龙芯俱乐部

LOONGSON CLUB

一步步跟我学智龙

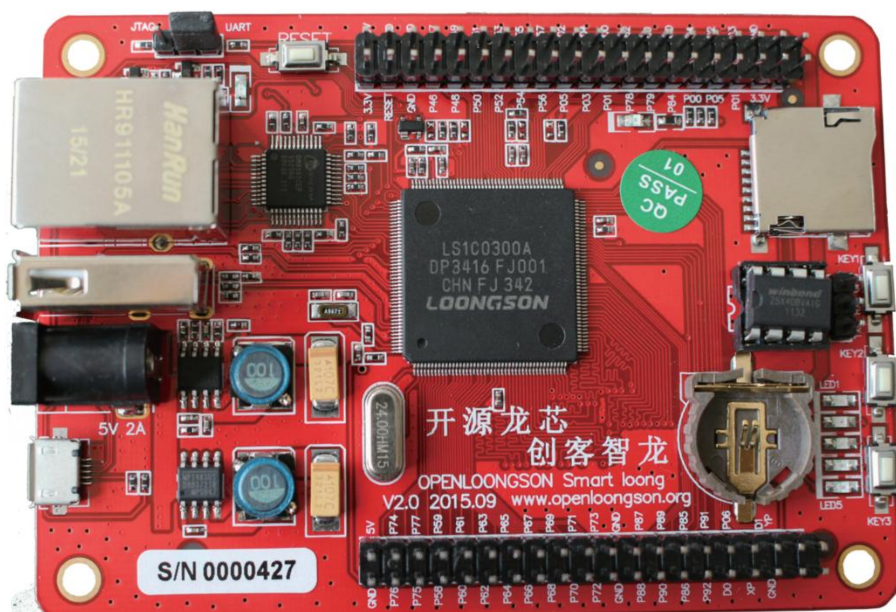
Linux 系统

龙芯智龙开源创客主板入门教程

版本号：V1.0.1

2017/5/31

孙冬梅



网址：www.loongsonclub.com
qq/手机/微信：13776573997
地址：南京市浦口区浦滨路 150 号中科创新广场

前言

本教程不仅用于 MIPS 架构的龙芯 1 号芯片的嵌入式系统开发，还可用于基于 Linux 操作系统的嵌入式系统，具有广泛的适用性。其中，在内核原理、应用程序编写方面，与市场常见 ARM 架构芯片相比，其开发过程是通用的，包括虚拟机编译、程序下载、调试、应用开发、内核裁减。

教程从开始编写到完工，经历了近 1 年半的时间。本教程最初不是为了出书，而是为了小范围内的学习交流及开发过程的记录。2017 年 4 月 25 日，有幸参加了龙芯新品发布会，也与龙芯生态圈建立了联系，感触很深。在了解了关于国产芯片的发展历史和现状后，觉得有必要将这一开发记录整理并参与更大范围的交流，以推进国产芯片的推广应用。

教程在撰写过程中，经历了各种困难。其中最大的困难在于恒心和毅力。因为本人不是企业的员工，没有将龙芯应用于产品的开发经历，没有现成的实例和成品的代码，更没有开发产品的压力。好在拥有最重要一点，对于嵌入式系统浓厚的兴趣。再加上对于操作系统内核有一定基础，对于全部的外设接口都能熟练操作，虽然以前用的是其它系统和其它芯片平台。尽管基础很好，但面对一个新的芯片、新的系统，还是花了一些心思。记得在调试串口的时候，由于粗心没有看硬件图，导致一个小问题连续花了一周的时间才解决，解决的过程曲折且有意思、并且有经验教训。我想，正是经历这无数多的问题，并且不放弃，孜孜不倦的寻找答案，才能积累到现在一眼就发现问题的能力。这期间，经常是一段不到 100 行的代码，查找相关起源、结果，就要花 1 周的时间。所以，通过看这个教程来学习操作系统，不仅是要进行相关验证性实验，更是要进行创造性开发，才能巩固提高。因为实现一个功能的测试，在代码正确的情况下，验证过程只有 30 分钟就能完成。可是设计开发者实现这个功能，可能写了 1 个月之久。所以不要羡慕资深开发者的火眼金睛和开发速度，那是用时间和精力换来的，在嵌入式系统这一行，没有站在巨人的肩膀上这一说法，只有沿着巨人走过的路再走一遍，才能达到巨人的水平。

本教程中的所有代码，部分来自于网络，部分自己撰写，但都全部调试并在智龙 V2.0 上运行通过。其中，部分应用程序与其它 ARM 架构的系统是通用的。教程由三部分组成：入门基础篇、中级编程篇、高级驱动篇。入门基础篇包含了从一个小白过渡到系统程序员的基础内容；中级编程篇不仅包含了操作系统的基本操作：文件、进程、线程、管道、消息、内存、锁、信号、网络，还包含了基本的硬件接口操作：GPIO、按键、RTC、UART。高级驱动篇则包含了驱动程序的编写和高级硬件接口操作：字符设备、I2C 总线、SPI 总线、CAN 总线、ADC、LCD。教程中所有的代码均开源在 GITHUB 和百度网盘上。

整篇内容由浅入深，方便自行阅读，也可以跟着配套视频自行学习。

GITHUB：https://github.com/sundm75/Loongson-Smartloong-V2.0

孙冬梅

2017.05.31

于南京工业大学电控学院测控教研室

目录

前言......I
目录......II
基础篇入门......1
1．背景知识......1
1.1 为何要如此麻烦地安装 ubuntu/redhat?......1
1.2 装完 linux 后想要编译和下载程序还要做些什么？......1
1.3 虚拟机常用目录......1
1.4 linux 下编程还需要安装的软件......2
1.4.1 TFTP......2
1.4.2 PuTTY......2
1.5 一点常识......4
1.6 本教程使用的智龙开发板......5
2．虚拟机安装 linux 操作系统......7
2.1 下载 vmware player 并安装......7
2.2 下载 ubuntu 桌面系统......7
2.3 安装 ubuntu......7
2.4 进入终端或者命令行方式......9
2.5 建立 root 用户并自动登录......10
2.6 安装 vmwaretools......10
2.7 安装必要的软件......12
2.8 查看相关版本、信息......12
3．安装工具链、编译内核、制作文件系统......14
3.1 安装交叉编译工具 gcc-4.3-ls232......14
3.2 编译和烧写 pmon......14
3.3 编译和烧写内核......15
3.4 制件根文件系统......17
3.4.1 配置和编译 busybox......17
3.4.2 创建文件系统目录......18
3.4.3 创建系统配置文件......18
3.4.4 拷贝库文件......20
3.5 制作根文件系统镜像......21
3.5.1 安装镜像文件制作工具......21
3.5.2 制作根文件系统镜像文件......23
3.5.2.1 cramfs 文件系统......23
3.5.2.2 yaffs2 文件系统......23
3.5.2.3 ubifs 文件系统......23
3.5.3 烧写根文件系统......24
4．使用 buildroot 构建根文件系统......25
4.1 获取 buildroot......25
4.2 系统构建......25
4.3 烧写根文件系统镜象......25

4.4 根文件系统软件包的定制......25
5．简单应用编程 Helloworld......27
6．简单驱动程序编写......28
6.1 驱动的原理及编写流程......28
6.2 驱动模块的加载与卸载......28
6.3 最简单的 Linux 驱动......28
6.4 驱动的编译和执行......29
6.5 内核配置驱动......32
6.6 led 子系统剖析......34
6.7 led_trigger 接口分析......41
中级篇应用......43
7．Linux 应用编程......43
7.1 linux 应用编程的基础知识......43
7.2 文件 I/O 编程......43
7.3 进程和线程......45
7.4 多进程操作......46
7.5 进程间的通信......47
7.5.1 管道......48
7.5.2 消息队列......52
7.5.3 共享内存......53
7.6 多线程操作......55
7.6.1 线程控制......55
7.6.2 线程属性......56
7.6.3 互斥锁......58
7.6.4 信号量......60
7.7 网络编程......62
7.7.1 网络编程基础概念......62
7.7.2 网络编程实例......65
7.7.3 网络编程小结......73
7.8 应用编程总结......73
8．开发板硬件接口编程......74
8.1 点亮一个 LED 灯......74
8.1.1 LED 的操作接口......74
8.1.2 LED 控制......74
8.1.3 在程序中操作 LED 灯......74
8.3 GPIO 硬件编程......75
8.3.1 GPIO 和 sysfs 操作接口......76
8.3.2 GPIO 基本操作......77
8.2.3 在 C 程序中操作 GPIO......77
8.3 按键应用层编程......79
8.3.1 按键操作接口......79
8.3.2 在程序中操作按键......82
8.4 SD 卡和 U 盘......83
8.4.1 U 盘......83

8.4.2 SD 卡 84
8.5 RTC 时钟 84
8.6 串口读写..85
8.6.1 串口硬件说明..85
8.6.2 用 minicom 操作串口..85
8.6.3 用接口操作串口..87
8.6.4 在程序中操作串口..87
9．NFS 文件系统搭建 90
9.1 在虚拟机端安装 NFS 90
9.2 配置虚拟机 NFS 90
9.3 配置单板机 NFS 91
9.4 使用 NFS 91
1）在单板机上挂载 nfs 服务 91
2）建立网络文件系统..92
高级篇驱动..93
10．配置 Eclipse 编程..93
10.1 用 eclipse 开发应用程序..93
10.2 用 eclipse 开发 内核模块..96
11. 一个简单的字符设备驱动..104
11.1 主设备号和次设备号...104
1)设备编号的表达...104
2)分配和释放设备编号...105
11.2 重要的数据结构...106
1)文件操作 file_operations...106
2)文件结构 struct file 107
3)inode 结构 107
11.3 字符设备的注册..109
11.4 具体实例...109
1) file_operations 结构体设计 112
2) 模块初始化、模块卸载函数实现...112
3) 读写函数的实现...113
4) 驱动程序编译..113
5) 驱动程序编译和加载...113
6) 驱动程序测试..114
11.5 一些有用的资料..115
11.6 例子修改成模块注销自动删除设备节点..115
12. misc 杂项设备驱动 117
12.1misc 使用的结构体和函数 117
12.2 为什么要有 misc 设备 118
12.3 内核源码..119
1) miscdevice 结构体...119
2) misc_register 函数...119
3）misc_deregister 函数...120
12.4 具体实例..121

13. PWM 控制输出............124
13.1 利用 LED_PWM............124
13.2 自己编写驱动文件............131
14. I2C 驱动............139
14.1 Linux I2C 设备驱动编写............139
14.1.1 I2C adapter............140
14.1.1.1 SMBus 与 I2C 的区别............140
14.1.1.2 I2C driver............141
14.1.1.3 I2C client............142
14.1.2 I2C 子系统驱动模块的 API............143
14.1.3 I2C client 的注册............144
14.1.3.1 使用总线号声明设备 i2c_register_board_info............145
14.1.3.2 枚举设备 i2c_new_device 或者 i2c_new_probed_device............146
14.1.3.3 从用户空间初始化 I2C 设备............148
14.1.4 I2C driver............149
14.1.4.1 I2C 设备驱动............149
14.1.4.2 关于 I2C 设备驱动的小总结............151
14.1.5 I2C adapter 的注册............151
14.1.5.1 注册方法............151
14.1.5.2 使用场景............151
14.1.5.3 物理 i2c 总线的编号查询............153
14.1.6 I2C tools 使用............154
14.1.6.1 下载安装............154
14.1.6.2 I2C 总线扫描............154
14.1.6.3 I2C 设备查询............155
14.1.6.4 寄存器内容导出............155
14.1.6.5 寄存器内容写入............156
14.1.6.6 寄存器内容读出............156
14.1.7 内核模块分析............156
14.2 实例分析 at24cxx............158
14.2.1 注册新设备............159
14.2.2 注册新驱动............159
14.2.3 对 i2c 驱动的操作............159
14.2.4 编译用的 Makefile............159
14.2.5 测试应用编程 test_at24cxx.c............159
14.3 实例分析 DS3231............160
15. SPI 总线和设备驱动架构............162
15.1 SPI 概述............162
15.1.1 硬件结构............162
15.1.1.1 工作时序............162
15.1.2 软件架构............163
5.1.2.1 SPI 控制器驱动程序............163
5.1.2.2 SPI 通用接口封装层............164
5.1.2.3 SPI 协议驱动程序............164

5.1.2.4 SPI 通用设备驱动程序......164
15.2 SPI 通用接口层......164
15.2.1 SPI 设备模型的初始化......164
15.2.2 spi_master 结构......165
15.2.3 spi_device 结构......166
15.2.4 spi_driver 结构......167
15.2.5 spi_message 和 spi_transfer 结构......168
15.3 SPI 控制器驱动......170
15.3.1 定义控制器设备......170
15.3.2 注册 SPI 控制器的 platform_driver......171
15.3.3 注册 spi_master......172
15.3.4 实现 spi_master 结构的回调函数......173
15.4 SPI 数据传输的队列化......173
15.4.1 spi_transfer 的队列化......174
15.4.2 spi_message 的队列化......174
15.4.3 队列以及工作线程的初始化......175
15.4.4 队列化的工作机制及过程......176
15.5 实例分析-驱动编写之 SPI 设备静态注册 spidev.c......178
15.6 实例分析-驱动编写之 SPI 设备动态注册 spike.c......181
15.7 编写测试程序......186
16. CAN 总线驱动开发......191
16.1 智龙开发板硬件 CAN 接口......191
16.2 SocketCAN......193
16.2.1 概述--什么是 Socket CAN？......193
16.2.2 动机--为什么使用 socket API 接口？......193
16.2.3 Socket CAN 详解......194
16.2.3.1 接收队列......194
16.2.3.2 发送帧的本地回环......195
16.2.3.3 网络安全相关......195
16.2.3.4 网络故障监测......196
16.2.4 如何使用 Socket CAN......196
16.2.4.1 使用 can_filter 的原始套接字 (RAW socket)......198
16.2.4.1.1 原始套接字选项 CAN_RAW_FILTER......198
16.2.4.1.2 原始套接字选项 CAN_RAW_ERR_FILTER......199
16.2.4.1.3 原始套接字选项 CAN_RAW_LOOPBACK......200
16.2.4.1.4 原始套接字选项 CAN_RAW_RECV_OWN_MSGS......200
16.2.4.2 广播管理协议套接字 (SOCK_DGRAM)......200
16.2.4.3 面向连接的传输协议 (SOCK_SEQPACKET)......200
16.2.4.4 无连接的传输协议 (SOCK_DGRAM)......200
16.2.5 Socket CAN 核心模块......200
16.2.5.1 can.ko 模块的参数......201
16.2.5.2 procfs 接口......201
16.2.5.3 写一个自己的 CAN 协议模块......201
16.2.6 CAN 网络驱动......202

16.2.6.1 常见设置......202
16.2.6.2 发送帧的本地回环......202
16.2.6.3 CAN 控制器的硬件过滤......202
16.2.6.4 虚拟的 CAN 驱动 (vcan)......203
16.2.6.5 CAN 网络设备驱动接口......203
16.2.6.5.1 Netlink 接口--设置/获取设备属性......203
16.2.6.5.2 设置 CAN 的比特_时序......205
16.2.6.5.3 启动和停止 CAN 网络设备......205
16.2.6.6 支持 Socket CAN 的硬件......206
16.2.7 学习 Socket CAN 的相关资源......206
16.2.8. 贡献者名单......206
16.3 测试工具......207
16.4 Socket CAN 在智龙上的测试-使用工具 iproute2......207
16.4.1 下载编译......207
16.4.2 配置运行命令......210
16.5 Socket CAN 在智龙上的测试-使用工具 canutils......210
16.5.1 安装 libsocketcan......210
16.5.2 安装 canutils......211
16.5.3 使用 canutils......213
16.6 编写 CAN 的 socket 收发测试程序 canapp......214
16.6.1 程序设计说明......214
16.6.2 程序发送示例......215
16.6.3 程序接收示例......216
16.6.4 发送接收测试......217
17. LCD 应用开发......219
17.1 硬件接口......219
17.2 GPIO 口操作函数......219
17.3 LCD 操作......225
17.4 编写 Makefile......227
17.5 代码及运行结果......228
17.5.1 显示字符......228
17.5.2 显示图片......228
17.5.3 显示汉字......229
17.5.3 画线、圆圈......232
18. ADC 应用开发......236
18.1 配置 ADC 驱动......236
18.2 硬件管脚分配......236
18.3 应用测试......237
18.4 应用层编程......237
19. 内核访问外设 I/O 资源......239
19.1 MIPS 的内存映射......239
19.2 动态映射(ioremap)方式......240
19.3 静态映射(map_desc)方式......245
19.4 mmap 系统调用......247

19.4.1mmap 系统调用 247
19.4.2 系统调用 mmap()用于共享内存的两种方式 249
19.4.3 mmap 进行内存映射的原理 249
19.4.4 内存映射的步骤: 253
19.5 mmap 编程示例 253
附录 1 一天一个 linux 命令 255
1) 解压 255
2) 用 cat 命令将字符串追加到文件中 256
3) echo 256
4) 常用 cat 命令 257
5) tree 显示目录结构 258
6) 防火墙 259
7) 内核启动参数 259
8) printk 内核打印 261
9) linux 下显示隐藏文件 264
10) linux cp 说明 264
11) subsys_initcall 说明 270
12) rm 删除目录 274
13) tar 打包 274
附录 2 VIM 图例及常用操作 275
附录 3 busybox 下载及配置 277
附录 3 PMON 常用命令 283
附录 4 module_init 和 module_exit 分析 285
附录 5.创建一个与你的驱动程序对应的设备节点 287
附录 6 linux kernel 从入口到 start_kernel 的代码分析 296
附录 7 linux 字符设备驱动总结之：全自动创建设备及节点 317
附录 8 LINUX 下的文件结构介绍 322
附录 9.无线网卡（RTL8188EU）驱动编译、使用 DHCP 配置无线网络 334
1 驱动编译进入内核 334
2 下载 wifitools.tar.gz 使用 336
3 建立连接 337
4 使用 wpa_supplicant 连接无线网络 338
5 安装 wpa_supplicant 339
6 继续使用 wpa_supplicant 连接无线网络 340
7 配置 DHCP 347
8 DHCP 配置成开机启动 349
附录 10. 错误集锦 349
1. intel_rapl 错误 349
2. rc-local.service 349
附录 11 git 命令 350
附录 12 在 pmon 中使命令 devcp 可以进行坏块处理和支持 yaffs2 烧写 353
附录 13 QT 安装使用 354
附录 14 GDB 使用 354
附录 15 makefile 经典教程 359

0 Makefile 很重要..359
1 Makefile 介绍..360
2 Makefile 总述..367
3 Makefile 书写规则..369
4 Makefile 书写命令..377
5 使用变量..382
6 使用函数..411
7 make 的运行..434
8 隐含规则..450
9 使用 make 更新函数库文件..478
10 后序...482

基础篇入门

1. 背景知识

本章针对编程人员进行嵌入式系统的入门基本培训，实现相同功能的方法不仅限于所介绍的，着重于入门开发环境搭建、工具使用、各种必须的过程等的介绍，对于编程本身不作深入研究。

文中所有代码均在智龙 V2.0 开发板上运行实践验证，所有截图均为实例运行结果，没有虚假及盗用。所有引用他人的博客或者文章，均标注有出处，并在智龙上修改运行后整理后发布。

1.1 为何要如此麻烦地安装 ubuntu/redhat?

习惯了在集成的 IDE（集成编译环境）中，编辑、修改、编译、下载，不知道 CPU 里跑的程序与实际编写的代码之间的关系。那么现在就要从 CPU 结构起，慢慢地熟悉构造、原理。

单板机里的跑的操作系统，就是一个框架，加上了对应的驱动才能操作线路板。操作系统+驱动构成了全部的源码；源码要经过编译后变成机器码，才能在芯片当中运行。源码到机器码的过程就是编译，不同架构的芯片要使用不同编译器。51 架构的用 C51，ContexM3 和 M4 的用 ARM 编译器，这两种都在 IAR 或者 MDK 或者 Keil 中集成，且在 windows 中运行。

那么在 ARM 或者 MIPS 架构芯片中运行的基于 linux 操作系统程序，必须在 linux 系统下用相关的编译器进行编译。在 Windows 下进行编译几乎是不可能的，就算能，那在 linux 下也不能用所以就有了装 linux 系统的必要。如果在 windows 下进行 linux 操作系统的操作，只能用虚拟机，Linux 内核使用大量的 GCC 拓展，而且整个工程是用 makefile 来控制，在 Windows 下虽然有对应的 GCC 工具和 make 工具，但配置起来都比较麻烦！既然选择学习 linux 内核，那么 linux 下 C 语言编程和操作系统应该都有一定的基础 而且现在桌面版的 linux 系统已经很人性化，像 fedora Ubuntu 都很好用了，不需要繁杂的配置！本文介绍利用虚拟机安装 ubuntu 的操作系统，及其后续的相关工作，并介绍过程及方法。

1.2 装完 linux 后想要编译和下载程序还要做些什么？

必要程序的安装，包括：

1）安装交叉编译工具 。

2）编译生成 BIOS/uboot/pmon、内核。

3）使用 buildroot 或者 busybox 构建根文件系统。

4）编写应用程序。

5）编写驱动程序。

1.3 虚拟机常用目录

以下为本文使用的虚拟机中的目录：

内核源码的目录：/Workstation/tools/kernel/linux-3.0.82-openloongson 。

交叉编译工具链目录：/opt/gcc-4.3-ls232。

制作的根文件系统目录：/Workstation/tools/makefs/rootfs。

桌面机系统使用的 IP：193.169.2.215，用于 tftp 传送文件。

1.4 linux 下编程还需要安装的软件

1.4.1 TFTP

Tftpd32 是一个集成 DHCP, TFTP, SNTP 和 Syslog 多种服务的袖珍网络服务器包，同时提供 TFTP 客户端应用，tsize, blocksize 和 timeout 支持等等。这里主要用于文件的传递。

TFTP 应用于主机（win 操作系统）：

把 tftpd32 下载到机器上，双击 tftpd32.exe，就出现如图的配置界面。

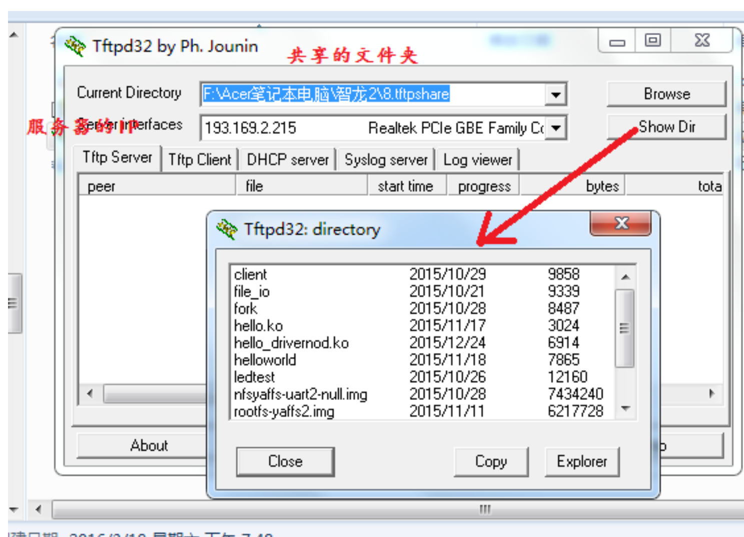


图 1.1 Windows 下配置 TFTP 过程示意图

配置好 tftp 后，紧接着要配置控制台软件 putty。

TFTP 还可在虚拟机（Linux 操作系统）中使用，网络上很多方法，具体方法这里不再叙述。

1.4.2 PuTTY

PuTTY 是一个 Telnet、SSH、rlogin、纯 TCP 以及串行接口连接软件。较早的版本仅支持 Windows 平台，在最近的版本中开始支持各类 Unix 平台，并打算移植至 Mac OS X 上。除了官方版本外，有许多第三方的团体或个人将 PuTTY 移植到其他平台上，像是以 Symbian 为基础的移动电话。PuTTY 为一开放源代码软件，主要由 Simon Tatham 维护，使用 MIT licence 授权。随着 Linux 在服务器端应用的普及，Linux 系统管理越来越依赖于远程。在各种远程登录工具中，Putty 是出色的工具之一。Putty 是一个免费的、Windows 32 平台下的 telnet、rlogin 和 ssh 客户端，但是功能丝毫不逊色于商业的 telnet 类工具。目前最新的版本为 0.63。

把 Putty 下载到机器（主机 win 操作系统）上，双击 putty.exe，就出现如图的配置界面：

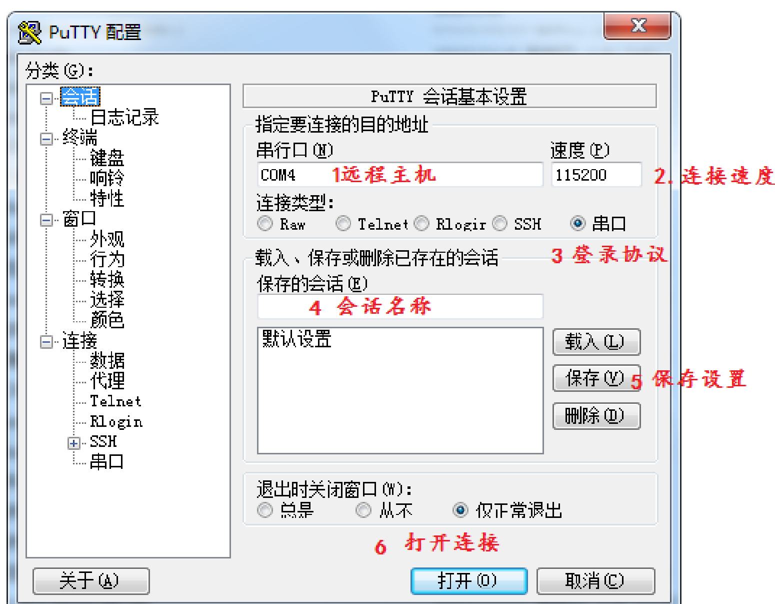


图 1.2 Windows 下配置 PuTTY 过程示意图

配置完成后，可打开如下的界面。

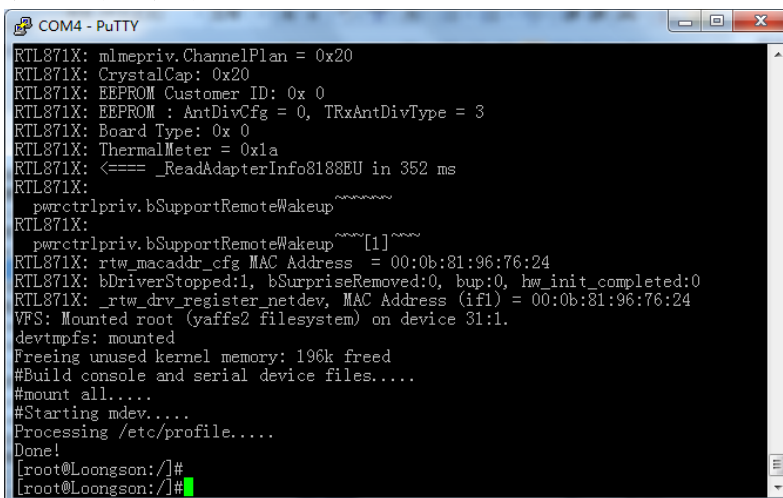


图 1.3 控制台界面

通过 Putty 可进行 PMON 配置和控制台操作 for OPENLOONGSON Smart Loong V2。

进入 pmon 方法：加电后按空格键。

PMON 中配置 IP 命令行代码：

```
ifaddr syn0 192.168.*.*     //IP 地址起临时作用，断电后无效
set ifconfig syn0:192.168.*.*    //重启后，IP 地址固定存在
```

pmon 下载内核操作命令行代码：

```
mtd_erase /dev/mtd0                                          //擦除内核数据
devcp tftp://193.169.2.215/vmlinuz /dev/mtd0                 //下载内核
set al /dev/mtd0                    //启动参数，自动从 nandflash 的 mtd0 分区 load
内存，设置板在一上电时自动执行 load 内核到内存操作
mtd_erase /dev/mtd1                                          //擦除根文件数据
devcp tftp://193.169.2.215/rootfs-yaffs2.img /dev/mtd1 yaf nw    //烧写文件系统 rootfs-yaffs2.img
set append " root=/dev/mtdblock1"                         //根目录位置，块设备
set append " $append console=ttyS2,115200"                //设置串口 3，115200 波特率
set append " $append  noinitrd init=/linuxrc rw rootfstype=yaffs2"     //noinitrd 代表没有使用 ramdisk；
init=/linuxrc 是指内核启动起来后进入系统中运行的第一个脚本，挂载之后文件系统是只读的，所以就加了
个 rw；rootfstype=yaffs2 指明文件系统类型为 yaffs2 不然没法挂载根分区
set append " $append  video=ls1bfb:480x272-16@60 fbcon=rotate:1 consoleblank=0"     //fbcon=rotate:1 标示
```

```
屏幕可旋转；consoleblank=0 禁用屏幕白色待机
PMON>reboot
```

Pmon 中运行 tftp 远程系统（如 RTT）命令行代码：

```
PMON>setal tftp://192.168.3.10/rtthread.elf     //远程系统 的 IP 为 192.168.3.10
PMON>reboot
```

下载并运行一个程序命令行代码：

```
ifconfig eth0 up                          //在文件系统里启动网口
ifconfig eth0 192.168.1.244               //文件系统里配置网络 ip(ip 不要和主机一样)
tftp -r ledtest -g 193.169.2.215          //下载编译好的 LED 文件
chmod u+x ledtest                         //给予权限
./ledtest                                 //运行 LED 文件
nohup /ledtest &                          //在后台不间断运行 LED 文件
```

几个不常用的命令行代码：

```
jobs                                      //查看
fg %n                                     //后 ctrl+c 关闭
udhcpc -b -i eth0 -p /var/run/udhcpc.pid -R      //自动获取 IP 几秒之后获取不到 进入后台继续获取
ls –lih                                   //显示文件的详细资料
```

1.5 一点常识

当开发板从贴片厂下线，里面是没有任何程序的，这时一般通过 JTAG 接口烧写第一个程序，就是 pmon，借助 pmon 可以使用网口或者 SD 卡下载更加复杂的系统程序等，这在后面的章节中你可以看到。除此之外，JTAG 接口在开发中最常见的用途是单步调试，不管是市面上常见的 JLINK 还是 ULINK，以及其他的仿真调试器，最终都是通过 JTAG 接口连接的。标准的 JTAG 接口是 4 线： TMS、 TCK、 TDI、 TDO，分别为模式选择、时钟、数据输入和数据输出线，加上电源和地，一般总共 6 条线就够了；为了方便调试，大部分仿真器还提供了一个复位信号。因此，标准的 JTAG 接口是指是否具有上面所说的 JTAG 信号线，并不是 20Pin 或者 10Pin 等这些形式上的定义表现。这就如同 USB 接口，可以是方的，也可以扁的，还可以是其他形式的，只要这些接口中包含了完整的 JTAG 信号线，都可以称为标准的 JTAG 接口。本智龙开发板提供了包含完整 JTAG 标准信号的 4 Pin JTAG 接口。说明一下，对于打算致力于 Linux 或者 RTThread 开发的初学者而言， JTAG 接口基本是没有任何意义和用途的，因为大部分开发板都已经提供了完善的 BSP，这包括最常用的串口和网络以及 USB 通讯口，当系统装载了可以运行的 Linux 或者 RTThread 系统，用户完全可以通过这些高级操作系统本身所具备的功能进行各种调试，这时是不需要 JTAG 接口的；即使可以进行跟踪，但鉴于操作系统本身结构复杂，接口繁多，单步调试犹如大海捞针，毫无意义。

