Pointer 寄存器

PCI 设备相关的扩展配置信息。

- Interrupt Line 寄存器

记录当前 PCI 设备使用的中断向量号。在系统软件对 PCI 设备进行配置时写入，用于识别 PCI 中断的请求引脚连接到中断控制器的哪个输入端。在 PC 环境中，这个中断就是 00H 到 0FH 的数值，表示就是 IRQ0 到 IRQ15。

- Interrupt Pin 寄存器

保存 PCI 设备使用的 4 个中断引脚中的哪一个，值为 1~4。

- Base Address Register 0~5 寄存器

简称 BAR 寄存器。BAR 寄存器保存 PCI 设备使用的地址空间的基地址，该基地址保存的是该设备在 PCI 总线域中的地址。

- Command 寄存器

表示 PCI 设备的命令。初始化时值为 0，此时 PCI 设备只能够接收配置请求总线事务，不能接收任何存储器或者 I/O 请求；合理设置该寄存器之后，才能访问该设备的存储器或者 I/O 空间。

- Status 寄存器

保存 PCI 设备的状态。

- Latency Timer 寄存器

该寄存器用来控制 PCI 设备占用 PCI 总线的时间，因为在 PCI 总线中，多个设备共享总线带宽。

22.1.5 PCI 的发展和衍生标准

PCI-X，在 1998 年由 IBM、HP 以及 Compaq 等公司制定，采用 64 位总线宽度及并发机制，可以达到 133MHz 带宽的传输速度，其有更多的引脚，而且所有的连接设备会共享所有可用的带宽，与 PCI 的协议类似。PCI-X 目前有 1.0 版本和 2.0 版本。

PCI Express，简称 PCI-E，沿用了现有的 PCI 编程概念及通信标准，但基于更快的串行通信系统，其连接创建在一个双向的串行点对点连接基础之上。Intel 公司是该接口的主要支持者，PCI Express 曾经被称为 3GIO。PCI Express 仅应用于内部互连。由于 PCI Express 基于现有的 PCI 系统，因此只需修改物理层而无须修改软件就可将现有 PCI 系统转换为 PCI Express。PCI Express 拥有更快的速率，以取代几乎全部现有的内部总线。PCI Express 设备能够支持热拔插以及热交换特性，支持的 3 种电压分别为 +3.3V、3.3Vaux（辅助电压）以及 +12V。PCI Express 有多个版本，最高可以达到 2 GB/s 的速度。

22.2 PCI 总线的驱动框架

Linux 系统 PCI 总线驱动框架是一个典型的总线结构，由 PCI 核心、PCI 设备和 PCI 驱动几个部分组成。PCI 驱动通过 PCI 总线访问和控制连接于 PCI 总线的 PCI 设备，并以某种驱动接口的形式提供给内核空间和用户空间调用。

PCI 总线的 sys 文件系统包括了下面两个目录。

- /sys/bus/pci/devices/：各个 PCI 的设备，表示为 XXXX:XX:XX.X 的形式。
- /sys/bus/pci/drivers/：各个 PCI 的驱动。

其中，/sys/bus/pci/deivices/中的各个字母是到/sys/devices/XXXX:XX 的连接。

PCI 系统的头文件在 include/linux/目录中，源代码在 driver/pci/目录中。

PCI 系统主要的头文件是 pci.h，其他头文件包括 pci_ids.h、pci_regs.h、pci-acpi.h、

pci-aspm.h、pci-dma.h、pcieport_if.h、pci_hotplug.h。

pci_regs.h 中包括了 PCI 总线各个寄存器偏移量的定义，如下所示：

```
#define PCI_VENDOR_ID      0x00 /* 16 bits */
#define PCI_DEVICE_ID      0x02 /* 16 bits */
```

pci_regs.h 当中定义了各种 PCI 命令，如下所示：

```
#define PCI_COMMAND        0x04 /* 16 bits */
#define PCI_COMMAND_IO      0x1  /* Enable response in I/O space */
#define PCI_COMMAND_MEMORY  0x2  /* Enable response in Memory space */
#define PCI_COMMAND_MASTER  0x4  /* Enable bus mastering */
#define PCI_COMMAND_SPECIAL 0x8  /* Enable response to special cycles */
#define PCI_COMMAND_INVALIDATE 0x10 /* Use memory write and invalidate */
#define PCI_COMMAND_VGA_PALETTE 0x20 /* Enable palette snooping */
#define PCI_COMMAND_PARITY  0x40 /* Enable parity checking */
#define PCI_COMMAND_WAIT    0x80 /* Enable address/data stepping */
#define PCI_COMMAND_SERR    0x100 /* Enable SERR */
#define PCI_COMMAND_FAST_BACK 0x200 /* Enable back-to-back writes */
#define PCI_COMMAND_INTX_DISABLE 0x400 /* INTx Emulation Disable */
```

PCI 命令是用来控制 PCI 总线的。通常可以通过写 PCI 总线控制器寄存器的方式向 PCI 总线输出一个命令，含义就是让 PCI 总线执行某个工作。

pci_regs.h 当中定义了各种 PCI 状态，如下所示：

```
#define PCI_STATUS        0x06 /* 16 bits */
#define PCI_STATUS_INTERRUPT 0x08 /* Interrupt status */
#define PCI_STATUS_CAP_LIST 0x10 /* Support Capability List */
#define PCI_STATUS_66MHZ    0x20 /* Support 66 Mhz PCI 2.1 bus */
// ..... 省略部分内容
#define PCI_CLASS_NOT_DEFINED 0x0000
#define PCI_CLASS_NOT_DEFINED_VGA 0x0001
```

PCI 通常表示 PCI 总线返回的信息。通常可以由 PCI 总线控制器的寄存器获取到。

pci_ids.h 当中定义的各种 PCI 设备类型如下所示：

```
#define PCI_BASE_CLASS_STORAGE 0x01          // PCI 存储器类型
#define PCI_CLASS_STORAGE_SCSI 0x0100
#define PCI_CLASS_STORAGE_IDE 0x0101
#define PCI_CLASS_STORAGE_FLOPPY 0x0102
#define PCI_CLASS_STORAGE_IPI 0x0103
#define PCI_CLASS_STORAGE_RAID 0x0104
#define PCI_CLASS_STORAGE_SATA 0x0106
#define PCI_CLASS_STORAGE_SATA_AHCI 0x010601
#define PCI_CLASS_STORAGE_SAS 0x0107
#define PCI_CLASS_STORAGE_OTHER 0x0180

#define PCI_BASE_CLASS_NETWORK 0x02          // PCI 网络类型
#define PCI_CLASS_NETWORK_ETHERNET 0x0200
#define PCI_CLASS_NETWORK_TOKEN_RING 0x0201
#define PCI_CLASS_NETWORK_FDDI 0x0202
#define PCI_CLASS_NETWORK_ATM 0x0203
#define PCI_CLASS_NETWORK_OTHER 0x0280
```

其中，基本类型（BASE_CLASS）包括了存储器（STORAGE）、网络（NETWORK）、显示器（DISPLAY），每种基本类型下面又有子类型。

PCI 系统的主要头文件是 `pci.h`，当中的主要结构如下所示。

- `pci_bus`: PCI 总线本身。
- `pci_dev`: PCI 总线上的一个设备。
- `pci_ops`: 底层体系结构相关的 PCI 操作。
- `pci_device_id`: 表示每个 PCI 设备都有一组参数。
- `pci_driver`: 一个 PCI 设备的驱动程序。

`pci_device_id` 结构如下所示:

```
#define PCI_ANY_ID (~0)           // PCI_ANY_ID 的实际数值为 0xFFFFFFFF
struct pci_device_id {
    __u32 vendor, device;         // vendor 和 device 的 ID (可以为 PCI_ANY_ID)
    __u32 subvendor, subdevice;   // 子系统及其设备的 ID (可以为 PCI_ANY_ID)
    __u32 class, class_mask;      // 类和类的掩码
    kernel_ulong_t driver_data;    // 驱动程序的私有数据
};
```

`pci_device_id` 结构中的 `id` 成员可以设置为 `PCI_ANY_ID`，表示可使用任意的 `id`。

`pci_driver` 是一个典型的总线驱动的结构:

```
struct pci_driver {
    struct list_head node;
    char *name;
    const struct pci_device_id *id_table;    // 表示设备的 ID
    int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
    void (*remove) (struct pci_dev *dev);
    int (*suspend) (struct pci_dev *dev, pm_message_t state);
    int (*suspend_late) (struct pci_dev *dev, pm_message_t state);
    int (*resume_early) (struct pci_dev *dev);
    int (*resume) (struct pci_dev *dev);
    void (*shutdown) (struct pci_dev *dev);
    struct pci_error_handlers *err_handler;
    struct device_driver driver;
    struct pci_dynids dynids;
};
```

`pci_driver` 中包括了驱动中常有的 `probe`、`remove` 等函数指针。`id_table` 成员变量用于记录当前这个驱动能够驱动哪些设备，就是 `pci_device_id` 结构的数组。在系统启动的时候，PCI 总线会扫描连接到这个总线上的设备，同时为每一个设备建立一个 `pci_dev` 结构。

`pci.h` 当中定义的函数有两类：一类是 PCI 核心部分内部使用的，另一类是给各个 PCI 设备使用的。

`pci.h` 当中的几个内部使用的函数如下所示:

```
void pcibios_scan_specific_bus(int busn);
extern struct pci_bus *pci_find_bus(int domain, int busnr);
void pci_bus_add_devices(const struct pci_bus *bus);
struct pci_bus *pci_create_bus(struct device *parent, int bus,
                               struct pci_ops *ops, void *sysdata);
struct pci_bus *pci_add_new_bus(struct pci_bus *parent, struct pci_dev *dev,
                                int busnr);
extern void pci_remove_bus(struct pci_bus *b);
extern void pci_remove_bus_device(struct pci_dev *dev);
```

以上几个函数由 PCI 的核心部分使用，与其他总线驱动类似，用于建立 PCI 总线。各个 PCI 设备使用的函数如下所示：

```
struct pci_bus *pci_find_next_bus(const struct pci_bus *from);
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device,
                                struct pci_dev *from);
int pci_bus_read_config_byte(struct pci_bus *bus, unsigned int devfn,
                             int where, u8 *val);
int pci_bus_read_config_word(struct pci_bus *bus, unsigned int devfn,
                              int where, u16 *val);
int pci_bus_read_config_dword(struct pci_bus *bus, unsigned int devfn,
                               int where, u32 *val);
int pci_bus_write_config_byte(struct pci_bus *bus, unsigned int devfn,
                              int where, u8 val);
int pci_bus_write_config_word(struct pci_bus *bus, unsigned int devfn,
                              int where, u16 val);
int pci_bus_write_config_dword(struct pci_bus *bus, unsigned int devfn,
                                int where, u32 val);
struct pci_ops *pci_bus_set_ops(struct pci_bus *bus, struct pci_ops *ops);
```

对于一个 PCI 驱动（pci_driver），实现的核心就是通过这些 PCI 函数来访问和控制总线，进而访问和控制 PCI 设备。

注册 PCI 驱动的函数如下所示：

```
int __must_check __pci_register_driver(struct pci_driver *, struct module *,
                                       const char *mod_name);
#define pci_register_driver(driver) \
    __pci_register_driver(driver, THIS_MODULE, KBUILD_MODNAME)
void pci_unregister_driver(struct pci_driver *dev);
```

PCI 核心部分的代码在 driver/pci 目录中，主要由 pci.c、pci-driver.c、irq.c、search.c、pci-sysfs.c 等文件组成。

22.3 PCI 设备的驱动

由于 PCI 设备的自动配置特性，各个 PCI 总线设备的驱动程序独立性较强，与板级的关系不大。

22.3.1 PCI 的桩实现

driver/pci/目录中的 pci-stub.c 提供了 PCI 的桩实现。这是一个非常简单的例子，提供了向内核注册一个 PCI 驱动的方法。本驱动可以通过 insmod 动态插入。

pci_driver 结构的定义如下所示。

```
static struct pci_driver stub_driver = {
    .name      = "pci-stub",
    .id_table   = NULL,           // 表示使用动态的 id
    .probe     = pci_stub_probe,
};
```

负责探测的 pci_stub_probe()函数是一个空实现。由于 PCI 设备通常只在关机的时候增

加或者删除, 开机后 PCI 设备是确定的, 因此 PCI 的探测功能仅在模块插入的时候有效果。

信息定义和负责初始化的 `pci_stub_init()` 函数如下所示:

```
module_param_string(ids, ids, sizeof(ids), 0);
MODULE_PARM_DESC(ids, "Initial PCI IDs to add to the stub driver, format is "
    "\"vendor:device[:subvendor[:subdevice[:class[:class_mask]]]]\""
    " and multiple comma separated entries can be specified");
static int __init pci_stub_init(void)
{
    char *p, *id;
    int rc;
    rc = pci_register_driver(&stub_driver);    // 注册 pci_driver 结构
    if (rc) return rc;
    p = ids;
    // 将模块的参数 ids 作为基本的信息
    while ((id = strsep(&p, ",")) {
        unsigned int vendor, device, subvendor = PCI_ANY_ID,
            subdevice = PCI_ANY_ID, class=0, class_mask=0;
        int fields;
        fields = sscanf(id, "%x:%x:%x:%x:%x:%x",    // 从 id 中得到 vendor 等信息
            &vendor, &device, &subvendor, &subdevice, &class, &class_mask);
    // ..... 省略错误处理
        rc = pci_add_dynid(&stub_driver, vendor, device,    // 增加动态的 id
            subvendor, subdevice, class, class_mask, 0);
    // ..... 省略错误处理
    }
    return 0;
}
```

`pci_add_dynid()` 需要 `vendor` 等信息, 对于 PCI 的桩实现, 可以在 `insmod` 的时候输入这些信息。模块的初始化将根据这些信息注册动态的 `id`。

22.3.2 网卡的 PCI 实现

Intel 8255x 10/100 Mb/s 是一个系列的 PCI 网卡, 包括 82558、82562 等设备。其驱动程序被称为 Intel(R) PRO/100+。本驱动代码的路径为 `drivers/net/e100.c`。驱动的核心功能是基于 PCI 总线的接口进行操作的, 实现之后是一个网络设备 (`net_device`)。

基本信息的定义如下所示:

```
#define INTEL_8255X_ETHERNET_DEVICE(device_id, ich) {\
    PCI_VENDOR_ID_INTEL, device_id, PCI_ANY_ID, PCI_ANY_ID, \
    PCI_CLASS_NETWORK_ETHERNET << 8, 0xFFFFF00, ich }
static DEFINE_PCI_DEVICE_TABLE(e100_id_table) = {    // PCI 设备 (pci_device_id) 的表
    INTEL_8255X_ETHERNET_DEVICE(0x1029, 0),
    INTEL_8255X_ETHERNET_DEVICE(0x1030, 0),
    INTEL_8255X_ETHERNET_DEVICE(0x1031, 3),
    INTEL_8255X_ETHERNET_DEVICE(0x1032, 3),
    INTEL_8255X_ETHERNET_DEVICE(0x1033, 3),
    // ..... 省略部分内容
    { 0, }
};
MODULE_DEVICE_TABLE(pci, e100_id_table);
```

`DEFINE_PCI_DEVICE_TABLE` 宏用于定义一个 `pci_device_id` 的数组, 此处不同的数值就是 `device` 的 `id`、`driver_data`。Vendor 等值都是相同的。

pci_driver 结构 e100_driver 的定义如下所示:

```
static struct pci_driver e100_driver = {
    .name = DRV_NAME,          // 值为"e100"
    .id_table = e100_id_table, // 引用 pci_device_id 的数组
    .probe = e100_probe,
    .remove = __devexit_p(e100_remove),
#ifdef CONFIG_PM
    .suspend = e100_suspend,
    .resume = e100_resume,
#endif
    .shutdown = e100_shutdown,
    .err_handler = &e100_err_handler,
};
```

e100_driver 结构当中的主要成员是表示 id_table 的 e100_id_table, 以及探测函数 e100_probe()。

设备的探测函数 e100_probe() 如下所示:

```
static int __devinit e100_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    struct net_device *netdev;
    struct nic *nic;
    int err;
    // ..... 省略错误处理
    netdev->netdev_ops = &e100_netdev_ops; // 网络设备的初始化
    SET_ETHTOOL_OPS(netdev, &e100_ethtool_ops);
    netdev->watchdog_timeo = E100_WATCHDOG_PERIOD;
    strncpy(netdev->name, pci_name(pdev), sizeof(netdev->name) - 1);
    nic = netdev_priv(netdev); // 驱动的私有结构
    netif_napi_add(netdev, &nic->napi, e100_poll, E100_NAPI_WEIGHT);
    nic->netdev = netdev;
    nic->pdev = pdev;
    nic->msg_enable = (1 << debug) - 1;
    nic->mdio_ctrl = mdio_ctrl_hw;
    pci_set_drvdata(pdev, netdev);
    if ((err = pci_enable_device(pdev))) { // 使能 PCI 设备
    // ..... 省略错误处理
    }
    if (!(pci_resource_flags(pdev, 0) & IORESOURCE_MEM)) { // PCI 资源操作
    // ..... 省略错误处理
    }
    if ((err = pci_request_regions(pdev, DRV_NAME))) { // PCI 申请
    // ..... 省略错误处理
    }
    if ((err = pci_set_dma_mask(pdev, DMA_BIT_MASK(32)))) {
    // ..... 省略错误处理
    }
    SET_NETDEV_DEV(netdev, &pdev->dev);
    if (use_io)
        netif_info(nic, probe, nic->netdev, "using i/o access mode\n");
    nic->csr = pci_iomap(pdev, (use_io ? 1 : 0), sizeof(struct csr));
    if (!nic->csr) {
    // ..... 省略错误处理
    }
    if (ent->driver_data)
        nic->flags |= ich;
```

```

else
    nic->flags &= ~ich;
    e100_get_defaults(nic);                // 获取 E100 的 nic
// ..... 省略: 初始化几个锁
    e100_hw_reset(nic);                    // E100 的复位操作
    pci_set_master(pdev);                  // 设置 PCI 总线
// ..... 省略: 初始化定时器和队列
    if ((err = e100_alloc(nic))) {
// ..... 省略错误处理
    }
    if ((err = e100_eeeprom_load(nic)))      // 读取网卡 EEPROM
        goto err_out_free;
    e100_phy_init(nic);                    // E100 的物理初始化
    memcpy(netdev->dev_addr, nic->eeeprom, ETH_ALEN);
    memcpy(netdev->perm_addr, nic->eeeprom, ETH_ALEN);
// ..... 省略部分内容
    pci_pme_active(pdev, false);
    strcpy(netdev->name, "eth%d");          // 网络设备的名称
    if ((err = register_netdev(netdev))) { // 注册网络设备
// ..... 省略错误处理
    }
    nic->cbs_pool = pci_pool_create(netdev->name, nic->pdev,
                                   nic->params.cbs.max * sizeof(struct cb), sizeof(u32), 0);
    return 0;
// ..... 省略错误处理
}