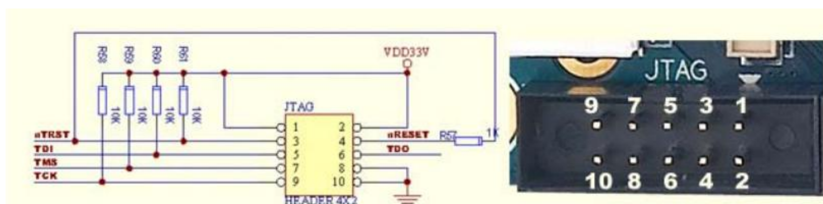


图 1.4 JTAG 接口示意图

想一想手头使用的 PC 机就知道了，或许从没有见过甚至听过有谁会在 PC 主板上插一个仿真器，来调试 PCI 这样接口的 Windows7 或者 Linux 驱动。这就是为什么你经常见到或者听到那么多人在讲驱动“移植”，因为大部分人都是参考前辈的实现来做驱动的。JTAG 仅对那些不打算采用操作系统，或者采用简易操作系统(例如 uCosII 等)的用户有用。大部分开发板所提供的 Bootloader 或者 PMON 已经是一个基本完好的系统了，因此也不需要

单步调试。

1.6 本教程使用的智龙开发板

智龙开发板 V2.0 V1.0 及其差异如图 1.5 所示。

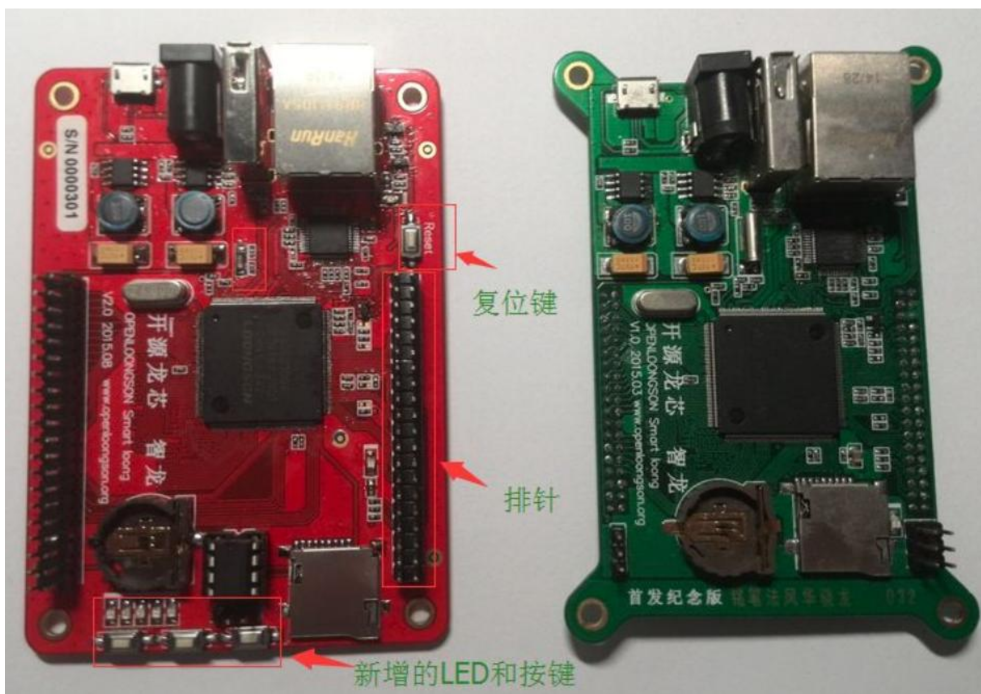


图 1.5 智龙开发板 V2.0 和 V1.0

智龙 V2.0 细节展示如图 1.6 所示。

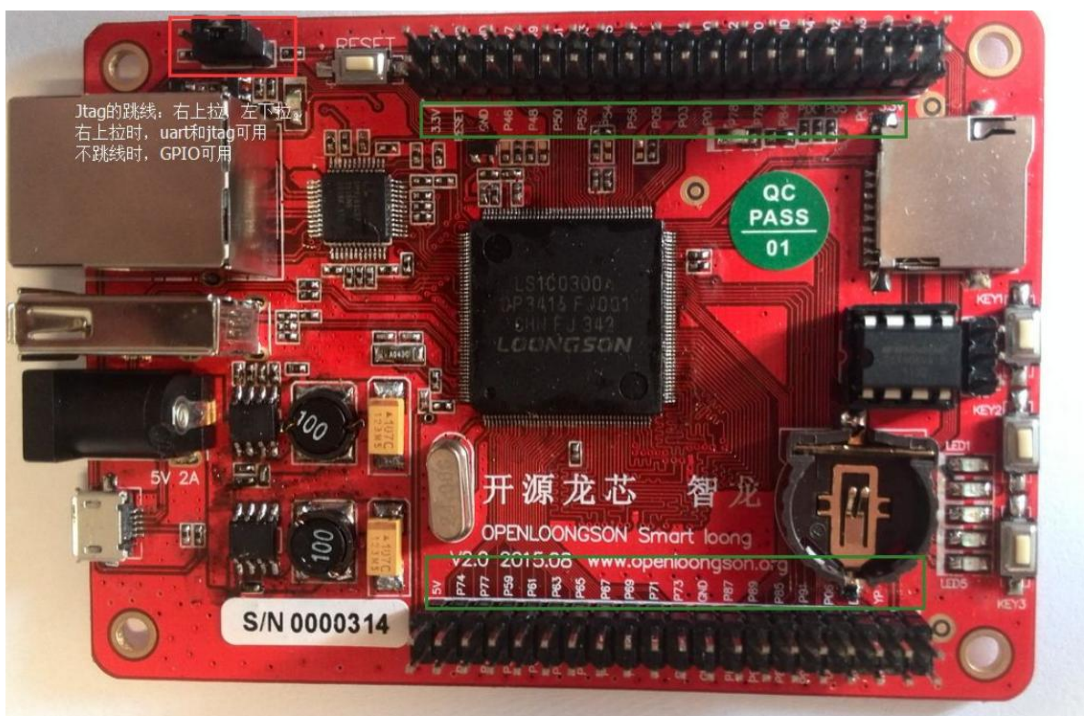


图 1.6 智龙开发板 V2.0 细节展示

开发板具体模块分布如图 1.7 所示。

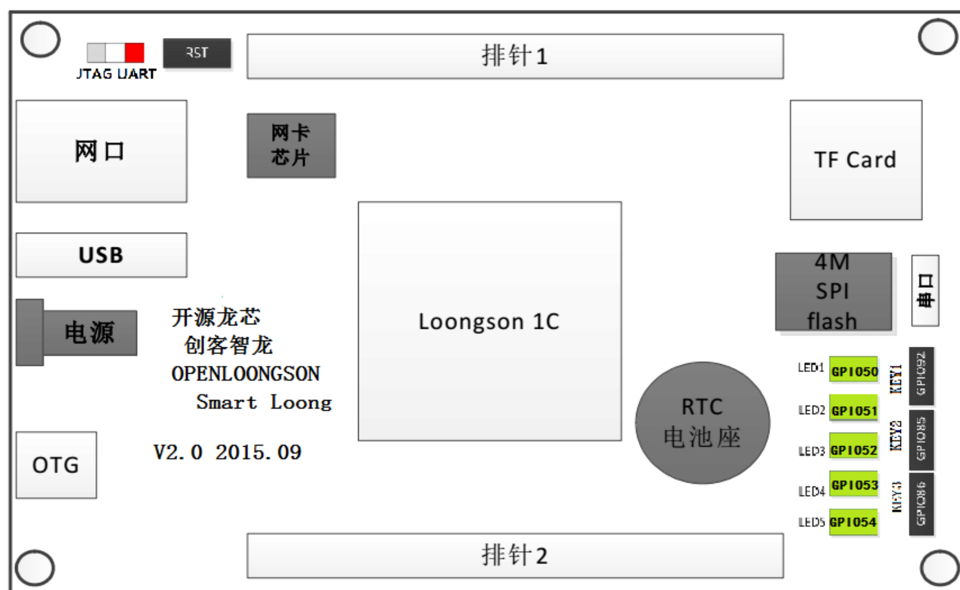


图 1.7 智龙开发板具体模块分布图

2. 虚拟机安装 linux 操作系统

2.1 下载 vmware player 并安装

常规操作，安装方法或网络查询，这里不再赘述。

2.2 下载 ubuntu 桌面系统

到以下网址选择合适的版本：
http://www.ubuntu.com/download/desktop 。

2.3 安装 ubuntu

新建虚拟机，导入 ubuntu 桌面系统光盘映象。

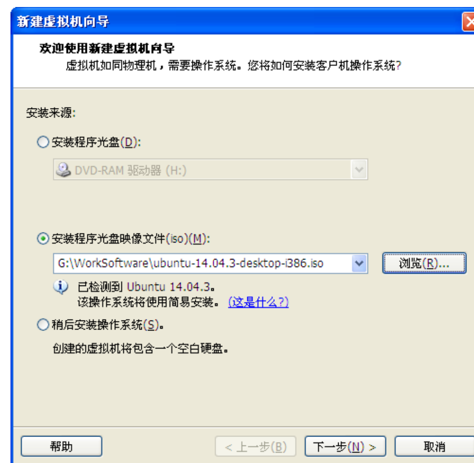


图 2.1 安装 ubuntu 界面

自定义硬件中进行设置，设置好后，点击完成。

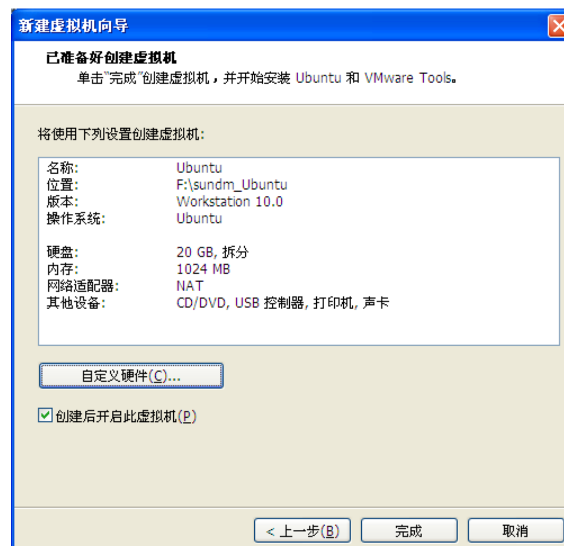


图 2.2 虚拟机安装位置设置

显示器中将加速 3D 去掉。

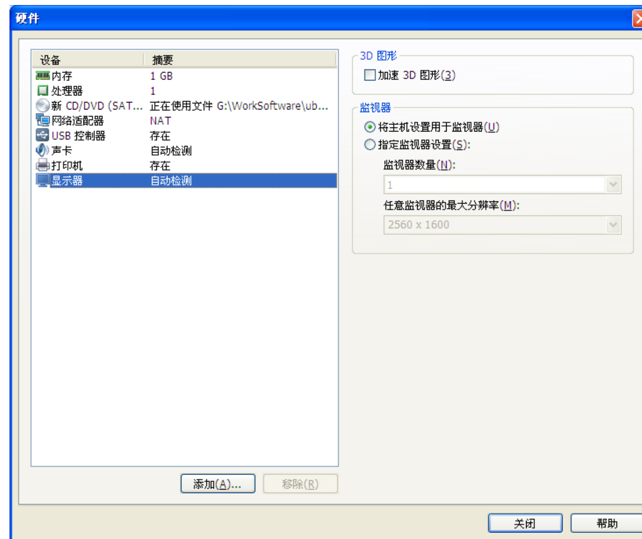


图 2.3 虚拟机中显示器设置

网络适配器中网络连接选用桥接模式。

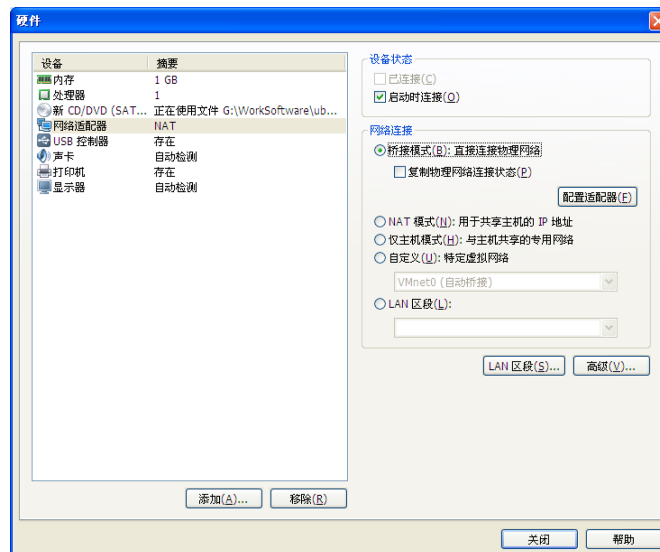


图 2.4 虚拟机中网络适配器设置

配置适配器中，如果有两个网卡，则都勾选上。

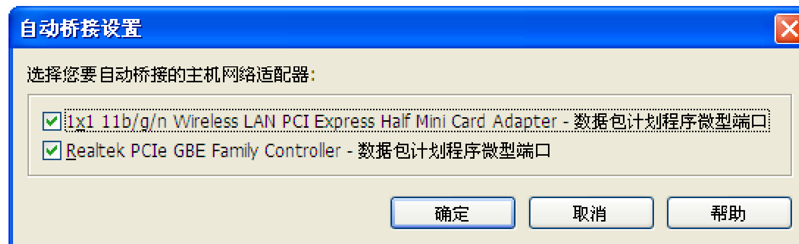


图 2.6 虚拟机中桥接设置

不要更新，否则速度太慢。

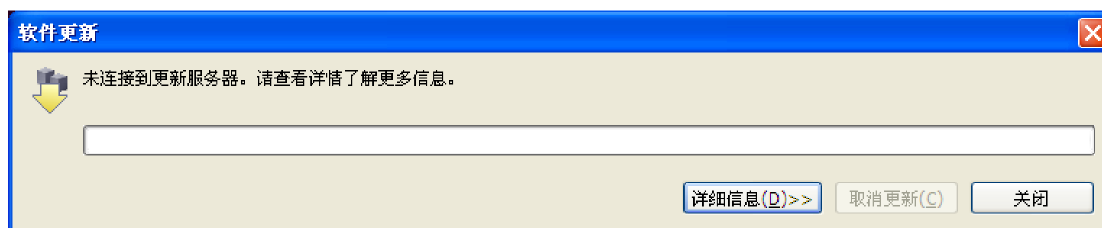


图 2.7 虚拟机中软件更新设置

开始进行自动安装。

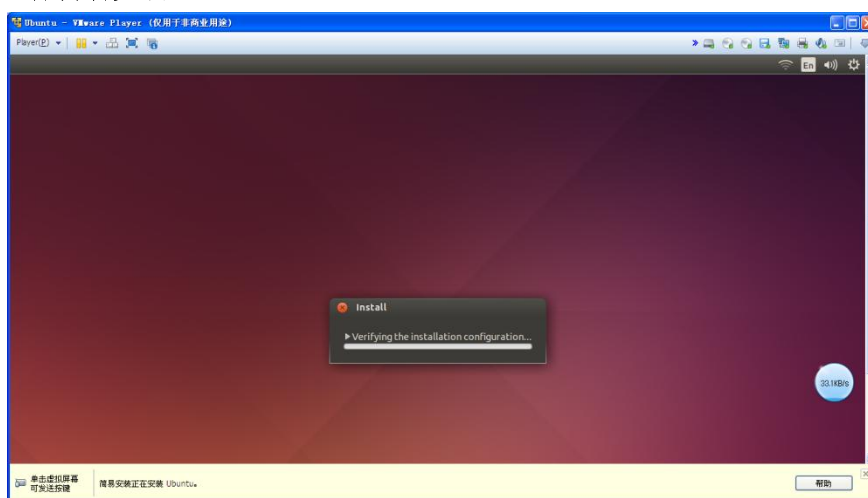


图 2.8 虚拟机中 ubuntu 操作系统安装

很好看的界面。

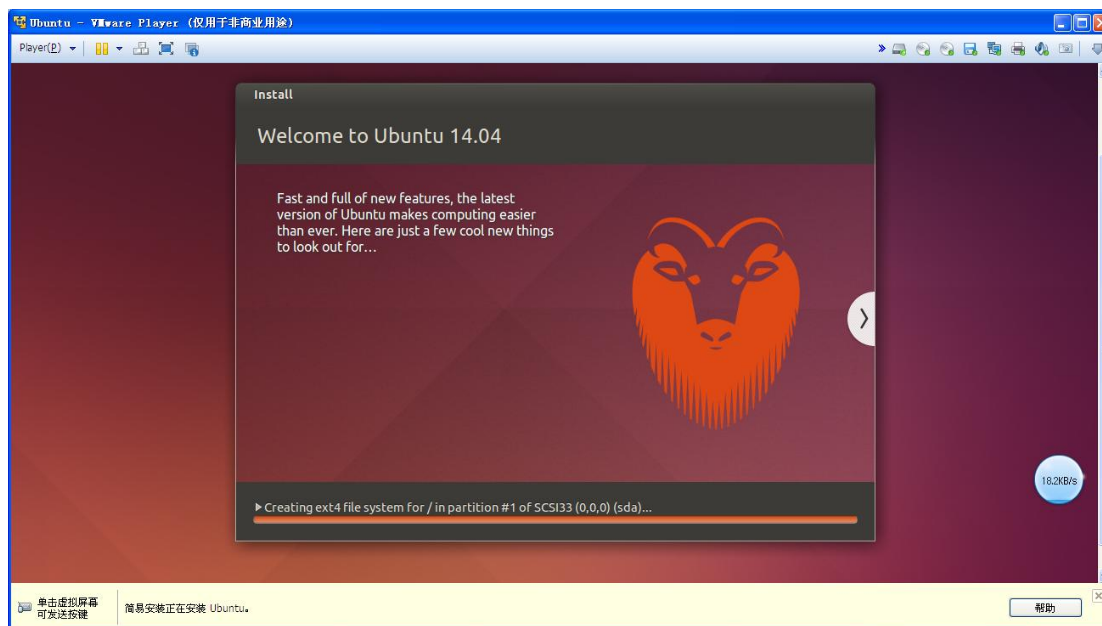


图 2.9 虚拟机中 ubuntu 安装进行界面

2.4 进入终端或者命令行方式

打开 ubuntu 终端，有三种方法：

1）桌面虚拟终端

在菜单中找，也可以直接运行 gnome-terminal 或 xterm 等。

2）按制台终端

取消 gdm 服务，或同时按 Ctrl+Alt+F2。

恢复桌面系统：Ctrl+Alt+F7。

3）远程登录

在物理机中运行相应工具，远程登录。

2.5 建立 root 用户并自动登录

1）建立 root 用户

```
sundm@ubuntu:/$ sudo passwd root
[sudo] password for sundm:          #输入当前用户密码
Enter new UNIX password:            #输入 root 密码
Retype new UNIX password:           #确认 root 密码
passwd: password updated successfully
sundm@ubuntu:/$ su    #切换至 root 用户
Password:
root@ubuntu:/#     #命令进入 root 用户了
```

2）设置 root 用户自动登录

```
gedit /usr/share/lightdm/lightdm.conf.d/50-ubuntu.conf
```

添加以下代码：

```
[SeatDefaults]
autologin-guest=false #不允许 guest 登录
autologin-user=root
user-session=ubuntu
greeter-show-manual-login=true#手工输入登陆系统的用户名和密码
```

在刚修改完 root 权限自动登录后，发现开机出现以下提示：

```
Error found when loading /root/.profile
stdin:is not a tty
```

需要修改 profile 文件。

```
gedit /root/.profile
```

打开文件后找到“mesg n”，将其更改为“tty -s && mesg n”。

2.6 安装 vmwaretools

VMware Tools 是 VMware 虚拟机中自带的一种增强工具，只有在 VMware 虚拟机中安装好了 VMware Tools，才能实现主机与虚拟机之间的文件共享，实现文件在虚拟机之间的复制粘贴；并可以根据自身需要自由切换显示屏幕的尺寸。

安装 vmware tools 使用 linux.iso 映像文件。打开 Vmware Tools，选中某个虚拟机，点击虚拟机设置。该步骤相当于把光盘插入光驱中。

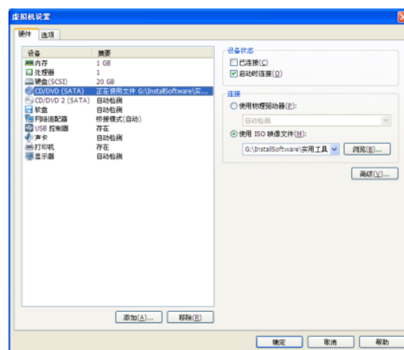


图 2.10 虚拟机安装 vmware tools

运行虚拟机，选择 Player，在可移动设备中选择 CD/DVD，点击连接。

图 2.10 虚拟机选择虚拟光盘操作图

弹出显示光盘内容的界面，后将 解压到/opt/下

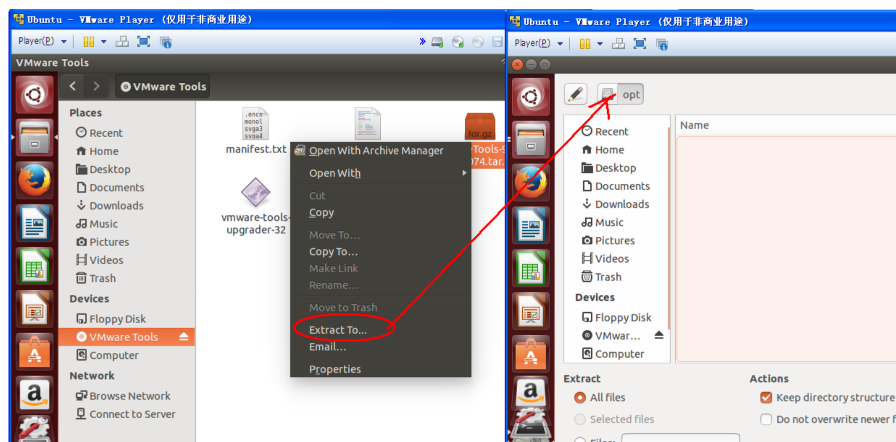


图 2.11 虚拟机安装 vmware tools 解压过程

依次执行以下指令，安装 vmware tools，安装过程中选择默认设置，一路按回车。

```
#进入 opt 文件夹
cd /opt
# 解压文件
tar zxvf VMwareTools-9.6.2-1688356.tar.gz
# 进行安装
cd vmware-tools-distrib/
./vmware-install.pl
#最后重启系统便完成了 Vmware Tools 安装
reboot
```

在终端输入 vm，然后按 2 次 tab 键（自动补齐），看系统有没有把 vmwaretools 的命令补齐，如果补齐了，则安装成功。

```
root@ubuntu:/opt/vmware-tools-distrib# vm
vmmouse_detect              vmware-rpctool
vmstat                      vmware-toolbox-cmd
vm-support                  vmware-uninstall-tools.pl
vmtoolsd                    vmware-user
vmware-checkvm              vmware-vmblock-fuse
vmware-config-tools.pl      vmware-xdg-detect-de
vmwarectrl                  vmware-xferlogs
vmware-hgfsclient
```

2.7 安装必要的软件

1）更新源

安装软件基本用的是 apt-get install,但是要先更新源：

```
apt-get update
```

以后如果提示：

```
Unable to locate packet    //无法找到包
```

就必须执行更新源的操作。后面安装所有的程序就都能用 apt-get install 命令了(网络安装比较方便)。如果出现以下提示：

```
Some packages could not be authenticated
```

表示在 Ubuntu 中没有设置软件源，下面开始进行设置软件源：打开 Ubuntu 软件中心，点击“编辑”，再点击“软件源”点击图中的“下载自”，选择“其它站点”，选择“最佳服务器”，然后电脑就会进行一些测试，然后电脑会根据你的网络状况选择最好的服务器，然后选择确认。

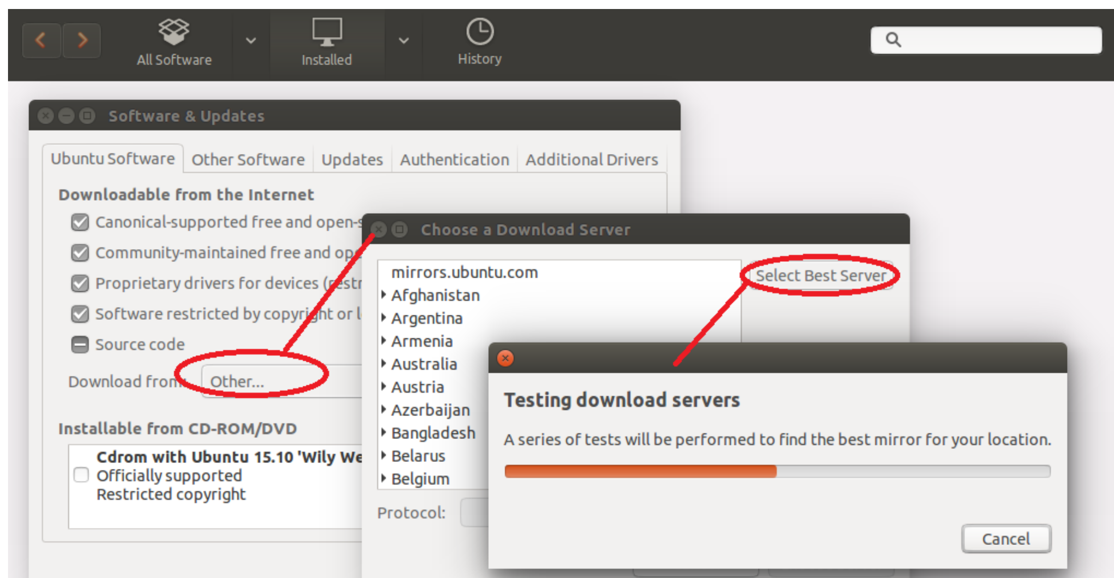


图 2.12 虚拟机安装更新源过程

2）安装以下软件（非必要，可选）

辅助编译 rtthread 的工具：

```
apt-get install python
apt-get install scons
```

利用 tftp 传输文件的工具：

```
apt-get install xinetd
apt-get install tftp-hpa
apt-get install tftpd-hpa
tftp-hpa 是 client，tftpd-hpa 是 server
```

完整的 vim 编辑器：

```
apt-get install vim
```

下载内核源码：

```
apt-get install Linux-source
```

安装 git 工具：

```
apt-get install git
```

2.8 查看相关版本、信息

1） 查看 linux 版本

```
cat /etc/issue
```

或者

```
uname -a
```

或者

```
cat /proc/version
Linux version 4.2.0-16-generic (buildd@lgw01-22) (gcc version 5.2.1 20151003 (Ubuntu 5.2.1-21ubuntu2) ) #19-Ubuntu SMP Thu Oct 8 14:46:51 UTC 2015
```

或者

```
uname -r
4.2.0-16-generic
```

2） 查看 gcc 版本

```
gcc -version
gcc -v
man gcc
```

3）查看系统位数

```
getconf WORD_BIT
```

4）查看文件属性：

```
file *
```

5） 查看 gcc 版本

```
gcc -version
gcc -v
man gcc
gcc -dumpversion
```

补充说明：

/proc 文件系统，它不是普通的文件系统，而是系统内核的映像，也就是说，该目录中的文件是存放在系统内存之中的，它以文件系统的方式为访问系统内核数据的操作提供接口。而使用命令“uname -a"的信息就是从该文件获取的，当然用命令直接查看它的内容也可以达到同等效果。另外，加上参数"a"是获得详细信息，如果不加参数为查看系统名称。

在 Linux 内核的顶层 Makefile 中的顶端就有，格式为

```
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 82
EXTRAVERSION =
NAME = Sneaky Weasel
```

以上的版本号就是 3.082。

3. 安装工具链、编译内核、制作文件系统

3.1 安装交叉编译工具 gcc-4.3-ls232

解压缩交叉编译工具到/opt 下：

```
tar zxvf gcc-4.3-ls232.tar.gz -C /opt
```

指定编译路径,添加环境变量(类似 win 系统中的 PATH)，打开当前用户下的.bashrc 文件：

```
vi ~/.bashrc
```

在其末尾添加语句：

```
export PATH=/opt/gcc-4.3-ls232/bin:$PATH
```

改变完~/.bashrc 文件后，重新加载路径

```
source ~/.bashrc
```

3.2 编译和烧写 pmon

安装编译工具 pmoncfg 需要依赖的库：

```
apt-get install bison
apt-get install flex
```

安装编译 PMON 还依赖于工具 makedepend:

```
apt-get install xutils-dev
```

将源码包 pmon-ls1x-openloongson.tar.gz 拷入/Workstation/tools/，并解压到该目录下，并进入 pmoncfg 目录。(后面所有的工作文件都放在这个目录下：/Workstation)

```
cd /Workstation/tools/pmon/pmon-ls1x-openloongson/tools/pmoncfg
```

执行编译命令，将生成的可执行文件拷贝到交叉编译工具链的 bin 目录下。

```
make
cp pmoncfg /opt/gcc-4.3-ls232/bin
```

进入相应的目录, 执行编译命令

```
cd /Workstation/tools/pmon/pmon-ls1x-openloongson/zloader.ls1c
make cfg all tgt=rom CROSS_COMPILE=mipsel-linux-
```

最后生成 gzrom.bin。

采用编程器，将 gzrom.bin 文件烧写进 Winboard 25X40 芯片。

注意事项：

1）配置文件

```
/Workstation/tools/pmon/pmon-ls1x-openloongson/Targets/LS1X/conf/ls1c
```

该配置文件内容与 ls1c_300a_openloongson 相同，ls1c_300a_openloongson 是该开发板的备份配置文件。

2）nand flash 分区

在/Workstation/tools/pmon-ls1x-openloongson/Targets/LS1X/dev/ls1x_nand.c 文件中设置分区。

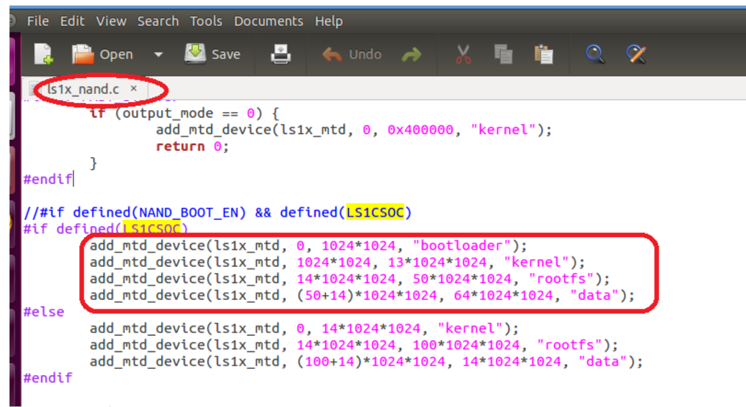


图 3.1 虚拟机中 ls1x_nand.c 文件显示分区的相关配置

```
#if defined(LS1CSOC)
    add_mtd_device(ls1x_mtd, 0, 1024*1024, "bootloader");
    add_mtd_device(ls1x_mtd, 1024*1024, 13*1024*1024, "kernel");
    add_mtd_device(ls1x_mtd, 14*1024*1024, 50*1024*1024, "rootfs");
    add_mtd_device(ls1x_mtd, (50+14)*1024*1024, 64*1024*1024, "data");
```

分区说明：

1）bootloader 1MByte 保留给 nand 启动用。

2）kernel 13MByte 用于烧录内核。

3）rootfs 50MByte 用于烧录根文件系统。

4）data 64MByte 可以用作其他。

注意分区大小要与 linux kernel 中的一致

bootloader 分区保留给 nand 启动用，所以 pmon 中烧录内核和根文件系统的命令为：

```
devcp tftp://193.169.2.215/vmlinux /dev/mtd1
mtd_erase /dev/mtd2
devcp tftp://193.169.2.215/rootfs-yaffs2.img /dev/mtd2 yaf nw
```

就是/dev/mtd0 不用，注意不要烧录错误。

如果用 spiflash 启动，不用 nand 启动的话可以根据自己使用的情况修改分区，注意 linux 内核中也要修改。

3.3 编译和烧写内核

安装图形化配置工具 Ncurses。

```
apt-get install libncurses5-dev
```

拷备内核源码包 linux-3.0.82-openloongson.tar.gz，并解压到内核源码包根目录/Workstation/Loongson_1C/BSP/Linux_Kernel 下。

```
tar zxvf linux-3.0.82-openloongson.tar.gz
cd linux-3.0.82-openloongson
cp arch/mips/configs/ls1c300a_openloongson_v2.0_defconfig .config
```

运行图形化配置命令：

```
make ARCH=mips CROSS_COMPILE=mipsel-linux- menuconfig
```

执行后进入内核配置主菜单界面，由于源码中已经全部配置好，这里不需要进行任何改动，直接保存退出。

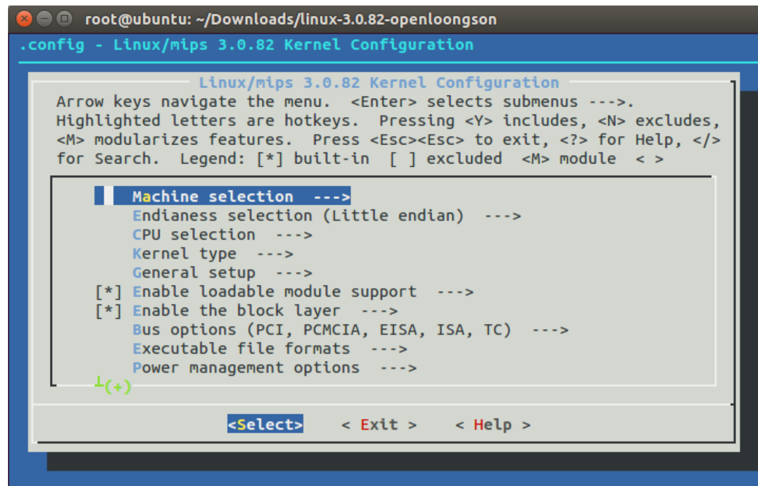


图 3.2 虚拟机中配置内核界面

查看平台文件 arch/mips/loongson/ls1x/ls1c/ls1c300a_openloongson_v2.0_platform.c

```
static struct mtd_partition ls1x_nand_partitions[] = {
	{
		.name		= "bootloader",
		.offset		= MTDPART_OFS_APPEND,
		.size	= 1024*1024,
	},	{
		.name		= "kernel",
		.offset		= MTDPART_OFS_APPEND,
		.size	= 13*1024*1024,
	},	{
		.name		= "rootfs",
		.offset		= MTDPART_OFS_APPEND,
		.size	= 50*1024*1024,
	},	{
		.name		= "data",
		.offset		= MTDPART_OFS_APPEND,
		.size	= MTDPART_SIZ_FULL,
	},
};
```

由于在 PMON 的启动中创建的三个 MTD 如下：

```
Creat MTD partitions on "ls1x-nand": name="kernel" size=14680064Byte
Creat MTD partitions on "ls1x-nand": name="os" size=104857600Byte
Creat MTD partitions on "ls1x-nand": name="data" size=14680064Byte
```

所以要将平台文件的 ls1x_nand_partitions 改成原来的，否则内核不能启动：

```
static struct mtd_partition ls1x_nand_partitions[] = {
{
		.name		= "kernel",
		.offset		= MTDPART_OFS_APPEND,
		.size	= 14*1024*1024,
	},	{
		.name		= "rootfs",
		.offset		= MTDPART_OFS_APPEND,
		.size	= 50*1024*1024,
	},	{
		.name		= "data",
		.offset		= MTDPART_OFS_APPEND,
		.size	= MTDPART_SIZ_FULL,
	},
};
```

配置完内核后，执行编译命令

```
make ARCH=mips CROSS_COMPILE=mipsel-linux-
```

编译完成后，在当前目录下生成内核镜像文件 vmlinux（未压缩）， vmlinuz（压缩后），

将单板机进入 pmon(在终端按住空格键，开机)，先擦除数据，再烧写其中一个均可。

```
mtd_erase /dev/mtd0
devcp tftp://193.169.2.215/vmlinuz3.0.82 /dev/mtd0
```

3.4 制件根文件系统

3.4.1 配置和编译 busybox

下载 busybox 网址： http://www.busybox.net/downloads/。

Busybox 是一个集成了一百多个最常用 linux 命令和工具的软件，它甚至还集成了一个 http 服务器和一个 telnet 服务器，而所有这一切功能却只有区区 1M 左右的大小。Busybox 的完全可定制性，提供了非常灵活，宜于扩展的结构。

因为最新版本的 busybox 要依赖更新版的内核头文件，我们使用的是 linux 3.0.82 的内核，而它当中不具备新版的 busybox 所具备的某些功能，所以编译过程中很可能会出错,这里下载并使用 1.23.0 版本。

Busybox 的配置方法类似于 linux 内核的配置。下载解压 busybox-1.23.0.tar.bz2 工具包后，进入 busybox-1.23.0 目录，运行“make menuconfig”，根据需要选择需使用的模块，保存退出后会在本地生成一个.config 文件，它指定 busybox 在编译的过程中需要包含哪些功能。

将工具包拷到虚拟机 /Workstation/tools/makefs 下解压 Busybox 源码包,后进入目录。

```
tar jxvf busybox1.23.0.tar.bz2
cd busybox-1.23.0
```

运行图形化配置命令：

```
make menuconfig
```

配置选项简述，其它按默认配置：

```
Busybox Settings    --->
Build Options    --->(以下为二选一)
[*] Build BusyBox as a static binary (no shared libs)（静态编译）
[*] Build shared libbusybox（动态编译）
```

指定交叉编译器,交叉编译器的绝对路径前缀，根据自己情况修改：

```
(/opt/gcc-4.3-ls232/bin/mipsel-linux-) Cross Compiler prefix
```

定制库：

```
Busybox Library Tuning    --->
[*] vi-style line editing commands
[*] Username completion(文件系统识别 PS1，以命令行提示符)
[*] Fancy shell prompts
```

其它组件：

```
Miscellaneous Utilities    --->
[ ] ubiattach
[ ] ubidetach
[ ] ubimkvol
[ ] ubirmvol
[ ] ubirsvol
[ ] ubiupdatevol
[ ] ionice
```

配置完成后清除：

```
make clean all
```

编译安装：

```
make install
```

则在 Busybox-1.23.0 目录下有_install 这个目录，正是所需要的。

另外也可以使用命令：

```
make CONFIG_PREFIX=/Workstation/tools/makefs/rootfs/ install
```

将生成的 install 安装的指定的目录下。

本代码安装到/Workstation/tools/makefs/rootfs/目录下。

3.4.2 创建文件系统目录

在 3.4.1 配置和编译 busybox 中创建的 rootfs/目录下，创建文件系统目录：

```
cd /Workstation/tools/makefs/rootfs/
mkdir dev home proc tmp var etc lib mnt sys opt root etc/init.d var/log
```

3.4.3 创建系统配置文件

进入刚创建的“rootfs”目录中，创建根文件系统的必要的文件:

```
cd /Workstation/tools/makefs/rootfs/
```

（1）etc/inittab 文件

说明：inittab 文件是 init 进程的配置文件，系统启动后所访问的第一个脚本文件，后续启动的文件都由它指定。用 VI 编辑 inittab 文件：

```
vi etc/inittab
```

添加如下内容：

```
::sysinit:/etc/init.d/rc.sysinit#指定系统启动后首先执行的文件

#Example of how to put a getty on a serial line (for a terminal)
ttyS2::respawn:-/bin/sh#串口终端，串口号 ttyS2 要与启动参数一致
#tty1::respawn:-/bin/sh#用于开发板屏幕终端显示

#Stuff to do when restarting the init process
#::restart:/sbin/init

#Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot#捕捉 ctrl+alt+del 键，重启文件系统
::shutdown:/bin/umount -a -r#当关机时卸载所有文件系统
#::shutdown:/sbin/swapoff -a 然后保存退出。
```

上面 ttyS3::respawn:-/bin/sh 中的 ttyS3 是指终端的控制台输出口，如想使用 ttyS1 或其他口，则需修改此配置，同时在文件系统启动参数也要相应修改对应的串口。（参考“3.5.3 烧写根文件系统”）

（2）etc/ init.d/rc.sysinit 文件

说明：这是一个脚本文件，可以在里面添加想自动执行的命令。以下命令配置环境变量、主机名、dev 目录环境、挂接/etc/fstab 指定的文件系统、建立设备节点与设置 IP。

打开文件 rc.sysinit :

```
vi etc/init.d/rc.sysinit
```

添加如下内容：

```
#!/bin/sh

#Set binary path
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

#Config dev enviornment
mount -t tmpfs -o size=64k,mode=0755 tmpfs /dev
mkdir -p /dev/pts #为 telnetd 创建 pts 目录
mount -t devpts devpts /dev/pts #挂载 pts 目录

#Build console and serial device files
echo "#Build console and serial device files....."
mknod -m 600 /dev/console          c 5 1 #建立设备文件
mknod -m 600 /dev/ttyS2            c 4 66 #建立设备文件

#mount all filesystem defined in "/etc/fstab"
echo "#mount all....."
/bin/mount -a

echo "#Starting mdev....."
echo /sbin/mdev > /proc/sys/kernel/hotplug #  设置热插拔事件处理程序为 mdev
```

```
/sbin/mdev -s #设备节点维护程序 mdev 初始化

#Set ip
ifconfig eth0 193.169.2.230 up
ifconfig lo 127.0.0.1
```

在这个启动脚本中，使用了 mdev。在应用启动后，使用 medv -s 自动创建设备节点。

（3）etc/fstab 文件

说明：执行 mount -a 时挂接/etc/fstab 指定的文件系统。

打开文件 fstab:

```
vi etc/fstab
```

添加如下内容：

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

（4）etc/profile 文件

说明：inittab 中执行了这样一个语句“respawn:-/bin/sh”。启动/bin/sh 程序时会启动 ash 的配置信息，而它就是/etc/profile，sh 会把 profile 的所有配置全部都运行一遍，因此用户可以把自己的启动程序放在这里。

打开文件 profile:

```
vi etc/profile
```

添加如下内容：

```
#!/bin/sh
#/etc/profile:system-wide .profile file for the Bourne shells
echo "Processing /etc/profile....."

#set search library path
export LD_LIBRARY_PATH=/lib:/usr/lib

#set user path
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

#Set PS1 modify command prompt
export PS1='[\u@\h:\w]\$'

#Set hostname
/bin/hostname "Loongson"
HOSTNAM=/bin/hostname

export PS1 HOSTNAME

#Set ll aliae
alias ll="ls -l"
echo "Done!"
```

（5）修改系统配置文件权限

```
chmod 755 /Workstation/tools/makefs/rootfs/etc/*
chmod 755 /Workstation/tools/makefs/rootfs/etc/init.d/rc.sysinit
```

（6）拷贝 Busybox 文件

将 ”3.4.1 配置和编译 busybox” 中编译安装好的 Busybox 文件拷贝到 rootfs 目录中：

```
cp /Workstation/tools/makefs//busybox-1.23.0/_install/* . –ra
```

如果是使用以下命令创建的 rootfs 目录，则可以省去以上拷备的过程。

```
make CONFIG_PREFIX=/Workstation/tools/makefs/rootfs/ install
```

除了以上文件之外，还制作以下文件及目录，放在 etc 目录下。

目录 hotplug 下也有一些文件。

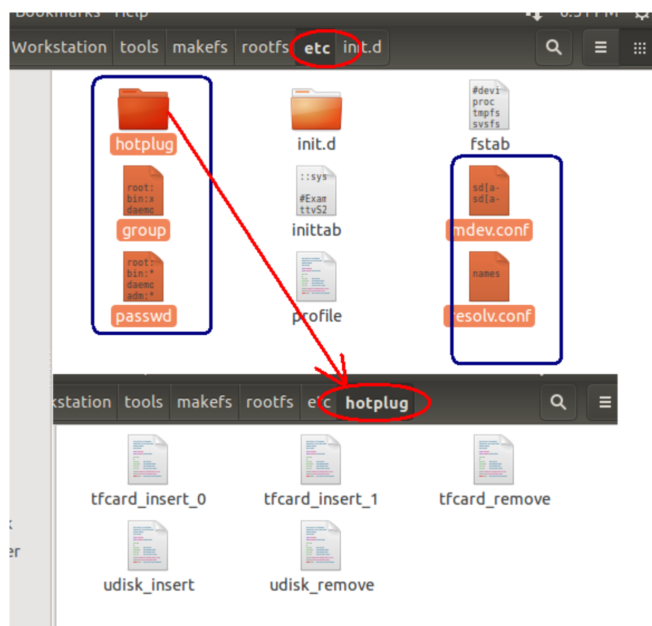


图 3.3 hotplug 文件夹内容拷备

3.4.4 拷贝库文件

提示：配置 Busybox 若选择静态编译则省略该步骤。

动态编译 Busybox 制作文件系统，需将交叉编译工具里的加载器和动态库文件到根文件系统里，交叉编译工具的库在/opt/gcc-4.3-ls232/sysroot/lib/中:

librt.so.1、ld.so.1、libc.so.6、libcrypt.so.1、libm.so.6、libdl.so.2、libpthread.so.0、libresolv.so.2

和目录/opt/gcc-4.3-ls232/mipsel-linux/lib/中的两个库文件：

libgcc_s.so.1、 libstdc++.so.6

将以上文件拷贝到根文件系统下的 lib 目录。

这些库文件中除了 libgcc_s.so.1 之外都是链接文件，因此拷贝这些库文件也要拷贝被链接的文件。在相应的目录中执行“ll”命令可查看这些库文件的链接关系。在执行拷贝命令时需要在命令后加“-L”参数，即随符号链接拷贝原文件。

使用脚本快速拷贝所需的基本动态库：

```
vi cplib.sh
```

添加如下内容：

```
#!/bin/bash
# copy mipsel-linux lib

libdir=/opt/gcc-4.3-ls232
fsdir=/Workstation/tools/makefs/rootfs/lib

cp   -L $libdir/sysroot/lib/librt.so.1 $fsdir
cp   -L $libdir/sysroot/lib/ld.so.1 $fsdir
cp   -L $libdir/sysroot/lib/libc.so.6 $fsdir
cp   -L $libdir/sysroot/lib/libcrypt.so.1 $fsdir
cp   -L $libdir/sysroot/lib/libm.so.6 $fsdir
cp   -L $libdir/sysroot/lib/libdl.so.2 $fsdir
cp   -L $libdir/sysroot/lib/libpthread.so.0 $fsdir
cp   -L $libdir/sysroot/lib/libresolv.so.2 $fsdir

cp   -L $libdir/mipsel-linux/lib/libgcc_s.so.1          $fsdir
cp   -L $libdir/mipsel-linux/lib/libstdc++.so.6        $fsdir
```

运行脚本：

```
 chmod +x cplib.sh
```

```
./cplib.sh
```

```
root@ubuntu:/Workstation/tools/makefs# ls ./rootfs/lib -ll
total 19000
-rwxr-xr-x 1 root root  648978 Mar 19 09:03 ld.so.1
-rwxr-xr-x 1 root root  106876 Mar 19 09:03 libcrypt.so.1
-rwxr-xr-x 1 root root 8407178 Mar 19 09:03 libc.so.6
-rwxr-xr-x 1 root root  102446 Mar 19 09:03 libdl.so.2
-rw-r--r-- 1 root root 3102260 Mar 19 09:03 libgcc_s.so.1
-rwxr-xr-x 1 root root 1141320 Mar 19 09:03 libm.so.6
-rwxr-xr-x 1 root root  911634 Mar 19 09:03 libpthread.so.0
-rwxr-xr-x 1 root root  251417 Mar 19 09:03 libresolv.so.2
-rwxr-xr-x 1 root root  223448 Mar 19 09:03 librt.so.1
-rwxr-xr-x 1 root root 4538060 Mar 19 09:03 libstdc++.so.6
```

图 3.4 压缩前 libgcc_s.so.1 文件属性

可以看到：libgcc_s.so.1 大小为:3M。

为了减少根文件系统的库大小，使用交叉编译工具即 gcc-4.7.3-mips32 的 strip 工具来处理库文件，把二进制文件中的包含的符号表和调试信息删除掉,可有效减少库文件大小。

```
mipsel-linux-strip rootfs/lib/*so*
```

压缩后：libgcc_s.so.1 大小为:246K

```
root@ubuntu:/Workstation/tools/makefs# mipsel-linux-strip rootfs/lib/*so*
root@ubuntu:/Workstation/tools/makefs# ls ./rootfs/lib -ll
total 3648
-rwxr-xr-x 1 root root  133744 Mar 19 09:36 ld.so.1
-rwxr-xr-x 1 root root   39876 Mar 19 09:36 libcrypt.so.1
-rwxr-xr-x 1 root root 1490696 Mar 19 09:36 libc.so.6
-rwxr-xr-x 1 root root   10900 Mar 19 09:36 libdl.so.2
-rw-r--r-- 1 root root  246208 Mar 19 09:36 libgcc_s.so.1
-rwxr-xr-x 1 root root  513992 Mar 19 09:36 libm.so.6
-rwxr-xr-x 1 root root  108256 Mar 19 09:36 libpthread.so.0
-rwxr-xr-x 1 root root   75876 Mar 19 09:36 libresolv.so.2
-rwxr-xr-x 1 root root   42756 Mar 19 09:36 librt.so.1
-rwxr-xr-x 1 root root 1080000 Mar 19 09:36 libstdc++.so.6
root@ubuntu:/Workstation/tools/makefs#
```

图 3.5 压缩后 libgcc_s.so.1 文件属性

至此，根文件系统（目录/Workstation/tools/makefs/rootfs）制作完成。

3.5 制作根文件系统镜像

3.5.1 安装镜像文件制作工具

所使用的工具有：

（1）zlib-1.2.8.tar.gz 依赖工具

（2）cramfs-1.1.tar.gz 制作 cramfs 文件系统工具

（3）yaffs2-d43e901.tar.gz 制作 yaffs2 文件系统工具

（4）lzo-2.09.tar.gz 依赖工具

（5）e2fsprogs-1.42.13.tar.gz 依赖工具

（6）mtd-utils-1.5.1.tar.bz2 制作 ubifs 文件系统工具

将以上工具拷备至 Workstaion/tools/makefs 下，并解压

（1）安装依赖工具 zlib

zlib 下载地址：http://www.zlib.net/

```
tar zxvf 1.zlib-1.2.8.tar.gz
```

```
cd zlib-1.2.8
./configure
make
make install
cd ..
```

（2）安装制作 cramfs 文件系统镜像文件工具 mkcramfs

提示：也可使用 Ubuntu 系统自带的制作 cramfs 文件系统工具 mkfs.cramfs

```
tar zxvf 2.cramfs-1.1.tar.gz
cd cramfs-1.1
make
```

在当前目录下生成 mkcramfs，将其拷贝到/usr/bin 目录下。

```
cp mkcramfs /usr/bin
 cd ..
```

（3）安装制作 yaffs2 文件系统镜像文件工具 mkyaffs2image

```
tar zxvf 3.yaffs2-d43e901.tar.gz
cd yaffs2-d43e901/utils
make
```

在当前目录下生成 mkyaffs2image，将其拷贝到/usr/bin 目录下。

```
cp mkyaffs2image /usr/bin
cd ../..
```

（4）安装依赖工具 lzo

lzo 下载地址：http://www.oberhumer.com/opensource/lzo/download/

```
tar zxvf 4.lzo-2.09.tar.gz
cd lzo-2.09
./configure --build= i686-linux-gnu --prefix=/Workstation/tools/makefs/install
make
make install
cd ..
```

（5）安装依赖工具 e2fsprogs（也可以用 apt-get install e2fsprogs 安装）

e2fsprogs 下载地址：http://sourceforge.net/projects/e2fsprogs/files/e2fsprogs/

```
tar zxvf 5.e2fsprogs-1.42.13.tar.gz
cd e2fsprogs-1.42.13
./configure --build= i686-linux-gnu --prefix=/Workstation/tools/makefs/install1
make
make install
cd lib/uuid
make install
cd ../../..
```

（6）安装 ubifs 文件系统镜像文件制作工具

来自于：http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=23089249&id=4385560

mtd-utils 依赖于 zlib、lzo、e2fsprogs 提供的库,所以编译 mtd-utils 之前,需要先编译 zlib、lzo、 e2fsprogs,并安装到编译工具相应目录下。

mtd-utils 下载地址：ftp://ftp.infradead.org/pub/mtd-utils/

使用命令安装 ubifs 文件系统镜像文件制作工具：mkfs.ubifs 和 ubinize 两个工具

```
apt-get install mtd-utils
```

另一种方法：

用安装包来安装 ubifs 文件系统镜像文件制作工具。

```
tar jxvf 6.mtd-utils-1.5.2.tar.bz2
cd mtd-utils-1.5.2
```

修改 Makefile

```
vi Makefile
```

在版本说明“VERSION = 1.5.2”一行后面添加如下内容：

```
PREFIX = /Workstation/tools/makefs/install/mtd
DEPEND = /Workstation/tools/makefs/install
ZLIBCPPFLAGS = -I/usr/local/include
ZLIBLDFLAGS = -L/usr/local/lib
LZOCPPFLAGS = -I$(DEPEND)/include
LZOLDFLAGS = -L$(DEPEND)/lib
LDFLAGS += $(ZLIBLDFLAGS) $(LZOLDFLAGS)
CFLAGS ?= -O2 -g $(ZLIBCPPFLAGS) $(LZOCPPFLAGS)
```

修改 common.mk

```
vi common.mk
```

找到并注释掉“PREFIX=/usr”一行，如下：

```
#PREFIX=/usr
```

编译安装。

```
WITHOUT_XATTR=1 make
make install
cd ..
```

将在安装目录 install/mtd/sbin 下生成的可执行文件 mkfs.ubifs 和 ubinize 拷贝到/usr/bin 目录下。

```
cp install/mtd/sbin/mkfs.ubifs /usr/bin
cp install/mtd/sbin/ubinize /usr/bin
```

3.5.2 制作根文件系统镜像文件

使用安装好镜像文件制作工具来制作在“4.制件根文件系统”中完成构建的根文件系统目录“rootfs”的镜像文件。

3.5.2.1 cramfs 文件系统

```
mkcramfs rootfs/ rootfs-cramfs.img
chmod +r rootfs-cramfs.img//添加可读权限，避免出现无法烧写的情况
```

或使用系统自带的工具

```
mkfs.cramfs rootfs/ rootfs-cramfs.img
chmod +r rootfs-cramfs.img//添加可读权限，避免出现无法烧写的情况
```

3.5.2.2 yaffs2 文件系统

```
mkyaffs2image rootfs/ rootfs-yaffs2.img
chmod +r rootfs-yaffs2.img//添加可读权限，避免出现无法烧写的情况
```

3.5.2.3 ubifs 文件系统

1）使用 mkfs.ubifs 命令将 rootfs 文件夹制作为 UBIFS 镜像，具体命令为：

```
mkfs.ubifs -r rootfs -m 2048 -e 129024 -c 370 -o ubifs.img
```

以上命令的含义为将 rootfs 文件夹制作为 UBIFS 文件系统镜像，输出的镜像名为 ubifs.img，-m 参数指定了最小的 I/O 操作的大小，也就是 NAND FLASH 一个 page 的大小，-e 参数指定了逻辑擦除快的大小，-c 指定了最大的逻辑块号。

通过此命令制作的出的 UBIFS 文件系统镜像可在 u-boot 下使用 ubi write 命令烧写到

NAND FLASH 上。

2）新建配置文件 ubinize.cfg

ubinize.cfg 为一些配置参数：

```
[ubifs]
mode=ubi
image=ubifs.img
vol_id=0
vol_size=45MiB
vol_type=dynamic
vol_alignment=1
vol_name=rootfs
vol_flags=autoresize
```

提示：1C300B 开发板使用 2K 页大小的 Nand Flash 芯片，文件系统分区大小为 50M。配置文件中的 vol_size 的大小不能大于文件系统分区的大小，vol_size 的值约等于-e 参数（逻辑擦除块大小）乘以-c 参数（最大逻辑擦除块数）。

使用 ubinize 命令可将使用 mkfs.ubifs 命令制作的 UBIFS 文件系统镜像转换成可直接在 FLASH 上烧写的格式（带有 UBI 文件系统镜像卷标）：

```
ubinize -o ubi.img -m 2048 -p 128KiB -s 2048 -O 2048 ubinize.cfg
```

通过此命令生成的 ubi.img 可直接使用 NAND FLASH 的烧写命令烧写到 FLASH 上。

提示：内核配置 UBIFS 的支持请参考“4.1 配置和编译 busybox”。

3.5.3 烧写根文件系统

将单板机进入 pmon(在终端按住空格键，开机)，在 pmon 中先擦除数据，再烧写*.img 文件。

```
mtd_erase /dev/mtd1
devcp tftp://193.169.2.215/rootfs-yaffs2.img /dev/mtd1 yaf nw
```

设置文件系统起动参数

```
set append " root=/dev/mtdblock1"                          //根目录位置，块设备
set append " $append console=ttyS2,115200"                 //设置串口 2，115200 波特率
set append " $append   noinitrd init=/linuxrc rw rootfstype=yaffs2"       //noinitrd 代表没有使用 ramdisk；init=/linuxrc 是指内核启动起来后进入系统中运行的第一个脚本，挂载之后文件系统是只读的，所以就加了个 rw；rootfstype=yaffs2 指明文件系统类型为 yaffs2 不然没法挂载根分区
set append " $append   video=ls1bfb:480x272-16@60 fbcon=rotate:1 consoleblank=0"      //fbcon=rotate:1 标示屏幕可旋转；consoleblank=0 禁用屏幕白色待机
```

4. 使用 buildroot 构建根文件系统

本节与 《3.4》中的功能是完全一致的。

https://github.com/pengphei/smartloong-sphinx/blob/master/source/cn/loongson1c_buildroot_guide.rst 的帖子详细说明了 buildroot 龙芯 1C 的支持指南。以下均按照贴操作，稍加改动。

4.1 获取 buildroot

可以从 buildroot 官网(http://buildroot.uclibc.org/download.html)获取 buildroot 源码包，buildroot 基本上三个月更新一次，这里实际下载的源码包是用 git clone 下载 pengphei 已经配置好的源码 https://github.com/pengphei/buildroot，基于 2015.02 发布版本。也可以从工具包里拷备：buildroot-loongson.tar.gz。

```
git clone https://github.com/pengphei/buildroot.git
```

4.2 系统构建

为了构建出支持智龙开发板可用的镜像格式。在原本的 Buildroot 环境中添加了 yaffs2img 文件系统支持。该包与 Buildroot 中原有的 yaffs2 文件系统构建并不相同，需要特别注意。

智龙开发板的构建命令：

```
cd buildroot
cp configs/loongson1c_smartloong_defconfig .config
make
```

上述命令与 linux 内核的配置工具和使用方法完全相同。只是 buildroot 将会从网络上下载自己所需要的基础软件包以及构建工具链，在首次构建时，代码的下载和构建将会需要比较长的时间(时间不是一般的长，在准备好 20M 宽带的前提下，用了将近 3 小时)。

4.3 烧写根文件系统镜象

在 make 命令执行完成之后，会生成 output/images/rootfs.yaffs2img 文件，该文件即为可以烧录到智龙开发板中的根文件系统镜像。

同《3.5.3》将根文件系统镜像 rootfs.yaffs2img 烧录到智龙开发板中。

4.4 根文件系统软件包的定制

在 4.2 系统构建过程中，如果下载软件时间过长，可以将提供的已经下载好的软件 dl.tar.gz（共 304M）解压到 自己拷备的 buildroot/dl 目录下，可减少下载时间。

如果希望根据自己的需要，添加自己需要的软件包，可以执行如下命令进行根文件系统软件包的定制：

```
    make menuconfig
```

下面配置几个比较重要的。

配置芯片架构：

```
Target option --->
        Target Architecture --->
(*)MIPS ( little endian )
```

配置交叉工具链：

```
Toolchain --->
        Toolchain --->
                (*) Sourcery CodeBench MIPS 2014.11
```

系统配置：选择设备节点的配置表，选择 system/device_table_dev.txt，否则后面在 dev 目录下将不会生成各 种设备节点。

```
System configuration    --->
System hostname    --->
SmartLoong     //配置主机名称,选择自己喜欢的名字
Path to the permission tables    --->
system/device_table_dev.txt    //否则后面在 dev 目录下将不会生成各 种设备节点
getty options--->
       TTY port
                ttyS2 //配置控制台串口
       Baudrate
              115200 //对应串口设置的波特率
```

根文件系统的类型设置：

```
Filesysytem images --->
       [*] yaffs2img root filesystem
```

5．简单应用编程 Helloworld

在虚拟机终端输入

```
vi helloworld.c
```

添加如下内容：

```
#include <stdio.h>

int main(void)
{
	printf("Hello World!\n");
	return 0;
}
```

编译程序:

```
mipsel-linux-gcc helloworld.c -o helloworld
```

有时可能会出现错误：

```
root@ubuntu:/Workstation/examples/AppProg/1.helloworld# mipsel-linux-gcc helloworld.c -o helloworld
helloworld.c: In function 'main':
helloworld.c:5: error: stray '\342' in program
helloworld.c:5: error: stray '\200' in program
helloworld.c:5: error: stray '\234' in program
```