```

e100_probe()函数中实现了一个网络设备（net_device）及其操作（net_device_ops）。本驱动的核心就是通过 PCI 设备的操作实现了一个网络设备，以网络设备作为对上层的接口。

e100_hw_init()函数负责 e100 硬件的初始化，本函数结构如下所示：

```

static int e100_hw_init(struct nic *nic)
{
    int err;
    e100_hw_reset(nic);
    netif_err(nic, hw, nic->netdev, "e100_hw_init\n");
    if (!in_interrupt() && (err = e100_self_test(nic)))    return err;
    if ((err = e100_phy_init(nic)))                        return err;
    if ((err = e100_exec_cmd(nic, cuc_load_base, 0)))      return err;
    if ((err = e100_exec_cmd(nic, ruc_load_base, 0)))      return err;
    if ((err = e100_load_ucose_wait(nic)))                return err;
    if ((err = e100_exec_cb(nic, NULL, e100_configure)))   return err;
    if ((err = e100_exec_cb(nic, NULL, e100_setup_iaaddr))) return err;
    if ((err = e100_exec_cmd(nic, cuc_dump_addr,
                             nic->dma_addr + offsetof(struct mem, stats))))
        return err;
    if ((err = e100_exec_cmd(nic, cuc_dump_reset, 0)))    return err;
    e100_disable_irq(nic);
    return 0;
}

```

e100_hw_init()函数当中调用的各种内容都是通过 PCI 对 e100 硬件的操作来完成的。其中，主要的一个函数是 e100_exec_cmd()，用于执行网卡的各种命令如下所示。

```

#define E100_WAIT_SCB_TIMEOUT 20000 /* we might have to wait 100ms!!! */
#define E100_WAIT_SCB_FAST 20      /* delay like the old code */
static int e100_exec_cmd(struct nic *nic, u8 cmd, dma_addr_t dma_addr)

```

```

{
    unsigned long flags;
    unsigned int i;
    int err = 0;
    spin_lock_irqsave(&nic->cmd_lock, flags);
    for (i = 0; i < E100_WAIT_SCB_TIMEOUT; i++) { // 根据延时进行操作
        if (likely(!ioread8(&nic->csr->scb.cmd_lo))) break;
        cpu_relax();
        if (unlikely(i > E100_WAIT_SCB_FAST))    udelay(5);
    }
    // ..... 省略错误处理
    iowrite8(cmd, &nic->csr->scb.cmd_lo); // 进行实际的操作
err_unlock:
    spin_unlock_irqrestore(&nic->cmd_lock, flags);
    return err;
}

```

e100_exec_cmd()参数中的 nic 就是本驱动定义的网络相关的信息。

e100_exec_cb()函数是执行过程中的一个回调，在 e100_hw_init()等函数当中被调用。这个函数的实现如下所示：

```

static int e100_exec_cb(struct nic *nic, struct sk_buff *skb,
    void (*cb_prepare)(struct nic *, struct cb *, struct sk_buff *))
{
    struct cb *cb;
    unsigned long flags;
    int err = 0;
    spin_lock_irqsave(&nic->cb_lock, flags);
    // ..... 省略错误处理
    cb = nic->cb_to_use;
    nic->cb_to_use = cb->next;
    nic->cbs_avail--;
    cb->skb = skb;
    // ..... 省略错误处理
    cb_prepare(nic, cb, skb); // 准备函数
    cb->command |= cpu_to_le16(cb_s);
    wmb();
    cb->prev->command &= cpu_to_le16(~cb_s);
    while (nic->cb_to_send != nic->cb_to_use) {
        if (unlikely(e100_exec_cmd(nic, nic->cuc_cmd, // 执行命令
            nic->cb_to_send->dma_addr))) {
            // ..... 省略错误处理
            break;
        } else {
            nic->cuc_cmd = cuc_resume;
            nic->cb_to_send = nic->cb_to_send->next;
        }
    }
    // ..... 省略错误处理
    return err;
}

```

参数中的 cb_prepare 是回调函数，用于实际的回调准备工作，其参数中的 nic 结构用于网络的控制，cb 结构是实际的回调函数，sk_buff 函数是 Socket Buffer。本函数表示在 PCI 操作的过程中，某些功能需要和网络设备结合起来。

第 23 章

音频系统和驱动

23.1 音频系统概述

音频系统需要将音频硬件的功能体现到 Linux 的软件系统。音频硬件通常由两个方面组成：一个是负责控制和处理数据的音频模块，另一个是数字模拟转换的音频 Codec 芯片。在嵌入式系统中，音频模块通常在嵌入式处理器的内部，而音频 Codec 芯片在嵌入式处理器的外部。在桌面电脑中，声卡通常是集成了上述两个部分的 PCI 卡，如果音频芯片集成在主板上，逻辑上的接口也是与之类似的。

软件与音频硬件交互的时候，通常具有两个方面的功能：一个是控制方面，另一个是数据方面。

- 控制方面：提供音频的参数设置，例如：采样率、通道、音量控制等方面。
- 数据方面：音频数据的传输，通常为 PCM（脉冲编码调制）格式。

音频系统在 Linux 中被称为 Sound（声音）。Linux 音频的框架有标准的 OSS 和 ALSA 两种架构。OSS 是 UNIX 通用的音频系统，相对陈旧和简单；随着增加功能的需要，Linux 后来又引入了 ALSA 架构。

由于历史的原因，Linux 系统的音频驱动在一个独立的 sound 目录中，而不像其他模块一样在 driver 目录中。sound/sound_core.c 文件为 OSS 和 ALSA 公用。

23.2 OSS 架构

OSS（Open Sound System，开放声音系统）是一种 UNIX 上通用的音频系统架构。1992 年，Hannu Savolainen 创造了 OSS。OSS 目前可用于多个主流类 UNIX 操作系统。OSS 可以在 4 种授权选择下发布：其中 BSD 许可、GPL 许可和 Common Development and Distribution 许可是自由软件授权，此外还有私有许可。

OSS 系统的网站是 <http://www.opensound.com/oss.html>。

23.2.1 OSS 系统的结构

OSS 提供了数字音频方面的功能，也提供了 MIDI (Musical Instrument Digital Interface, 乐器数字接口) 方面的功能。OSS 系统的所有 API 都是 UNIX 标准的，此种接口也被称为 UNIX Audio API。因此在所有支持 OSS 的系统中，用户空间的软件都非常类似。

OSS 音频驱动的架构如图 23-1 所示。

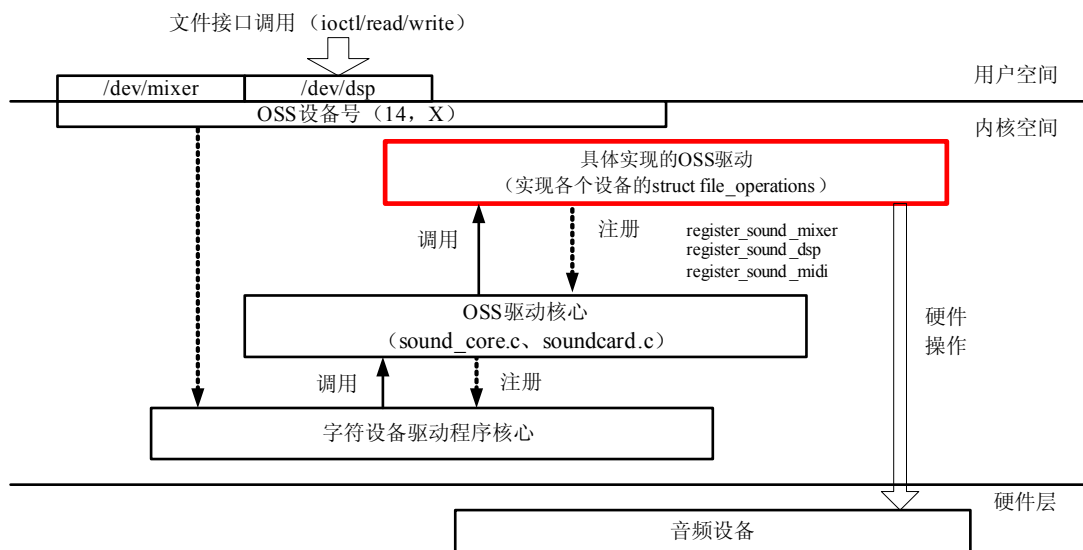


图 23-1 OSS 音频驱动的架构

OSS 驱动向用户空间提供了字符设备的节点，一个 OSS 音频系统的实现会提供多个字符设备节点。OSS 的主设备号为 14，次设备号由各个设备单独定义。

OSS 的几种设备文件如下所示。

- /dev/mixer: 音频控制设备，次设备号为 0。控制音频硬件的混音器，调整音量大小，选择音源。
- /dev/dsp: 音频数据设备，次设备号为 3。读此设备进行录音，写此设备进行放音。
- /dev/sequencer: 次设备号为 1。访问声卡内置的或者连接在 MIDI 端口的合成器。
- /dev/sndstat: 次设备号为 6。测试声卡，读这个文件可以显示声卡驱动的信息。
- /dev/midiXX: MIDI 端口，次设备号为 2、18、34。

在用户空间操作 OSS 数据音频，通常使用 /dev/mixer 设备节点进行 ioctl 调用，完成控制操作，对 /dev/dsp 设备节点使用 write 调用进行放音，使用 read 调用进行录音。

/dev/dsp 设备具有一些变种，例如 /dev/dspW、/dev/audio，它们的区别在于采样的编码不同，/dev/dsp 使用 8 位无符号数的线性编码，/dev/audio 使用 μ 律编码（用于与 SunOS 的兼容），/dev/dspW 使用 16 位有符号数的线性编码。

OSS 的指标主要有采样速率（例如：电话为 8kb/s，DVD 为 96kb/s）、通道数目（单声道、立体声）、采样分辨率（8 位、16 位）。

OSS 系统的主要头文件如下所示。

- `include/linux/soundcard.h`: OSS 驱动的主要头文件。
- `include/linux/sound.h`: 定义 OSS 驱动的次设备号和注册函数。

OSS 系统程序源代码的核心是音频系统公用的 `sound/sound_core.c` 文件, 还有在 `sound/oss/` 目录中的内容, 主要源代码文件包括 `soundcard.c`、`audio.c`, 以及 `sound_timer.c`。OSS 系统的配置宏为 `CONFIG_SOUND_OSS`。

23.2.2 OSS 系统的核心

`sound.h` 中定义了设备类型, 如下所示。

```
#define SND_DEV_CTL      0 // 控制设备: /dev/mixer
#define SND_DEV_SEQ      1 // 顺序器设备: /dev/sequencer (如 FM 合成器和 MIDI 输出)
#define SND_DEV_MIDIN    2 // 原始 MIDI 访问
#define SND_DEV_DSP      3 // 数字音频: /dev/dsp
#define SND_DEV_AUDIO    4 // Sparc 兼容的数字音频: /dev/audio
#define SND_DEV_DSP16    5 // 16 位采样的数字音频
#define SND_DEV_UNUSED    6
#define SND_DEV_AWFM     7
#define SND_DEV_SEQ2     8 // 第二个顺序器: /dev/sequencer
#define SND_DEV_SYNTH    9 // 原始同步访问设备: /dev/synth
#define SND_DEV_DMFM     10 // 原始同步访问设备: /dev/dmfm
#define SND_DEV_UNKNOWN11 11
#define SND_DEV_ADSP     12 // 类似 /dev/dsp
#define SND_DEV_AMIDI    13 // 类似 /dev/midi
#define SND_DEV_ADMIDI   14 // 类似 /dev/dmmidi (onsolete)
```

其中主要的设备就是用于数字音频的 `SND_DEV_DSP` (设备节点: `/dev/mixer`) 和用于控制的 `SND_DEV_CTL` (设备节点: `/dev/dsp`)。

`sound.h` 中定义了各种 OSS 设备的注册和注销函数, 如下所示。

```
extern int register_sound_special(const struct file_operations *fops, int unit);
extern int register_sound_special_device(const struct file_operations *fops,
                                         int unit, struct device *dev);
extern int register_sound_mixer(const struct file_operations *fops, int dev);
extern int register_sound_midi(const struct file_operations *fops, int dev);
extern int register_sound_dsp(const struct file_operations *fops, int dev);
extern void unregister_sound_special(int unit);
extern void unregister_sound_mixer(int unit);
extern void unregister_sound_midi(int unit);
extern void unregister_sound_dsp(int unit);
```

驱动程序对 OSS 音频的实现就是实现一个 `file_operations`, 然后可以注册成 `dsp` 设备或者 `mixer` 设备。`file_operations` 的 `ioctl` 由 OSS 驱动核心定义, 需要根据驱动的具体情况来实现。

`soundcard.h` 中数据设备 (`/dev/dsp`) 的 `ioctl` 命令如下所示:

```
#define SNDCTL_DSP_RESET      _SIO ('P', 0)
#define SNDCTL_DSP_SYNC      _SIO ('P', 1)
#define SNDCTL_DSP_SPEED      _SIOWR('P', 2, int)
#define SNDCTL_DSP_STEREO     _SIOWR('P', 3, int)
#define SNDCTL_DSP_GETBLKSIZE _SIOWR('P', 4, int)
```

```
#define SNDCTL_DSP_SAMPLESIZE      SNDCTL_DSP_SETFMT
#define SNDCTL_DSP_CHANNELS        _SIOWR('P', 6, int)
#define SOUND_PCM_WRITE_CHANNELS  SNDCTL_DSP_CHANNELS
#define SOUND_PCM_WRITE_FILTER     _SIOWR('P', 7, int)
#define SNDCTL_DSP_POST            _SIO ('P', 8)
#define SNDCTL_DSP_SUBDIVIDE       _SIOWR('P', 9, int)
#define SNDCTL_DSP_SETFRAGMENT     _SIOWR('P',10, int)
```

OSS 数字设备（dsp）主要通过读、写进行数据操作。ioctl 提供额外的控制命令，用于设置速度、通道数、是否立体声等。

soundcard.h 中混音设备（/dev/mixer）的 ioctl 命令如下所示：

```
typedef struct mixer_info
{
    char id[16];
    char name[32];
    int modify_counter;
    int fillers[10];
} mixer_info;
typedef struct _old_mixer_info /* Obsolete */
{
    char id[16];
    char name[32];
} _old_mixer_info;
#define SOUND_MIXER_INFO          _SIOR ('M', 101, mixer_info)
#define SOUND_OLD_MIXER_INFO      _SIOR ('M', 101, _old_mixer_info)
typedef unsigned char mixer_record[128];
#define SOUND_MIXER_ACCESS        _SIOWR('M', 102, mixer_record)
#define SOUND_MIXER_AGC           _SIOWR('M', 103, int)
#define SOUND_MIXER_3DSE         _SIOWR('M', 104, int)
#define SOUND_MIXER_PRIVATE1      _SIOWR('M', 111, int)
#define SOUND_MIXER_PRIVATE2      _SIOWR('M', 112, int)
```

SOUND_MIXER_INFO 控制命令的参数为 mixer_info 结构，表示一个混音器的信息，用于将驱动程序的信息返回给用户空间。SOUND_MIXER_PRIVATE1 等几个命令为私有的命令号，参数类型为整数，每个驱动程序可以自定义。