出现此问题的原因是，源代码中存在汉语时的字符，如“”、，等，将其改过来即可！

将 helloworld 从虚拟机拷出来，并拷备入单板，在 putty 中执行命令：

```
[root@Loongson:/]#tftp -r helloworld -g 193.169.2.215
helloworld          100% |*******************************|   7865    0:00:00 ETA
 [root@Loongson:/]#./helloworld
-/bin/sh: ./helloworld: Permission denied
[root@Loongson:/]#chmod u+x helloworld
[root@Loongson:/]#./helloworld
Hello World![root@Loongson:/]#
```

刚开始运行 helloworld 命令的时候，权限不够，采用语句 chmod u+x helloworld 添加权限后，能够正常运行。

6. 简单驱动程序编写

6.1 驱动的原理及编写流程

从上到下，一个软件系统可以分为：应用程序、库、操作系统（内核）、驱动程序。其中，linux 内核就是由各种驱动组成的，内核源码中大约 85%是各种驱动程序的代码。内核中驱动程序种类齐全，可以在同类型驱动的基础上进行修改以符合具体不同的单板。

app 与驱动的关系如图 6.1 所示。

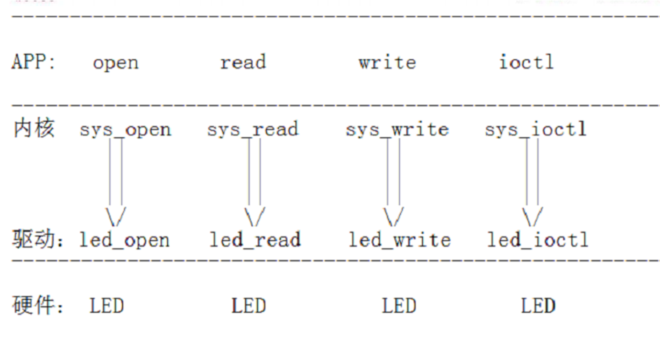


图 6.1 驱动程序与 APP 的关系图

驱动程序编写的流程：

1）查看原理图、数据手册，了解设备的操作方法。

2）在内核中找到相近的驱动程序，以之为模板进行开发。

3）实现驱动程序的初始化：注册 、传入文件名。

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

5）实现中断服务（不是每个设备驱动必须）。

6）编译驱动程序到内核，或者用 insmod 加载。

7）编写应用程序，用来测试驱动。

6.2 驱动模块的加载与卸载

可以将驱动程序静态编译进内核，也可以将它作为模块在使用时加载。使用方法：

1） 手工加载和挂载。模块的扩展名为 *.ko，使用 insmod 命令或者 modprobe 命令加载到内核，使用 rmmod 命令卸载，使用 lsmod 命令查看内核中已经加载了哪些模块。

2） 配置某个目录下的 Kconfig 和 Makefile，然后 make menuconfig 来配置。配置内核时，如果某个配置项设为[m]，就表示它将会被编译成一个模块。配置内核时，如果某个配置项设为[*]，就表示它将会被编译进内核。

一般来说，开发过程中经常使用的是方法 1，对于方法 2 是出厂的时候用的。

6.3 最简单的 Linux 驱动

Linux 驱动必须束缚在内核规定的某个架构下。

模块的入口函数也称模块加载函数，当执行 insmod 或 modprobe 命令加驱动模块到内核时，驱动模块的入口函数就会自动被内核执行。至于模块入口数需要完成什么工作，这就由编程决定了。

模块出口函数也称为模块卸载函数，当执行 rmmod 命令卸载驱动模块，驱动模块的出口函数就会自动被内核执行。至于模块出口函数需要完成什么作，这个也是由编程决定的。

许可证(LICENSE)声明该模块的许可权限，如果不声明，就会收到内核的警告。

模块参数、模块作者。模块导出符号、模块描述是可选的。

最简单的 hello 驱动源码：

```
/*
 * Name:hello.c
 * Copyright (C) 2015 sundm75   (17164830@qq.com)
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
#include <linux/module.h>
#include <linux/kernel.h>
/* 驱动程序的入口函数 */
static int __init hello_init(void)
{
       printk(KERN_WARNING "Hello_init!\n");
       printk(KERN_WARNING "Hello,world!\n");
       return 0;
}
/* 驱动程序的出口函数 */
static void __exit hello_exit(void)
{
     printk(KERN_WARNING "Hello_exit!\n");
     printk(KERN_WARNING "Goodbye,world!\n");
}
/*
 * 用于修饰入口/出口函数，
 * 告诉内核驱动程序的入口/出口函数在哪里
 */
module_init(hello_init);
module_exit(hello_exit);
/* 该驱动支持的协议、作者、描述 */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("sundm75");
MODULE_DESCRIPTION("Board First module test");
```

6.4 驱动的编译和执行

编写 Makefile：

```
#定义生成的目标
obj-m := hello.o
#定义目录变量
KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson
PWD := $(shell pwd)
all:
# make 文件
	make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS_COMPILE=mipsel-linux-
clean:
	rm -rf *.o   *.mod.c *.ko 以上代码注意:
```

1）make-C.....这一行中，行首要 tab 缩进，不是空格。rm –rf.....这一行，行首也是缩进。

2）/root/Downloads/linux-3.0.82-openloongson 是虚拟机的内核源码的目录，要根据自己内核位置进行相应修改，而内核源码一定要编译过，而且单板机上下载的 vmlinuz 版本也是这个编译过的版本，否则会出现编译错误或者编译正确后无法加载错误。

make -C $(Kdir) M=`pwd` modules 该命令是 make modules 命令的扩展，-C 选项的作用是指将当前的工作目录转移到制定的目录，即（KDIR）目录，程序到（shellpwd）当前目录查找模

块源码，将其编译、生成.ko 文件。

$(PWD) 指当前目录的全路径名称。

`pwd`即获取当前路径。 ``(注意`不是单引号,`是 Tab 键上面那个)作用是执行指令。

$1 是获取函数的第一个参数值。

以下是模块加载不成功错误，提示 version 版本问题。

```
[root@Loongson:/]#insmod hello.ko
hello: disagrees about version of symbol module_layout
insmod: can't insert 'hello.ko': invalid module format
```

Makefile 代码说明：

1）定义生成目标

如果模块源码目录中的 Kbuild 或 Makefile 中没有定义编译目标时，编译过程最终是什么都没生成的。

以下一行定义了要生成的目标：

```
obj-m: = <module_name>.o
```

obj-m 变量是指外部模块，其后面的<module_name>.o 指定最终生成<module_name>.ko 模块。在默认情况下，内核源码编译系统会将<module_name>.c 编译成<module_name>.o，并最终链接生 成<module_name>.ko。如果<module_name>.ko 需要多个源文件时， Makefile 中要按以下形式编写：

```
obj-m: = <module_name>.o
<module_name>-objs: = src1.o src2.o ….
```

2）Kbuild 文件

```
make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS_COMPILE=mipsel-linux-
```

make -C$(KDIR) 指明跳转到内核源码目录下，读取那里的 Makefile; M=$(PWD) 表明返回到当前的目录继续读入执行当前的 Makefile

3）头文件

在内核源码树中，头文件的存放规则如下：

（1）如果该头文件定义的是模块内部的接口，则头文件放在模块所在的目录下

（2）如果头文件中内容在内核其他子系统中使用，则放在 include/linux

将以上 hello.c 和 Makefile 放在虚拟机某一文件夹下，运行 make：

```
root@ubuntu:~/Downloads/developdrivers/1th_hello# make
make -C /root/Downloads/linux-3.0.82-openloongson M=/root/Downloads/developdrivers/1th_hello modules ARCH=mips CROSS_COMPILE=mipsel-linux-
make[1]: Entering directory `/root/Downloads/linux-3.0.82-openloongson'
  CC [M]  /root/Downloads/developdrivers/1th_hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /root/Downloads/developdrivers/1th_hello/hello.ko
make[1]: Leaving directory `/root/Downloads/linux-3.0.82-openloongson'
```

以上提示说明：首先进入内核源码目录，编译出 hello.o 的文件，运行 MODPOST 生成临时的 hello.mod.c 文件，而后根据此文件编译出 hello.mod.o,之后连接 hello.o 和 hello.mod.o 文件，得到模块目标文件 hello.ko，最后离开 linux 内核所在的目录回到当前目录。运行 ls 命令查看当前目录下文件：

```
hello.c     hello.ko        hello.mod.o   Makefile      modules.order
hello.c~    hello.mod.c     hello.o       Makefile~     Module.symvers
```

Module.symvers 文件包含了内核以及编译后的模块导出的所有符号。对于每一个符号，相应的 CRC 校验值也被保存，Module.symvers 每一行数据格式如下：

```
<CRC>              <Symbol>              <module>
0x2d036834         scsi_remove_host      drivers/scsi/scsi_mod
```

Module.symvers 文件主要有以下用途：

1）列出 vmlinux 和所有模块的导出函数

2.列出所有符号的 CRC 校验值

modules.order 文件记录了 Makefile 中模块出现的顺序，modprobe 通过它来确定解决多个模块匹配的别名（指定模块的绝对路径）。以下是 modules.order 文件内容：

```
kernel//root/Downloads/developdrivers/1th_hello/hello.ko
```

hello.mod.c 文件产生了 ELF(linux 所采用的可执行/可连接的文件格式)的 2 个节，_module 和.modinfo。以下是 hello.mod.c 文件内容：

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
  .name = KBUILD_MODNAME,
  .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
  .exit = cleanup_module,
#endif
  .arch = MODULE_ARCH_INIT,
};

static const struct modversion_info ____versions[]
__used
__attribute__((section("__versions"))) = {
        { 0x2dec7b1e, "module_layout" },
        { 0x27e1a049, "printk" },
};

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";
```

将生成 hello.ko 用 tftp 传入单板机：

```
tftp -r hello.ko -g 193.169.2.215
```

在单板机上运行：

```
[root@Loongson:/]#insmod hello.ko
Hello_init!
Hello,world!
```

再卸载模块：

```
[root@Loongson:/]#rmmod hello.ko
rmmod: can't change directory to '/lib/modules': No such file or directory
```

错误说明没有该目录。查找原因为 busybox 的配置，在 3.4.1 配置和编译 busybox 一节中，采用如下配置，其中 Module Utilities 为简化配置，没有 rmmod 等选项。

```
--- Applets
Linux Module Utilities    --->
[*] Simplified modutils
[*]      Accept module options on modprobe command line (NEW)
[*]      Skip loading of already loaded modules (NEW)
--- Options common to multiple modutils
[ ] Try to load module from a mmap'ed area
(/lib/modules) Default directory containing modules
(modules.dep) Default name of modules.dep
```

现在采用配置选项如下：

```
[*] modinfo
[ ] Simplified modutils
[*]      insmod
[*]      rmmod
```

```
[*]     lsmod
[*]         Pretty output
[*]     modprobe
[*]         Blacklist support
[*]     depmod
--- Options common to multiple modutils
[ ] Support version 2.2/2.4 Linux kernels
[ ] Try to load module from a mmap'ed area
[*] Support tainted module checking with new kernels
[*] Support for module.aliases file
[*] Support for module.symbols file
(/lib/modules) Default directory containing modules
(modules.dep) Default name of modules.dep
```

现在重新编译 busybox,制作根文件系统并下载至单板机，如《3.4.1 配置和编译 busybox》进行操作。

现在运行加载、显示、卸载命令及结果如下：

```
[root@Loongson:/]#insmod hello.ko
Hello_init!
Hello,world!
[root@Loongson:/]#lsmod
Module                  Size    Used by     Not tainted
hello                   587     0
[root@Loongson:/]#rmmod hello.ko
Hello_exit!
Goodbye,world!
```

模块能够正常地加载与卸载。

6.5 内核配置驱动

内核有很多文件，但不是每个文件都是必须编译的，这就是内核裁剪。内核裁剪是通过配置内核编译选项来裁剪的。在源码根目录下使用命令进行配置。

```
make ARCH=mips CROSS_COMPILE=mipsel-linux- menuconfig
```

其中驱动模块也能像配置菜单那样，将驱动配置到内核中。配置步骤如下：

1）拷备 hello.c 到源码目录的 drivers/char 目录下。

2）修改 drivers/char 目录下的 Konfig，添加代码：

```
config HELLO_MODULE
        tristate " Hello test module "
        depends on LS1C_MACH
        help
          This is first driver for hello
```

依赖定义 depends on 或 requires 指此菜单的出现是否依赖于另一个定义。

这样当 make menuconfig 时 ，将会出现 Hello test module 选项。该 HELLO_MODULE 配置项只对 LS1C_MACH 处理器有效，即只有在选择了 LS1C_MACH，该菜单才可见(可配置)。LS1C_MACH 处理器的选择在内核目录下/Arch/Mips/Loongson/Kconfig 文件中定义

```
config LS1C_MACH
        bool "Loongson 1C board"
        select BOOT_ELF32
        select CEVT_R4K if ! MIPS_EXTERNAL_TIMER
        select CSRC_R4K if ! MIPS_EXTERNAL_TIMER
        select SYS_HAS_CPU_LOONGSON1C
#       select CPU_HAS_WB
        select DMA_NONCOHERENT
        select IRQ_CPU
        select SYS_HAS_EARLY_PRINTK
        select SYS_SUPPORTS_32BIT_KERNEL
        select SYS_SUPPORTS_HIGHMEM
        select SYS_SUPPORTS_LITTLE_ENDIAN
        select USB_ARCH_HAS_HCD
        help
```

```
        GZ Loongson 1C family machines utilize the ls232 revision of
        Loongson processor.
```

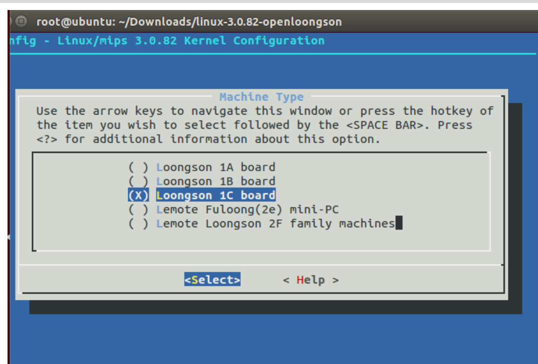


图 6.2 虚拟机中驱动配置选项

3）修改 driver/char 目录下的 Makefile，添加内容：

```
obj-$(CONFIG_HELLO_MODULE)      += hello.o
```

当运行 make menucofig 时，会发现 Hello test module 选项，假如选择了此项。该选择就会保存在.config 文档中。当编译内核时，将会读取.config 文档，当发现 HELLO_MODULE 选项为 yes 时，系统在调用/driver/char/下的 makefile 时，将会把 hello.o 加入到内核中。即可达到编译时内核的目的。

4）回到 linux 源码的根目录，使用 make menuconfig 命令进行配置。

进入 Device Driver-->Character devices-->后可以看如图的配置。

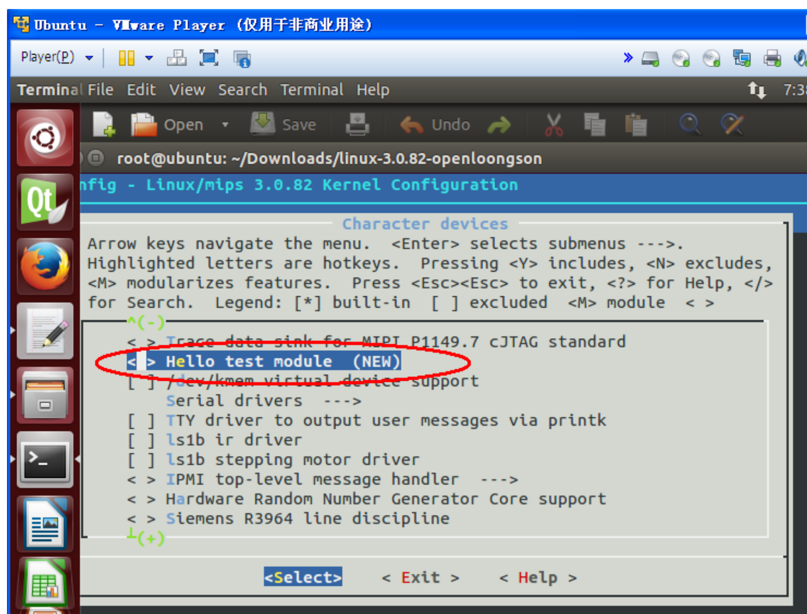


图 6.3 虚拟机中新出现模块选项

保存配置后，如上图所示保存为<M>，即以模块方式加载进内核，如果保存为<*>，表示编译进内核。输入 make modules 命令就可以编译模块。

```
root@ubuntu:~/Downloads/linux-3.0.82-openloongson# make ARCH=mips CROSS_COMPILE=mipsel-linux- modules
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
  CALL    scripts/checksyscalls.sh
<stdin>:1554:2: warning: #warning syscall sendmmsg not implemented
  Building modules, stage 2.
  MODPOST 3 modules
  LD [M]  drivers/char/hello.ko
```

然后 make modules_install，把编译好的模块拷贝到系统目录下（一般是/lib/modules/）；

```
root@ubuntu:~/Downloads/linux-3.0.82-openloongson# make ARCH=mips CROSS_COMPILE=mipsel-linux-
```

```
modules_install
  INSTALL drivers/char/hello.ko
  INSTALL drivers/scsi/scsi_wait_scan.ko
  INSTALL drivers/usb/gadget/g_mass_storage.ko
  DEPMOD   3.0.82
```

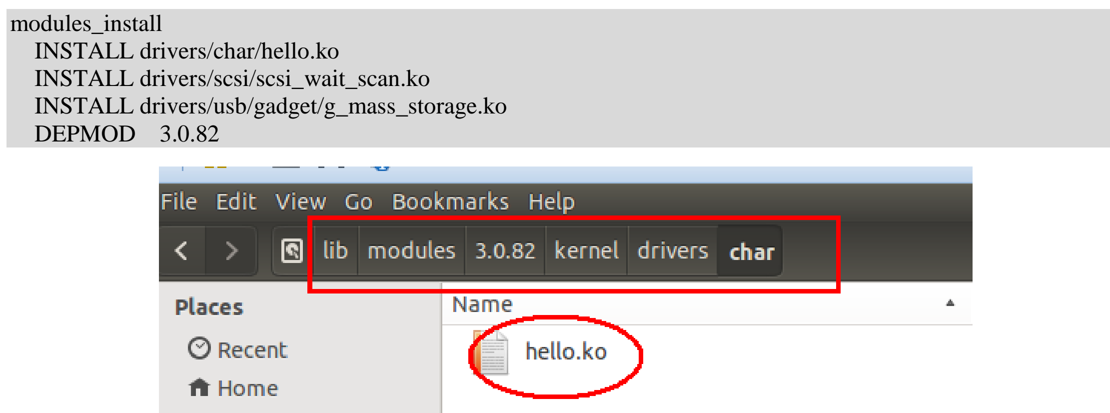


图 6.4 虚拟机中编译成功 ko 模块

最后生成的模块文件（.KO）就在\lib\modules***\下；找到需要添加的模块文件，如果是开发板的话就将此文件下载到你的开发板；最后 insmod 模块名，就可以动态加载模块。

注：在 make modules 命令之后用，即可编译内核模块。拷贝到系统目录下的目的是方便使用。一般加载驱动使用 modprobe ***命令，该命令从系统目录下查找名为***的模块。其实也可以不做 make modules_install，但是这样的话就需要每次手动从编译目录里运行 insmod。运行 modules_install 的另一个作用是会运行 depmod 去生成 modules.dep 文件，该文件记录了模块之间的依赖关系。这样当运行 modprobe ***（自动处理可载入模块）的时候就能够把***所依赖的模块一并加载。

make 写好的驱动程序之后会生成.ko 文件，此 ko 文件就是编译之后生成的模块文件，也就是 Makefile 文件中 obj-m 缩生成的文件，然后需要将此文件加载到模块，即使用 insmod 或者 modprobe 命令将生成的模块文件(.ko 文件加载进内核)，但是此时所写好的应用程序还是不能运行，需要在/dev 下创建设备节点，手动创建设备节点的方法是：mknod 设备节点名称 设备类型 主设备号 次设备号。例如：mknod memdev c 260 0，创建好之后会在/dev 目录下看到一个字符设备名字为 memdev 的类型。然后此时应用程序才能正常运行。

此处讲述的就是不用使用 mknod 手动创建设备节点，采用的是在源文件中添加代码使之加载模块的时候自动创建设备节点，从而达到高效的作用。

在源程序中添加头文件#include <linux/device.h>,需要使用头文件里面的 struct class 结构,class_create、device_create、device_destroy、class_destroy 函数，具体使用方法参见头文件，在此不再赘述。

chr_dev_init(void)函数。可以看到，所有字符设备的初始化函数（IDE_INT_init 之类）都要添加在这里。

自动创建设备节点见附录 4，附录 5。

6.6 led 子系统剖析

透过 LED 驱动研究驱动编写及使用的过程。来自于以下博文，根据智龙作了适当修改：http://blog.csdn.net/shiyi_2012/article/details/7456165

menuconfig 中定义：

```
[*]   LED Class Support
      *** LED drivers ***
<*>   LED Support for GPIO connected LEDS
[*]     Platform device bindings for GPIO LEDs
[*]   LED Trigger support
      *** LED Triggers ***
<*>   LED Timer Trigger
<*>   LED Heartbeat Trigger
```

```
<*>   LED backlight Trigger
<*>   LED GPIO Trigger
<*>   LED Default ON Trigger
```

平台文件 ls1c300a_openloongson_v2.0_platform.c 添加对 LED 驱动的支持：

```
#if defined(CONFIG_LEDS_GPIO) || defined(CONFIG_LEDS_GPIO_MODULE)
#include <linux/leds.h>
struct gpio_led gpio_leds[] = {
	{
		.name			= "led_green",
		.gpio			= 50,
		.active_low		= 1,
//		.default_trigger	= "timer", /* 瑙﹀彂鏂瑰紡 */
		.default_trigger	= NULL,
		.default_state		= LEDS_GPIO_DEFSTATE_ON,
	}, {
		.name			= "led_yellow",
		.gpio			= 51,
		.active_low		= 1,
//		.default_trigger	= "heartbeat", /* 瑙﹀彂鏂瑰紡 */
		.default_trigger	= NULL,
		.default_state		= LEDS_GPIO_DEFSTATE_ON,
	}, {
		.name			= "led_blue",
		.gpio			= 52,
		.active_low		= 1,
//		.default_trigger	= "timer",  /* 瑙﹀彂鏂瑰紡 */
		.default_trigger	= NULL,
		.default_state		= LEDS_GPIO_DEFSTATE_ON,
	},	{
		.name			= "led_red",
		.gpio			= 53,
		.active_low		= 1,
//		.default_trigger	= "timer",  /* 瑙﹀彂鏂瑰紡 */
		.default_trigger	= NULL,
		.default_state		= LEDS_GPIO_DEFSTATE_ON,
	}, {
		.name			= "led_orange",
		.gpio			= 54,
		.active_low		= 1,
//		.default_trigger	= "timer",  /* 瑙﹀彂鏂瑰紡 */
		.default_trigger	= NULL,
		.default_state		= LEDS_GPIO_DEFSTATE_ON,
	},
};

static struct gpio_led_platform_data gpio_led_info = {
	.leds		= gpio_leds,
	.num_leds	= ARRAY_SIZE(gpio_leds),
};

static struct platform_device leds = {
	.name		= "leds-gpio",
	.id	= -1,
	.dev	= {
		.platform_data	= &gpio_led_info,
	}
};
#endif //#if defined(CONFIG_LEDS_GPIO) || defined(CONFIG_LEDS_GPIO_MODULE)
```

在 kconfig 文件中：

```
config LEDS_GPIO   //相当于定义了 LEDS_GPIO
	tristate "LED Support for GPIO connected LEDs"//在 menuconfig 中显示了引号中的内容
	The code to use these bindings can be selected below.
```

```
config LEDS_GPIO_PLATFORM //相当于定义了  LEDS_GPIO_PLATFORM
	bool "Platform device bindings for GPIO LEDs"//在 menuconfig 中显示
```

Makefile 中

```
# LED Core
obj-$(CONFIG_NEW_LEDS)			+= led-core.o
obj-$(CONFIG_LEDS_CLASS)		+= led-class.o
obj-$(CONFIG_LEDS_TRIGGERS)		+= led-triggers.o
```

写之前，先看一张图：

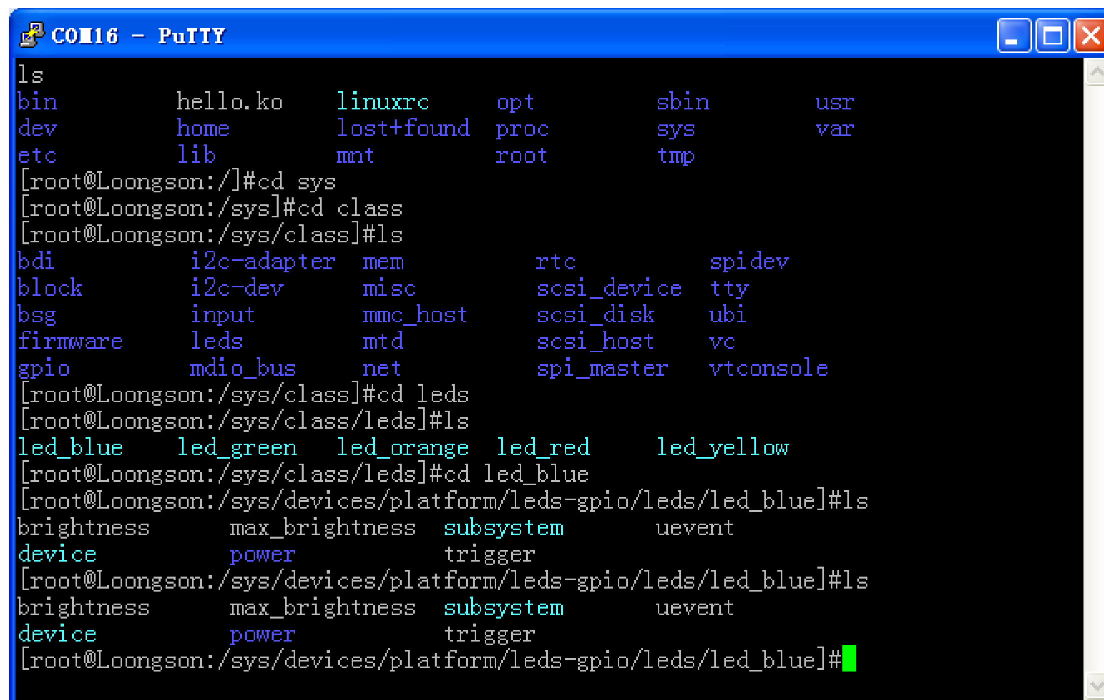


图 6.5 开发板控制台显示 LED 平台设备界面

关于 LED 驱动程序，Linux 自身携带了 LED 驱动，且是脱离平台的，即 LED 子系统。操作起来十分简单。但是它的实质却不是那么容易，其中最重要的 led-class.c 文件。

LED 的驱动位于内核 driver/leds 目录下。核心文件有：led-class.c leds-s3c24xx.c、leds-gpio.c。先看其中一个文件 led-core.c 文件。

一看就知道和类 class 脱不了关系。class 有何作用呢？首先何所谓类呢？就是一组设备具有共同性而抽象出来的。 leds-gpio.c 的功能是在 class 里面实现的。在 led-class.c 文件实现的功能总的来说就是先建立一个类 leds，然后在该类下建立一个设备节点，最后就在该设备节点下载建立几个属性文件。而建立类的交给函数 leds_init(void)来完成，而在该类下建立设备节点，以及在该节点下建立属性文件，并对属性文件实现读写操作。

先看看第一个，就是 init 加载文件，第一句也是核心句，就是建立一个类 leds。并且函数返回值赋值给了 leds_class，最终在设备注册时给了 led_cdev->dev。

在 led_class.c 中实现了 LED 模块的初始化

LED 的初始化函数如下：

```
static int __init leds_init(void)
{
	leds_class = class_create(THIS_MODULE, "leds");//为设备创建一个 class
	if (IS_ERR(leds_class))
		return PTR_ERR(leds_class);
	leds_class->suspend = led_suspend;
	leds_class->resume = led_resume;
	leds_class->dev_attrs = led_class_attrs;
	return 0;
}
```

```
static void __exit leds_exit(void)
{
	class_destroy(leds_class);
}
```

由于 leds_class = class_create(THIS_MODULE, "leds")，这个将在 sys 目录下产生文件即产生 leds 类的文件名，第一个参数指定所属的模块，第二个指定了设备的名字。

而接下来的，第二句 IS_ERR（leds-class）就是判断 leds-class 是否正确产生。接下来的都是函数指针。leds_class->suspend = led_suspend 等是函数指针，上面有具体函数实现。其中 suspend()是在设备休眠时调用，resume()是恢复设备时调用。第一个函数 suspend()函数的实现，其实它就是调了 led_suspend (led_cdev, 0)函数，

```
static int led_suspend(struct device *dev, pm_message_t state)
{
	struct led_classdev *led_cdev = dev_get_drvdata(dev);

	if (led_cdev->flags & LED_CORE_SUSPENDRESUME)//LED_CORE_SUSPENDRESUME=(1 << 16)
		led_classdev_suspend(led_cdev);

	return 0;
}
```

这里定义了一个结构体 led_cdev：

```
struct led_classdev {
	const char		*name;
	int			brightness;
	int			max_brightness;
	int			flags;

	/* Lower 16 bits reflect status */
#define LED_SUSPENDED		(1 << 0)
	/* Upper 16 bits reflect control information */
#define LED_CORE_SUSPENDRESUME	(1 << 16)

	/* Set LED brightness level */
	/* Must not sleep, use a workqueue if needed */
	void		(*brightness_set)(struct led_classdev *led_cdev,
					  enum led_brightness brightness);
	/* Get LED brightness level */
	enum led_brightness (*brightness_get)(struct led_classdev *led_cdev);

	/*
	 * Activate hardware accelerated blink, delays are in milliseconds
	 * and if both are zero then a sensible default should be chosen.
	 * The call should adjust the timings in that case and if it can't
	 * match the values specified exactly.
	 * Deactivate blinking again when the brightness is set to a fixed
	 * value via the brightness_set() callback.
	 */
	int		(*blink_set)(struct led_classdev *led_cdev,
				     unsigned long *delay_on,
				     unsigned long *delay_off);

	struct device		*dev;
	struct list_head	node;			/* LED Device list */
	const char		*default_trigger;	/* Trigger to use */

	unsigned long		blink_delay_on, blink_delay_off;
	struct timer_list	blink_timer;
	int			blink_brightness;

#ifdef CONFIG_LEDS_TRIGGERS
	/* Protects the trigger data below */
	struct rw_semaphore	trigger_lock;

	struct led_trigger	*trigger;
```

```
        struct list_head     trig_list;
        void                 *trigger_data;
#endif
};
```

这个 led_suspend (led_cdev, 0)函数是数据结构体 led_classdev 里的成员，其中 brightness_set 是指向一个函数，在 leds-gpio.c 里 create_gpio_led()时，指向 gpio_led_set 函数。

```
static int __devinit create_gpio_led(const struct gpio_led *template,
        struct gpio_led_data *led_dat, struct device *parent,
        int (*blink_set)(unsigned, int, unsigned long *, unsigned long *))
{
……
        led_dat->cdev.brightness_set = gpio_led_set;
……
}
```

create_gpio_led()函数是在 gpio_led_probe ()中调用

```
static int __devinit gpio_led_probe(struct platform_device *pdev)
{
        ……
                ret = create_gpio_led(&pdata->leds[i],
                                &priv->leds[i],
                                &pdev->dev, pdata->gpio_blink_set);
        ……
}
```

gpio_led_probe 在结构体中定义：

```
static struct platform_driver gpio_led_driver = {
        .probe          = gpio_led_probe,
        .remove         = __devexit_p(gpio_led_remove),
        .driver         = {
                .name   = "leds-gpio",
                .owner  = THIS_MODULE,
                .of_match_table = of_gpio_leds_match,
        },
};
```

而以上结构体是在平台驱动注册时就已经定义：

```
static int __init gpio_led_init(void)
{
        return platform_driver_register(&gpio_led_driver);
}

static void __exit gpio_led_exit(void)
{
        platform_driver_unregister(&gpio_led_driver);
}
module_init(gpio_led_init);
module_exit(gpio_led_exit);
```

总而来说，实现的功能是设备挂起时候，就 level 赋一个值，0 还是 1 就根据 active_low 的选择。

```
void led_classdev_suspend(struct led_classdev *led_cdev)
{
        led_cdev->flags |= LED_SUSPENDED;
        led_cdev->brightness_set(led_cdev, 0);
}
```

调用以下函数：

```
        void            (*brightness_set)(struct led_classdev *led_cdev,
                                  enum led_brightness brightness);
```

现在说一下 led_resume ()，其实也等同上面一样，最终用 led_cdev->brightness 赋值给 level。

```
static int led_resume(struct device *dev)
{
        struct led_classdev *led_cdev = dev_get_drvdata(dev);
```

```
	if (led_cdev->flags & LED_CORE_SUSPENDRESUME)
		led_classdev_resume(led_cdev);

	return 0;
}
```

到这里 leds_init 函数就完成了，最后通过 subsys_initcall(leds_init)使得 leds_init 在系统启动时候就会被初始化。

```
subsys_initcall(leds_init);	//定义 MODULE 的情况下对 subsys_initcall 的定义，等价于使用 module_init,见附录 1 -9)
module_exit(leds_exit);
```

总结一下，leds_init 函数在系统启动的时候就会被调用。实现的功能就是在 sys/class 目录下生成 leds 类目录，还有就是实现挂起和恢复时候，执行 brightness_set(led_cdev, *)函数。

再看下 gpio_led_set 作了什么工作：

```
static void gpio_led_set(struct led_classdev *led_cdev,
	enum led_brightness value)
{
	struct gpio_led_data *led_dat =
		container_of(led_cdev, struct gpio_led_data, cdev);
	int level;

	if (value == LED_OFF)
		level = 0;
	else
		level = 1;

	if (led_dat->active_low)
		level = !level;

	/* Setting GPIOs with I2C/etc requires a task context, and we don't
	 * seem to have a reliable way to know if we're already in one; so
	 * let's just assume the worst.
	 */
	if (led_dat->can_sleep) {
		led_dat->new_level = level;
		schedule_work(&led_dat->work);
	} else {
		if (led_dat->blinking) {
			led_dat->platform_gpio_blink_set(led_dat->gpio, level,
							NULL, NULL);
			led_dat->blinking = 0;
		} else
			gpio_set_value(led_dat->gpio, level);
	}
}
```

gpio_set_value 函数中，通过

```
*led_dat =		container_of(led_cdev, struct gpio_led_data, cdev);
```

获得 led_dat ，再通过

```
gpio_set_value(led_dat->gpio, level);
```

设置 led 的值。

接下来就主要剩下 led_classdev_register 函数。

前面说了就是产生几个文件。其中第一个就是设备节点。该函数第一句

```
led_cdev->dev = device_create(leds_class, parent, 0, led_cdev, "%s", led_cdev->name);
```

函数原型

```
device_create(struct class *class, struct device *parent,dev_t devt, void *drvdata, const char *fmt, ...)，
```

代码实现的一个主要函数是

dev = device_create_vargs(class, parent, devt, drvdata, fmt, vargs)，实现在对应目录下产生

设备节点。先看看各个参数意思，第一个参数指定所要创建的设备所从属的类，第二个参数是这个设备的父设备，如果没有就指定为 NULL，第三个参数是设备号，第四个参数是设备名称，第五个参数是从设备号。看一下实参，第一个实参 leds_class，这个在哪里出现的呢？在前面 leds_init 函数的建立类的返回值，所以其实就是在前面的 leds 类下建立设备文件节点。

现在先把代码剖出来：

```
int led_classdev_register(struct device *parent, struct led_classdev *led_cdev)
{
	led_cdev->dev = device_create(leds_class, parent, 0, led_cdev,
				      "%s", led_cdev->name);
	if (IS_ERR(led_cdev->dev))
		return PTR_ERR(led_cdev->dev);

#ifdef CONFIG_LEDS_TRIGGERS
	init_rwsem(&led_cdev->trigger_lock);
#endif
	/* add to the list of leds */
	down_write(&leds_list_lock);
	list_add_tail(&led_cdev->node, &leds_list);
	up_write(&leds_list_lock);

	if (!led_cdev->max_brightness)
		led_cdev->max_brightness = LED_FULL;

	led_update_brightness(led_cdev);

	init_timer(&led_cdev->blink_timer);
	led_cdev->blink_timer.function = led_timer_function;
	led_cdev->blink_timer.data = (unsigned long)led_cdev;

#ifdef CONFIG_LEDS_TRIGGERS
	led_trigger_set_default(led_cdev);
#endif

	printk(KERN_DEBUG "Registered led device: %s\n",
			led_cdev->name);

	return 0;
}
```

第一步：调用了 device_register 函数向系统注册 了一个 struct device（可以在/sys/class/leds/目录下找到这个设备，比如 lcd 背光驱动的 lcd-backlight）。

第二步：加入 led_cdev 加入到链表 leds_list 中。

第三步： 调用 led_update_brightness 函数更新 led 亮度值。

第四步： 调用 init_timer 初始化一个定时器。

这个函数使用 EXPORT_SYMBOL_GPL(led_classdev_register)导出来，在 leds-gpio.c 的 gpio_led_probe()函数中调用，并注册 LED 类设备，下面溯源。

侦测函数(在 leds-gpio.c 中)：

```
gpio_led_probe(struct platform_device *pdev)
```

调用：

```
ret = create_gpio_led(&pdata->leds[i], &leds_data[i],	&pdev->dev, pdata->gpio_blink_set);
```

而 create_gpio_led(const struct gpio_led *template,struct gpio_led_data *led_dat, struct device *parent,int (*blink_set)(unsigned, unsigned long *, unsigned long *))

就调用 ret = led_classdev_register(parent, &led_dat->cdev);

前面说了建立设备节点，现在再细讲一下。主要看看 parent 指向的是 pdev->dev。由于 pdev 是平台设备，所以得关系到平台设备问题。其中 pdev->dev 对应平台设备下的设备。所以呢，对于设备节点的设备名，跟踪一下 leds-gpio.c 代码，就知道：

leds-gpio.c 里的 cdev 就对应上面的 led_cdev

```
led_dat->cdev.name = template->name;    //在函数 creat_gpio_led
```

而 template 又对应于 pdata->leds[i]，

struct gpio_led_platform_data *pdata = pdev->dev.platform_data;

简而言之，就是在 gpio_led_probe 函数中，获取平台信息 platform_data，作为参数传给函数 creat_gpio_led 的 template 参数，最后通过该参数付给了 led_dat->cdev.name。

所以创建的 dev 节点的名字由平台设备的信息决定的。

现在再来看看在哪里生成属性文件，看函数：

```
device_create_file(led_cdev->dev, &dev_attr_brightness);
```

主要是看参数 led_cdev->dev，这个又是指向哪里，其实就是建立设备节点时候的返回值，可以看看上面。所以就在设备节点目录下建立属性文件，当然后面几个建立属性文件都一样。

说道这里，led-class.c 就完啦，剩下没讲的函数要不就是属性读写函数，要不就是卸载函数。总结一下，一般来说，用 class_creat 添加类，在 device_creat 在类目录下建立设备文件，还可以在设备节点下建立属性文件，实现对设备的操作，但是该操作一般就是读写，是通过命令实现的。

那张图就展示了在 class 下建立 5 个设备节点：led_blue…等，每个设备节点下建立属性文件，其中有一个 brigtness，往这个文件执行命令,cat 是读出,echo 是写入，如：我的板子执行 echo 1 >brightness 时候，第一颗灯亮。执行 echo 0 >brightness 时候，第一颗灯不亮。对于为什么会亮，会什么又会灭，在 LED 移植篇再讲解。

接下来 leds-gpio.c，里面主要就是 platform 模型，即涉及到存放什么硬件资源内核，怎么存放，然后又怎么去取出来。

6.7 led_trigger 接口分析

源码位置：/driver/leds/led-triggers.c。

1）注册触发器

```
int led_trigger_register(struct led_trigger *trigger);
```

这个函数注册的 trigger 会被加入全局链表 **trigger_list**，这个链表头是在 /driver/leds/led-triggers.c 定义的。

此外，这个函数还会遍历所有的已注册的 led_classdev，如果有哪个 led_classdev 的默认触发器和自己同名，则调用 led_trigger_set 将自己设为那个 led 的触发器。

led_classdev 注册的时候也会调用 led_trigger_set_default 来遍历所有已注册的触发器，找到和 led_classdev.default_trigger 同名的触发器则将它设为自己的触发器。

2）注销触发器

```
void led_trigger_unregister(struct led_trigger *trigger);
```

这个函数做和注册相反的工作，并把所有和自己建立连接的 led 的 led_classdev.trigger 设为 NULL。

3）设置触发器上所有的 led 为某个亮度

```
void led_trigger_event(struct led_trigger *trigger, enum led_brightness brightness);
```

4）注册触发器的简单方法

指定一个名字就可以注册一个触发器，注册的触发器通过**tp 返回，但是这样注册的触发器没有 active 和 deactivede。

```
void led_trigger_register_simple(const char *name, struct led_trigger **tp);
```

相对应的注销函数为：

```
void led_trigger_unregister_simple(struct led_trigger *trigger);
```

5）触发器和 led 的连接

```
void led_trigger_set(struct led_classdev *led_cdev, struct led_trigger *trigger); //建立连接。建立连接的时候会调用触发器的 activate 方法
void led_trigger_remove(struct led_classdev *led_cdev); //取消连接。取消连接的时候会调用触发器的 deactivate 方法
void led_trigger_set_default(struct led_classdev *led_cdev); //在所有已注册的触发器中寻找 led_cdev 的默认触发器并调用 led_trigger_set 建立连接
```

总结一下 led、led_classdev、led_trigger 的关系：

图 6.5 led、led_classdev、led_trigger 的关系图

trigger 好比是控制 LED 类设备的算法，这个算法决定着 LED 什么时候亮什么时候暗。LED trigger 类设备可以是现实的硬件设备，比如 IDE 硬盘，也可以是系统心跳等事件。

中级篇应用

7. Linux 应用编程

7.1 linux 应用编程的基础知识

Linux 应用编程，需要学习的功底有：C 语言,数据结构，其中重点学习数据结构中的链表，至于树、图这些稍微复杂的数据结构可以稍微了解。

有了上面的基础，再来学习 Linux 应用编程，以后写起应用程序来，能起到事半功倍的效果。Linux 应用编程需要学习的内容有：

1）库函数的熟悉使用，比如 memcpy,memset,strstr,strcpy 等等。

2）SHELL 编程，很多游戏行业，必须熟悉 SHELL 编程。

3）文件 IO 编程，最常用的应用编程。

4）进程编程，进程间通信。

5）多线程编程。

6）网络编程。

7.2 文件 I/O 编程

在 Linux 操作系统里，什么是文件？以前说过，在 Linux 中，几乎一切都可以看做是文件。串口、打印机、硬盘等设备都可以看做是文件，学过驱动的应该都知道 /dev/xxx 都是设备文件。大多数情况下，这些文件都会涉及的函数接口一般有以下 5 个， open，read,write,ioctl,close。目录在 Linux 环境下确实也是一个文件，只不过打开目录文件，用的函数不再是 open/read 而是 opendir/readdir 接口来读取目录。另外，应用程序工程师不必了解上面提到的系统接口函数是如何实现的。总结一下： Linux 中的文件主要分为 4 种：普通文件、 目录文件、链接文件和设备文件。

在 Linux 环境下，所有的设备和文件的操作都使用文件描述符来进行的。文件描述符是一个非负整数，它是一个索引值，并指向内核中每个进程打开的记录表。当打开一个显存或创建一个新文件时，内核就向进程返回一个文件描述符，当需要读写文件时，也需要把文件描述符作为参数传递给相应函数。一般来说，一个进程的启动，都会打开 3 个文件：标准输入、标准输出、标准出错。 这 3 个文件分别对应文件描述为 0、 1、 2（也就是宏替换 STDIN_FILENO、 STDOUT_FILENO 和 STDERR_FILENO）。基于文件描述符的 I/O 操作是 Linux 中最常用的操作之一。

不带缓存的 IO 操作又称底层 IO 操作。文件底层 I/O 操作的系统调用主要用 到 5 个函数： open()、 close()、 read()、 write()、 lseek()。这些函数的特点是不带缓存的，直接对文件进行操作。

函数说明：

表 7.1 文件底层 I/O 操作函数

名称	作　　用
open	打开或者创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

read	从指定文件描述符中读出数据放到缓存区中，并返回实际读入的字节。
write	将缓存区中的数据写到指定文件描述符中，并返回实际写入的字节数。
lseek	在指定文件描述符中将文件指针定位到相应的位置。
close	关闭指定文件描述符的文件。

函数参数

在 Linux 系统中，拥有非常多的 API 接口，这些函数接口有非常多的参数，参数中又有很多的选择，如何记住这些参数？常用的函数，用多了就记住了。三种最常用的方法：

例如要查询 open()函数有哪些参数，参数的用法。

方法一、 在 linux 命令行中，使用 man open（有些 API 函数是使用 man 2 xxx）

方法二、在谷歌上，直接搜索想要查询的函数。

方法三、 去查阅书籍，看 API 函数用法，或者直接看书籍上的源码。

实例源码：fileio.c，熟练运用文件 I/O 中 open、 close、 write、 read、 lseek 函数的操作。首先打开当前目录下的 hello.c 文件，如果此文件不存在则先创建，接着写入“hello,I'm Loongson,This is file io test!”，此时文件指针位于文件尾部，接着再使用 lseek 函数将文件指针移到文件开始处，并读出 15 个字节，并将其打印出来。

```
/* fileio.c*/
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc ,char **argv)
{
	int fd,size;
	char *buf = "hello,I'm Loongson,This is    file io test!";
	char buf_r[20];
	int len = strlen(buf);

	/*  首先调用 open 函数，如果当前目录下没有 hello.c 则创建
	 *  以读写方式打开文件，并指定权限为可读可写
	 */
	if((fd = open("./hello.c",O_CREAT | O_TRUNC | O_RDWR,0666))<0)
	{
		/*  错误处理  */
		printf("open fail\n");
		exit(1);
	}
	else
		printf("open file:hello.c fd = %d\n",fd);

	/*  调用 write 函数，将 buf 中的内容写入 hello.c */
	if((size = write(fd,buf,len) ) < 0)
	{
		printf("write fail\n");
		exit(1);
	}
	else
		printf("write: %s\n",buf);

	/*  调用 lseek 函数将文件指针移到文件起始位置，并读出文件中的 15 个字节  */
	lseek(fd,0,SEEK_SET);
	if((size = read(fd,buf_r,15)) <0)
	{
		printf("read fail\n");
		exit(1);
```

```
        }
        else
        {
                buf_r[15] = '\0';
                printf("read from hello.c and the content is %s\n",buf_r);
        }

        /* 最后调用 close 函数关闭打开的文件 */
        if(close(fd) < 0)
        {
                printf("close fail\n");
                exit(1);
        }
        else
                printf("close hello.c\n");

        return 0;
}
```

步骤：编译、下载、运行

编写 Makefile 文件（在虚拟机上）：

```
fileio : fileio.o
        mipsel-linux-gcc -o fileio fileio.o
fileio.o : fileio.c
        mipsel-linux-gcc -c fileio.c -o fileio.o
clean :
        rm fileio.o fileio
```

上述 4 行字必须存为 Makefile 文件，注意第 2、4 行必须以 Tab 键缩进，不能以空格键缩进。放入与 fileio.c 相同目录下，然后执行 make 命令即可编译程序，执行"make clean"可清除编译出来的结果。

将生成的 fileio 通过 tftp 传入开发板，修改权限，运行（在开发板上）。

```
[root@Loongson:/]#tftp -r fileio -g 193.169.2. 215
fileio               100% |*******************************|   9338    0:00:00 ETA
[root@Loongson:/]#chmod u+x fileio
[root@Loongson:/]#./fileio
open file:hello.c fd = 3
write: hello,I'm Loongson,This is    file io test!
read from hello.c and the content is hello,I'm Loong
close hello.c
[root@Loongson:/]#ls
bin            fileio         lib            mpu6050.ko     sys            usr
dev            hello.c        linuxrc        opt            testi2c        var
devdemo.ko     home           lost+found     proc           testmpu6050    virtdev.ko
ds3231.ko      i2ctools       memdev.ko      root           tmp
etc            ledoff         mnt            sbin           tools
[root@Loongson:/]#cat hello.c
hello,I'm Loongson,This is    file io test![root@Loongson:/]#
```

结果分析：

文件描述符为 3，是因为通常每一个进程都首先打开 0、1、2 文件，也就是标准输入、标准输出、标准出错处理，所以这里打开的 fd 为 3。

7.3 进程和线程

进程是系统中程序执行和资源分配的基本单位。相应地，线程是一个进程内的基本调度单位，也可以成为轻量级进程。线程是在共享内存中并发的多道执行路径，它们共享一个进程的资源，如文件描述符和信号处理。因此，就大大减少了上下文的切换开销。一个进程内的多线程共享一个用户地址空间。由于线程共享了进程的资源和地址空间。因此，任何线程对系统资源的操作都会给其他线程带来影响，这样一来就要实现多线程之间的同步。在多线程系统中，线程与进程的关系如图 7.1 示。

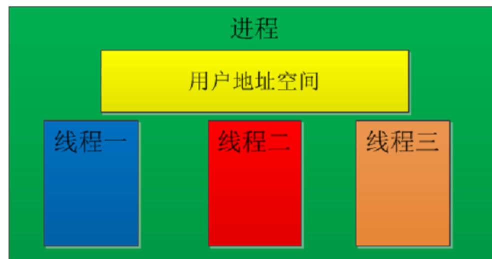


图 7.1 线程、进程关系示意图

7.4 多进程操作

进程是一个程序一次执行的过程。它和程序的本质区别是，程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念。而进程是一个动态的概念，它是程序执行的过程，包含了动态创建、调度和消亡的整个过程。它是程序执行和资源管理的最小单位。因此，对系统而言，当用户在系统中键入命令执行一个程序的时候，它将启动一个进程。

进程也是用一系列数字来代表进程号的。这个进程号称之为 PID ，对应地父进程号就是 PPID 。其中 PID 唯一地标识一个进程，它们都是非零的正整数。

在 Linux 中获得当前进程的 PID 和 PPID 的系统调用函数为 getpid 和 getppid， 下面简单示例 getpid.c ：(后面举例便不再编写 Makefile 文件 ，可采用 5.简单应用编程中的命令)。

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main(void)
{
printf("The PID of the process is:%d\n",getpid());
printf("The PPID of the process is:%d\n",getppid());
return 0;
}
```

虚拟机中编译：

```
mipsel-linux-gcc getpid.c -o getpid
```

开发板中运行结果如下：

```
[root@Loongson:/]#./getpid
The PID of the process is:70
The PPID of the process is:50
```

进程是程序的执行过程，根据它的声明期可以划分为三种状态：

执行态：该进程正在执行，即进程正在占用 CPU。

就绪态：进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间片。

阻塞态: 进程还不具备占用 CPU 的权力，若等待时间发生可将其唤醒。

这三种状态之间有四种可能的转换关系：

1)执行态->就绪态

2)执行态->阻塞态

3)就绪态->执行态

4)阻塞态->就绪态

它们之间的转换关系图如下图 7.2 所示。注意：阻塞态是不能直接跳跃到执行态的。

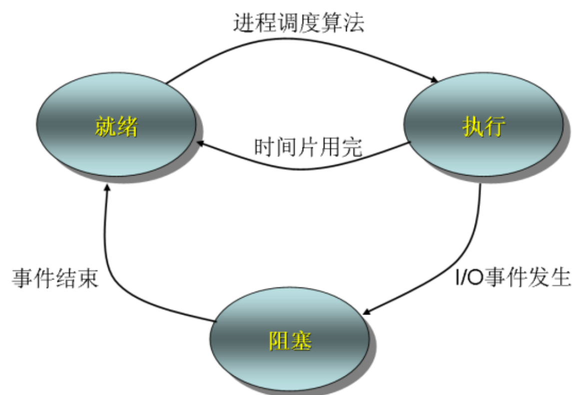


图 7.2 进程三种状态转换示意图

在进程中使用 fork 函数，会创建一个新进程，新的进程称为子进程，原来的进程则称为父进程。这函数看起来没什么特别的， fork 函数的特别之处在于函数执行一次却返回两个值。重点是，父进程返回的是子进程的进程号，而子进程返回 0。因此，可以通过返回值来判定该进程是父进程还是子进程。

fork 函数实例 fork.c：

```
/*fork.c*/
#include<unistd.h>
#include<stdio.h>
#include<error.h>
#include<stdlib.h>
int main()
{
	pid_t result;
	result = fork();
	if(result==-1)
	{
		perror("fork:");
		exit(1);
	}
	else if(result==0)
	{
		printf("Current value is %d. In child process, child PID = %d\n",result,getpid());
	}
	else
	{
		printf("Current value is %d. In father process, father PID = %d\n",result,getppid());
	}
	return 0;
}
```

虚拟机中编译：

```
mipsel-linux-gcc fork.c -o fork
```

运行结果如下：

```
[root@Loongson:/]#./fork
Current value is 76. In father process, father PID = 50
[root@Loongson:/]#Current value is 0. In child process, child PID = 76
```

从实例中可以看出，使用 fork 函数新建了一个子进程，其中父进程返回子进程的 PID = 76，而子进程的返回值为 0。

7.5 进程间的通信

进程是一个程序的一次执行的过程。这里所说的进程一般是指运行在用户态的进程，而由于处于用户态的不同进程之间是彼此通信的。例如：一个进程把数据写到通信媒介(如管道)上，另一个进程就可以从媒介(如管道)中取出数据。经常听到的管道、消息队列、共享内

存、信号量、套接字都是常见的进程间的通信方式。本节主要讲解前三种通信方式。

7.5.1 管道

管道是 Linux 系统中最古老的进程间通信方式，它是作用是把一个程序的输出直接连接到另一个程序的输入。例如在 shell 中输入命令:ls | more 这条命令的作用是列出当前目录下的所有文件和子目录，如果内容超过一页则自动进行分页。符号“|”就是 shell 为“ ls”和“ more”命令建立的一条管道，它将 ls 的输出直接送进了 more 的输入，如图 7.3 所示：

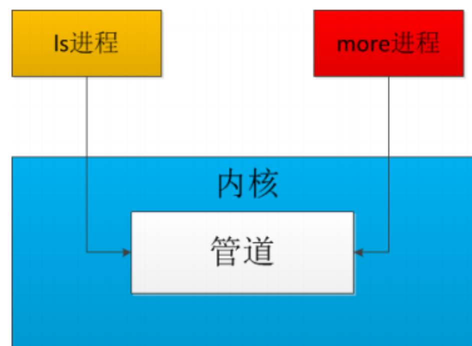


图 7.3 进程与管道关系示意图

1)无名管道

无名管道具有如下特点：

a、它只能用于具有亲缘关系的进程之间通信，例如父子进程或者兄弟进程之间。

b、它是一个半双工的通信模式，具有固定的读端口和写端口。

c、管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 函数。但它不是普通的文件，并不属于其他任何文件系统，并且只存在与内存中。在 Linux 中在文件属性中带有 p(pipe)的文件就是管道文件。一个进程向管道中写的内容被管道的另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

通过 pipe 函数创建无名管道，如果成功创建则打开两个文件描述符，分别是 fd[0] 和 fd[1]，其中 fd[0] 固定用于管道读端， fd[1] 固定用于管道写端。

无名管道的关闭只需要将这两个文件描述符关闭即可，就像关闭普通文件描述符那样通过 close 函数分别关闭各个文件描述符。

举例 pipe.c ：

```
/* pipe.c */
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
	int pipe_fd[2];
	if(pipe(pipe_fd)<0)
	{
		printf("Pipe creat error\n");
		exit(1);
	}
	else
	{
		printf("Pipe creat success\n");
	}
	close(pipe_fd[0]);
	close(pipe_fd[1]);
}
```

虚拟机中编译：

```
mipsel-linux-gcc pipe.c -o pipe
```

程序使用 pipe 函数创建一个无名管道，之后再将其关闭，执行结果如下：

```
[root@Loongson:/]#./pipe
Pipe creat success
[root@Loongson:/]#
```

无名管道读写实例：pipe_rw.c，首先创建无名管道，然后使用 fork 函数创建子进程，通过关闭父进程的读描述符和子进程的写描述符，建立起它们之间的管道通信，最终达到父进程写入数据，子进程读出数据的效果。

```
/* pipe_rw.c */
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main()
{
	int pipe_fd[2];
	pid_t pid;
	char buf_r[100];
	char *p_wbuf;
	int r_num;
	memset(buf_r,0,sizeof(buf_r));

	if(pipe(pipe_fd)<0)
	{
		printf("Creat pipe error\n");
		return 1;
	}
	if((pid=fork())==0)
	{
		close(pipe_fd[1]);
		sleep(2);
		if((r_num=read(pipe_fd[0],buf_r,100))>0)
		{
			printf("%d numbers read from pipe is %s\n",r_num,buf_r);
		}
		close(pipe_fd[0]);
		exit(0);
	}
	else
	{
		close(pipe_fd[0]);
		if(write(pipe_fd[1],"Hello",5)!=-1)
		{
			printf("Pipe write1 success.\n");
		}
		if(write(pipe_fd[1]," pipe",5)!=-1)
		{
			printf("Pipe write2 success.\n");
		}
		close(pipe_fd[1]);
		sleep(3);
		exit(0);
	}
	return 0;
}
```

虚拟机中编译：

```
mipsel-linux-gcc pipe_rw.c -o pipe_rw
```

执行结果如下：

```
[root@Loongson:/]#./pipe_rw
```

```
Pipe write1 success.
Pipe write2 success.
10 numbers read from pipe is Hello pipe
```

2)有名管道

有名管道可以使互不相关的两个进程实现彼此通信。有名管道又称 FIFO，(first In first OUT)即先进先出，对有名管道的读总是从开始处读数据，对它的写则把数据添加至末尾，它不支持 lseek 等文件定位操作。

有名管道的创建使用 mkfifo() 函数，在创建管道成功之后，就可以使用 open、read、write 这些函数了。需要注意的是，对于普通文件进行读写时，不会出现阻塞问题，而读写有名管道就有阻塞的可能。如果需要读写非阻塞，那么在 open 函数中设定为 O_NONBLOCK。

为了证明有名管道能够让任意两个进程之间进行通信，要编写两个程序。 一个用于读取管道中的数据(读进程)，另一个用于写数据到管道(写进程)。

程序源码 fifo_read.c：

```
/* fifo_read.c */
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<errno.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define FIFO "/tmp/myfifo"

int main(int argc,char **argv)
{
	char buf_r[100];
	int fd,nread;
	if((mkfifo(FIFO,O_CREAT|O_EXCL)<0)&(errno!=EEXIST))
	{
		printf("Can not creat fifo\n");
	}
	else
	{
		printf("Creat fifo success\nPreparing for reading bytes..\n");
	}
	memset(buf_r,0,sizeof(buf_r));

	fd=open(FIFO,O_RDONLY|O_NONBLOCK,0);// 非阻塞
	if(fd==-1)
	{
		perror("open:");
		exit(1);
	}
	while(1)
	{
		memset(buf_r,0,sizeof(buf_r));
		if((nread=read(fd,buf_r,100))==-1)
		{
			if(errno==EAGAIN)
			{
				printf("No data yet\n");
			}
		}
		else
		{
			printf("Read %s from FIFO\n",buf_r);
			sleep(1);
		}
	}
	pause();
	ulink(FIFO);
```

```
}
```

fifo_write.c：

```
/* fifo_write.c*/
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<errno.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define FIFO "/tmp/myfifo"

int main(int argc,char **argv)
{
	int fd;
	char buf_w[100];
	int nwrite;
	fd=open(FIFO,O_WRONLY|O_NONBLOCK,0);
	if(fd==-1)
	{
		if(errno==ENXIO)
		printf("open error:no reading process");
	}
	if(argc==1)
	{
		printf("Plese sent something\n");
	}
	strcpy(buf_w,argv[1]);
	if((nwrite=write(fd,buf_w,100))==-1)
	{
		if(errno==EAGAIN)
		printf("The FIFO has not been read yet.Please try latter.\n");
	}
	else
	{
		printf("write %s to the FIFO\n",buf_w);
	}

}
```

虚拟机中编译：

```
mipsel-linux-gcc fifo_read.c -o fifo_read
mipsel-linux-gcc fifo_write.c -o fifo_write
```

为了能够较好地观察实验结果，需要将这读进程在后台运行。首先启动读管道，由于这是非阻塞的，并且写进入还没有启动，所以没有数据给读进程读取。一旦写进程运行后，并将数据写入管道中，读进程马上将数据从管道中读取出来。 执行结果如下：

```
[root@Loongson:/]# ./fifo_read & //运行读进程
[root@Loongson:/]#Creat fifo success
Preparing for reading bytes..
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
Read	from FIFO
```

```
./fifo_write Loongson //运行写进程
write Loongson to the FIFO
[root@Loongson:/]#Read Loongson from FIFO //读进程中读出刚才写入的“Loongson”
Read    from FIFO
Read    from FIFO
```

7.5.2 消息队列

消息队列就是一个消息的列表。用户可以从消息队列中添加消息、读取消息等。消息队列具有一定的 FIFO 特性，但它可以实现消息的随机查询，比 FIFO 具有更大的优势。同时，这些消息又是存在于内核中的，由“队列 ID”来标识。

消息队列的实现包括创建或打开消息队列、添加消息、读取消息和控制消息 队列这四种操作。

表 7.2 消息操作函数

名称	作　　用
msgget	创建或者打开消息队列，消息队列的数据会受系统的限制。
msgsnd	添加消息，将消息添加到已打开的消息队列的末尾。
msgrcv	读取消息队列，可以指定读取某一条消息。
msgctl	控制消息队列