23.2.3 OSS 系统的实现

各种具体硬件为 OSS 实现的声卡驱动，主要都在 sound/oss 目录中。几个相关的驱动程序如下所示。

- au1550_ac97.c 和 ac97_codec.c：实现了 Au1550/Au1200 AC97 声卡驱动。
- vwsnd.c：实现了 SGI Visual Workstation 的声卡驱动。
- swarm_cs4297a.c：实现了 Crystal 的 CS4297a 为 Swarm 系统的驱动。

随着 Linux 系统的发展，使用 OSS 编写的音频驱动程序已经不是主流，因此驱动之中的相关代码并不是很多。

在基于 OSS 的驱动程序架构中，用户空间主要使用/dev/mixer 进行控制，使用/dev/dsp 进行数据流的输入和输出。

23.3 ALSA 架构

ALSA (Advanced Linux Sound Architecture, 高级 Linux 声音体系) 是 Linux 内核中专用的音频系统架构。ALSA 项目在 1998 已经开始, 起初一直作为一个单独的软件包开发, 直到 2002 年它被引入 Linux 内核的开发版本。从 Linux 2.6 版本开始, ALSA 成为 OSS 的替代者, 作为 Linux 内核的标准音频架构。

ALSA 是完全开放源代码的音频驱动。除了像 OSS 那样提供一组内核驱动程序模块之外, ALSA 还专门为简化应用程序的编写提供用户空间的函数库。与 OSS 提供的基于 ioctl 等原始编程接口相比, ALSA 函数库使用起来更方便一些。

通过 ALSA 用户空间的库, 开发人员可以方便、快捷地开发出自己的应用程序, 对驱动程序的控制细节则留给函数库进行内部处理。ALSA 的开发建议应用程序开发者使用音频函数库, 而不是通过系统调用直接调用驱动程序。

ALSA 系统的网站为: <http://www.alsa-project.org/>。

23.3.1 ALSA 系统的结构

ALSA 音频驱动的架构如图 23-2 所示。

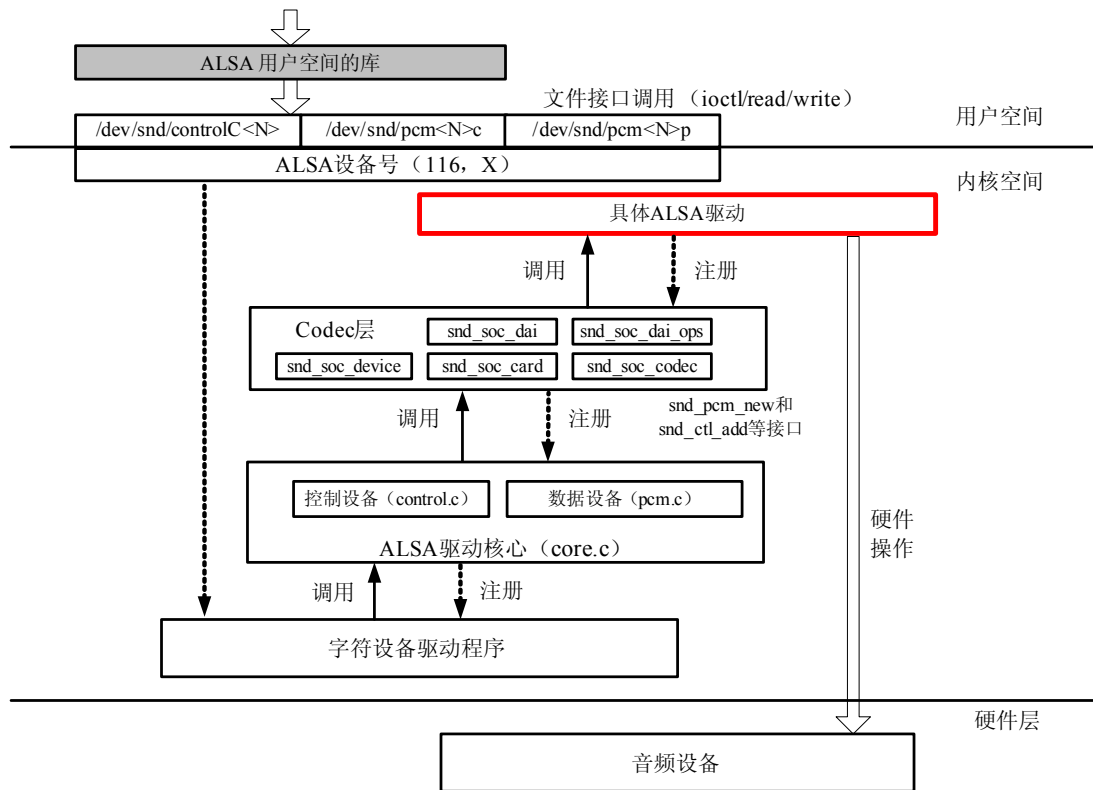


图 23-2 ALSA 音频驱动的架构

ALSA 驱动提供字符设备的接口，其主设备号是 116，通常在 `/dev/snd/` 目录中。用户空间的设备节点根据声卡分为若干组，每组有若干个次设备号。ALSA 主要的设备节点如下所示。

- `/dev/snd/controlCX`：某个声卡的主控制设备。
- `/dev/snd/pcm<N>c`：某个声卡数据输入设备，用于 PCM 捕获（Capture）。
- `/dev/snd/pcm<N>p`：某个声卡数据输出设备，用于 PCM 播放（Play）。
- `/dev/snd/seq`：顺序器设备。
- `/dev/snd/timer`：定时器设备。

ALSA 设备的次设备号没有根据类型分配，在每个系统中可能不同。

`proc` 文件系统的 `/proc/asound/` 目录提供了 ALSA 系统的信息，其中几个主要的文件如下所示。

- `devices` 文件：ALSA 设备的列表。
- `cards` 文件：声卡的列表。
- `version` 文件：ALSA 驱动的版本。
- `card<N>/` 目录：每个声卡的信息，其中包括 `id` 文件、`pcm<N>c` 和 `pcm<N>p` 目录。
- `timers` 文件：定时器设备的信息。
- `seq/` 目录：顺序器设备的目录。

表示设备的 `/proc/asound/devices` 文件如下所示：

```
$ cat /proc/asound/devices
1:          : sequencer
2: [ 0- 2]: digital audio capture
3: [ 0- 0]: digital audio playback
4: [ 0- 0]: digital audio capture
5: [ 0- 0]: hardware dependent
6: [ 0]    : control
33:        : timer
```

第一列的数字代表了次设备号。

`/proc/asound/cards` 文件的内容通常具有下列格式：

```
<N> [    <Name>    ]: <snd_soc_codec>名称 - <snd_soc_card>名称
                        <snd_soc_card>名称
```

最前面的序号表示每个声卡设备的序号，同 `/dev/snd/control<N>` 的 `<N>` 含义等价。

ALSA 驱动程序可以支持模拟 OSS 驱动对用户空间的接口。打开两个内核选项之后，可以同时提供 OSS 的 `dsp` 和 `mixer` 设备节点。

ALSA 驱动程序的头文件如下所示。

- `include/asoundef.h`：IEC958 的和 MIDI 的整型常量的定义。
- `include/sound/asound.h`：ALSA 驱动的接口文件。
- `include/sound/core.h`：ALSA 驱动的主要头文件，定义核心数据结构和具体驱动的注册函数。
- `include/sound/pcm.h`：用于数据通道 PCM 的头文件。

- `include/sound/control.h`: 用于控制的头文件。
- `include/sound/soc.h`: 芯片层的头文件。

ALSA 驱动程序的核心实现包括在 `sound/core/` 目录中的 `sound.c`、`pcm.c` 和 `control.c` 等几个文件中。

ALSA 驱动芯片层实现的内容大部分在 `sound/soc/codecs` 目录中。

23.3.2 ALSA 系统的核心

`asoundef.h` 头文件包括了 IEC958 的和 MIDI 的整型常量的定义。IEC958 是数字音频接口标准 (Digital Audio Interface)，默认的音频格式是 16 位，可扩展至 24 位，音频范围允许采样频率为：44.1kHz (CD)、48 kHz (DAT，数字录音带)、32 kHz (DSR)。

ALSA 声卡的注册和注销函数如下所示：

```
static inline int snd_register_device(int type, struct snd_card *card, int dev,
                                     const struct file_operations *f_ops, void *private_data,
                                     const char *name){}
int snd_unregister_device(int type, struct snd_card *card, int dev);
```

ALSA 系统的某一个设备实现后，使用 `snd_register_device_for_dev()` 进行注册，用户空间的设备节点则根据实现的情况产生。

`asound.h` 头文件中定义了几个方面的内容，以 `SNDRV_PCM_IOCTL_` 为开头的是数据设备的 `ioctl` 号，以 `snd_pcm_` 为开头的是数据设备当中所需的数据结构，以 `SNDRV_CTL_IOCTL_` 为开头的是控制设备的 `ioctl` 号，`snd_ctl_` 是控制设备当中所需的数据结构。

`asound.h` 头文件大部分的内容在 `__KERNEL__` 宏之外，并不依赖内核，因此在用户空间调用 ALSA 驱动程序的时候，也需要使用这个文件。

`core.h` 是 ALSA 驱动的主要头文件，其中的几个主要数据结构如下所示。

- `snd_device_ops`: 表示一个音频设备的操作。
- `snd_device`: 表示一个音频设备，其中包括若干个 `snd_card`。
- `snd_card`: 表示一个声卡，可以独立进行操作。

`snd_device` 表示一个系统的音频设备，音频设备当中可以自定义操作，在这个设备当中可以包括若干个声卡 (`snd_card`)，每个声卡是一组可以独立操作的单元。

23.3.3 ALSA 系统芯片层

1. ALSA 系统芯片层的核心

在 `soc.h` 中定义芯片级别的实现，这个层次的内容位于 ALSA 驱动的较下层，几个核心结构如下所示。

- `snd_soc_pcm_stream`: 音频硬件的数据流。
- `snd_soc_ops`: 用于音频硬件的操作，包括开、关、参数设置等。
- `snd_soc_codec`: 对音频硬件的编解码芯片的实现。
- `snd_soc_dai_link`: 声卡中 DAI 的接口，其中包含了 `snd_soc_op`。

- **snd_soc_card**: 一个声卡, 包含了 **snd_soc_dai_link** 的数组。
- **snd_soc_device**: 一个音频硬件接口。

dai 的含义为数字音频接口 (Digital Audio Interface), 表示针对某个硬件实现的 ALSA 驱动程序层。

一个具体平台的声音驱动层的实现, 要根据 **soc.h** 中定义的内容完成, 其中的一些函数如下所示:

```
void snd_soc_free_pcms(struct snd_soc_device *socdev);
int snd_soc_new_pcms(struct snd_soc_device *socdev, int idx, const char *xid);
```

例如, SOC 的通用函数 **snd_soc_new_pcms()**调用了 **snd_pcm_new()**, 进而调用 ALSA 核心建立 PCM 设备, **snd_soc_cnew()**则调用 **snd_ctl_new1()**函数建立控制设备。

soc-dai.h 文件是数字音频接口层的头文件, **snd_soc_dai** 结构表示 DAI 层的实现, **snd_soc_dai_ops** 结构表示了 DAI 层的操作。

snd_soc_dai 结构的内容如下所示:

```
struct snd_soc_dai_ops {
    int (*set_sysclk)(struct snd_soc_dai *dai,      // DAI 时钟的配置
                     int clk_id, unsigned int freq, int dir);
    int (*set_pll)(struct snd_soc_dai *dai, int pll_id, int source,
                  unsigned int freq_in, unsigned int freq_out);
    int (*set_clkdiv)(struct snd_soc_dai *dai, int div_id, int div);
    int (*set_fmt)(struct snd_soc_dai *dai, unsigned int fmt); // DAI 格式的设置
    int (*set_tdm_slot)(struct snd_soc_dai *dai,
                       unsigned int tx_mask, unsigned int rx_mask, int slots, int slot_width);
    int (*set_channel_map)(struct snd_soc_dai *dai,
                          unsigned int tx_num, unsigned int *tx_slot,
                          unsigned int rx_num, unsigned int *rx_slot);
    int (*set_tristate)(struct snd_soc_dai *dai, int tristate);
    int (*digital_mute)(struct snd_soc_dai *dai, int mute); // 静音的处理
    // PCM 音频的操作, 被 soc-core 部分调用
    int (*startup)(struct snd_pcm_substream *, struct snd_soc_dai *);
    void (*shutdown)(struct snd_pcm_substream *, struct snd_soc_dai *);
    int (*hw_params)(struct snd_pcm_substream *,
                    struct snd_pcm_hw_params *, struct snd_soc_dai *);
    int (*hw_free)(struct snd_pcm_substream *, struct snd_soc_dai *);
    int (*prepare)(struct snd_pcm_substream *, struct snd_soc_dai *);
    int (*trigger)(struct snd_pcm_substream *, int, struct snd_soc_dai *);
    snd_pcm_sframes_t (*delay)(struct snd_pcm_substream *, struct snd_soc_dai *);
};
```

snd_soc_dai_ops 结构当中的 **startup**、**shutdown**、**hw_params** 等函数指针将被 SOC 芯片的核心部分调用, 参数类型 **snd_pcm_substream** 表示 PCM 数据流的操作。

snd_soc_dai 结构的主要内容如下所示:

```
struct snd_soc_dai {
    char *name;          // DAI 的描述
    unsigned int id;
    // .....省略部分内容: AC97 的特殊定义
    int (*probe)(struct platform_device *pdev, struct snd_soc_dai *dai);
    void (*remove)(struct platform_device *pdev, struct snd_soc_dai *dai);
    int (*suspend)(struct snd_soc_dai *dai);
```

```

int (*resume)(struct snd_soc_dai *dai);
struct snd_soc_dai_ops *ops;           // 表示 DAI 的操作
struct snd_soc_pcm_stream capture;     // DAI 的能力
struct snd_soc_pcm_stream playback;
unsigned int symmetric_rates:1;
struct snd_soc_codec *codec;          // 运行时的信息
// .....省略部分内容: 私有数据、链表等
};

```

snd_soc_dai 结构当中包含了表示操作的 snd_soc_dai_ops, 以及 snd_soc_codec 的指针。snd_soc_dai 的注册函数如下所示:

```

int snd_soc_register_dai(struct snd_soc_dai *dai);
void snd_soc_unregister_dai(struct snd_soc_dai *dai);
int snd_soc_register_daIs(struct snd_soc_dai *dai, size_t count);
void snd_soc_unregister_daIs(struct snd_soc_dai *dai, size_t count);

```

snd_soc_dai 和 snd_soc_dai_ops 结构是 ALSA 具体芯片驱动程序所需要实现的主要内容。snd_soc_dai_ops 作为 snd_soc_dai 结构中的一个指针, 之后通过 snd_soc_register_dai() 等函数注册。

2. PCI AC97 的 ALSA 系统芯片层的实现

ac97.c 是针对 AC97 声卡的核心实现部分。

AC97 声卡的实现除了 ac97.c 之外, 还有 sound/pci/ac97/ 目录中的内容。

ac97.c 当中具有如下的定义:

```

static struct snd_soc_dai_ops ac97_dai_ops = {
    .prepare = ac97_prepare,           // 定义 AC97 的 prepare 操作
};
struct snd_soc_dai ac97_dai = {
    .name = "AC97 HiFi",
    .ac97_control = 1                  // 表示用于 AC97 的控制
    .playback = {
        .stream_name = "AC97 Playback", // 用于播放的设备
        .channels_min = 1,               .channels_max = 2,
        .rates = STD_AC97_RATES,         .formats = SND_SOC_STD_AC97_FMTS, },
    .capture = {
        .stream_name = "AC97 Capture",   // 用于捕获的设备
        .channels_min = 1,               .channels_max = 2,
        .rates = STD_AC97_RATES,         .formats = SND_SOC_STD_AC97_FMTS, },
    .ops = &ac97_dai_ops,
};

```

ac97_dai 结构包括了播放和捕获设备, 分别用于输出和输入。

本驱动的探测函数 ac97_soc_probe() 的实现如下所示:

```

static int ac97_soc_probe(struct platform_device *pdev)
{
    struct snd_soc_device *socdev = platform_get_drvdata(pdev); // 平台的私有数据
    struct snd_soc_card *card = socdev->card;
    struct snd_soc_codec *codec;
    struct snd_ac97_bus *ac97_bus;
    struct snd_ac97_template ac97_template;
    int i;
    int ret = 0;
    socdev->card->codec = kzalloc(sizeof(struct snd_soc_codec), GFP_KERNEL);
    if (!socdev->card->codec) return -ENOMEM;
}

```

```

    codec = socdev->card->codec;           // 填充 snd_soc_codec 结构
    mutex_init(&codec->mutex);
    codec->name = "AC97";                  // snd_soc_codec 的名称
    codec->owner = THIS_MODULE;
    codec->dai = &ac97_dai;                 // DAI 的结构 snd_soc_dai
    codec->num_dai = 1;
    codec->write = ac97_write;              // 读写函数
    codec->read = ac97_read;
    INIT_LIST_HEAD(&codec->dapm_widgets);
    INIT_LIST_HEAD(&codec->dapm_paths);
    ret = snd_soc_new_pcm(socdev, SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1);
    if (ret < 0) goto err;
    ret = snd_ac97_bus(codec->card, 0, &soc_ac97_ops, NULL, &ac97_bus);
    if (ret < 0) goto bus_err;
    memset(&ac97_template, 0, sizeof(struct snd_ac97_template));
    ret = snd_ac97_mixer(ac97_bus, &ac97_template, &codec->ac97);
    if (ret < 0) goto bus_err;
    for (i = 0; i < card->num_links; i++) { // 循环处理 snd_soc_card 的每一个声卡
        if (card->dai_link[i].codec_dai->ac97_control) {
            snd_ac97_dev_add_pdata(codec->ac97,
                                    card->dai_link[i].cpu_dai->ac97_pdata);
        }
    }
    return 0;
// .....省略错误处理
}

```

此处探测函数的实现逻辑为，平台设备的私有数据是一个 `snd_soc_device` 结构，从这个结构中得到一个 `snd_soc_card` 结构，再将 `snd_soc_card` 结构中的 `snd_soc_codec` 结构进行填充。驱动程序所实现的 `snd_soc_dai` 就作为 `snd_soc_card` 中的 `dai` 指针。其中使用的 `snd_ac97_template` 结构是针对 AC97 声卡的特定结构。

`ac97_prepare()`函数的实现如下所示：

```

static int ac97_prepare(struct snd_pcm_substream *substream,
                        struct snd_soc_dai *dai)
{
    struct snd_pcm_runtime *runtime = substream->runtime;
    struct snd_soc_pcm_runtime *rtd = substream->private_data;
    struct snd_soc_device *socdev = rtd->socdev;
    struct snd_soc_codec *codec = socdev->card->codec;
    int reg = (substream->stream == SNDRV_PCM_STREAM_PLAYBACK) ?
        AC97_PCM_FRONT_DAC_RATE : AC97_PCM_LR_ADC_RATE;
    return snd_ac97_set_rate(codec->ac97, reg, runtime->rate);
}

```

其中调用的 `snd_ac97_set_rate()`函数来自基于 PCI 的 AC97 声卡的公共实现。

3. 嵌入式芯片的 ALSA 系统芯片层的实现

各个嵌入式芯片中有 AC97 的，也有 i2s 的：例如，S3c-ac97.c 是三星平台 AC97 声卡的实现，s3c64xx-i2s.c 是三星平台 I2S 声卡的实现。

s3c64xx-i2s.c 的核心实现如下所示：

```

struct snd_soc_dai s3c64xx_i2s_dai[MAX_I2SV3];
EXPORT_SYMBOL_GPL(s3c64xx_i2s_dai);
static struct snd_soc_dai_ops s3c64xx_i2s_dai_ops;

```

```

static __devinit int s3c64xx_iis_dev_probe(struct platform_device *pdev)
{
    struct s3c_i2sv2_info *i2s;
    struct snd_soc_dai *dai;
    int ret;
// .....省略错误处理
    i2s = &s3c64xx_i2s[pdev->id];                // 基于 i2s 的私有数据结构
    dai = &s3c64xx_i2s_dai[pdev->id];            // 获取 snd_soc_dai 类型
    dai->dev = &pdev->dev;
    dai->name = "s3c64xx-i2s";                    // snd_soc_dai 的名称
    dai->id = pdev->id;
    dai->symmetric_rates = 1;
    dai->playback.channels_min = 2;                // 播放设备的参数
    dai->playback.channels_max = 2;
    dai->playback.rates = S3C64XX_I2S_RATES;
    dai->playback.formats = S3C64XX_I2S_FMTS;
    dai->capture.channels_min = 2;                // 捕获设备的参数
    dai->capture.channels_max = 2;
    dai->capture.rates = S3C64XX_I2S_RATES;
    dai->capture.formats = S3C64XX_I2S_FMTS;
    dai->probe = s3c64xx_i2s_probe;                // snd_soc_dai 的探测操作
    dai->ops = &s3c64xx_i2s_dai_ops;                // snd_soc_dai_ops 结构
    i2s->feature |= S3C_FEATURE_CDCLKCON;            // 驱动私有数据的处理
    i2s->dma_capture = &s3c64xx_i2s_pcm_stereo_in[pdev->id];
    i2s->dma_playback = &s3c64xx_i2s_pcm_stereo_out[pdev->id];
// .....省略错误处理: 驱动私有结构 s3c_i2sv2_info 的初始化, 播放设备和捕获设备
    i2s->iis_cclk = clk_get(&pdev->dev, "audio-bus");
// .....省略错误处理
    clk_enable(i2s->iis_cclk);
    ret = s3c_i2sv2_probe(pdev, dai, i2s, 0);    // i2s 总线的注册
    if (ret) goto err_clk;
    ret = s3c_i2sv2_register_dai(dai);    // 其中调用 snd_soc_register_dai() 注册
    if (ret != 0) goto err_i2sv2;
    return 0;
// .....省略错误处理
}

```

探测函数 `s3c64xx_iis_dev_probe()` 在接口方面就是实现了 `snd_soc_dai` 结构, 填充数据结构后向系统注册。对于驱动硬件方面的内容, 通过调用 `i2s` 的内容完成。`s3c64xx_i2s_probe()` 完成了 GPIO 的设置。

`s3c_i2sv2_register_dai()` 函数属于 S3C 系列处理器通用的音频系统功能函数。其实现的代码在 `sound/soc/s3c24xx/` 目录的 `s3c-i2s-v2.c` 文件中, 如下所示:

```

int s3c_i2sv2_register_dai(struct snd_soc_dai *dai)
{
    struct snd_soc_dai_ops *ops = dai->ops;    // 得到 snd_soc_dai_ops 结构
    ops->trigger = s3c2412_i2s_trigger;        // 赋值 snd_soc_dai_ops 中的各个函数指针
    if (!ops->hw_params) ops->hw_params = s3c_i2sv2_hw_params;
    ops->set_fmt = s3c2412_i2s_set_fmt;
    ops->set_clkdiv = s3c2412_i2s_set_clkdiv;
    ops->set_sysclk = s3c_i2sv2_set_sysclk;
    if (!ops->delay) ops->delay = s3c2412_i2s_delay;
    dai->suspend = s3c2412_i2s_suspend;
    dai->resume = s3c2412_i2s_resume;
    return snd_soc_register_dai(dai);        // 注册 snd_soc_dai 结构
}

```

s3c2412_i2s_trigger()等函数的实现，被赋值成 snd_soc_dai_ops 结构中的各个函数指针。它们都是通过读写 S3C 处理器 i2s 总线的寄存器来实现的。

23.3.4 ALSA 的用户空间

ALSA 系统不仅包括内核中的驱动程序，也提供了用户空间的支持。在用户空间使用 ALSA 驱动程序的时候，通常通过 ALSA 用户空间的支持完成。

ALSA 用户空间的支持包括下面几个部分。

- **alsa-lib-<version>**: ALSA 用户空间的库。
- **alsa-utils-<version>**: aplay、amixer 的命令行工具，基于 ALSA 用户空间的库。
- **alsa-tools-<version>**: 额外的工具。

ALSA 驱动、alsa-lib 和 alsa-utils 之间的关系如图 23-3 所示。

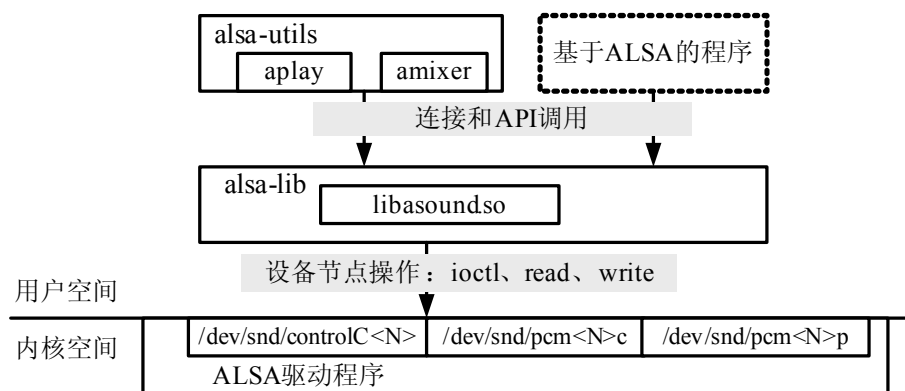


图 23-3 ALSA 驱动、alsa-lib 和 alsa-utils 之间的关系

1. alsa-lib

ALSA 库（alsa-lib）通过 ioctl 等接口调用 ALSA 驱动程序的设备节点。在用户空间对于 ALSA 驱动的调用，通常要调用用户空间的 ALSA 库的接口，而不是直接调用 ALSA 驱动程序。

ALSA 库经过编译后，生成 libasound.so 动态库，并提供 asoundlib.h 作为对其他程序统一的 API 头文件，这个头文件包括了 asoundef.h、global.h、pcm.h、control.h 等 ALSA 系统的头文件。

在数据设备的操作函数 pcm.h 头文件中，几个函数如下所示：

```

int snd_pcm_open(snd_pcm_t **pcm, const char *name,
                 snd_pcm_stream_t stream, int mode);
int snd_pcm_close(snd_pcm_t *pcm);
int snd_pcm_prepare(snd_pcm_t *pcm);
int snd_pcm_reset(snd_pcm_t *pcm);
int snd_pcm_start(snd_pcm_t *pcm);
int snd_pcm_pause(snd_pcm_t *pcm, int enable);
snd_pcm_sframes_t snd_pcm_writeti(snd_pcm_t *pcm,
                                  const void *buffer, snd_pcm_uframes_t size);
    
```

```

snd_pcm_sframes_t snd_pcm_readi(snd_pcm_t *pcm,
                                void *buffer, snd_pcm_uframes_t size);
snd_pcm_sframes_t snd_pcm_writen(snd_pcm_t *pcm,
                                 void **bufs, snd_pcm_uframes_t size);
snd_pcm_sframes_t snd_pcm_readn(snd_pcm_t *pcm,
                                 void **bufs, snd_pcm_uframes_t size);

```

`snd_pcm_t` 表示 PCM 数据设备的操作句柄，各种操作函数的参数很多，各种定义也在 ALSA 库的头文件中定义。这些函数的实现都基于 `/dev/snd/pcm<N>c` 和 `/dev/snd/pcm<N>` 设备节点。

在控制设备的操作函数 `control.h` 头文件中，几个函数如下所示：

```

int snd_ctl_open(snd_ctl_t **ctl, const char *name, int mode);
int snd_ctl_close(snd_ctl_t *ctl);

int snd_ctl_read(snd_ctl_t *ctl, snd_ctl_event_t *event);
int snd_ctl_wait(snd_ctl_t *ctl, int timeout);
const char *snd_ctl_name(snd_ctl_t *ctl);
snd_ctl_type_t snd_ctl_type(snd_ctl_t *ctl);

```

`snd_ctl_t` 表示控制数据设备的操作句柄，`snd_ctl_type_t` 表示几个控制的类型。

2. alsa-utils

ALSA 驱动程序需要特殊的 `ioctl` 操作进行设置，因此在用户空间不能直接利用 `read/write` 的方式进行 PCM 格式数据的输出。ALSA 工具（`alsa-utils`）提供了 `aplay`、`amixer` 等命令行工具。

`aplay` 工具的命令行参数如下所示：

```

Usage: aplay [OPTION]... [FILE]...
-h, --help                help
    --version              print current version
-l, --list-devices         list all soundcards and digital audio devices
-L, --list-pcms            list device names
-D, --device=NAME         select PCM by name
-q, --quiet                quiet mode
-t, --file-type TYPE       file type (voc, wav, raw or au)
-c, --channels=#           channels
-f, --format=FORMAT        sample format (case insensitive)
-r, --rate=#               sample rate
-d, --duration=#           interrupt after # seconds
-M, --mmap                 mmap stream
-N, --nonblock              nonblocking mode
-F, --period-time=#        distance between interrupts is # microseconds
-B, --buffer-time=#        buffer duration is # microseconds
    --period-size=#        distance between interrupts is # frames
    --buffer-size=#        buffer duration is # frames
-A, --avail-min=#          min available space for wakeup is # microseconds
-R, --start-delay=#        delay for automatic PCM start is # microseconds
                           (relative to buffer size if <= 0)
-T, --stop-delay=#         delay for automatic PCM stop is # microseconds from xrun
-v, --verbose               show PCM structure and setup (accumulative)
-V, --vumeter=TYPE         enable VU meter (TYPE: mono or stereo)
-I, --separate-channels    one file for each channel
    --disable-resample      disable automatic rate resample

```

```
--disable-channels  disable automatic channel conversions
--disable-format    disable automatic format conversions
--disable-softvol   disable software volume control (softvol)
--test-position     test ring buffer position
--test-coef=#       test coefficient for ring buffer position (default 8)
                    expression for validation is: coef * (buffer_size / 2)
--test-nowait       do not wait for ring buffer - eats whole CPU
```

其中最主要的设置内容格式就是--format=FORMAT 参数后面的 FORMAT, 包括了以下一些值: S8 (8 位有符号)、U8 (8 位无符号)、S16_LE (16 位有符号小端)、S16_BE (16 位有符号大端)、U16_LE (16 位无符号小端)、U16_BE (16 位无符号大端) 等。

在命令行直接键入如下命令, 可以播放一个由数据组成的 PCM 文件:

```
$ aplay -f cd <pcm.data.file>
```

-fcd 参数的格式表示 16 位小端、44100 采样率、立体声 (stereo)。

amixer 工具的命令行参数如下所示:

```
$ amixer --help
Usage: amixer <options> [command]
Available options:
  -h,--help          this help
  -c,--card N        select the card
  -D,--device N      select the device, default 'default'
  -d,--debug         debug mode
  -n,--nocheck       do not perform range checking
  -v,--version       print version of this program
  -q,--quiet         be quiet
  -i,--inactive      show also inactive controls
  -a,--abstract L    select abstraction level (none or basic)
  -s,--stdin         Read and execute commands from stdin sequentially
Available commands:
  scontrols          show all mixer simple controls
  scontents          show contents of all mixer simple controls (default command)
  sset sID P         set contents for one mixer simple control
  sget sID           get contents for one mixer simple control
  controls           show all controls for given card
  contents           show contents of all controls for given card
  cset cID P         set control contents for one control
  cget cID           get control contents for one control
```

amixer 命令用于通过各种控制器 (control) 进行设置和信息获取操作。

在命令行直接执行 amixer 命令, 将列出当前 ALSA 系统中的控制句柄。

除此之外, alsa-utils 中还有 alsactl、alsaloop 等命令行工具。

第 24 章

视频系统和驱动

24.1 视频系统概述

视频系统通常包括视频的输入和输出。

视频输入一般对应于摄像头硬件。在桌面计算机上，视频输入硬件可以是 USB 的摄像头。在嵌入式系统中，视频输入硬件一般就是摄像头和处理器当中的视频处理单元，视频处理单元在某些处理器中被称为 ISP（Image Signal Processing）。

视频输出则对应于某些显示硬件的独立视频显示层。这种视频显示层通常可以提供独立的显存进行操作，与主显示层进行硬件叠加之后显示到屏幕上，此种机制通常都由嵌入式处理器的显示部分直接提供，有时也被称为 Overlay，表示显示叠加层。

Linux 当中称之为视频有两个方面：源代码 video 目录指的是主显示输出，也就是帧缓冲（Frame Buffer）驱动部分，表示对基本图形层的显示支持；源代码 media 目录针对的是对多媒体视频的支持部分。Linux 的多媒体视频方面已经形成了标准的 Video For Linux 驱动程序架构。

Video For Linux 提供了几个方面的接口：视频捕获（Video Capture）、视频叠加（Video Overlay）、视频输出（Video Output）、视频叠加输出（Video Output Overlay）、编解码（Codec）、收音机（Radio）、VBI、Teletext、RDS。其中以视频方面的接口为主，视频中又以视频捕获（也就是视频输入）为主。

24.2 Video for Linux 系统

Video for Linux（简称 V4L）是 Linux 内核中标准的关于视频的驱动子系统。目前主要使用的是 V4L 的第二个版本 Video4Linux2（简称 V4L2）。Video for Linux 的第一个版本主要提供了对摄像头，也就是 Video 输入部分的支持，第二个版本又加入了对视频输出、收音机（Radio）设备的支持。

Video for Linux 视频捕获部分已经是 Linux 当中标准的摄像头架构，USB 和嵌入式 SOC 上的摄像头硬件均可使用这个驱动架构。V4L2 的视频输出部分类似于 FrameBuffer 驱动，却又提供了视频的队列功能。

24.2.1 基本结构

Video for Linux 驱动为用户空间提供了字符设备的设备节点，每一个 Video for Linux 设备可以拥有一个节点。Video for Linux 字符设备，主设备号为 81，设备具有多种类型，每种类型有自己的次设备号范围和设备节点名称。

- /dev/video<N>：次设备号为 0~63，表示视频设备，包括捕获设备和输出设备。
- /dev/radio<N>：次设备号为 64~127，表示收音机（Radio）设备。
- /dev/vtx<N>：次设备号为 192~223，表示 Teletext 设备。
- /dev/vbi<N>：次设备号为 224~255，表示 VBI（Vertical Blank Interrupt）设备。