相关函数原型：

```
/* 所需头文件 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
/* 函数原型 */
int msgget(key_t key, int msgflg);
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);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

消息队列实例, msg.c: 如何使用消息队列进行进程间通信，它包括消息队列的创建、消息的发送/读取，消息的撤销等操作。使用了 ftok 函数，是系统 IPC 键值的格式转换函数，它根据 pathname 指定的文件（或目录）名称，以及 proj_id 参数指定的数字，ftok 函数为 IPC 对象生成一个唯一性的键值。

```
/* msg.c */
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define BUFSIZE 512
struct message{
long msg_type;
char msg_text[BUFSIZE];
};
int main()
{
        int qid;
        key_t key;
        int len;
        struct message msg;
        if((key=ftok(".",'a'))==-1)
        {
                perror("ftok:");
                exit(1);
        }
```

```
        if((qid=msgget(key,IPC_CREAT|066))==-1)
        {
                perror("msgget:");
                exit(1);
        }
        printf("Open queue %d\n",qid);
        puts("Please enter the message to the queue:");
        if((fgets((&msg)->msg_text,BUFSIZE,stdin))==NULL)
        {
                puts("no message.");
                exit(1);
        }
        msg.msg_type=getpid();
        len=strlen(msg.msg_text);
        if((msgsnd(qid,&msg,len,0))<0)
        {
                perror("msgsnd:");
                exit(1);
        }
        if(msgrcv(qid,&msg,BUFSIZE,0,0)<0)
        {
                perror("msgrcv");
                exit(1);
        }
        printf("Message is %s\n",(&msg)->msg_text);
        if((msgctl(qid,IPC_RMID,NULL))<0)
        {
                perror("msgctl:");
                exit(1);
        }
        exit(0);
}
```

执行结果如下：

```
[root@Loongson:/]#./msg
Open queue 98304
Please enter the message to the queue:
I'm Loongson,I love Linux.
Message is I'm Loongson,I love Linux.
```

7.5.3 共享内存

内核专门留出了一块内存，可以由需要访问的进程将其映射到自己的私有地址空间，不同进程可以及时看到某进程对共享内存的数据进行更新。采用内存共享通信机制的好处是效率非常高，因为进程可以直接读写内存，不再需要进行数据的拷贝。由于多个进程都可以对共享内存进行读写数据，要引进某种同步机制，如互斥锁和信号量等。

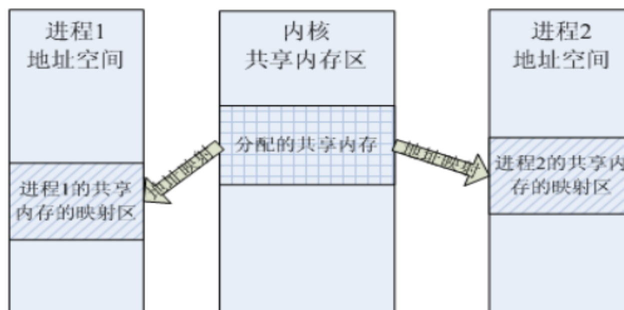


图 7.4 共享内存操作示意图

分为两个步骤，第一步是创建共享内存，使用函数 shmget，也从内存中获得一段共享内存区域。第二步映射共享内存，把这段刚创建的共享内存映射到具体的进程空间去，使用函数 shmat。完成这二步后，就可以使用不带缓存的 I/O 读写命令对其进行操作。如果要撤销映射，使用函数 shmdt 实现。

函数原型：

```
/* 所需头文件 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
/* 函数原型 */
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

共享内存实例：shmadd.c。首先创建一个共享内存区，大小为 2K，然后将其映射到本进程中，最后再解除映射。注意本实例中，使用了 ipcs 命令，它的作用是报告进程间通信机制状态，一般用于查看共享内存、消息队列等各种进程间通信机制的情况，这里巧妙的利用 system 函数用于调用 shell 命令 ipcs。

```
/* shmadd.c */
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h>

#define BUFSIZE 2048
int main()
{
	int shmid;
	char *shmadd;
	if((shmid=shmget(IPC_PRIVATE,BUFSIZE,0666))<0)
	{
		perror("shmget:");
		exit(1);
	}
	else
	{
		printf("Create share memory success,the shmid is:%d\n",shmid);
	}
	system("ipcs -m");
	if((shmadd=shmat(shmid,0,0))<(char *)0)
	{
		peror("shmat:");
		exit(1);
	}
	else
	{
		printf("Mat memory success,the share address is matted to %x\n",shmadd);
	}
	system("ipcs -m");
	if((shmdt(shmadd))<0)
	{
		perror("shmdt:");
		exit(1);
	}
	else
	{
		printf("Delete memory success\n");
	}
	system("ipcs -m");
	exit(0);
}
```

运行结果：

```
[root@Loongson:/]#./shmadd
Create share memory success,the shmid is:0

------ Shared Memory Segments --------
key         shmid       owner       perms       bytes       nattch      status
0x00000000 0            root        666         2048        0
```

```
Mat memory success,the share address is matted to 2abec000

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       666        2048       1

Delete memory success

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       666        2048       0
```

从运行结果可知， nattch 的值随着共享内存状态的变化而变化，共享内存的值根据不同的系统可能不同。

7.6 多线程操作

pthread 线程库是由 POSIX 提供的一套通用的线程库，具有很好的移植性。pthread 线程库使用了内核级线程来完成，目的是为了提高线程的并发性。

7.6.1 线程控制

创建线程用的函数是 pthread_create；线程的退出，一般用 pthread_exit。线程的退出函数使用 pthread_exit，而进程的退出函数使用 exit。如果使用 exit 函数使进程结束，此时进程中的所有线程都会因进程的结束而结束。 pthread_join 函数用于将当前线程挂起，等待线程的结束，它是一个线程阻塞函数，调用它的函数将一直到被等待的线程结束。

```
/* 所需头文件 */
#include <pthread.h>
/* 函数原型 */
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);
```

实例 thread.c。本实例创建两个线程，第一个线程是在程序运行到中途时调用 pthread_exit 主 动退出，然后睡眠 2s；第二个线程正常运行退出。在主进程中收集这两个线程的退出信息，并释放资源。从这个实例可以看出，这两个线程是并发运行的。

```
/* thread.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/* 线程一 */
void thread1(void)
{
	int i = 0;
	while(i++ < 3)
	{
		printf("This is pthread1.\n");
		if(i == 2)
			pthread_exit(0);
		sleep(2);
	}
}
/* 线程二 */
void thread2(void)
{
	int i = 0;
	while(i++ < 3)
```

```
        {
                printf("This is pthread2.\n");
        }
        pthread_exit(0);
}
int main()
{
        pthread_t id1,id2;
        int ret;
        /* 分别创建线程 1、2 */
        ret = pthread_create(&id1,NULL,(void *)thread1,NULL);
        if(ret != 0)
        {
                printf("Create pthread1 error\n");
                exit(1);
        }
        ret = pthread_create(&id2,NULL,(void *)thread2,NULL);
        if(ret != 0)
        {
                printf("Create pthread2 error\n");
                exit(1);
        }
        /* 等待线程结束 */
        pthread_join(id1,NULL);
        pthread_join(id2,NULL);
        return 0;
}
```

编译时出现错误：

```
root@ubuntu:/Workstation/examples/AppProg/5.threadapp# mipsel-linux-gcc thread.c -o thread
/tmp/cckPNR18.o: In function `main':
thread.c:(.text+0x178): undefined reference to `pthread_create'
thread.c:(.text+0x1ec): undefined reference to `pthread_create'
thread.c:(.text+0x254): undefined reference to `pthread_join'
thread.c:(.text+0x278): undefined reference to `pthread_join'
collect2: ld returned 1 exit status
root@ubuntu:/Workstation/examples/AppProg/5.threadapp# mipsel-linux-gcc thread.c -o thread
/tmp/ccCEUZGf.o: In function `main':
pthread.c:(.text+0x1ec): undefined reference to `pthread_create'
pthread.c:(.text+0x260): undefined reference to `pthread_create'
pthread.c:(.text+0x2c8): undefined reference to `pthread_join'
pthread.c:(.text+0x2ec): undefined reference to `pthread_join'
collect2: ld returned 1 exit status
```

错误报告说的是没有声明以上这些函数，为什么没有声明呢?因为 Linux 系统中本身并不包含线程库。那怎么办呢？最简单的方法就是在编译的后面添加上线程库的参数项 -pthread。

```
执行命令和结果如下:
root@ubuntu:/Workstation/examples/AppProg/5.threadapp# mipsel-linux-gcc thread.c -o thread -pthread
root@ubuntu:/Workstation/examples/AppProg/5.threadapp#
```

运行结果如下

```
[root@Loongson:/]#./thread
This is pthread2.
This is pthread2.
This is pthread2.
This is pthread1.
This is pthread1.
```

7.6.2 线程属性

在上一个实例中，pthread_create 函数的第二个参数，就是线程的属性，被设置为 NULL 表示采用默认的属性。线程的多数属性都是可以修改的。这些属性主要包括绑定属性、分离属性、堆栈地址、堆栈大小、优先级。其中系统默认的属性为非绑定、非分离、缺省 1M 大小的堆栈，与父进程同样级别的优先级。关于这些属性的概念，可去查阅相关资料。

下面重点讲解如何对这些属性进行设置，这些设置有固定的步骤。通常，首先调用 pthread_attr_init 函数进行初始化，然后调用相应的属性设置函数。如果要设置绑定属性那么使用 pthread_attr_setscope，如果要设置分离属性则使用 pthread_attr_setdetachstate ，设置线程优先级则使用 pthread_attr_getscheparam 和 pthread_attr_setchedparam。完成这些属性的设置后，就可以调用 pthread_create 函数来创建线程了。。

实例：pthread.c 创建两个线程，第一个线程的线程属性设置为绑定、分离属性；第二个 线程设置为默认的属性，即非绑定、非分离属性。

```
/* pthread.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/* 线程一 */
void thread1(void)
{
	int i = 0;
	while(i++ < 6)
	{
		printf("This is pthread1.\n");
		if(i == 2)
			pthread_exit(0);
		sleep(2);
	}
}
/* 线程二 */
void thread2(void)
{
	int i = 0;
	while(i++ < 3)
	{
		printf("This is pthread2.\n");
	}
	sleep(1);
	pthread_exit(0);
}
int main()
{
	pthread_t id1,id2;
	int ret;

	pthread_attr_t attr;
	/* 初始化线程 */
	pthread_attr_init(&attr);
	/* 设置线程绑定属性 */
	pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);
	/* 设置线程分离属性 */
	pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
	/* 分别创建线程 1、2 */
	ret = pthread_create(&id1,&attr,(void *)thread1,NULL);
	if(ret != 0)
	{
		printf("Create pthread1 error\n");
		exit(1);
	}
	ret = pthread_create(&id2,NULL,(void *)thread2,NULL);
	if(ret != 0)
	{
		printf("Create pthread2 error\n");
		exit(1);
	}
	/* 等待线程结束 */
```

```
        pthread_join(id1,NULL);
        pthread_join(id2,NULL);
        return 0;
}
```

```
执行结果如下：
[root@Loongson:/]#./pthread
This is pthread2.
This is pthread2.
This is pthread2.
This is pthread1.
```

读者可以使用 free 命令查看运行前后的内存使用情况，你会发现程序运行完后，系统就回收了内存

```
[root@Loongson:/]#free
              total         used         free       shared      buffers
Mem:          23316         9384        13932            0            0
-/+ buffers:                9384        13932
Swap:             0            0            0
```

7.6.3 互斥锁

互斥锁的来源:

平时听的最多的并不是线程，而是多线程。一提到线程，肯定首先想到的是多线程。由于线程共享进程的资源和地址空间，因此在对这些资源进程操作时，必须考虑的问题是，资源访问的唯一性。简单的说，就是对资源的访问每次只能有一个线程，这时就需要引入同步与互斥机制。在 POSIX 中线程同步的方法，主要有互斥锁(mutex)和信号量。

互斥锁的操作:

互斥锁的操作主要包括以下几个步骤：

1). 互斥锁初始化： pthread_mutex_init

2). 互斥锁上锁： pthread_mutex_lock

3). 互斥锁判断上锁： pthread_mutex_trylock

4). 互斥锁解锁： pthread_mutex_unlock

5). 消除互斥锁： pthread_mutex_destroy

其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其它未占有互斥锁的现场在希望得到互斥锁时，是否需要阻塞等待。快速锁是指调用线程会阻塞知道拥有互斥锁的线程解锁为止。递归互斥锁能够成功返回并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立刻返回一个错误信息。

互斥锁的实例:mutex.c

该实例使用快速互斥锁来实现对计数变量 lock_var 的加一、打印操作，从而进一步认识互斥锁，进一步理解上锁、解锁的实质。

```
/* mutex.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

/*  创建快速互斥锁  */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
/*  待观察计数变量  */
int lock_var;
timer_t end_time;

/*  线程一  */
```

```
void thread1(void)
{
        int i = 0;
        while(time(NULL) < end_time)
        {
                if(pthread_mutex_lock(&mutex)!=0)
                {
                        perror("pthread_mutex_lock");
                }
                else
                {
                        printf("pthread1:pthread1 lock the variable\n");
                        for(i =0;i < 2;i++)
                        {
                                sleep(1);
                                lock_var++;
                        }
                }
                /* 互斥锁解锁 */
                if(pthread_mutex_unlock(&mutex)!=0)
                {
                        perror("pthread_mutex_unlock");
                }
                else
                {
                        printf("pthread1:pthread1 unlock the variable\n");
                        sleep(1);
                }
        }
}
/* 线程二 */
void thread2(void)
{
        int ret;
        while(time(NULL) < end_time)
        {
                /* 判断是否已经被上锁 */
                ret = pthread_mutex_trylock(&mutex);
                if(ret==EBUSY)
                {
                        /* 忙 */
                        printf("pthread2:the variable is lock by pthread1\n");
                }
                else
                {
                        /* 不忙 */
                        if(ret!=0)
                        {
                                perror("pthread_mutex_trylock");
                                exit(1);
                        }
                        else
                        {
                                printf("pthread2:pthread2 got lock. The variable is %d\n",lock_var);
                        }
                        /* 互斥锁解锁 */
                        if(pthread_mutex_unlock(&mutex)!=0)
                        {
                                perror("pthread_mutex_unlock");
                        }
                        else
                        {
                                printf("pthread2:pthread2 unlock the variable\n");

                        }
                }
```

```
        sleep(3);
        }
}
int main()
{
        pthread_t id1,id2;
        int ret;
        end_time = time(NULL) + 10;

        /* 快速互斥锁的初始化 */
        pthread_mutex_init(&mutex,NULL);
        /* 分别创建线程 1、2 */
        ret = pthread_create(&id1,NULL,(void *)thread1,NULL);
        if(ret != 0)
        {
                printf("Create pthread1 error\n");
                exit(1);
        }
        ret = pthread_create(&id2,NULL,(void *)thread2,NULL);
        if(ret != 0)
        {
                printf("Create pthread2 error\n");
                exit(1);
        }
        /* 等待线程结束 */
        pthread_join(id1,NULL);
        pthread_join(id2,NULL);
        return 0;
}
```

执行结果如下：

```
[root@Loongson:/]#./mutex
pthread2:pthread2 got lock. The variable is 0
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
pthread1:pthread1 unlock the variable
pthread2:pthread2 got lock. The variable is 2
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
pthread1:pthread1 unlock the variable
pthread2:pthread2 got lock. The variable is 4
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
pthread1:pthread1 unlock the variable
pthread2:pthread2 got lock. The variable is 6
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
pthread1:pthread1 unlock the variable
```

从结果可以看出，快速互斥锁如上面所述，如果已经有线程上锁了，就会一直等待该线程解锁，才能对互斥锁进行上锁操作。

7.6.4 信号量

信号量的介绍

信号量其实就是一个非负的整数计数器，是操作系统中所用的 PV 原语，主要应用于进程或线程间的同步与互斥。其工作原理也很简单， PV 原语就是对整数计数器信号量 sem 进行操作，一次 P 操作使 sem 减一，而一次 V 操作使 sem 加一。 当信号量 sem 的值大于等于零时，该线程具有访问公共资源的权限；相反，当信号量 sem 的值小于零时，该线程就阻塞直到信号量 sem 的值大于等于 0 为止。

信号量的操作函数说明

表 7.3 信号量操作函数

名称	作　　用
sem_init	用于创建一个信号量，并能初始化它的值。
sem_wait	相当于 P 操作，将信号量的值减一，会阻塞进程。
sem_trywait	相当于 P 操作，将信号量的值减一，立刻返回，不会阻塞。
sem_post	相当于 V 操作，将信号量的值加一，同时发出信号唤醒等待进程。
sem_getvalue	获得信号量当前值。
sem_destroy	删除信号量

信号量的实例

将 7.6.3 节互斥锁的实例简单的变换一下，使用信号量的机制来对 lock_var 的操作，使信号量为的值为 1，其实就相当于互斥锁了。

```
/* sem.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>

/* 定义一个信号量 */
sem_t sem;
/* 待观察计数变量 */
int lock_var;
timer_t end_time;

/* 线程一 */
void thread1(void)
{
	int i = 0;
	while(time(NULL) < end_time)
	{
		/* P 操作，使得 sem 减一 */
		sem_wait(&sem);
		for(i =0;i < 2;i++)
		{
			sleep(1);
			lock_var++;
			printf("lock_var = %d\n",lock_var);
		}
		printf("pthread1:lock_var = %d\n",lock_var);
		/* V 操作，使得 sem 加一 */
		sem_post(&sem);
		sleep(1);
	}
}
/* 线程二 */
void thread2(void)
{
	int ret;
	while(time(NULL) < end_time)
	{
		/* P 操作，使得 sem 减一 */
		sem_wait(&sem);
		printf("pthread2:pthread2 got lock;lock_var = %d\n",lock_var);
		/* V 操作，使得 sem 加一 */
		sem_post(&sem);
		sleep(3);
	}
}
int main()
```

```
{
	pthread_t id1,id2;
	int ret;
	end_time = time(NULL) + 10;

	/* 初始化信号量为 1 */
	ret = sem_init(&sem,0,1);
	if(ret != 0)
	{
		printf("sem_init error\n");
		exit(1);
	}
	/* 分别创建线程 1、2 */
	ret = pthread_create(&id1,NULL,(void *)thread1,NULL);
	if(ret != 0)
	{
		printf("Create pthread1 error\n");
		exit(1);
	}
	ret = pthread_create(&id2,NULL,(void *)thread2,NULL);
	if(ret != 0)
	{
		printf("Create pthread2 error\n");
		exit(1);
	}
	/* 等待线程结束 */
	pthread_join(id1,NULL);
	pthread_join(id2,NULL);
	return 0;
}
```

执行结果如下：

```
[root@Loongson:/]#./sem
pthread2:pthread2 got lock;lock_var = 0
lock_var = 1
lock_var = 2
pthread1:lock_var = 2
pthread2:pthread2 got lock;lock_var = 2
lock_var = 3
lock_var = 4
pthread1:lock_var = 4
pthread2:pthread2 got lock;lock_var = 4
lock_var = 5
lock_var = 6
pthread1:lock_var = 6
pthread2:pthread2 got lock;lock_var = 6
lock_var = 7
lock_var = 8
pthread1:lock_var = 8
```

这里使用的是一个信号量的机制，当然也可以使用多个信号量来实现线程间的同步，限于篇幅，这里就不再介绍了。

7.7 网络编程

7.7.1 网络编程基础概念

TCP/IP 基本概念

TCP/IP 协议（ Transmission Control Protocol / Internet Protocol）叫做传输控制/网际协议，又叫网络通信协议。实际上，它包含了上百个功能的协议，如 ICMP（互联网控制信息协议）、 FTP（文件传输协议）、 UDP（用户数据报协议）、 ARP（地址解析协议）等。 TCP 负责发现传输的问题，一旦有问题就会发出重传的信号，直到所有数据安全正确地传输到目

的地。而 IP 就是给因特网的每一 台电脑规定一个地址。

IP 地址、端口与域名

IP 地址的作用是标识计算机的网卡地址，每一台计算机都有唯一个 IP 地址。在程序中是通过 IP 地址来访问一台计算机的。 IP 地址具有统一的格式， IP 地址的长度是 32 位的二进制数值，4 个字节。为了便于记忆，通常化为十进制的整数来表示。如：192.168.1.100。在 Linux 虚拟机终端输入命令，可查看到本机的 IP 地址和 MAC 地址。

```
root@ubuntu:/# ifconfig
ens33      Link encap:Ethernet   HWaddr 00:0c:29:94:2c:1b
           inet6 addr: fe80::20c:29ff:fe94:2c1b/64 Scope:Link
           UP BROADCAST RUNNING MULTICAST   MTU:1500   Metric:1
           RX packets:4 errors:0 dropped:0 overruns:0 frame:0
           TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:272 (272.0 B)   TX bytes:842 (842.0 B)
           Interrupt:19 Base address:0x2000

lo         Link encap:Local Loopback
           inet addr:127.0.0.1   Mask:255.0.0.0
           inet6 addr: ::1/128 Scope:Host
           UP LOOPBACK RUNNING   MTU:65536   Metric:1
           RX packets:15895 errors:0 dropped:0 overruns:0 frame:0
           TX packets:15895 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:1209466 (1.2 MB)   TX bytes:1209466 (1.2 MB)
```

端口，是指计算机中为了标识同一计算机中不同程序访问网络而设置 编号。每个程序在访问网络时都会分配一个标识符，程序在访问网络或接受访问时，会用这个标识符表示这一网络数据属于这个程序。端口号其实是一个 16 位的无符号整数（unsigned short）也就是 0~65535。不同编号范围的端口号有不同的作用。低于 256 的端口是系统保留端口号，主要用于系统进程通信。如 WWW 服务使用的是 80 号端口， FTP 服务使用的是 21 号端口。不在这一范围内的端口号是自由端口号，在编程时可以调用这些端口号。

域名,用来代替 IP 地址来标识计算机的一种直观名称。如百度网址 IP 是，180.97.33.108，没有任何逻辑含义，不便于记忆。我们一般选择 www.baidu.com 这个域名来代替 IP 地址。可以使用命令：ping baidu.com 来查看一个域名对应的 IP 地址为。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping www.baidu.com

正在 Ping www.a.shifen.com [180.97.33.108] 具有 32 字节的数据:
来自 180.97.33.108 的回复: 字节=32 时间=4ms TTL=50
来自 180.97.33.108 的回复: 字节=32 时间=4ms TTL=50
来自 180.97.33.108 的回复: 字节=32 时间=4ms TTL=50
来自 180.97.33.108 的回复: 字节=32 时间=5ms TTL=50

180.97.33.108 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程的估计时间(以毫秒为单位):
    最短 = 4ms，最长 = 5ms，平均 = 4ms

C:\Users\Administrator>
```

图 7.5 windows 中查看 IP

套接字

套接字（socket），在网络中用来描述计算机中不同程序与其他计算机程序的通信方式。人们常说的 socket 其实是一种特殊的 IO 接口，它也是一种文件描述符。

套接字类型

1） 流式 socket（ SOCK_STREAM） 流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证 了数据传输的正确性和顺序性。

2） 数据报 socket（ SOCK_DGRAM） 数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输， 是无序的，并且不保证是可靠、无差错的。它使用的数据报协议是 UDP。

3） 原始 socket 原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

套接字（也叫套接口）由三个参数构成： IP 地址、端口号、传输层协议，以区分不同应用程序进程间的网络通信与连接。 socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的建立连接、数据传输等操作都是通过 socket 来实现的。

套接字数据结构

C 程序进行套接字编程时，常会使用到 sockaddr 数据类型和 sockaddr_in 数据类型，用于保存套接字信息。

```
struct sockaddr
{
/*地址族，就是一些协议类型的集合*/
unsigned short sa_family;
/*14 字节的协议地址，包含该 socket 的 IP 地址和端口号*/
char sa_data[14];
};
struct sockaddr_in
{
short int sa_family; /*地址族*/
unsigned short int sin_port; /*端口号*/
struct in_addr sin_addr; /*IP 地址*/
/*全填 0，保持与 struct sockaddr 同样大小*/
unsigned char sin_zero[8];
};
```

上面这二个数据类型是等效的，可以相互转化， 但一般都使用 sockaddr_in 这种形式。sa_family 常见值如下：

```
AF_INET: IPv4 协议
AF_INET6:  IPv6 协议
AF_LOCAL: UNIX 域协议
```

域名、主机名与 IP 地址转换

通常，使用过程中都不愿意记忆一连串的 IP 地址， 因此，使用主机名将会是一个很好的选择。在 Linux 中，有一些函数可以实现主机名和地址间的转换。 其中 gethostbyname 是将主机名转换为 IP 地址， 而 gethostbyaddr 是将 IP 地址转换为主机名。 这两个函数都涉及一个 hostent 结构体，那么先认识一下这个结构体：

```
struct hostent
{
char *h_name; /*正式主机名*/
char **h_aliases; /*主机别名*/
int h_addrtype; /*地址类型*/
int h_length; /*地址长度*/
char **h_addr_list; /*指向 IPv4 或 IPv6 的地址指针数组*/
}
```

数据存储优先级顺序

计算机数据存储分高字节优先和低字节优先。即大端、小端的问题。而 Internet 上数据是以高位字节优先顺序在网络上传输的，但是 ARM 等一些 CPU， 除了摩托罗拉公司的 CPU 是大端的，常见的 CPU 都是小端格式存储数据的。所以有必要对这两个字节存储优先顺序进行互相转化一些。这里有： htos、 ntohs、 htonl、 ntohl 四个函数实现网络字节序和主机字节序的转化，这里的 h 代表 host，n 代表 network，s 代表 short，l 代表 long。通常 16 位的 IP 端口号 用 s 代表，而 IP 地址用 l 来代表。 函数原型：

```
uint16_t htons(uint16_t host16bit)
uint32_t htonl(uint32_t host32bit)
uint16_t ntohs(uint16_t net16bit)
uint32_t ntohs(uint32_t net32bit)
```

7.7.2 网络编程实例

网络基础编程主要介绍传输层中的 TCP 和 UDP 协议， TCP 和 UDP 是两种不同的网络传输方式。

（1）TCP 协议

通常应用程序通过打开一个 socket 来使用 TCP 服务， TCP 管理到其他 socket 的数据传递。可以说，通过 IP 源/目的可以唯一的区分网络中两个设备的关联，通过 socket 的源/目的可以唯一的区分网络中两个应用程序的关联。

TCP 三次握手协议

TCP 对话通过三次握手来初始化，三次握手的目的是使数据段的发送和接收同步；告诉其他主机其一次可接收的数据量，并建立虚连接。下面简单描述了三次握手的过程： 1、初始化主机通过一个同步标志置位的数据段发出会话请求； 2、接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发生的数据段的起始字节的顺序号、应答并带有将受到的下一个数据段的字节顺序号。3、请求主机再回送一个数据段，并带有确认顺序号和确认号。 TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体将回送一个数据报，其中包含有一个确认序号，它的意思是希望收到下一个数据报的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送发重发该数据报。

TCP 协议编程相关函数说明

网络上绝大多数的通信服务采用服务器机制(client/server)， TCP 提供的是一种可靠的、面向连接的服务。

表 7.4 网络操作函数

名称	作　　用
socket	用于建立一个 socket 连接。
bind	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求。
connect	面向连接的客户程序使用 connect 函数来配置 socket ，并与远端服务器建立一个 TCP 连接。
listen	使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务器，请求保存在此队列中，直到程序处理他们。
accept	让服务器接收客户的连接请求。
close	停止在该 socket 上的任何数据操作。
send	数据发送函数
recv	数据接收函数

TCP 编程实例：该实例分为服务器端(server)和客户端(client)，其中服务器端首先建立起 socket，接着绑定本地端口，接着建立于客户端的联系，并接收客户端发送的消息。而客户

端则建立 socket 之后，调用 connect 函数来与服务器端建立连接，连接上后，调用 send 函数发送数据到服务器端。 服务器端的源码如下：

```
/* server.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PORT    4321
#define BUFFER_SIZE 1024
#define MAX_QUE_CONN_NM 5

int main(void)
{
	struct sockaddr_in server_addr,client_addr;
	int sin_size,recvbytes;
	int ser_fd,cli_fd;
	char buf[BUFFER_SIZE];

	/*  建立 socket 连接，IPv4 协议，字节流套接字  */
	if((ser_fd = socket(AF_INET,SOCK_STREAM,0))==-1)
	{
		perror("socket");
		exit(1);
	}
	printf("Socket id = %d\n",ser_fd);

	/*  初始化 sockaddr_in 结构体  */
	server_addr.sin_family = AF_INET;
	server_addr.sin_port = htons(PORT);
	server_addr.sin_addr.s_addr = INADDR_ANY;
	bzero(&(server_addr.sin_zero),8);

	/*  绑定函数 bind */
	if(bind(ser_fd,(struct sockaddr *)&server_addr,sizeof(struct sockaddr))==-1)
	{
		perror("bind");
		exit(1);
	}
	printf("Bind success!\n");

	/*  调用 listen 函数，进行监听  */
	if(listen(ser_fd,MAX_QUE_CONN_NM)==-1)
	{
		perror("listen");
		exit(1);
	}
	printf("Listening......\n");

	/*  调用 accept 函数，等待客户端的连接  */
	if((cli_fd = accept(ser_fd,(struct sockaddr *)&client_addr,&sin_size))==-1)
	{
		perror("accept");
		exit(1);
	}
	printf("Have client ready for connecting\n");

	/*  调用 recv 函数接收客户端的请求  */
	memset(buf,0,sizeof(buf));
	if((recvbytes = recv(cli_fd,buf,BUFFER_SIZE,0))==-1)
	{
```

```
        perror("recv");
        exit(1);
    }
    /* 将收到的信息(客服端发来的信息)打印出来 */
    printf("Received a message:%s\n",buf);

    /* 关闭 socket */
    close(ser_fd);
    return 0;
}
```

客户端的源码如下：

```
/* client.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PORT    4321
#define BUFFER_SIZE 1024

int main(int argc,char *argv[])
{
    struct sockaddr_in server_addr;
    int sockfd,sendbytes;
    char buf[BUFFER_SIZE];
    struct hostent *host;

    /* 指定输入参数为 3 个，否则出错 */
    if(argc != 3)
    {
        printf("Usage: ./clinet IP address Text\n");
        exit(1);
    }

    /* 地址解析函数 */
    if((host = gethostbyname(argv[1]))==NULL)
    {
        perror("gethostbyname");
        exit(1);
    }
    memset(buf,0,sizeof(buf));
    sprintf(buf,"%s",argv[2]);

    /* 建立 socket 连接，IPv4 协议，字节流套接字 */
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))==-1)
    {
        perror("socket");
        exit(1);
    }
    printf("Socket id = %d\n",sockfd);

    /* 初始化 sockaddr_in 结构体 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    bzero(&(server_addr.sin_zero),8);

    /* 调用 connect 函数主动发起对服务器的连接 */
    if((connect(sockfd,(struct sockaddr *)&server_addr,sizeof(struct sockaddr)))==-1)
    {
```

```
            perror("connect");
            exit(1);
        }
        printf("Connect server success!\n");

        /* 发送消息给服务器端 */
        if((sendbytes = send(sockfd,buf,strlen(buf),0))==-1)
        {
            perror("send");
            exit(1);
        }
        sleep(3);
        /* 关闭 socket */
        close(sockfd);
        return 0;
}
```

服务器端执行步骤:

先在开发板运行以下程序：