Video for Linux 驱动的架构如图 24-1 所示。

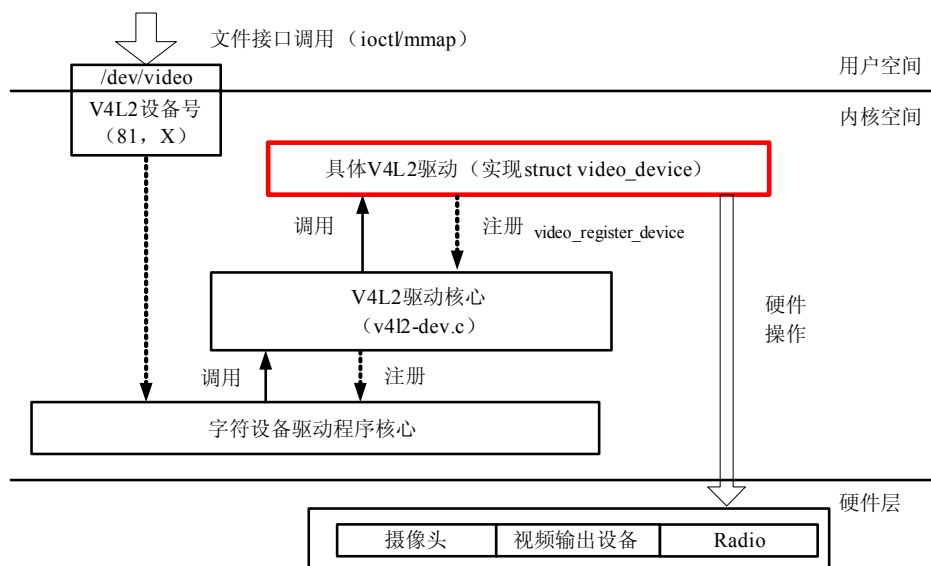


图 24-1 Video for Linux 驱动的架构

Video for Linux 驱动的主要头文件包括 include/linux/ 目录中的 videodev.h 和 videodev2.h，前者是 Video For Linux 第一版的头文件，后者是 Video For Linux 第二版的头文件；include/media/ 目录当中有众多名为 v4l2-XXX.h 的头文件，它们为 Video For Linux 第二版的各层实现提供接口。

Video for Linux 驱动核心实现在 driver/media/video/ 目录中，主要的文件是 v4l2-dev.c 和 v4l2-common.c。v4l1-compat.c 文件用于建成 V4L1 的功能。

Video for Linux 的两个版本并非简单的功能增加，其视频设备的接口形式完全不同，但是 V4L2 可以兼容 V4L1。

Video for Linux 驱动的 Video 设备在用户空间通过各种 ioctl 调用进行控制，并且可以使用 mmap 进行内存映射。

24.2.2 Video for Linux 的核心结构

Video for Linux 的主体功能都是通过各种 ioctl 命令完成的，主要的操作就是控制命令和内存操作。

1. 主要的 ioctl 命令

videodev2.h 文件中定义了 Video for Linux 设备驱动主要使用的 ioctl 命令，V4L2 的 ioctl 号按照 Linux 规范的方式进行定义，用字符'V'表示驱动类型。

几个主要的命令如下所示：

```
#define VIDIOC_QUERYCAP    _IOR('V', 0, struct v4l2_capability)    // 查询能力
#define VIDIOC_G_FMT       _IOWR('V', 4, struct v4l2_format)      // 获得格式
#define VIDIOC_S_FMT       _IOWR('V', 5, struct v4l2_format)      // 设置格式
#define VIDIOC_REQBUFS     _IOWR('V', 8, struct v4l2_requestbuffers) // 申请内存
#define VIDIOC_QUERYBUF    _IOWR('V', 9, struct v4l2_buffer)      // 查询内存
#define VIDIOC_QBUF        _IOWR('V', 15, struct v4l2_buffer)     // 将内存加入队列
#define VIDIOC_DQBUF       _IOWR('V', 17, struct v4l2_buffer)     // 从队列取出内存
#define VIDIOC_STREAMON    _IOW('V', 18, int)                    // 开始流
#define VIDIOC_STREAMOFF   _IOW('V', 19, int)                    // 停止流
#define VIDIOC_G_CTRL      _IOWR('V', 27, struct v4l2_control)    // 得到控制器
#define VIDIOC_S_CTRL      _IOWR('V', 28, struct v4l2_control)    // 设置控制器
```

V4L2 视频设备的主要功能由下面几个 ioctl 命令提供。

- **VIDIOC_QUERYCAP**：用于查询设备的能力，从内核的驱动中获取相关的信息；V4L2_CAP_VIDEO_CAPTURE 表示具有视频捕获（摄像头）能力；V4L2_CAP_VIDEO_OUTPUT 表示具有视频输出能力；V4L2_CAP_VIDEO_OVERLAY 表示具有视频叠加能力。
- **VIDIOC_G_FMT** 和 **VIDIOC_S_FMT**：用于灵活地进行流数据格式的获取和设置。
- **VIDIOC_REQBUFS**：申请内存，参数中包括内存的类型和来源。
- **VIDIOC_QUERYBUF**：查询内存，参数表示实际的内存。
- **VIDIOC_QBUF** 和 **VIDIOC_DQBUF**：将数据加入视频队列和取出。
- **VIDIOC_STREAMON** 和 **VIDIOC_STREAMOFF**：用于开关视频流。
- **VIDIOC_G_CTRL** 和 **VIDIOC_S_CTRL**：各种控制，每种控制有自己的 id。
- **VIDIOC_G_FBUF**、**VIDIOC_S_FBUF** 和 **VIDIOC_OVERLAY**：叠加预览方面的内容。

2. 设备的能力

能力是驱动程序反映给用户空间的信息，VIDIOC_QUERYCAP 命令用于查询 v4l2 驱动的能力。

VIDIOC_QUERYCAP 的参数结构 v4l2_capability 如下所示：

```
struct v4l2_capability {
    __u8 driver[16];          // 驱动的名称，例如"btv"
    __u8 card[32];           // 卡的信息，例如：i.e. "Hauppauge WinTV"
```

```
__u8 bus_info[32];          // 总线信息, 例如: "PCI:" + pci_name(pci_dev)
__u32 version;              // 版本信息
__u32 capabilities;         // 能力值
__u32 reserved[4];         // 保留值
};
```

v4l2_capability 当中的 capabilities 是用整数表示的各种能力值。

表示能力的各种定义如下所示:

```
#define V4L2_CAP_VIDEO_CAPTURE      0x00000001    // 视频捕获设备
#define V4L2_CAP_VIDEO_OUTPUT      0x00000002    // 视频输出设备
#define V4L2_CAP_VIDEO_OVERLAY     0x00000004    // 可以做视频叠加
#define V4L2_CAP_VBI_CAPTURE       0x00000010    // 原始 VBI 捕获设备
#define V4L2_CAP_VBI_OUTPUT        0x00000020    // 原始 VBI 输出设备
#define V4L2_CAP_SLICED_VBI_CAPTURE 0x00000040    // sliced 的 VBI 捕获设备
#define V4L2_CAP_SLICED_VBI_OUTPUT 0x00000080    // sliced 的 VBI 输出设备
#define V4L2_CAP_RDS_CAPTURE       0x00000100    // RDS 数据捕获
#define V4L2_CAP_VIDEO_OUTPUT_OVERLAY 0x00000200  // 可以做视频输出的叠加
#define V4L2_CAP_HW_FREQ_SEEK     0x00000400    // 可以做硬件频率的 s 搜索
#define V4L2_CAP_RDS_OUTPUT        0x00000800    // RDS 编码器
#define V4L2_CAP_TUNER             0x00010000    // 具有 tuner
#define V4L2_CAP_AUDIO             0x00020000    // 具有音频能力
#define V4L2_CAP_RADIO             0x00040000    // 收音机 (Radio) 设备
#define V4L2_CAP_MODULATOR        0x00080000    // 具有调节器 (modulator)
#define V4L2_CAP_READWRITE         0x01000000    // 支持读、写的调用
#define V4L2_CAP_ASYNCIO           0x02000000    // 支持异步 I/O
#define V4L2_CAP_STREAMING         0x04000000    // 支持流 I/O 的 ioctl
```

各种能力的值是按位进行或的关系, 每个设备可能包含一种或者几种能力。一个视频捕获设备通常具有的能力就是 V4L2_CAP_VIDEO_CAPTURE 和 V4L2_CAP_STREAMING。

3. 视频内存的格式

格式表示了每个视频驱动内部视频内存的信息, VIDIOC_G_FMT 和 VIDIOC_S_FMT 两个命令用于获取和设置格式, 两个控制命令的参数格式为 v4l2_format。

v4l2_format 结构的内容如下所示:

```
struct v4l2_format {
    enum v4l2_buf_type type;
    union {
        struct v4l2_pix_format    pix;    // 用于 V4L2_BUF_TYPE_VIDEO_CAPTURE
        struct v4l2_window        win;    // 用于 V4L2_BUF_TYPE_VIDEO_OVERLAY
        struct v4l2_vbi_format    vbi;    // 用于 V4L2_BUF_TYPE_VBI_CAPTURE
        struct v4l2_sliced_vbi_format sliced;
                                           // 用于 V4L2_BUF_TYPE_SLICED_VBI_CAPTURE
        __u8 raw_data[200];          // 自定义的
    } fmt;
};
```

v4l2_format 的首要内容是 v4l2_buf_type 表示几种不同视频内存, 每种视频内存所使用的格式不同, 因此使用一个联合体 fmt 来表示。

表示视频内存类型的枚举值如下所示:

```
enum v4l2_buf_type {
    V4L2_BUF_TYPE_VIDEO_CAPTURE    // 表示视频缓冲区的类型
    = 1,
```

```

V4L2_BUF_TYPE_VIDEO_OUTPUT      = 2,
V4L2_BUF_TYPE_VIDEO_OVERLAY     = 3,
V4L2_BUF_TYPE_VBI_CAPTURE       = 4,
V4L2_BUF_TYPE_VBI_OUTPUT        = 5,
V4L2_BUF_TYPE_SLICED_VBI_CAPTURE = 6,
V4L2_BUF_TYPE_SLICED_VBI_OUTPUT  = 7,
V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY = 8,
V4L2_BUF_TYPE_PRIVATE           = 0x80,
};

```

`v4l2_buf_type` 表示视频缓冲区的类型，几个枚举值和能力的定义有相似的对应关系。对于视频捕获设备，格式的类型就是 `v4l2_pix_format`，这个结构的内容如下所示：

```

struct v4l2_pix_format {
    __u32          width;           // 视频内存尺寸
    __u32          height;          // 视频内存的像素格式
    enum v4l2_field field;
    __u32          bytesperline;    // 每行的字节，0 表示不使用这个特性
    __u32          sizeimage;
    enum v4l2_colorspace colorspace;
    __u32          priv;           // 私有的数据
};

```

`v4l2_pix_format` 结构中有基本的表示宽度、高度和颜色格式的信息。视频颜色格式使用无符号 32 位数来表示，就使 `V4L2_PIX_FMT_` 等开头的宏，分为 RGB、YUV、灰度等情况。

几种颜色格式的定义如下所示：

```

#define v4l2_fourcc(a, b, c, d)\
    (((__u32)(a) | ((__u32)(b) << 8) | ((__u32)(c) << 16) | ((__u32)(d) << 24))
#define V4L2_PIX_FMT_RGB565X v4l2_fourcc('R', 'G', 'B', 'R') /* 16 RGB-5-6-5 BE */
#define V4L2_PIX_FMT_RGB32   v4l2_fourcc('R', 'G', 'B', '4') /* 32 RGB-8-8-8-8 */
#define V4L2_PIX_FMT_GREY     v4l2_fourcc('G', 'R', 'E', 'Y') /* 8 Greyscale */
#define V4L2_PIX_FMT_PAL8     v4l2_fourcc('P', 'A', 'L', '8') /* 8 8-bit palette */
#define V4L2_PIX_FMT_YUV422P  v4l2_fourcc('4', '2', '2', 'P') /* 16 YVU422 planar */
#define V4L2_PIX_FMT_YUV411P  v4l2_fourcc('4', '1', '1', 'P') /* 16 YVU411 planar */
#define V4L2_PIX_FMT_NV12     v4l2_fourcc('N', 'V', '1', '2') /* 12 YCbCr 4:2:0 */
#define V4L2_PIX_FMT_NV21     v4l2_fourcc('N', 'V', '2', '1') /* 12 YCrCb 4:2:0 */

```

根据 `v4l2_fourcc` 宏的定义，颜色格式实际上是 4 个字符拼成的 32 位无符号数。对于视频系统，通常使用的就是某种 YUV 的格式。

4. 视频内存的操作

视频内存的操作是用户空间操作 V4L2 驱动的主要工作。

表示视频内存的来源枚举值如下所示：

```

enum v4l2_memory {
    V4L2_MEMORY_MMAP      = 1,      // 来自内核的映射内存
    V4L2_MEMORY_USERPTR   = 2,      // 来自用户空间的内存
    V4L2_MEMORY_OVERLAY   = 3,      // 叠加层的内存
};

```

`v4l2_memory` 表示内存的来源：`V4L2_MEMORY_MMAP` 表示映射内核空间内存，

V4L2_MEMORY_USERPTR 表示使用用户空间内存，V4L2_MEMORY_OVERLAY 表示叠加内存。在视频设备操作的过程中，内存来自内核还是用户空间，对性能有一定的区别。

表示内存的两个结构体 v4l2_requestbuffers 和 v4l2_buffer 如下所示：

```
struct v4l2_requestbuffers {           // 申请的视频内存类型
    __u32          count;
    enum v4l2_buf_type  type;         // 内存的类型
    enum v4l2_memory    memory;       // 申请内存的来源
    __u32          reserved[2];
};

struct v4l2_buffer {                 // 表示所使用的视频内存
    __u32          index;
    enum v4l2_buf_type  type;         // 表示视频缓冲区的类型
    __u32          bytesused;
    __u32          flags;
    enum v4l2_field      field;
    struct timeval      timestamp;     // 时间戳
    struct v4l2_timecode timecode;
    __u32          sequence;
    enum v4l2_memory    memory;       // 内存的来源
    union {
        __u32          offset;        // 内存的偏移量
        unsigned long   userptr;      // 内存存在用户空间的指针
    } m;
    __u32          length;            // 内存的长度
    __u32          input;
    __u32          reserved;
};
```

v4l2_requestbuffers 结构是 VIDIOC_REQBUFS 命令操作的参数。v4l2_buffer 是 VIDIOC_QUERYBUF 命令操作的参数。在用户空间操作 v4l2 驱动的时候，两个操作常常依次进行。

很多视频设备都支持流式操作，通过 VIDIOC_STREAMON 和 VIDIOC_STREAMOFF 两个命令进行开关。

视频的实际操作则通过 VIDIOC_QBUF 和 VIDIOC_DQBUF 两个命令完成，其中使用参数的类型也是 v4l2_buffer。

用户空间和内核空间视频内存的交流通常通过下面的流程进行：

- 使用 VIDIOC_QBUF 将内存依次加入队列。
- 最后一块内存队列加完之后，要循环从第一块开始。
- 调用 VIDIOC_DQBUF，等待驱动返回内存，此调用具有阻塞作用。

一个具有 4 块视频内存的队列的操作示例如图 24-2 所示。

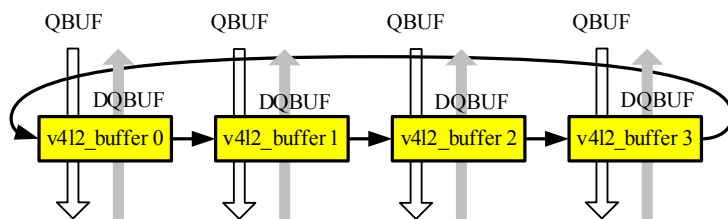


图 24-2 一个具有 4 块视频内存的队列的操作示例

提示：VIDIOC_DQBUF 命令的语义本身也和阻塞有关，在阻塞的过程中可以等待硬件的中断处理，并完成视频内存的填充。

如果驱动程序支持用户空间的内存（V4L2_MEMORY_USERPTR），驱动程序则必须支持 VIDIOC_REQBUFS、VIDIOC_QBUF、VIDIOC_DQBUF、VIDIOC_STREAMON 和 VIDIOC_STREAMOFF 几个 ioctl 命令，还要实现 select() 函数。

如果使用内存映射（V4L2_MEMORY_MMAP）方式，视频内存来自于内核空间，用户空间需要调用 mmap() 函数从设备节点中得到映射的内存。

一段使用单内存的映射方法的示例代码如下所示：

```
struct v4l2_requestbuffers reqbuf;           // 准备请求的内存结构
struct {
    void *start;
    size_t length;
} *buffers;
unsigned int i;
memset(&reqbuf, 0, sizeof(reqbuf));
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory = V4L2_MEMORY_MMAP;           // 内存映射的方式
reqbuf.count = 20;                          // 请求内存的数目
if (-1 == ioctl(fd, VIDIOC_REQBUFS, &reqbuf)) { // 请求内存
    // .....省略错误处理部分
}
// .....省略部分内容：返回的 reqbuf.count 数目不一定和申请的相同
buffers = calloc(reqbuf.count, sizeof(*buffers)); // 分配几个内存结构
assert(buffers != NULL);
for (i = 0; i < reqbuf.count; i++) {         // 循环申请内存
    struct v4l2_buffer buffer;               // 内存结构
    memset(&buffer, 0, sizeof(buffer));
    buffer.type = reqbuf.type;
    buffer.memory = V4L2_MEMORY_MMAP;
    buffer.index = i;
    if (-1 == ioctl(fd, VIDIOC_QUERYBUF, &buffer)) { // 进行内存的查询
        // .....省略错误处理部分
    }
    buffers[i].length = buffer.length;       // 得到内存长度，用于 munmap()
    buffers[i].start = mmap(NULL, buffer.length, // 根据长度和偏移量映射内存
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, buffer.m.offset);
    if (MAP_FAILED == buffers[i].start) {
        // .....省略错误处理部分
    }
}
}
```

由此，已经通过 mmap() 函数从内核中映射了几块内存。

操作完成后，清除内存映射代码如下所示：

```
for (i = 0; i < reqbuf.count; i++)
    munmap(buffers[i].start, buffers[i].length);
```

5. 控制功能

V4L2 提供了统一的额外控制函数，主要的控制内容通过 VIDIOC_G_CTRL 和 VIDIOC_S_CTRL 两个命令实现，控制功能的参数类型是 v4l2_control。

控制相关的内容结构如下所示：

```
struct v4l2_control {                // 表示 V4L2 控制的结构
    __u32      id;
    __s32      value;
};
#define V4L2_CID_BASE                (V4L2_CTRL_CLASS_USER | 0x900)
#define V4L2_CID_USER_BASE          V4L2_CID_BASE
#define V4L2_CID_PRIVATE_BASE        0x08000000    // 驱动私有的 ioctl 控制号
#define V4L2_CID_USER_CLASS          (V4L2_CTRL_CLASS_USER | 1)
#define V4L2_CID_BRIGHTNESS          (V4L2_CID_BASE+0)
#define V4L2_CID_CONTRAST              (V4L2_CID_BASE+1)
#define V4L2_CID_SATURATION            (V4L2_CID_BASE+2)
#define V4L2_CID_HUE                  (V4L2_CID_BASE+3)
#define V4L2_CID_AUDIO_VOLUME          (V4L2_CID_BASE+5)
#define V4L2_CID_AUDIO_BALANCE         (V4L2_CID_BASE+6)
#define V4L2_CID_AUDIO_BASS            (V4L2_CID_BASE+7)
```

控制其实就是一个命令和一个有符号 32 位数表示的参数。V4L2_CID_BRIGHTNESS 等是 V4L2 驱动统一定义的控制命令。驱动程序也可以自定义一些命令，供用户空间的程序调用。

24.2.3 Video for Linux 的其他方面

1. 视频叠加设备

视频叠加设备（Overlay）提供了在图形输出设备上叠加一块内存的功能，图形设备的主显示层（primary graphics surface）通常是 RGB 颜色格式，叠加的内存通常是 YUV 颜色格式，硬件可以支持将两块显存合并然后输出显示。视频叠加设备用于输出，与常常用于摄像头的视频捕获设备的硬件完全不同。

叠加显示层和主显示层的关系如图 24-3 所示。

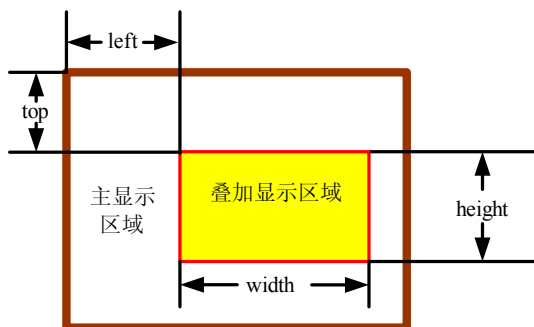


图 24-3 叠加显示层和主显示层的关系

V4L2 设备如果支持视频叠加设备，在调用 VIDIOC_QUERYCAP 查询能力的时候，应返回 V4L2_CAP_VIDEO_OVERLAY 能力。

VIDIOC_OVERLAY 是一个 ioctl 命令，用于使能和禁止视频叠加设备，如下所示：

```
#define VIDIOC_OVERLAY                _IOW('V', 14, int)
```

视频叠加设备和视频捕获设备的操作基本相同，但是调用 `VIDIOC_S_FMT` 设置的颜色格式不同，`v4l2_format` 当中的 `fmt` 共用体将使用 `v4l2_window` 成员。

`v4l2_window` 成员的内容如下所示：

```
struct v4l2_window {
    struct v4l2_rect    w;           // 表示区域的矩形
    enum v4l2_field     field;
    __u32               chromakey;   // 颜色键
    struct v4l2_clip    __user *clips; // 用于内存的剪裁处理
    __u32               clipcount;
    void               __user *bitmap; // 按位的掩码
    __u8               global_alpha;  // 全局的 Alpha 值
};
```

`v4l2_rect` 结构具有 `left`、`top`、`width` 和 `height` 这 4 个成员，表示的就是视频叠加区域在主显示区域的位置和大小。

一个视频叠加的驱动程序可以通过 `v4l2_window` 结构实现几个额外的特性，几个方面的语义如下所示：

- `chromakey` 成员（`__u32` 类型）为色度键，表示只有在主显示层（`primary graphics surface`）是这种颜色的时候，才会显示视频叠加层。
- `bitmap` 成员（`void*` 类型）的每一个位对应每一个像素，表示一个掩码，含义是只有这个掩码置位的时候，视频叠加层才会被显示。
- `clips` 成员（`v4l2_clip` 类型）和 `clipcount`（`__u32` 类型）成员，表示剪裁区域的数组，这个剪裁区域内的视频叠加层不会被显示。
- `global_alpha`（`__u8` 类型）表示视频叠加层全局的 `ALPHA` 数值，表示视频叠加层的透明度。

以上几个方面的特性，通常需要和视频叠加设备的硬件特性结合起来，根据硬件的特性有选择地实现或者不实现。

2. 视频输出设备

视频输出设备用于输出模拟的视频信号，输出的内容可以是静止的图像，也可以是图像序列。

`V4L2` 设备如果支持视频输出设备，在调用 `VIDIOC_QUERYCAP` 查询能力的时候，应返回 `V4L2_CAP_VIDEO_OUTPUT` 能力。

视频输出设备调用 `VIDIOC_S_FMT` 设置不同的颜色格式，`v4l2_format` 当中的 `fmt` 共用体将使用 `v4l2_pix_format` 成员。

视频输出设备数据流的操作可以通过实现 `write()` 调用或者 `mmap()` 调用完成。

提示：视频输出设备与视频叠加设备在性质上类似，但是格式与视频捕获设备相同。

3. 视频叠加输出设备

视频叠加输出设备用于输出一个 `FrameBuffer` 到视频输出设备上，作为视频叠加的输出。

此功能常常用于支持 OSD (on-screen display) 的电视设备,也就是用叠加的输出层表示出现在电视屏幕上的菜单。

V4L2 设备如果支持视频叠加输出设备,在调用 VIDIOC_QUERYCAP 查询能力的时候,应返回 V4L2_CAP_VIDEO_OUTPUT_OVERLAY 能力。

视频叠加输出设备相关的两个 ioctl 命令和相关结构如下所示:

```
struct v4l2_framebuffer {
    __u32          capability;
    __u32          flags;
    void           *base;      // 基地址
    struct v4l2_pix_format fmt; // 颜色格式
};
#define VIDIOC_G_FBUF _IOR('V', 10, struct v4l2_framebuffer) // 获得 FrameBuffer
#define VIDIOC_S_FBUF _IOW('V', 11, struct v4l2_framebuffer) // 设置 FrameBuffer
```

VIDIOC_G_FBUF 用于从驱动中获得 FrameBuffer, VIDIOC_S_FBUF 用于向驱动中设置 FrameBuffer, 二者的参数类型都是 v4l2_framebuffer。v4l2_framebuffer 的结构包括了 FrameBuffer 基地址和格式 v4l2_pix_format, v4l2_pix_format 当中的内容表示 FrameBuffer 的宽度、高度和颜色格式。

在用户空间操作的时候视频叠加输出设备通常和 FrameBuffer 设备结合起来使用,一段示例代码如下所示:

```
#include <linux/fb.h>
struct v4l2_framebuffer fbuf;
unsigned int i;
int fb_fd; // 表示 FrameBuffer 设备的文件描述符
if (-1 == ioctl (fd, VIDIOC_G_FBUF, &fbuf)) { // 操作 V4L2 设备
    // .....省略错误处理部分
}
for (i = 0; i < 30; ++i) { // 循环处理各个 FrameBuffer 设备
    char dev_name[16];
    struct fb_fix_screeninfo si;
    snprintf (dev_name, sizeof (dev_name), "/dev/fb%u", i);
    fb_fd = open (dev_name, O_RDWR); // 打开 FrameBuffer 设备
    if (-1 == fb_fd) {
        // .....省略错误处理部分
    }
    if (0 == ioctl (fb_fd, FBIOGET_FSCREENINFO, &si)) { // 设置 FrameBuffer
        if (si.smem_start == (unsigned long) fbuf.base) // 二者的地址相同
            break;
    } else {}
    close (fb_fd);
    fb_fd = -1;
}
}
```

此处实现的目的是从视频叠加输出设备中得到 FrameBuffer, 然后设置到 FrameBuffer 设备当中去。也就是查找到一个 FrameBuffer 设备作为 OSD, 其条件是二者的地址相同。

4. 其他功能

Video For Linux 有一些 ioctl 命令实际上与 Video 设备无关, 是专门为 Radio、Teletext、VBI 等设备使用的。

```

#define VIDIOC_G_INPUT      _IOR('V', 38, int)
#define VIDIOC_S_INPUT      _IOWR('V', 39, int)
#define VIDIOC_G_OUTPUT     _IOR('V', 46, int)
#define VIDIOC_S_OUTPUT     _IOWR('V', 47, int)
#define VIDIOC_ENUMOUTPUT   _IOWR('V', 48, struct v4l2_output)
#define VIDIOC_G_AUDOUT     _IOR('V', 49, struct v4l2_audioout)
#define VIDIOC_S_AUDOUT     _IOW('V', 50, struct v4l2_audioout)
#define VIDIOC_G_MODULATOR _IOWR('V', 54, struct v4l2_modulator)
#define VIDIOC_S_MODULATOR _IOW('V', 55, struct v4l2_modulator)
#define VIDIOC_G_FREQUENCY  _IOWR('V', 56, struct v4l2_frequency)
#define VIDIOC_S_FREQUENCY  _IOW('V', 57, struct v4l2_frequency)

```

以上几个方面的功能中，Radio（收音机）的使用较为广泛，如果 V4L2 设备支持收音机，在调用 VIDIOC_QUERYCAP 查询能力的时候，通常要返回 V4L2_CAP_RADIO 和 V4L2_CAP_TUNER 的组合。VIDIOC_S_MODULATOR、VIDIOC_S_FREQUENCY 等几个 ioctl 号用于 Radio 设备的控制。控制功能中也有几个是为 Radio 设备做的。

提示：当作为 Radio 设备使用的时候，V4L2 没有 Audio 数据通道的功能，只提供了 Radio 的控制接口。

24.2.4 Video for Linux 驱动接口

1. Video Buffer

Video Buffer 的接口是 Video For Linux 驱动所使用的视频内存，这些内存通常会以队列的形式存在。因此，视频的内存队列操作就是加入队列和取出队列。

在内核中，include/media/目录中的头文件 videobuf-core.h 定义了视频内存、视频内存的队列以及视频内存操作的相关内容。驱动程序可以使用这些类完成自己的实现。

表示视频内存的 videobuf_buffer 结构如下所示：

```

struct videobuf_buffer {
    unsigned int    i;                // 视频内存的索引
    u32             magic;
    unsigned int    width;            // 视频内存的宽
    unsigned int    height;           // 视频内存的高
    unsigned int    bytesperline;     // 每行的字节数，0 表示不使用
    unsigned long   size;             // 视频内存的尺寸
    unsigned int    input;
    enum v4l2_field field;
    enum videobuf_state state;        // 表示视频内存的状态
    struct list_head stream;          // 表示 QBUF 和 DQBUF 的链表
    struct list_head queue;           // 表示队列的链表
    wait_queue_head_t done;           // 表示内存是否处理完成
    unsigned int    field_count;
    struct timeval  ts;               // 表示视频的时间戳
    enum v4l2_memory memory;          // V4L2 的内存类型
    size_t          bsize;            // 内存的大小
    size_t          boff;             // 内存的偏移量
    unsigned long   baddr;            // 内存的地址
    struct videobuf_mapping *map;      // 用于使用 mmap 的内存

```

```

        int                privsize;           // 私有的指针及其大小
        void                *priv;
    };

```

videobuf_buffer 结构就是一个典型的内存，具有宽和高、内存类型等特性。它与 v4l2_buffer 不同的地方在于考虑了内存的队列特性，videobuf_state 类型的成员 state 表示这块内存的状态：VIDEObUF_NEEDS_INIT（需要初始化，0），VIDEObUF_PREPARED（准备好，1），VIDEObUF_QUEUED（加入队列，2），VIDEObUF_ACTIVE（活动中，3），VIDEObUF_DONE（完成，4），VIDEObUF_ERROR（错误，5），VIDEObUF_IDLE（空闲，6）。内存在队列中处理的时候，主要在 2、3、4 三个状态之中转化。

表示视频内存队列操作的 videobuf_queue_ops 结构如下所示：

```

struct videobuf_queue_ops {
    int (*buf_setup)(struct videobuf_queue *q,
        unsigned int *count, unsigned int *size);
    int (*buf_prepare)(struct videobuf_queue *q,
        struct videobuf_buffer *vb, enum v4l2_field field);
    void (*buf_queue)(struct videobuf_queue *q, struct videobuf_buffer *vb);
    void (*buf_release)(struct videobuf_queue *q, struct videobuf_buffer *vb);
};

```

videobuf_queue_ops 结构当中的 buf_queue 函数指针表示将内存加入队列，buf_release 表示释放内存。

videobuf_queue 结构如下所示：

```

struct videobuf_queue {
    struct mutex          vb_lock;
    spinlock_t            *irqlock;
    struct device          *dev;
    wait_queue_head_t     wait;           // 如果队列为空，则实现等待
    enum v4l2_buf_type     type;          // V4l2 的 Buffer 类型
    unsigned int           inputs;       // 用于 V4l2_BUF_FLAG_INPUT
    unsigned int           msize;
    enum v4l2_field        field;
    enum v4l2_field        last;         // 用于 V4l2_FIELD_ALTERNATE
    struct videobuf_buffer *bufs[VIDEO_MAX_FRAME]; // videobuf_buffer 指针数组
    const struct videobuf_queue_ops *ops;
    struct videobuf_qtype_ops *int_ops;
    unsigned int           streaming:1;
    unsigned int           reading:1;
    struct list_head       stream;       // 用于 mmap() + ioctl(QBUF/DQBUF)
    unsigned int           read_off;     // 表示视频流的链表，用于 read()
    struct videobuf_buffer *read_buf;
    void                   *priv_data;   // 私有的指针
};

```

videobuf_queue 表示视频内存的队列，它由若干个 videobuf_buffer 组成，其他成员用于控制这些内存的状态。

几个 videobuf_queue 的函数如下所示：

```

void *videobuf_queue_to_vaddr(struct videobuf_queue *q,
    struct videobuf_buffer *buf);
void videobuf_queue_core_init(struct videobuf_queue *q,

```

```

        const struct videobuf_queue_ops *ops, struct device *dev,
        spinlock_t *irqlock, enum v4l2_buf_type type,
        enum v4l2_field field, unsigned int msize,
        void *priv, struct videobuf_qtype_ops *int_ops);
int videobuf_queue_is_busy(struct videobuf_queue *q);
void videobuf_queue_cancel(struct videobuf_queue *q);
enum v4l2_field videobuf_next_field(struct videobuf_queue *q);

```

`videobuf_queue_core_init()` 函数用于初始化队列, 其他几个函数用于查询视频内存的状态和获取信息。

几个 `videobuf_queue` 的关键函数如下所示:

```

int videobuf_reqbufs(struct videobuf_queue *q,
                    struct v4l2_requestbuffers *req);
int videobuf_querybuf(struct videobuf_queue *q, struct v4l2_buffer *b);
int videobuf_qbuf(struct videobuf_queue *q, struct v4l2_buffer *b);
int videobuf_dqbuf(struct videobuf_queue *q, struct v4l2_buffer *b,
                  int nonblocking);
int videobuf_streamon(struct videobuf_queue *q);
int videobuf_streamoff(struct videobuf_queue *q);
void videobuf_stop(struct videobuf_queue *q);

```

以上几个函数的语义与 V4L2 接口的操作相同。其中, 只有 `videobuf_dqbuf()` 函数多了一个 `nonblocking` 参数。如果为非 0, 则表示使用不阻塞的操作方式。