```
[root@Loongson:/]#./server
Socket id = 3
Bind success!
Listening......
```

在上位机上用 TCP/UDP 工具建立连接，连接到 IP 地址为 193.169.2.230（开发板）,端口号为 4321 的服务器（开发板），并发送数据： Hello! Loongson, socket.

图 7.6 服务器调试窗口发送网络数据

开发板运行结果：

```
Have client ready for connecting
Received a message: Hello▒▒ Loongson, socket.
[root@Loongson:/]#
```

结果说明：开发板作为服务器打开 4321 端口监听，当 PC 机建立连接后并发送数据后，开发板接收打印后，关闭连接。

客户端执行步骤如下：

先在上位机上用 TCP/UDP 工具建立服务器 IP 地址为 193.169.2.215,端口号为 4321，并打开监听。开发板运行程序执行结果如下：

```
[root@Loongson:/]# ./client 193.169.2.215    HelloLoongson,socket!
Socket id = 3
```

```
Connect server success!
```

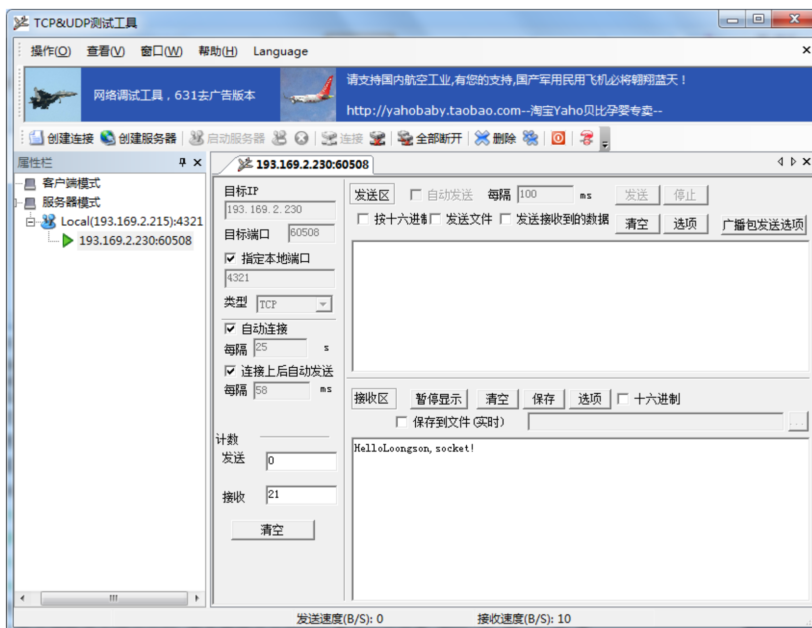


图 7.7 服务器调试窗口接收网络数据

开发板作为客户端，与（193.169.2.215）建立连接后发送数据（HelloLoongson,socket!）后关闭连接。注意：发送的数据(HelloLoongson,socket!)，中间不能留有空格，否则，不能正常运行。

（2）UDP 协议

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可以同时作为应用的客户或服务器方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。UDP 比 TCP 更能更好地解决实时性问题，如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议.

UDP 协议编程相关函数

所谓无连接的套接字通信，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信时，两个计算机之前没有建立连接的过程。需要处理的内容只是把信息发送到另外一个计算机，这种通讯方式比较简单，涉及函数也比较少。

表 7.5 UDB 编程函数

名称	作　　用
bind	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求。
close	停止在该 socket 上的任何数据操作。
sendto	数据发送函数
recvfrom	数据接收函数

UDP 编程实例 该实例同样分为服务器端(server)和客户端(client)，服务器端首先建立 socket，接着绑定本地端口，随后并没有 listen 监听客服端，也没有 accept 等待连接，只是在死循环里，直接等待接收数据。而客户端就更加简单，在建立 socket 后，直接调用 sendto 发送数据到服务器。这样就省去了很多 TCP 必须的步骤，而 UDP 正因为不是面向连接的，所以显得简单方便。值得注意的是， UDP 并不是可靠的通信方式。

```
/* udp_server.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PORT    8888
#define BUFFER_SIZE 1024

int main(void)
{
        struct sockaddr_in server_addr,client_addr;
        int sockfd;
        char buf[BUFFER_SIZE];
        int ret,len;

        /* 建立 socket 连接，IPv4 协议，数据报套接字 */
        if((sockfd = socket(AF_INET,SOCK_DGRAM,0))==-1)
        {
                perror("socket");
                exit(1);
        }
        printf("Socket id = %d\n",sockfd);

        /* 初始化 sockaddr_in 结构体 */
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(PORT);
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        bzero(&(server_addr.sin_zero),8);

        len = sizeof(server_addr);

        /* 绑定函数 bind */
        if(bind(sockfd,(struct sockaddr *)&server_addr,sizeof(server_addr))==-1)
        {
                perror("bind");
                exit(1);
        }
        printf("Bind success!\n");

        while(1)
        {
                /* 等待客户端发送过来的数据，一旦有数据就接收进来 */
                if((ret = recvfrom(sockfd,buf,sizeof(buf),0,(struct sockaddr *)&server_addr,&len))==-1)
                {
                        perror("recvfrom");
                        exit(1);
                }
                buf[len] = '0';
                printf("The message is %s\n",buf);

                if(strncmp(buf,"stop",4)==0)
                {
                        printf("Stop to run!\n");
                        break;
                }
                /* 关闭 socket */
                close(sockfd);
                exit(0);
        }
        return 0;
}
```

UDP 编程， 客户端的源码如下：

```
/* udp_client.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PORT    8888
#define BUFFER_SIZE 1024

int main(int argc,char *argv[])
{
        struct sockaddr_in server_addr;
        int sockfd;
        char buf[BUFFER_SIZE];
        int sendbytes;

        /* 指定输入参数为 2 个，否则出错 */
        if(argc != 2)
        {
                printf("Usage: ./clinet Text\n");
                exit(1);
        }

        memset(buf,0,sizeof(buf));
        sprintf(buf,"%s",argv[1]);

        /* 初始化 sockaddr_in 结构体 */
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(PORT);
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        bzero(&(server_addr.sin_zero),8);

        /* 建立 socket 连接，IPv4 协议，数据报套接字 */
        if((sockfd = socket(AF_INET,SOCK_DGRAM,0))==-1)
        {
                perror("socket");
                exit(1);
        }
        printf("Socket id = %d\n",sockfd);

        /* 发送消息给服务器端 */
        if((sendbytes = sendto(sockfd,buf,strlen(buf),0,(struct sockaddr *)&server_addr,sizeof(server_addr)))==-1)
        {
                perror("sendto");
                exit(1);
        }
        sleep(3);
        /* 关闭 socket */
        close(sockfd);
        return 0;
}
```

服务器端执行步骤:

先在开发板运行以下程序：

```
[root@Loongson:/]#./udp_server
Socket id = 3
Bind success!
```

在上位机上用 TCP/UDP 工具建立 UDP 连接，连接到 IP 地址为 193.169.2.230,端口号为 4321 的服务器（开发板），并发送数据： Hello! Loongson, socket.

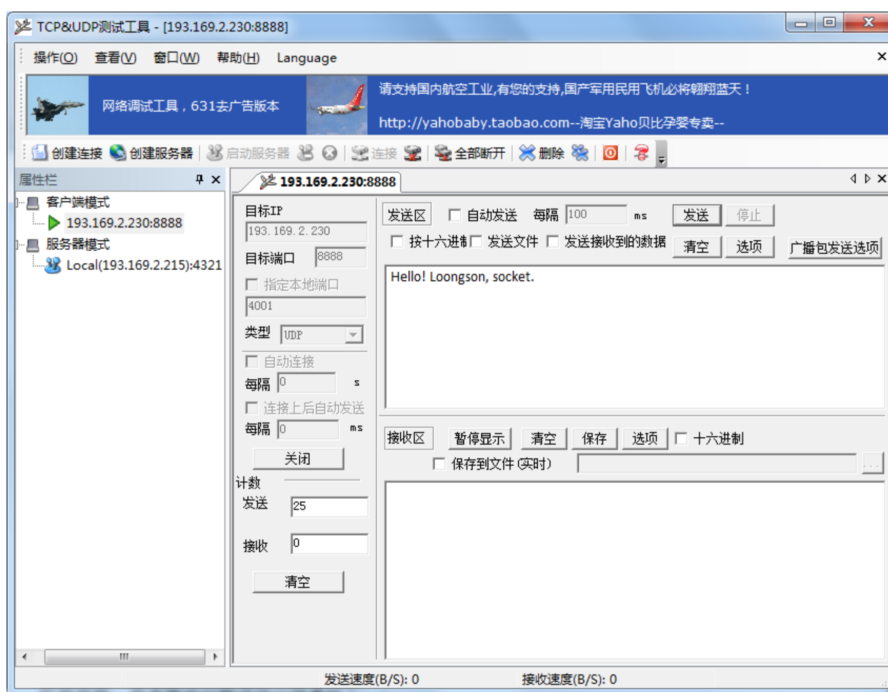


图 7.8 UDP 调试工具发送 UDP 数据

开发板运行结果：

```
The message is    Hello! Loongson0 socket.
```

结果说明：开发板作为 UDP 服务器打开 8888 端口监听，当 PC 机建立连接后并发送数据后，开发板接收打印后，关闭连接。

客户端执行步骤如下：

先在上位机上用 TCP/UDP 工具建立 UDP 服务器，IP 地址为 193.169.2.215(即为 PC 机的 IP 地址),端口号为 8888，并打开监听。开发板运行程序执行结果如下：

```
 [root@Loongson:/]#./udp_client 193.169.2.215    HelloLoongson,UDP!
Socket id = 3
[root@Loongson:/]#./udp_client 193.169.2.215    HelloLoongson,UDP!
Socket id = 3
[root@Loongson:/]#
```

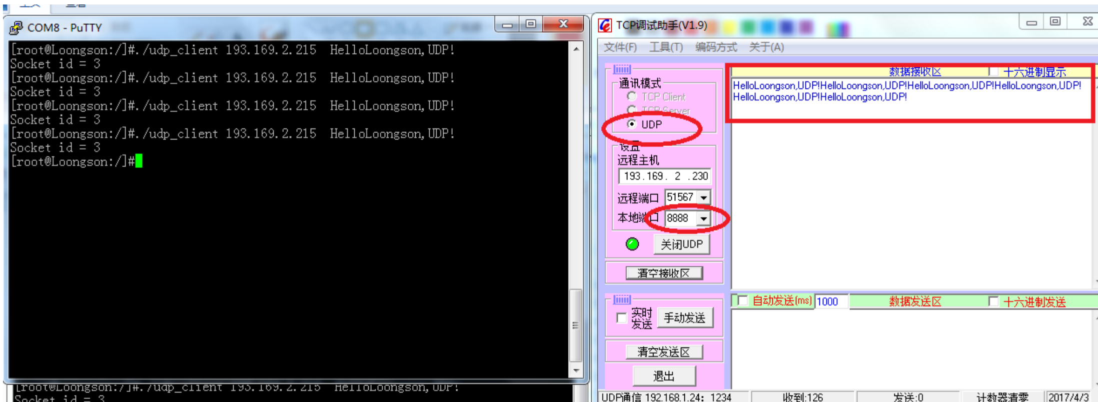


图 7.9 UDP 服务器与开发板通讯

开发板作为 UDP 客户端，建立连接后发送数据（HelloLoongson,UDP!）后关闭连接。注意：发送的数据(HelloLoongson,UDP!)，中间不能留有空格，否则，不能正常运行。

7.7.3 网络编程小结

因为有些初学者，没有接触过太多的网络编程，所以开头讲述了网络基础的概念，比如:TCP/IP 协议、 IP 地址、域名、端口、网络套接字等概念。介绍了 socket 的类型和定义，最后重点讲解了面向连接的套接字通信和无连接的套接字通信。通过 2 个实例，详细的分析了 TCP、 UDP 两种截然不同的编程。通过学习本章内容，应该能够熟练掌握 TCP、 UDP 的简单编程。

7.8 应用编程总结

到这里，Linux 应用编程就基本完成，如果能够熟练的将这些内容掌握，应用编程能力基本合格，后面要多去阅读更多的开源应用程序，甚至尝试着自己去编写更复杂的应用程序。用户还要亲自用手敲代码，动手操作一遍。不要因为取巧偷懒，觉得只要把教程看懂了，就能够掌握，这是不可能的。

8．开发板硬件接口编程

8.1 点亮一个 LED 灯

所有程序编程都是从点灯开始的，俗称“点灯大法”。开发板上的可控 LED 灯通常都是一端接高电平或 GND，另一端接 GPIO。通过操作 GPIO 来控制其点亮和熄灭。

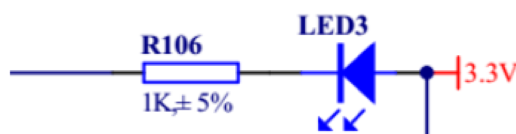


图 8.1 LED 发光二极管点亮示意图

如图所示，一个 LED 是由发光二极管组成，一端接高电平，另一端若接入高电平，则二极管不导通， LED 不会发光。另一端若接入低电平，则二极管导通， LED 发光。高低电平一般由 GPIO 输出。

8.1.1 LED 的操作接口

LED 操作接口位于/sys/class/leds 目录下。此目录下包含了关于 LED 操作的目录：

```
[root@Loongson:/]#ls /sys/class/leds
led_blue        led_orange      led_yellow      ls1x_pwm_led2
led_green       led_red         ls1x_pwm_led1
```

以 led_blue 为例，进入 led_blue 目录，该目录的内容为：

```
[root@Loongson:/sys/devices/platform/leds-gpio/leds/led_blue]#ls
brightness      max_brightness  subsystem       uevent
device          power           trigger
```

各个文件作用介绍如表 8.1 所示。

表 8.1 LED 操作函数说明

名称	作　　用
brightness	用于控制 LED 亮灭 （需要将 LED 灯设置为用户控制）。
Subsystem	符号链接，指向父目录。
trigger	写入“none ”可以将指示灯设置为用户控制;写入“heartbeat ”可以将指示灯设置为心跳灯;写入“nand-disk ”可以将指示灯设置为 NAND Flash 读写灯。
power	设备供电方面的相关信息。

8.1.2 LED 控制

```
[root@Loongson:/sys/devices/platform/leds-gpio/leds/led_blue]#echo heartbeat > trigger//指示灯设置为心跳灯
[root@Loongson:/sys/devices/platform/leds-gpio/leds/led_blue]#echo none >trigger //将指示灯设置为用户控制
[root@Loongson:/sys/devices/platform/leds-gpio/leds/led_blue]#echo 255 > brightness//点亮指示灯
[root@Loongson:/sys/devices/platform/leds-gpio/leds/led_blue]#echo 0 > brightness//熄灭指示灯
```

8.1.3 在程序中操作 LED 灯

C 程序中操作 LED，首先需要设置 trigger 属性。然后操作 brightness 属性， 设置 LED 点亮或熄灭。

实例源码：led.c 。 首先设置 LED trigger 属性为“ none”，然后设置 brightness 属性交替为 0 和 1。实现了 LED 每隔 1s 点亮一次。

```
/*led.c*/
#include <stdint.h>
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <string.h>

#define TRIGGER                     "trigger"
#define LED_PATH                "/sys/class/leds/"
#define LED_STATUS                  "brightness"
#define TRIGGER_NONE                "none"

int main(int argc,char **argv)
{
        char path[20],data[2];
        int fd, ret, flag;
        if(argv[1] == NULL) {
                printf("usage : ./led led_***    ");
                return 0;
        }

        strcpy(path, LED_PATH);
        strcat(path, argv[1]);
        strcat(path, "/" TRIGGER);
        fd = open(path, O_RDWR);
        if(fd < 0) {
                perror("open");
                return -1;
        }
        ret = write(fd, TRIGGER_NONE, strlen(TRIGGER_NONE));
        if(ret < 0) {
                perror("write");
                return -1;
        }
        close(fd);
        strcpy(path, LED_PATH);
        strcat(path, argv[1]);
        strcat(path, "/" LED_STATUS);
        fd = open(path, O_WRONLY);
              if(fd < 0) {
                          perror("open");
                          return -1;
              }
        for(;;)
        {
                data[0] = flag ? '0': '1';
                ret = write(fd, data, 1);
                if(ret < 0) {
                perror("write");
                return -1;
        }
                flag = !flag;
                sleep(1);
        }
        return 0;
}
```

在虚拟机中编译，后下载到开发板上运行，观察效果是开发板上的蓝灯每隔 1 秒闪烁。

```
[root@Loongson:/]#./led led_blue
```

8.3 GPIO 硬件编程

相比于 Linux 2.4， 2.6 及以上的内核可以使用系统中的 GPIOLIB 模块在用户空间提供

的 sysfs 接口，实现应用层对 GPIO 的独立控制。

8.3.1 GPIO 和 sysfs 操作接口

Linux 开发平台实现了通用 GPIO 的驱动，用户通过 Shell 命令或系统调用即能控制 GPIO 的输出和读取其输入值。其属性文件均在/sys/class/gpio 目录下，如：

```
[root@Loongson:/]#ls /sys/class/gpio
export          gpiochip0     gpiochip32    gpiochip64    gpiochip96    unexport
```

属性文件有 export 和 unexport。其余四个文件为符号链接（gpiochip0，gpiochip32，gpiochip64，gpiochip96），指向管理对应设备的目录，以 gpiochip0 为例，此目录下文件有：

```
[root@Loongson:/]# ls /sys/class/gpio/gpiochip0
base          label         ngpio         power         subsystem    uevent
```

以上文件用途如表 8.2 所示。

表 8.2 GPIO 操作

名称	路径	作　　用
export	/sys/class/gpio/export	导出 GPIO
unexport	/sys/class/gpio/unexport	将导出的 GPIO 从 sysfs 中清除
gpiochipN	/sys/class/gpio/gpiochipN/base	设备所管理的 GPIO 初始编号
	/sys/class/gpio/gpiochipN/label	设备信息
	/sys/class/gpio/gpiochipN/ngpio	设备所管理的 GPIO 总数
	/sys/class/gpio/gpiochipN/power	设备供电方面的相关信息
	/sys/class/gpio/gpiochipN/subsystem	符号链接，指向父目录
	/sys/class/gpio/gpiochipN/uevent	内核与 udev(自动设备发现程序)之间的通信接口

向 export 文件写入需要操作的 GPIO 排列序号 N，就可以导出对应的 GPIO 设备目录。操作命令如下：

```
# echo N > /sys/class/gpio/export
```

例如，导出序号为 68 的 GPIO 的操作接口，在 Shell 下，可以用如下命令：

```
[root@Loongson:/]#echo 50 > /sys/class/gpio/export
sh: write error: Device or resource busy
[root@Loongson:/]#echo 49 > /sys/class/gpio/export
```

通过以上操作后在/sys/class/gpio 目录下生成 gpioN 目录，通过读写该设备目录下的属性文件（位于 gpioN 下）就可以操作这个 GPIO 的输入和输出。以此类推可以导出其它 GPIO 设备目录。如果 GPIO50 已经被系统占用，导出时候会提示资源占用。

以排列序号为 49 的 GPIO 为例， 设备目录下有如下属性文件：

```
[root@Loongson:/]#ls /sys/class/gpio/gpio49
active_low    direction     edge          power         subsystem    uevent        value
```

文件用途如表示。

表 8.3 GPIO 属性操作

名称	路径	作　　用
active_low	/sys/class/gpio/gpioN/active_low	具有读写属性。用于决定 value 中的值是否翻转。 0 不翻转， 1 翻转。
edge	/sys/class/gpio/gpioN/edge	具有读写属性。设置 GPIO 中断，或检测中断是否发生。
subsystem	/sys/class/gpio/gpioN/subsystem	符号链接，指向父目录。
value	/sys/class/gpio/gpioN/value	具有读写属性。 GPIO 的电平状态设置或读取。
direction	/sys/class/gpio/gpioN/direction	具有读写属性。用于查看或设置 GPIO 输入输出
power	/sys/class/gpio/gpioN/power	设备供电方面的相关信息
uevent	/sys/class/gpio/gpioN/uevent	内核与 udev(自动设备发现程序)之间的通信接口

8.3.2 GPIO 基本操作

在应用层可以通过 Shell 命令操作 GPIO。通过以下步骤，可以控制 GPIO 输入输出。下面步骤是以 GPIO 的输入输出功能进行介绍。

1） 输入输出设置

GPIO 导出后默认为输入功能。向 direction 文件写入“ in”字符串，表示设置为输入功能；向 direction 文件写入“ out”字符串，表示设置为输出功能。读 direction 文件，会返回 in/out 字符串， in 表示当前 GPIO 作为输入， out 表示当前 GPIO 作为输出。方向查看和设置命令如下：

```
[root@Loongson:/]#cat /sys/class/gpio/gpio49/direction                //查看方向
in
[root@Loongson:/]#echo out > /sys/class/gpio/gpio49/direction         //设置为输出
[root@Loongson:/]#cat /sys/class/gpio/gpio49/direction
out
```

2) 输入读取

当 GPIO 被设为输入时， value 文件记录 GPIO 引脚的输入电平状态： 1 表示输入的是高电平； 0 表示输入的是低电平。通过查看 value 文件可以读取 GPIO 的电平，查看命令如下：

```
[root@Loongson:/]# echo in >/sys/class/gpio/gpio49/direction //设置 GPIO 排列序号为 49 的 GPIO 方向为输入
[root@Loongson:/]#cat /sys/class/gpio/gpio49/value//查看 GPIO 排列序号为 N 的 GPIO 电平
1
```

3）输出控制

当 GPIO 被设为输出时，通过向 value 文件写入 0 或 1（ 0 表示输出低电平； 1 表示输出高电平）可以设置输出电平的状态，例如：设置排列序号为 49 的 GPIO 的电平为高电平

```
[root@Loongson:/]# echo out > /sys/class/gpio/gpio49/direction #设置 GPIO 排列序号为 49 的 GPIO 方向为输出
[root@Loongson:/]# echo 0 > /sys/class/gpio/gpio49/value #输出低电平
[root@Loongson:/]# echo 1 > /sys/class/gpio/gpio49/value #输出高电平
```

8.2.3 在 C 程序中操作 GPIO

使用系统调用实现 GPIO 输入输出操作时， 首先需要使用 export 属性文件导出 GPIO，

```
#define EXPORT_PATH "/sys/class/gpio/export" //GPIO 设备导出设备
#define GPIO "49" //GPIO49
int fd_export = open(EXPORT_PATH, O_RDWR); //打开 GPIO 设备导出设备
...
write(fd_export, GPIO, strlen(GPIO)); //向 export 文件写入 GPIO 排列序号字符串
```

可以调用 write 函数向 direction 设备写入方向 in/out 字符串，将 GPIO 设置为输入（输出）， 如：

```
#define DIRECT_PATH "/sys/class/gpio/gpio49/direction" //GPIO 输入输出控制设备
int fd_dir, ret ;
fd_dir = open(DIRECT_PATH, O_RDWR); //打开 GPIO 输入输出控制设备
...
ret = write(fd_dir, direction, sizeof(direction)); //写入 GPIO 输入（ in）输出（ out）方向
```

GPIO 设置为输入时使用 read 系统调用读取 value 属性文件，就可以读取 GPIO 电平值。

GPIO 设置为输出时， 使用 write 系统调用向 value 属性文件写入 0 或 1 字符串，就可以设置 GPIO 电平值。如：

```
#define DEV_PATH "/sys/class/gpio/gpio49/value" //输入输出电平值设备
int fd_dev, ret ;
fd_dev = open(DEV_PATH, O_RDWR); //打开输入输出电平值设备
...
ret = read(fd_dev, buf, sizeof(buf)); //读取 GPIO 输入电平值
```

范例代码以 GPIO49 为例，实现 GPIO 的输入读取。首先通过 open()和 write()系统调用导出 GPIO，然后设置 GPIO 为输入，读取 GPIO 的输入值。操作范例如所程序所示。

```
/*gpio.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>
#include <string.h>
#define DEV_PATH "/sys/class/gpio/gpio49/value" //输入输出电平值设备
#define EXPORT_PATH "/sys/class/gpio/export" //GPIO 设备导出设备
#define DIRECT_PATH "/sys/class/gpio/gpio49/direction" //GPIO 输入输出控制设备
#define OUT "out"
#define IN "in"
#define GPIO "49" //GPIO49
#define HIGH_LEVEL "1"
#define LOW_LEVEL "0"
int main(int argc, char ** argv)
{
	static int fd_dev, fd_export, fd_dir, ret;
	char buf[10], direction[4];
	fd_export = open(EXPORT_PATH, O_WRONLY); //打开 GPIO 设备导出设备
	if(fd_export < 0) {
		perror("open export:");
		return -1;
	}
	write(fd_export, GPIO, strlen(GPIO));
	fd_dev = open(DEV_PATH, O_RDWR); //打开输入输出电平值设备
	if(fd_dev < 0) {
		perror("export write:");
		return -1;
	}
	fd_dir = open(DIRECT_PATH, O_RDWR); //打开 GPIO 输入输出控制设备
	if(fd_dir < 0) {
		perror("export write:");
		return -1;
	}
	ret = read(fd_dir, direction, sizeof(direction)); //读取 GPIO2_4 输入输出方向
	if(ret < 0) {
		perror("dir read:");
		close(fd_export);
		close(fd_dir);
		close(fd_dev);
		return -1;
	}
	printf("default directions:%s",direction);
	strcpy(buf, IN);
	ret = write(fd_dir, buf, strlen(IN));
	if(ret < 0) {
		perror("dir read:");
		close(fd_export);
		close(fd_dir);
		close(fd_dev);
		return -1;
	}
	ret = read(fd_dir, direction, sizeof(direction));
	if(ret < 0) {
		perror("dir read:");
		close(fd_export);
		close(fd_dir);
		close(fd_dev);
		return -1;
	}
	ret = read(fd_dev, buf, sizeof(buf)); //读取 GPIO2_4 输入电平值
	if(ret < 0) {
```

```
            perror("dir read:");
            close(fd_export);
            close(fd_dir);
            close(fd_dev);
            return -1;
        }
        printf("now directions:%sinput level:%s",direction,buf);
        close(fd_export);
        close(fd_dir);
        close(fd_dev);
        return 0;
}
```

程序执行结果：

```
[root@Loongson:/]#./gpio
default directions:in
now directions:in
input level:1
```

8.3 按键应用层编程

8.3.1 按键操作接口

开发板上有三个独立的按键，内核中已经加载了按键的驱动程序。

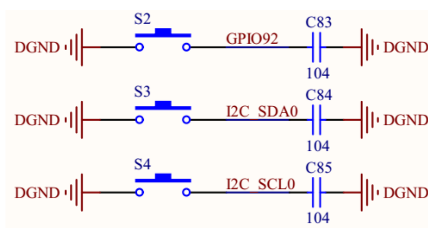


图 8.2 按键接口硬件原理图

配置内核的信息如下：

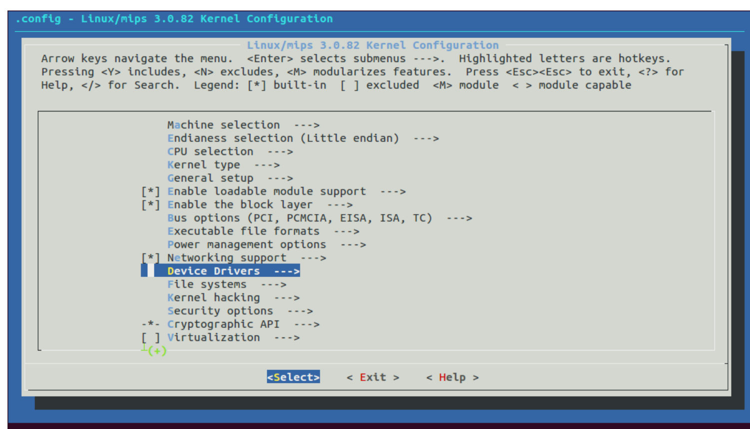


图 8.3 虚拟机中内核配置按键操作 1

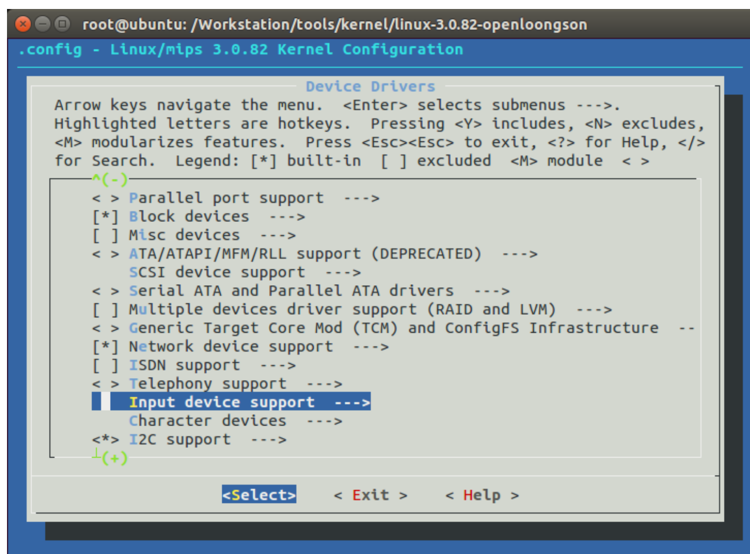


图 8.4 虚拟机中内核配置按键操作 2

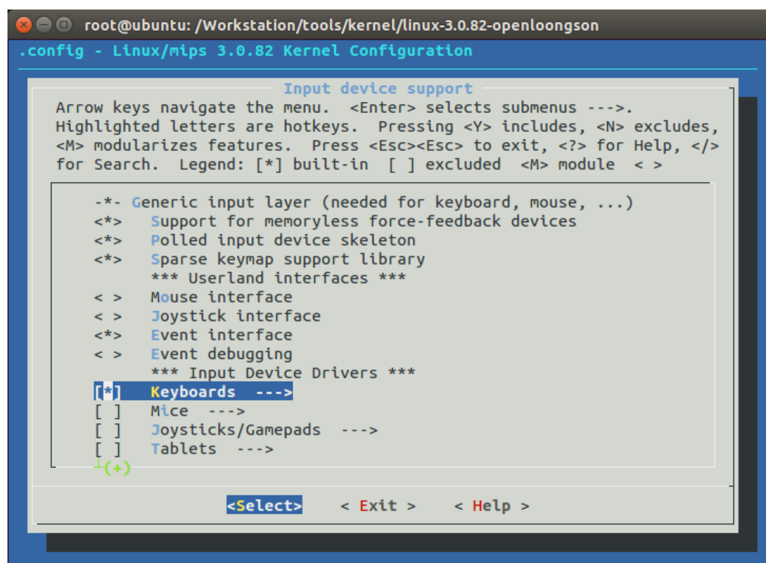


图 8.5 虚拟机中内核配置按键操作 3

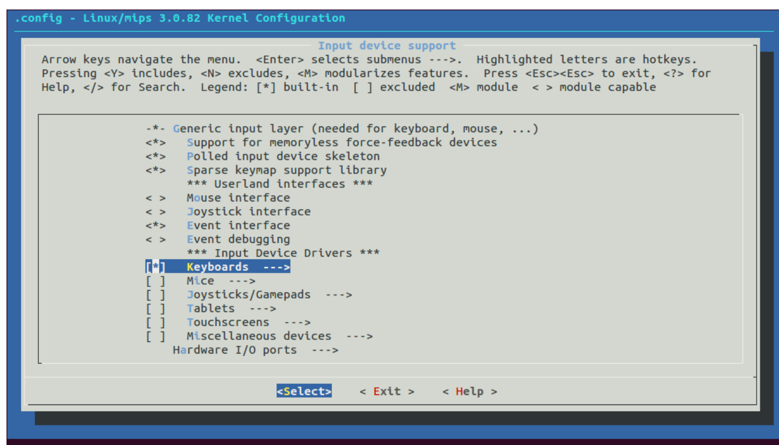


图 8.6 虚拟机中内核配置按键操作 4

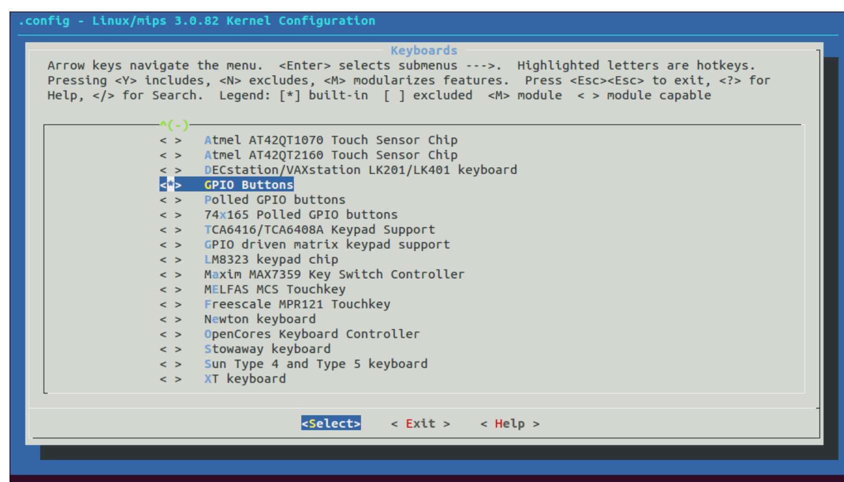


图 8.7 虚拟机中内核配置按键操作 5

驱动模块已经加载入内核，启动完成后，在/dev/input 目录生成设备文件，生成的设备文件为/dev/input/event0，应用程序可以读取按键事件，应用程序代码中必须包含<Linux/input.h>头文件。可以用以下命令查看设备信息：