`videobuf_queue` 有两种操作: 一种操作用于支持直接的 `read()` 调用, 另一种操作用于支持 `mmap()` 之后用 `ioctl` 进行 QBUF 和 DQBUF 操作。

用于支持 `read()` 调用的函数如下所示:

```

int videobuf_read_start(struct videobuf_queue *q);
void videobuf_read_stop(struct videobuf_queue *q);
ssize_t videobuf_read_stream(struct videobuf_queue *q,
                             char __user *data, size_t count, loff_t *ppos,
                             int vbihack, int nonblocking);
ssize_t videobuf_read_one(struct videobuf_queue *q,
                          char __user *data, size_t count, loff_t *ppos, int nonblocking);
unsigned int videobuf_poll_stream(struct file *file,
                                  struct videobuf_queue *q, poll_table *wait);

```

`videobuf_read_one()` 和 `videobuf_read_stream()` 函数用于帮助驱动程序实现 `read` 操作, 其中也需要考虑阻塞的问题, 因此具有 `nonblocking` 参数。`videobuf_poll_stream()` 函数用于配合实现 `poll` 操作。

用于支持 `mmap` 操作的函数如下所示:

```

int videobuf_mmap_setup(struct videobuf_queue *q,
                       unsigned int bcount, unsigned int bsize, enum v4l2_memory memory);
int videobuf_mmap_free(struct videobuf_queue *q);
int videobuf_mmap_mapper(struct videobuf_queue *q, struct vm_area_struct *vma);

```

以上几个函数用于帮助驱动程序实现 `mmap` 操作, `videobuf_mmap_mapper()` 是其中主要的实现函数。

2. Video for Linux 设备

v4l2-dev.h 头文件中定义的 video_device 是 V4L2 驱动程序实现者的核心结构，也就是用于定义一个具体 Video for Linux 驱动程序需要实现的内容。

video_device 结构如下所示：

```
struct video_device
{
    const struct v4l2_file_operations *fops;           // V4L 的文件操作
    struct device dev;                                // 表示 sys 文件系统当中的设备
    struct cdev *cdev;                                // 表示字符设备
    struct device *parent;                             // 父设备指针
    struct v4l2_device *v4l2_dev;                     // v4l2_device 的父链表
    char name[32];                                     // 设备的名称
    int vfl_type;                                     // 设备的操作
    int minor;                                         // 次设备号
    u16 num;                                           // 设备的数目
    // .....省略部分内容
    const struct v4l2_ioctl_ops *ioctl_ops;           // ioctl 的操作
};
```

具体的 Video for Linux 驱动程序实现 video_device 结构体，video_device 当中又以 v4l2_file_operations 结构最为重要。

v4l2_file_operations 表示 V4L2 的操作，如下所示：

```
struct v4l2_file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*ioctl) (struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                         unsigned long, unsigned long, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct file *);
    int (*release) (struct file *);
};
```

v4l2_file_operations 表示视频设备的操作，其中的函数指针实际上是 file_operations 的子集。Video for Linux 驱动要根据支持的情况，实现其中的一部分。

对于某个视频硬件，要实现 video_device 中的 v4l2_file_operation 结构体，还需要填充 video_device 结构的指针。在用户空间进行调用的时候，要调用到具体实现的各个函数指针。

v4l2-ioctl.h 文件定义了 v4l2_ioctl_ops 结构，如下所示：

```
struct v4l2_ioctl_ops {
    int (*vidioc_querycap) (struct file *file, void *fh, struct v4l2_capability *cap);
    int (*vidioc_g_priority) (struct file *file, void *fh, enum v4l2_priority *p);
    int (*vidioc_s_priority) (struct file *file, void *fh, enum v4l2_priority p);
    // 用于处理 VIDIOC_ENUM_FMT
    int (*vidioc_enum_fmt_vid_cap) (struct file *file, void *fh,
                                     struct v4l2_fmtdesc *f);
};
```

```

int (*vidioc_enum_fmt_vid_overlay) (struct file *file, void *fh,
                                     struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_vid_out)      (struct file *file, void *fh,
                                     struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_type_private) (struct file *file, void *fh,
                                     struct v4l2_fmtdesc *f);
// .....省略部分内容: 用于处理 VIDIOC_G_FMT
// .....省略部分内容: 用于处理 VIDIOC_S_FMT
// .....省略部分内容: 其他的函数类型
int (*vidioc_reqbufs) (struct file *file, void *fh,
                      struct v4l2_requestbuffers *b);
int (*vidioc_querybuf) (struct file *file, void *fh, struct v4l2_buffer *b);
int (*vidioc_qbuf)     (struct file *file, void *fh, struct v4l2_buffer *b);
int (*vidioc_dqbuf)    (struct file *file, void *fh, struct v4l2_buffer *b);
int (*vidioc_overlay)  (struct file *file, void *fh, unsigned int i);
// .....省略部分内容: 兼容 v4L1 的处理
int (*vidioc_g_fbuf)   (struct file *file, void *fh,
                      struct v4l2_framebuffer *a);
int (*vidioc_s_fbuf)   (struct file *file, void *fh,
                      struct v4l2_framebuffer *a);
int (*vidioc_streamon) (struct file *file, void *fh, enum v4l2_buf_type i);
int (*vidioc_streamoff) (struct file *file, void *fh, enum v4l2_buf_type i);
// .....省略部分内容: 其他的各种接口
long (*vidioc_default) (struct file *file, void *fh, // 私有的处理
                       int cmd, void *arg);
};

```

v4l2_ioctl_ops 结构当中的大部分函数指针都对应于 V4L2 驱动的 ioctl 接口命令。如果它们被实现了, 则直接作为一个接口来返回。

video_device 的类型、注册和注销函数如下所示:

```

#define VFL_TYPE_GRABBER 0
#define VFL_TYPE_VBI     1
#define VFL_TYPE_RADIO   2
#define VFL_TYPE_VTX     3
#define VFL_TYPE_MAX     4
int video_register_device(struct video_device *vfd, int type, int nr);
void video_unregister_device(struct video_device *vdev);

```

用 VFL_TYPE_开头的几个宏表示 V4L2 设备的类型, 一般使用 VFL_TYPE_GRABBER 表示视频设备。video_register_device() 函数中的 type 表示设备的类型, nr 表示设备的序号。

虽然 Video for Linux 驱动程序可以支持视频捕获设备, 也可以支持视频输出设备, 但是在实际的实现中, 捕获设备和输出设备的硬件结构相差很多。因此, 一个系统中的这两种驱动程序需要分别实现。

24.2.5 Video for Linux 驱动的实现层

Video for Linux 驱动的实现一般都以 video_device 为结构, 以自身系统的视频内存处理为关键。在实现的过程中, 考虑到硬件的特殊情况和相关点, 也有一些衍生的公共代码。

1. 内存中实现的 V4L2 设备

V4L2 的框架中有一个基于内存实现的驱动程序, 也被称为 memory-to-memory 设备,

它与具体的硬件无关，完全在内存中实现。

对于内存中的 V4L2 设备，include/media/目录中的 v4l2-mem2mem.h 文件是其头文件；v4l2-mem2mem.c 文件提供一些库函数，mem2mem_testdev.c 文件提供设备的注册接口。

mem2mem_testdev.c 是驱动程序的入口，定义名称为"m2m-testdev"的平台驱动。其探测函数 m2mtest_probe()如下所示：

```
static int m2mtest_probe(struct platform_device *pdev)
{
    struct m2mtest_dev *dev;           // 设备的内部结构
    struct video_device *vfd;          // 需要注册的结构
    int ret;
    dev = kzalloc(sizeof *dev, GFP_KERNEL);
    if (!dev) return -ENOMEM;
    spin_lock_init(&dev->irqlock);
    ret = v4l2_device_register(&pdev->dev, &dev->v4l2_dev); // 注册 v4l2 信息
    if (ret) goto free_dev;
    atomic_set(&dev->num_inst, 0);
    mutex_init(&dev->dev_mutex);
    vfd = video_device_alloc();         // 分配 video_device
    // .....省略错误处理
    *vfd = m2mtest_videodev;           // video_device 的实现
    ret = video_register_device(vfd, VFL_TYPE_GRABBER, 0); // 注册 video_device
    // .....省略错误处理
    video_set_drvdata(vfd, dev);        // 设置平台的私有数据
    snprintf(vfd->name, sizeof(vfd->name), "%s", m2mtest_videodev.name);
    dev->vfd = vfd;
    v4l2_info(&dev->v4l2_dev, MEM2MEM_TEST_MODULE_NAME
        "Device registered as /dev/video%d\n", vfd->num);
    setup_timer(&dev->timer, device_isr, (long)dev); // 启动一个 Timer
    platform_set_drvdata(pdev, dev);    // 设置平台的私有数据为 m2mtest_dev 类型
    dev->m2m_dev = v4l2_m2m_init(&m2m_ops); // 设置 m2m 驱动的结构
    // .....省略错误处理
    q_data[V4L2_M2M_SRC].fmt = &formats[0];
    q_data[V4L2_M2M_DST].fmt = &formats[0];
    return 0;
    // .....省略错误处理
}
```

m2mtest_probe()函数最主要的工作就是注册了 video_device 结构和启动了一个 Timer 模拟驱动的实现。调用 v4l2_device_register()函数、设置平台的私有数据为 m2mtest_dev 都是驱动程序的辅助工作。

v4l2-mem2mem.h 中定义的私有驱动操作结构 v4l2_m2m_ops 如下所示：

```
struct v4l2_m2m_ops {
    void (*device_run)(void *priv);    // (要求实现) 开始实际任务时调用
    int (*job_ready)(void *priv);      // (可选实现) 任务准备好
    void (*job_abort)(void *priv);     // (要求实现) 通知驱动中止目前的任务
};
```

v4l2-mem2mem.c 文件中定义的私有驱动结构 v4l2_m2m_dev 如下所示：

```
struct v4l2_m2m_dev {
    struct v4l2_m2m_ctx *curr_ctx;    // 表示 m2m 驱动的上下文
    struct list_head job_queue;        // 表示 m2m 驱动任务的队列
};
```

```
spinlock_t      job_spinlock;
struct v4l2_m2m_ops *m2m_ops;
};
```

v4l2_m2m_ops 是 memory-to-memory 视频设备内部使用的回调结构。

核心的结构 v4l2_file_operations 和 video_device 定义如下所示：

```
static const struct v4l2_file_operations m2mtest_fops = {
    .owner      = THIS_MODULE,
    .open       = m2mtest_open,
    .release    = m2mtest_release,
    .poll       = m2mtest_poll,
    .ioctl      = video_ioctl2,
    .mmap       = m2mtest_mmap,
};
static struct video_device m2mtest_videodev = {
    .name       = MEM2MEM_NAME,           // 名称为"m2m-testdev"
    .fops       = &m2mtest_fops,          // v4l2_file_operations 类型
    .ioctl_ops  = &m2mtest_ioctl_ops,     // v4l2_ioctl_ops 类型
    .minor      = -1,
    .release    = video_device_release,
};
```

v4l2_m2m_ops 是 memory-to-memory 的视频设备内部使用的回调结构，其中有几个函数指针由自己驱动所实现，ioctl 的实现者 video_ioctl2() 函数则是系统的默认实现。m2mtest_videodev 是 video_device 类型的。

v4l2-mem2mem.c 当中用于初始化的 v4l2_m2m_init() 函数如下所示：

```
struct v4l2_m2m_dev *v4l2_m2m_init(struct v4l2_m2m_ops *m2m_ops)
{
    struct v4l2_m2m_dev *m2m_dev;
    if (!m2m_ops) return ERR_PTR(-EINVAL);
    BUG_ON(!m2m_ops->device_run);
    BUG_ON(!m2m_ops->job_abort);
    m2m_dev = kzalloc(sizeof *m2m_dev, GFP_KERNEL); // 分配 v4l2_m2m_dev 结构
    if (!m2m_dev) return ERR_PTR(-ENOMEM);
    m2m_dev->curr_ctx = NULL;
    m2m_dev->m2m_ops = m2m_ops; // 设置 v4l2_m2m_ops 到 v4l2_m2m_dev 结构中
    INIT_LIST_HEAD(&m2m_dev->job_queue);
    spin_lock_init(&m2m_dev->job_spinlock);
    return m2m_dev; // 返回 v4l2_m2m_dev 结构
}
EXPORT_SYMBOL_GPL(v4l2_m2m_init);
```

v4l2_m2m_init() 函数的功能就是得到一个 v4l2_m2m_dev 结构。

mem2mem_testdev.c 当中的 m2mtest_open() 函数如下所示：

```
static int m2mtest_open(struct file *file)
{
    struct m2mtest_dev *dev = video_drvdata(file);
    struct m2mtest_ctx *ctx = NULL; // 私有数据的上下文就是 m2mtest_ctx
    ctx = kzalloc(sizeof *ctx, GFP_KERNEL); // 分配 m2mtest_ctx 结构
    if (!ctx) return -ENOMEM;
    file->private_data = ctx; // 设置私有数据的指针
    ctx->dev = dev; // m2mtest_ctx 当中包括 m2mtest_dev 成员
    ctx->translen = MEM2MEM_DEF_TRANSLEN; // 默认为 16MB
```

```

ctx->transtime = MEM2MEM_DEF_TRANSTIME; // 默认为 1000
ctx->num_processed = 0;
ctx->m2m_ctx = v4l2_m2m_ctx_init(ctx, dev->m2m_dev, queue_init);
// .....省略错误处理
atomic_inc(&dev->num_inst);
return 0;
}

```

m2mtest_open()函数就是 v4l2_file_operations 中 open 函数指针的实现，在视频驱动设备节点被打开的时候调用。打开文件的私有数据就是 m2mtest_ctx 类型，m2mtest_ctx 当中又包括了类型为 m2mtest_dev 的成员。

v4l2_ioctl_ops 结构类型为 v4l2_ioctl_ops，如下所示：

```

static const struct v4l2_ioctl_ops m2mtest_ioctl_ops = {
    .vidioc_querycap = vidioc_querycap,          // 对应VIDIOC_QUERYCAP
    .vidioc_enum_fmt_vid_cap = vidioc_enum_fmt_vid_cap, // 各种格式的操作
    .vidioc_g_fmt_vid_cap = vidioc_g_fmt_vid_cap,
    .vidioc_try_fmt_vid_cap = vidioc_try_fmt_vid_cap,
    .vidioc_s_fmt_vid_cap = vidioc_s_fmt_vid_cap,
    .vidioc_enum_fmt_vid_out = vidioc_enum_fmt_vid_out,
    .vidioc_g_fmt_vid_out = vidioc_g_fmt_vid_out,
    .vidioc_try_fmt_vid_out = vidioc_try_fmt_vid_out,
    .vidioc_s_fmt_vid_out = vidioc_s_fmt_vid_out,
    .vidioc_reqbufs = vidioc_reqbufs,           // 对应VIDIOC_REQBUFS
    .vidioc_querybuf = vidioc_querybuf,         // 对应VIDIOC_QUERYBUF
    .vidioc_qbuf = vidioc_qbuf,                 // 对应VIDIOC_QBUF
    .vidioc_dqbuf = vidioc_dqbuf,              // 对应VIDIOC_DQBUF
    .vidioc_streamon = vidioc_streamon,         // 对应VIDIOC_STREAMON
    .vidioc_streamoff = vidioc_streamoff,       // 对应VIDIOC_STREAMOFF
    .vidioc_queryctrl = vidioc_queryctrl,
    .vidioc_g_ctrl = vidioc_g_ctrl,             // 对应VIDIOC_G_CTRL
    .vidioc_s_ctrl = vidioc_s_ctrl,             // 对应VIDIOC_S_CTRL
};

```

m2mtest_ioctl_ops 是 m2mtest_videodev (video_device) 其中的一个成员。它实现的就是 ioctl 的各个命令。

其中实现查询的 vidioc_querycap()函数如下所示：

```

static int vidioc_querycap(struct file *file, void *priv,
                          struct v4l2_capability *cap)
{
    strncpy(cap->driver, MEM2MEM_NAME, sizeof(cap->driver) - 1);
    strncpy(cap->card, MEM2MEM_NAME, sizeof(cap->card) - 1);
    cap->bus_info[0] = 0;
    cap->version = KERNEL_VERSION(0, 1, 0);
    cap->capabilities = V4L2_CAP_VIDEO_CAPTURE | V4L2_CAP_VIDEO_OUTPUT
                      | V4L2_CAP_STREAMING;

    return 0;
}

```

memory-to-memory 设备同时具有视频的捕获和输出功能，再加上流式的功能，因此设置了 3 个能力。视频的捕获和输出功能共存情况在实际的硬件实现中基本是不可能的。

4 个用于流操作的核心函数的实现如下所示：

```

static int vidioc_qbuf(struct file *file, void *priv, struct v4l2_buffer *buf)

```

```

{
    struct m2mtest_ctx *ctx = priv;
    return v4l2_m2m_qbuf(file, ctx->m2m_ctx, buf);
}
static int vidioc_dqbuf(struct file *file, void *priv, struct v4l2_buffer *buf)
{
    struct m2mtest_ctx *ctx = priv;
    return v4l2_m2m_dqbuf(file, ctx->m2m_ctx, buf);
}
static int vidioc_streamon(struct file *file, void *priv, enum v4l2_buf_type type)
{
    struct m2mtest_ctx *ctx = priv;
    return v4l2_m2m_streamon(file, ctx->m2m_ctx, type);
}
static int vidioc_streamoff(struct file *file, void *priv, enum v4l2_buf_type type)
{
    struct m2mtest_ctx *ctx = priv;
    return v4l2_m2m_streamoff(file, ctx->m2m_ctx, type);
}

```

以上 4 个函数都是通过私有数据得到 `m2mtest_ctx` 类型的私有结构，然后调用 `v4l2-mem2mem.c` 文件中的公共函数实现功能。

`v4l2-mem2mem.c` 当中加入队列和从队列中取出的两个函数如下所示：

```

int v4l2_m2m_qbuf(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                 struct v4l2_buffer *buf)
{
    struct videobuf_queue *vq;
    int ret;
    vq = v4l2_m2m_get_vq(m2m_ctx, buf->type); // 从上下文中得到 videobuf_queue
    ret = videobuf_qbuf(vq, buf); // 调用核心的 Q
    if (!ret) v4l2_m2m_try_schedule(m2m_ctx);
    return ret;
}
EXPORT_SYMBOL_GPL(v4l2_m2m_qbuf);
int v4l2_m2m_dqbuf(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                 struct v4l2_buffer *buf)
{
    struct videobuf_queue *vq;
    vq = v4l2_m2m_get_vq(m2m_ctx, buf->type); // 从上下文中得到 videobuf_queue
    return videobuf_dqbuf(vq, buf, file->f_flags & O_NONBLOCK); // 调用核心的 DQ
}
EXPORT_SYMBOL_GPL(v4l2_m2m_dqbuf);

```

从上下文中得到 `videobuf_queue` 之后，下面的调用都是通过调用系统的 `videobuf_dqbuf()` 函数所实现的。

`v4l2-mem2mem.c` 当中流开关的两个函数如下所示：

```

int v4l2_m2m_streamon(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                     enum v4l2_buf_type type)
{
    struct videobuf_queue *vq;
    int ret;
    vq = v4l2_m2m_get_vq(m2m_ctx, type); // 从上下文中得到 videobuf_queue
    ret = videobuf_streamon(vq); // 调用 V4L2 核心的开始流
    if (!ret) v4l2_m2m_try_schedule(m2m_ctx); // 进入调度
    return ret;
}

```

```

}
EXPORT_SYMBOL_GPL(v4l2_m2m_streamon);
int v4l2_m2m_streamoff(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                      enum v4l2_buf_type type)
{
    struct videobuf_queue *vq;
    vq = v4l2_m2m_get_vq(m2m_ctx, type); // 从上下文中得到 videobuf_queue
    return videobuf_streamoff(vq);       // 调用 V4L2 核心的关闭流
}
EXPORT_SYMBOL_GPL(v4l2_m2m_streamoff);

```

除了调用的公共的驱动实现之外，当开始视频流的时候，关键的地方要调用驱动实现自己实现的 `v4l2_m2m_try_schedule()`，进行内存队列的调度。如果视频流没有开始，`v4l2_m2m_try_schedule()`函数当中又会调用 `v4l2_m2m_try_run()`函数。

`v4l2_m2m_try_run()`函数如下所示：

```

static void v4l2_m2m_try_run(struct v4l2_m2m_dev *m2m_dev)
{
    unsigned long flags;
    spin_lock_irqsave(&m2m_dev->job_spinlock, flags);
    if (NULL != m2m_dev->curr_ctx) { // 确定有另一个队列在运转
        spin_unlock_irqrestore(&m2m_dev->job_spinlock, flags);
        dprintk("Another instance is running, won't run now\n");
        return;
    }
    if (list_empty(&m2m_dev->job_queue)) { // 确定队列是空的
        spin_unlock_irqrestore(&m2m_dev->job_spinlock, flags);
        dprintk("No job pending\n");
        return;
    }
    m2m_dev->curr_ctx = list_entry(m2m_dev->job_queue.next, // 获得 v4l2_m2m_ctx
                                  struct v4l2_m2m_ctx, queue);
    m2m_dev->curr_ctx->job_flags |= TRANS_RUNNING;
    spin_unlock_irqrestore(&m2m_dev->job_spinlock, flags);
    m2m_dev->m2m_ops->device_run(m2m_dev->curr_ctx->priv); // 调用私有的运行结构
}

```

用于实现 `v4l2_m2m_ops` 的 `device_run()`函数又调用了 `device_process()`函数。`device_process()`函数的关键内容就是填充内存，模拟视频数据得到的情况。

`v4l2-mem2mem.c` 中的两个获得函数如下所示：

```

static struct v4l2_m2m_queue_ctx *get_queue_ctx(struct v4l2_m2m_ctx *m2m_ctx,
                                                enum v4l2_buf_type type)
{
    switch (type) {
        case V4L2_BUF_TYPE_VIDEO_CAPTURE: // 视频捕获的情况
            return &m2m_ctx->cap_q_ctx;
        case V4L2_BUF_TYPE_VIDEO_OUTPUT: // 视频输出的情况
            return &m2m_ctx->out_q_ctx;
        default:
            printk(KERN_ERR "Invalid buffer type\n");
            return NULL;
    }
}

struct videobuf_queue *v4l2_m2m_get_vq(struct v4l2_m2m_ctx *m2m_ctx,

```

```

        enum v4l2_buf_type type)
{
    struct v4l2_m2m_queue_ctx *q_ctx;
    q_ctx = get_queue_ctx(m2m_ctx, type);
    if (!q_ctx)        return NULL;
    return &q_ctx->q;
}

```

`get_queue_ctx()`函数用于获得上下文，根据捕获和输出的情况，在上下文中得到不同的内容。`v4l2_m2m_get_vq()`函数用于 `videobuf_queue` 类型的队列，这个队列是驱动在自己运行过程中组织的，此函数仅仅是得到它。

`mem2mem_testdev.c` 文件当中 `poll` 和 `mmap` 的两个调用实现函数如下所示：

```

static unsigned int m2mtest_poll(struct file *file,
                                struct poll_table_struct *wait)
{
    struct m2mtest_ctx *ctx = (struct m2mtest_ctx *)file->private_data;
    return v4l2_m2m_poll(file, ctx->m2m_ctx, wait);
}
static int m2mtest_mmap(struct file *file, struct vm_area_struct *vma)
{
    struct m2mtest_ctx *ctx = (struct m2mtest_ctx *)file->private_data;
    return v4l2_m2m_mmap(file, ctx->m2m_ctx, vma);
}

```

`v4l2-mem2mem.c` 中 `v4l2_m2m_poll()`函数的实现如下所示：

```

unsigned int v4l2_m2m_poll(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                          struct poll_table_struct *wait)
{
    struct videobuf_queue *src_q, *dst_q;
    struct videobuf_buffer *src_vb = NULL, *dst_vb = NULL;
    unsigned int rc = 0;
    src_q = v4l2_m2m_get_src_vq(m2m_ctx); // 从上下文中得到源队列
    dst_q = v4l2_m2m_get_dst_vq(m2m_ctx); // 从上下文中得到目标队列
    mutex_lock(&src_q->vb_lock);
    mutex_lock(&dst_q->vb_lock);
    if (src_q->streaming && !list_empty(&src_q->stream))
        src_vb = list_first_entry(&src_q->stream, struct videobuf_buffer, stream);
    if (dst_q->streaming && !list_empty(&dst_q->stream))
        dst_vb = list_first_entry(&dst_q->stream, struct videobuf_buffer, stream);
    // .....省略错误处理
    if (src_vb) {
        poll_wait(file, &src_vb->done, wait); // 由于源的等待
        if (src_vb->state == VIDEOBUF_DONE || src_vb->state == VIDEOBUF_ERROR)
            rc |= POLLOUT | POLLWRNORM;
    }
    if (dst_vb) {
        poll_wait(file, &dst_vb->done, wait); // 由于目标的等待
        if (dst_vb->state == VIDEOBUF_DONE || dst_vb->state == VIDEOBUF_ERROR)
            rc |= POLLIN | POLLRDNORM;
    }
    // .....省略错误处理
}
EXPORT_SYMBOL_GPL(v4l2_m2m_poll);

```

`v4l2_m2m_poll()`函数当中同时处理了由于源等待和由于目标等待的情况。

v4l2-mem2mem.c 中 v4l2_m2m_mmap() 函数的实现如下所示:

```
int v4l2_m2m_mmap(struct file *file, struct v4l2_m2m_ctx *m2m_ctx,
                  struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    struct videobuf_queue *vq;
    if (offset < DST_QUEUE_OFF_BASE) {
        vq = v4l2_m2m_get_src_vq(m2m_ctx);
    } else {
        vq = v4l2_m2m_get_dst_vq(m2m_ctx);
        vma->vm_pgoff -= (DST_QUEUE_OFF_BASE >> PAGE_SHIFT);
    }
    return videobuf_mmap_mapper(vq, vma);    // 调用核心的部分实现映射
}
EXPORT_SYMBOL(v4l2_m2m_mmap);
```

以上内容大部分是实现了 memory-to-memory 设备的结构,而真正让一个 V4L2 运转的,通常是一个中断。由于 memory-to-memory 设备基于软件,其中没有中断,因此替代的机制就是在探测 probe 阶段启动一个定时器。

定时器实现的函数 device_isr() 如下所示:

```
static void device_isr(unsigned long priv)
{
    struct m2mtest_dev *m2mtest_dev = (struct m2mtest_dev *)priv;
    struct m2mtest_ctx *curr_ctx;
    struct m2mtest_buffer *src_buf, *dst_buf;
    unsigned long flags;
    curr_ctx = v4l2_m2m_get_curr_priv(m2mtest_dev->m2m_dev);
    // ..... 省略部分内容
    src_buf = v4l2_m2m_src_buf_remove(curr_ctx->m2m_ctx); // 移除源中的上下文
    dst_buf = v4l2_m2m_dst_buf_remove(curr_ctx->m2m_ctx); // 移除目标中的上下文
    curr_ctx->num_processed++;
    if (curr_ctx->num_processed == curr_ctx->translen
        || curr_ctx->aborting) { // 表示已经完成或者中止
        dprintk(curr_ctx->dev, "Finishing transaction\n");
        curr_ctx->num_processed = 0; // 清零操作
        spin_lock_irqsave(&m2mtest_dev->irqlock, flags);
        src_buf->vb.state = dst_buf->vb.state = VIDEODONE;
        wake_up(&src_buf->vb.done); // 唤醒源队列中的完成队列
        wake_up(&dst_buf->vb.done); // 唤醒目标队列中的完成队列
        spin_unlock_irqrestore(&m2mtest_dev->irqlock, flags);
        v4l2_m2m_job_finish(m2mtest_dev->m2m_dev, curr_ctx->m2m_ctx);
    } else {
        spin_lock_irqsave(&m2mtest_dev->irqlock, flags);
        src_buf->vb.state = dst_buf->vb.state = VIDEODONE;
        wake_up(&src_buf->vb.done); // 唤醒源队列中的完成队列
        wake_up(&dst_buf->vb.done); // 唤醒目标队列中的完成队列
        spin_unlock_irqrestore(&m2mtest_dev->irqlock, flags);
        device_run(curr_ctx); // 再次让设备运行
    }
}
```

此处调用 wake_up() 唤醒的队列,刚好与 poll 操作当中的等待对应,含义就是如果处理没有完成,调用者就需要进行等待。

2. 硬件视频设备

硬件的视频照相机设备通常在硬件上包括摄像头传感器和视频处理单元两个部分，它们属于不同的硬件，因此在驱动程序当中二者通常分开使用。

内核源代码 `driver/media/video/` 目录中包括了很多 Video for Linux 驱动和相关部分的实现。几个主要内容如下：

- `soc_camera.c` 实现了一个通用的照相机 API，适用于直接连接于嵌入式处理器，使用 i2c 总线的摄像头系统，其中的 i2c 信息从板级配置中取出。
- `usbvideo` 和 `usbvision` 等子目录提供了 USB 视频设备的驱动。
- 多个摄像头传感器的实现文件：OmniVision（文件以 `ov` 开头）、Micron（文件以 `mt` 开头）、Philips（文件以 `SAA` 开头）、Texas Instruments（文件以 `tpv` 开头）。

`v4l2-int-device` 是一个典型的用于分开摄像头传感器和视频处理单元的结构，表示用于整合 Video for Linux 的设备。`v4l2-int-device.h` 头文件中定义了一个 `v4l2-int-device`，它作为具体设备和具体的摄像头传感器的中间层次。

Video for Linux 的核心部分和摄像头传感器控制部分都需要调用这个函数进行注册：Video for Linux 核心部分的注册类型为 `v4l2_int_type_master`（整型数 1），实现 `v4l2_int_master` 结构；摄像头传感器控制部分的注册类型为 `v4l2_int_type_slave`（整型数 2）。

基于 `v4l2-int` 实现的 Video for Linux 驱动程序如图 24-4 所示。

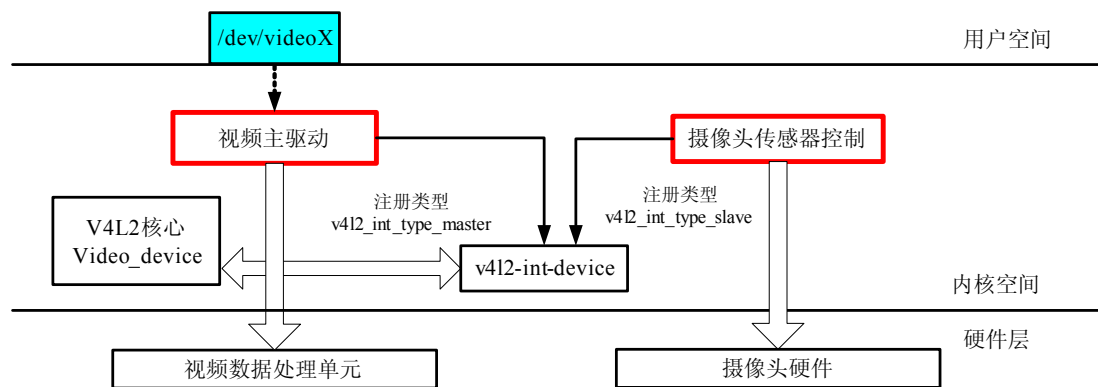


图 24-4 基于 `v4l2-int` 实现的 Video for Linux 驱动程序

`v4l2_int` 主设备 `v4l2_int_master` 的定义如下所示：

```
struct v4l2_int_master {
    int (*attach)(struct v4l2_int_device *slave);
    void (*detach)(struct v4l2_int_device *slave);
};
```

`v4l2_int` 控制类型的定义如下所示：

```
typedef int (v4l2_int_ioctl_func)(struct v4l2_int_device *);
typedef int (v4l2_int_ioctl_func_0)(struct v4l2_int_device *);
typedef int (v4l2_int_ioctl_func_1)(struct v4l2_int_device *, void *);
struct v4l2_int_ioctl_desc {
    int num; // 对应 v4l2_int_ioctl_num 类型
```

```
v4l2_int_ioctl_func *func;
};
```

`v4l2_int_ioctl_num` 是一个枚举类型，其中包括了 `vidioc_int_enum_fmt_cap_num`、`vidioc_int_queryctrl_num` 等数值，这些数值表示对 `v4l2_int` 从设备的控制命令。每一个 `v4l2_int_ioctl_desc` 就用于处理一个数值。

`v4l2_int` 从设备 `v4l2_int_slave` 的定义如下所示：

```
struct v4l2_int_slave {
    struct v4l2_int_device *master;
    char attach_to[V4L2_NAMESIZE];
    int num_ioctls;
    struct v4l2_int_ioctl_desc *ioctls;           // 用于控制的命令
};
```

从设备中包括一个 `v4l2_int_ioctl_desc` 类型的结构指针及其数目，其实就指向了一个由若干个处理命令组成的命令处理器数组。

`v4l2-int-device` 及其注册和注销函数如下所示：

```
struct v4l2_int_device {
    struct list_head head;
    struct module *module;
    char name[V4L2_NAMESIZE];
    enum v4l2_int_type type;           // 表示设备的类型
    union {
        struct v4l2_int_master *master; // 表示主设备
        struct v4l2_int_slave *slave;   // 表示从设备
    } u;
    void *priv;
};
void v4l2_int_device_try_attach_all(void);
int v4l2_int_device_register(struct v4l2_int_device *d);
void v4l2_int_device_unregister(struct v4l2_int_device *d);
```

`v4l2_int_device` 当中的 `u` 是一个共用体，它或者是一个主设备（`v4l2_int_master`），或者是一个从设备（`v4l2_int_slave`），`v4l2_int_device_register()` 函数将完成 `v4l2-int` 设备的注册，主设备和从设备注册的方式相同。

通过 `v4l2-int-device` 实现视频输入部分驱动的时候，相当于 `Video for Linux` 的核心部分调用摄像头传感器部分的函数。这种实现方式，方便了同一个嵌入式处理器使用不同摄像头传感器的情况——核心部分相同，只有摄像头传感器部分不同。