```
[root@Loongson:/dev/input]#cat /proc/bus/input/devices
I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="gpio-keys"
P: Phys=gpio-keys/input0
S: Sysfs=/devices/platform/gpio-keys/input/input0
U: Uniq=
H: Handlers=kbd event0
B: PROP=0
B: EV=100003
B: KEY=80c
```

I line：这行包含身份信息，显示了 bus type 是 19， vendor， product， version 等信息。

N line：这行包含了名字信息。

P line：这行包含了物理设备信息。

H line：这行包含了与设备关联的 handler drivers。

B line：这些行包含了显示设备能力的一些位域 (bitfield)。

本机键盘对应的事件类型是 event0。

cat /proc/interrupts 命令查看中断信息：

```
[root@Loongson:/]#cat /proc/interrupts
           CPU0
  5:        926   LS1X-INTC   serial
 13:       6522   LS1X-INTC   ls1x-nand
 21:          0   LS1X-INTC   rtc-mach0
 22:          0   LS1X-INTC   rtc-mach1
 23:          0   LS1X-INTC   rtc-mach2
 27:      41309   LS1X-INTC   rtc-tick
 32:        221   LS1X-INTC   ehci_hcd:usb1
 33:          1   LS1X-INTC   ohci_hcd:usb2
 34:         24   LS1X-INTC   dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb3
 35:        911   LS1X-INTC   eth0
149:          9   LS1X-INTC   0
150:         15   LS1X-INTC   1
162:          0        MIPS   cascade
163:          0        MIPS   cascade
164:          0        MIPS   cascade
165:          0        MIPS   cascade
166:          0        MIPS   cascade
167:     630346        MIPS   timer
ERR:         10
```

可用命令 cat /dev/input/event0 查看按键的事件信息，

```
[接收]3B A2 00 00 06 AD 0D 00 01 00 0B 00 01 00 00 00 3B A2 00 00 17 AD 0D 00 00 00 00 00 00 00 00 00 00
[接收]3C A2 00 00 D3 AC 00 00 01 00 0B 00 00 00 00 00 3C A2 00 00 DC AC 00 00 00 00 00 00 00 00 00 00 00
[接收]3C A2 00 00 7B 7A 0B 00 01 00 02 00 01 00 00 00 3C A2 00 00 8C 7A 0B 00 00 00 00 00 00 00 00 00 00
[接收]3C A2 00 00 D4 DB 0D 00 01 00 02 00 00 00 00 00 3C A2 00 00 E0 DB 0D 00 00 00 00 00 00 00 00 00 00
```

以上为 KEY2 按下、KEY2 抬起、KEY3 按下、KEY3 抬起调试串口接收到的 16 进制数据。

8.3.2 在程序中操作按键

C 程序中操作按键，首先打开设备 event0，然后读取设备，如果其属性为 EV_KEY，则有按键按下。再读取属性 value,相应的值对应的信息为：

表 8.4 按键键值表

2	-2	11	-11	3	-3
KEY3 pressed	KEY3 relaese	KEY2 pressed	KEY2 relaese	KEY1 pressed	KEY1 relaese

```
/* key.c*/
/ * Name              : key.c
  * Author          : sundm75
  * Version         : v1.0
  * Copyright       : 2016    by sundm75
  * Description : key in C, Ansi-style
  * Change Logs:
  * Date                  Author            Notes
  * 2016-09-25         sundm75           first version
  */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>
#define DEV_PATH "/dev/input/event0"      //difference is possible

int main(int argc, char ** argv)
{
        int keys_fd;
        struct input_event t;
        int direct = 0;
        keys_fd=open(DEV_PATH, O_RDONLY);
        if(keys_fd <= 0)
        {
                printf("open /dev/input/event0 device error!\n");
                return -1;
        }
        while(1)
        {
                if(read(keys_fd, &t, sizeof(t)) == sizeof(t))
                {
                        if(t.type==EV_KEY)
                                //if(t.value==0 || t.value==1)
                                {
                                        printf("key %d %s\n", t.code, (t.value) ? "Pressed" : "Released");
                                        direct = (t.value) ? 1 : (-1);//1 : pressed -1:Released
                                }
                }
        }
        close(keys_fd);
        return t.code * direct;//
        //2 -2                            11 -11                              3 -3
        //KEY3 pressed relaese    KEY2 pressed relaese    KEY1 pressed relaese
}
```

观察结果，当有键按下，调试串口打印如下信息:

```
[root@Loongson:/]#./key
key 2 Pressed
key 2 Released
key 11 Pressed
key 11 Released
key 11 Pressed
key 11 Released
key 2 Pressed
key 2 Released
```

8.4 SD 卡和 U 盘

8.4.1 U 盘

1）进入 linux 系统后，在 PuTTY 中输入命令 **fdisk –l**。

2）再执行 U 盘挂载命令 / # mount –t vfat /dev/sda1 /mnt，即把该设备节点/dev/sda1 挂载到/mnt 目录下，就可以看到 U 盘内容；执行 U 盘卸载命令， / # umount /mnt， 就可以安全卸载 U 盘。

```
[root@Loongson:/]#fdisk -l

Disk /dev/mtdblock0: 14 MB, 14680064 bytes
255 heads, 63 sectors/track, 1 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/mtdblock0 doesn't contain a valid partition table

Disk /dev/mtdblock1: 52 MB, 52428800 bytes
255 heads, 63 sectors/track, 6 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/mtdblock1 doesn't contain a valid partition table

Disk /dev/mtdblock2: 67 MB, 67108864 bytes
255 heads, 63 sectors/track, 8 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/mtdblock2 doesn't contain a valid partition table

Disk /dev/mmcblk0: 7744 MB, 7744782336 bytes
149 heads, 39 sectors/track, 2603 cylinders
Units = cylinders of 5811 * 512 = 2975232 bytes

        Device Boot      Start         End      Blocks  Id System
/dev/mmcblk0p1               2        2604     7559168   b Win95 FAT32

Disk /dev/sda: 2116 MB, 2116550656 bytes
2 heads, 63 sectors/track, 32808 cylinders
Units = cylinders of 126 * 512 = 64512 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1   *           1       32809     2066928   b Win95 FAT32
[root@Loongson:/]#mount -t vfat /dev/sda1 /mnt
[root@Loongson:/]#ls /mnt
! 360U????????              ???14??????
2017??????                  System Volume Information
[root@Loongson:/]# umount /mnt
[root@Loongson:/]#ls /mnt
[root@Loongson:/]#
```

8.4.2 SD 卡

1）开发板进入 linux 系统后，在终端中输入命令 ls /dev | grep mmc -l 看 是否存在 SD 卡的设备文件节点/dev/mmc0。

```
[root@Loongson:/]# ls /dev | grep mmc -
mmcblk0
mmcblk0p1
```

2） 创建 SD 卡临时挂载目录：

```
[root@Loongson:/]#mkdir /mnt/sd
[root@Loongson:/]#ls /mnt/
sd
```

3） 挂载 SD 卡文件系统： 即把 SD 的设备节点挂载到/mnt/sd 目录下，这样在此目录下读写 文件了。

输入 df –m 命令可以查看， TF 卡各分区的挂载的情况和分区的使用，挂载前：

```
[root@Loongson:/]#df -m
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/root                   50        15        35  31% /
devtmpfs                     0         0         0   0% /dev
tmpfs                        0         0         0   0% /dev
tmpfs                       12         0        12   0% /tmp
```

挂载之后输入命令显示：

```
[root@Loongson:/]#df -m
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/root                   50        15        35  31% /
devtmpfs                     0         0         0   0% /dev
tmpfs                        0         0         0   0% /dev
tmpfs                       12         0        12   0% /tmp
/dev/mmcblk0p1            7378         0      7378   0% /mnt
[root@Loongson:/]#
```

4)取消挂载

取消挂载，以下三条选择其一即可。

```
[root@Loongson:/]#umount /dev/mmcblk0p1 /mnt/sd
[root@Loongson:/]#umount /mnt/sd
[root@Loongson:/]#umount /dev/mmcblk0p1
```

现在就可以像操作 U 盘一样操作 SD 卡了。

8.5 RTC 时钟

设置并保存系统实时时钟。

Linux 中更改时间的方法一般使用 date 命令，为了把 lonngs 1C 内部带的时钟与 linux 系 统时钟同步，一般使用 hwclock 命令，下面是它们的使用方法：

（1）设置时间 2017-04-06 14:59

```
[root@Loongson:/]#date -s "2017-01-01 00:00"
Sun Jan  1 00:00:00 UTC 2017
[root@Loongson:/]#
```

（2）把刚刚设置的时间存入 loongson 1C 内部的 RTC

```
[root@Loongson:/]#hwclock -w
```

（3）开机时使用 hwclock -s 命令可以恢复 linux 系统时钟为 RTC，一般把该语句放入/etc/init.d/rcS 文件自动执行。

```
[root@Loongson:/]#hwclock -s
```

（4）读取 RTC 的时间

```
[root@Loongson:/]#hwclock
Sun Jan  1 00:01:08 2017   0.000000 seconds
```

注意： hwclock 命令需要读取/dev/rtc 设备节点， lonngs 1C 的 RTC 驱动注册后会在/dev 目 录下建立/dev/rtc0 节点，可以使用命令 ln -s rtc0 rtc 建立一个 rtc 的设备节点，

这样 hwclock 命令就可以读取 RTC 的时间。

```
COM8 - PuTTY
[root@Loongson:/]#hwclock -s
[root@Loongson:/]#hwclock
Sun Jan  1 00:01:08 2017  0.000000 seconds
[root@Loongson:/]#hwclock -s
[root@Loongson:/]#hwclock
Sun Jan  1 00:01:56 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:01:58 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:01:59 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:00 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:01 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:02 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:06 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:07 2017  0.000000 seconds
[root@Loongson:/]#hwclock
Sun Jan  1 00:02:07 2017  0.000000 seconds
[root@Loongson:/]#
```

图 8.8 控制台操作 RTC 时钟示意图

8.6 串口读写

8.6.1 串口硬件说明

开发板的串口如下所示。

UART1-（98，99） GPIO3、2 第四复用

UART2- （100，97）GPIO5、4 第四复用 用于控制台

19	EJTAG_TDI	99	2	CAMVSYNC	I2C_SDA1	CAN1_RX	UART1_RX		GPIO
20	EJTAG_RST	100	5	CAMDATA0	I2C_SCL2	PWM1	UART2_TX		GPIO
21	EJTAG_TMS	97	4	CAMDATA1	I2C_SDA2	PWM0	UART2_RX		GPIO
22	EJTAG_TDO	98	3	CAMHSYNC	I2C_SCL1	CAN1_TX	UART1_TX		GPIO
23	EJTAG_SEL	95	0	CAMCLKOUT	I2C_SDA0	CAN0_RX	UART3_RX	SDRAM_CS1	GPIO
24	EJTAG_TCK	96	1	CAMPCLKIN	I2C_SCL0	CAN0_TX	UART3_TX		GPIO

图 8.9 UART 硬件接口复用示意图

如果需要启用串口 1，必须在平台文件中恢复以下代码，打开串口 1 的第 4 复用：

```
/* UART1 */
__raw_writel(__raw_readl(LS1X_CBUS_FIRST0) & (~0x00060000), LS1X_CBUS_FIRST0);
__raw_writel(__raw_readl(LS1X_CBUS_FIRST3) & (~0x00000060), LS1X_CBUS_FIRST3);
__raw_writel(__raw_readl(LS1X_CBUS_SECOND1) & (~0x00000300), LS1X_CBUS_SECOND1);
__raw_writel(__raw_readl(LS1X_CBUS_SECOND2) & (~0x00003000), LS1X_CBUS_SECOND2);
__raw_writel(__raw_readl(LS1X_CBUS_FOURTHT0) | 0x0000000c, LS1X_CBUS_FOURTHT0);
```

使用以下命令查询当前控制台使用串口情况：

```
[root@Loongson:/]#tty
/dev/ttyS2
```

当前控制台使用串口 2。

8.6.2 用 minicom 操作串口

在 linux 下设置 pl2303 串口，在 ubuntu 下，接入接口则自动安装。插上 USB 转 TTL 小

板下，自动识别。

```
lsmod | grep pl2303      //首先查看 pl2303 驱动是否正确安装，在 ubuntu 下，接入接口则自动安装
ls /dev | grep ttyUSB*    //查看所使用的设备端口，在本机电脑上显示为 ttyUSB0
```

```
root@ubuntu:/# lsmod | grep pl2303
pl2303                 20480  0
usbserial              40960  1 pl2303
root@ubuntu:/# ls /dev | grep ttyUSB
ttyUSB0
root@ubuntu:/#
```

图 8.10 虚拟机中查看 PL2303 安装情况

以下进入 minicom 安装设置。

在虚拟机中安装 minicom。

```
root@ubuntu:~# apt-get install minicom
……..
Setting up minicom (2.7-1) ...
root@ubuntu:~#
```

打开终端，在命令行中输入“minicom -s”,进行设置。将串口号改为 ttyUSB0 .

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : Yes                                       |
| G - Software Flow Control : No                                        |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

图 8.11 虚拟中 minicom 界面

在 Modem and dialing 里进行配置，将 A、B、K 的配置字符改为空。

```
+--------------------[Modem and dialing parameter setup]---------------------+
|                                                                            |
| A - Init string .........                                                  |
| B - Reset string ........                                                  |
| C - Dialing prefix #1.... ATDT                                             |
| D - Dialing suffix #1.... ^M                                               |
| E - Dialing prefix #2.... ATDP                                             |
| F - Dialing suffix #2.... ^M                                               |
| G - Dialing prefix #3.... ATX1DT                                           |
| H - Dialing suffix #3.... ;X4D^M                                           |
| I - Connect string ...... CONNECT                                          |
| J - No connect strings .. NO CARRIER            BUSY                       |
|                           NO DIALTONE           VOICE                      |
| K - Hang-up string ......                                                  |
| L - Dial cancel string .. ^M                                               |
|                                                                            |
| M - Dial time ........... 45      Q - Auto bps detect ..... No             |
| N - Delay before redial . 2       R - Modem has DCD line .. Yes            |
| O - Number of tries ..... 10      S - Status line shows ... DTE speed      |
| P - DTR drop time (0=no). 1       T - Multi-line untag .... No             |
|                                                                            |
| Change which setting?     Return or Esc to exit. Edit A+B to get defaults. |
+----------------------------------------------------------------------------+
```

将串口线一端连接到开发板，另一端连接到主机虚拟机上。打开开发板电源开关，主机

也进入虚拟机系统后，运行“minicom”，就将开发板与主机进行相互通信。其它串口也类似。在 linux 系统中实现了与 Windows 中类似的 PuTTY 的界面。

8.12 虚拟机中实现控制台

在 minicom 中， ctrl+A 后再按 Q，退出系统；ctrl+A 后再按 Z，查看快捷键帮助。

8.6.3 用接口操作串口

由于在控制台已经配置好串口 2，下面在命令行启用以下命令操作串口 1：

```
stty -F /dev/ttyS1 115200    //配置串口 1 为波特率为 115200
```

用另外一根串口线连接到串口 1（GPIO3、2），在 windows 中打开一个串口助手。在开发板上运行以下命令：

```
echo "Hi, I am Loongson! ">/dev/ttyS1    //将数据”abc” 打印到串口 1（PC 机的串口助手）上
cat /dev/ttyS1    //串口 1 输入数据（来自于 PC 机的串口助手），打印到控制台上
```

控制台执行结果如下：

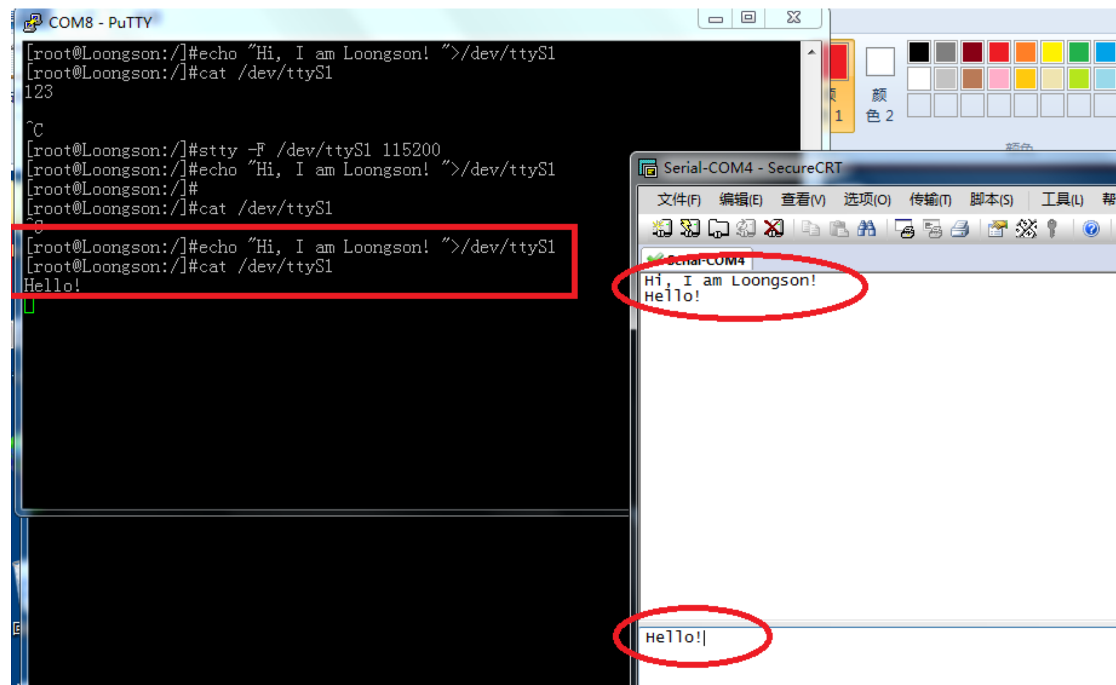


图 8.13 开发板控制台与串口助手合作操作串口

8.6.4 在程序中操作串口

实例程序 uart.c，程序中操作按键，首先打开设备/dev/ttyS1, 配置设备，然后向设备写字

符串"Hello! I am Loongson!"， 最后打印出从设备中读取的字符串。

```
/*uart.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <asm/termios.h>

#include "serial.h"

#define DEV_NAME    "/dev/ttyS1"


int main (int argc, char *argv[])
{
        int fd;
        int len, i,ret;
              char buf[] = "Hello! I am Loongson!";


        fd = open(DEV_NAME, O_RDWR | O_NOCTTY);
              if(fd < 0) {
                          perror(DEV_NAME);
                          return -1;
              }
          ret = set_port_attr (
                                          fd,
                                          B115200,            // B1200 B2400 B4800 B9600 .. B115200
                                          8,                  // 5, 6, 7, 8
                                          "1",           //    "1", "1.5", "2"
                                          'N',                // N(o), O(dd), E(ven)
                                          150,           //VTIME
                                          255 );              //VTIME

        if(ret < 0) {
                printf("set uart arrt faile \n");
                exit(-1);
        }

        len = write(fd, buf, sizeof(buf));
        if (len < 0) {
                printf("write data error \n");
                return -1;
        }

        len = read(fd, buf, sizeof(buf));
              if (len < 0) {
                          printf("read error \n");
                          return -1;
              }

        printf("%s \n", buf);

        return(0);
}
```

运行程序，观察结果， PC 机串口助手打印如下信息"Hello! I am Loongson!"，在 PC 串口助手中输入：Hello,I am Sundm75, 则此字符通过 uart1 传入开发板。

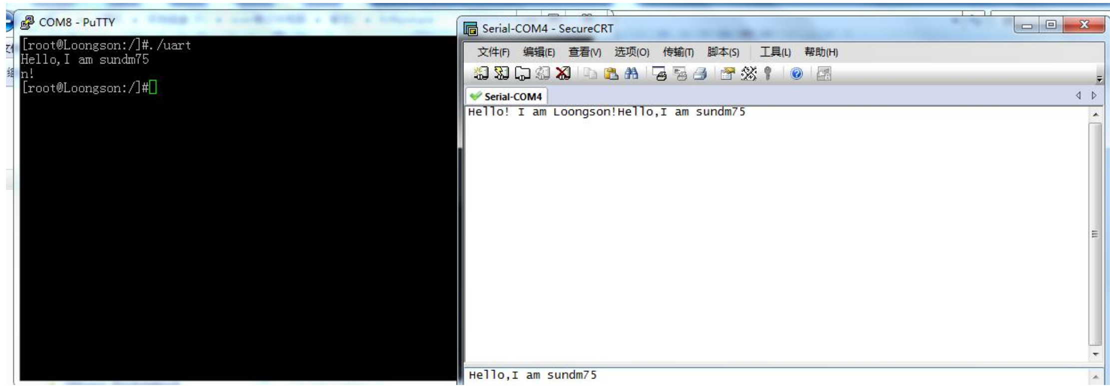


图 8.14 控制台中应用程序操作串口

9. NFS 文件系统搭建

应用程序的移植方式目前主要有四种，目前使用的都是第二种。

第一种：复制到介质（以 U 盘为例）

第二种：通过网络（tftp ）传输文件到开发板（文件较小，推荐使用）

第三种：置于根文件系统目录下制作文件系统镜像，再烧进开发板（文件很大，可以使用）

第四种：通过 NFS（网络文件系统）直接运行

下面介绍第四种。

NFS 不是传统意义上的文件系统，而是访问远程文件系统的协议。全名叫 Network File System，中文叫网络文件系统，是 Linux、 UNIX 系统的分布式文件系统的一个组成部分，可实现在不同网络上共享远程文件系统。NFS 由 Sun 公司开发，目前已经成为文件服务的一种标准之一 （RFC1904，RFC1813）。其最大的功能就是可以通过网络，让不同操作系统的计算机可以共享数据，所以可以把 NFS 看做是一个文件服务器。可以将远端所分享出来的档案系统，挂载 (mount) 在本地端的系统上，然後就可以很方便的使用远端的档案，而操作起来就像在本地操作一样，不会感到有甚麼不同。 NFS 缺点是其读写性能比本地硬盘要差一些。下面则是 nfs 搭建的步骤。

NFS 主要通过两个 daemon 来进行控制：

1.rpc.nfsd ------- 它用来控制客户端是否可以连接到，NFS server

2.rpc.mountd----它用来控制客户端连接后是否有权限对文件进行操作，主要是依据 /etc/exports 文件的设置

9.1 在虚拟机端安装 NFS

使用 apt-get 命令安装（有互联网）,注意 portmap 已经被 rpcbind 取代。

```
apt-get install nfs-common
apt-get install nfs-kernel-server
```

9.2 配置虚拟机 NFS

（1）修改配置文件

```
vim    /etc/exports
```

添加内容为：

```
/home/nfs/nfsrootfs 193.169.2.*(rw,no_root_squash,sync,no_subtree_check)
```

函数参数内容含义：

/home/nfs/nfsrootfs ：要共享的目录，需要先创建后改变权限 ；

*：网段内所有值 ；

rw：读写权限 ；

sync ：资料同步写入内在和硬盘 ；

no_root_squash ：nfs 客户端共享目录使用者权限；

no_subtree_check ：即使输出目录是一个子目录，nfs 服务器也不检查其父目录的权限，这样可以提高效率；

修改后保存退出。

（2）创建服务文件目录，更改权限

```
mkdir /home/nfs/nfsrootfs
```

```
chmod 777 /home/nfs/nfsrootfs
```

（3）启动端口转发；启动 NFS 服务

```
/etc/init.d/rpcbind restart
/etc/init.d/nfs-kernel-server restart
```

显示：

```
root@ubuntu:/etc/init.d# /etc/init.d/rpcbind restart
root@ubuntu:/etc/init.d# /etc/init.d/nfs-kernel-server restart
 * Stopping NFS kernel daemon                                        [ OK ]
 * Unexporting directories for NFS kernel daemon...                  [ OK ]
 * Exporting directories for NFS kernel daemon...                    [ OK ]
 * Starting NFS kernel daemon                                        [ OK ]
```

（4）显示共享出的目录

```
showmount  -e
```

显示：

```
Export list for ubuntu:
/home/nfs/nfsrootfs 193.169.2.*
```

（5）关闭服务器端

```
/etc/init.d/nfs-kernel-server stop
```

显示：

```
root@ubuntu:/etc/init.d# /etc/init.d/nfs-kernel-server stop
 * Stopping NFS kernel daemon                                        [ OK ]
 * Unexporting directories for NFS kernel daemon...                  [ OK ]
root@ubuntu:/etc/init.d# showmount  -e
clnt_create: RPC: Program not registered
```

9.3 配置单板机 NFS

单板机一端配置，主要是重新编译 1C 开发板的内核，再把此内核更新到开发板。

（1）编译 1C 板子内核，添加 nfs 功能 ，命令 make menuconfig 进行配置界面。

```
[*] Networking support   --->
--- Networking options
Networking options --->
                [*] TCP/IP networking
                [*] IP: kernel level autoconfiguration
                [*] IP: DHCP support
                [*] IP: BOOTP support
                [*] IP: RARP support
File Systems   --->
    [*] Network File Systems   --->
        ---  Network File Systems
        <*>  NFS client support
        [*]        NFS client support for NFS version 3
        [*]              NFS client support for the NFSv3 ACL protocol extension
        [*]        NFS client support for NFS version 4
        [*]              NFS client support for NFSv4.1 (EXPERIMENTAL)
        [*]   Root file system on NFS
```

（2）在虚拟机中编译内核

```
make ARCH=mips CROSS_COMPILE=mipsel-linux-
```

在 PMON 中下载到 1C 开发板上的/dev/mtd0 分区

```
mtd_erase /dev/mtd0
devcp tftp://193.169.2.215/vmlinuz /dev/mtd0
```

9.4 使用 NFS

1）在单板机上挂载 nfs 服务

（1）在虚拟机，把交叉编译后的需要共享的程序或者文件置于共享目录：/home/nfs/nfsrootfs

（2）在单板机中中，挂载虚拟机的 nfs 共享目录：

```
mount -t nfs -o nolock 193.169.2.104:/home/nfs/nfsrootfs    /mnt
```

其中 193.169.2.104 为虚拟机的 IP 地址，可用 ifconfig 命令看到 IP 地址。

这样就把共享目录挂到了单板机/mnt 目录。

（3）使用 NFS 运行程序：

```
cd /mnt
```

可以直接运行当前目录已经交叉编译的程序。

（4）取消挂载：

```
umount /mnt
```

2）建立网络文件系统

(1)在虚拟机上，把自己当前做好的根文件系统 rootfs（三.4.制作根文件系统）移置 nfsrootfs 目录下,同时需要确保相关文件的链接路径不能有错,(当真实环境的 rootfs 使用)，注意这里拷备的是整个目录，不是单个根文件系统镜像。

```
cp rootfs /home/nfs/nfsrootfs -rf
```

(2)重启板子,按空格键进入 pmon,中,设置启动参数:

```
set al /dev/mtd0
set append "g root=/dev/nfs rw"
set append "$append nfsroot=193.169.2.104:/home/nfs/nfsrootfs/rootfs noinitrd init=/linuxrc "
set append "$append ip=193.169.2.230:193.169.2.104:193.169.2.1:255.255.255.0::eth0:off"
set append "$append console=ttyS2,115200"
```

root=/dev/nfs ：指定根文件系统为 /dev/nfs，即 NFS；

rw ：根文件系统挂载为可读写。还可以有 ro 即只读的选项。

nfsroot=193.169.2.104:/home/nfs/nfsrootfs/rootfs :设置网络启动时的 NFS 根名字（虚拟机上根文件系统的位置），如果该字符串不是以 "/"、","、"."开始，默认指向"/tftp-boot";

noinitrd：代表没有使用 ramdisk；

init=/linuxrc ：设置内核执行的初始化进程名，如果该项没有设置，内核会按顺序尝试 /etc/init /bin/init，/sbin/init， /bin/sh，如果所有的都没找到，内核会抛出 kernel panic：的错误。

"ip="后面: 设置本机的 IP。此举是为了连接刚才设置的 IP。这里是一个静态的配置，配置的格式为 ip=本机的 IP 地址: 虚拟机的 IP:网关地址:网络掩码:本机的主机名:网络接口名:off 。如果是 DHCP 获取 IP，那很简单，直接 ip=dhcp 即可。这里配置说明如下：

第一项(193.169.2.230)是单板机的 IP(注意不要和局域网内其他 IP 冲突);

第二项(193.169.2.104)是虚拟机的 IP;

第三项(193.169.2.1)是单板机上网关(GW)的设置;

第四项(255.255.255.0)是子网掩码;

第五项是开发主机的名字(一般无关紧要,可随便填写)

eth0 是网卡设备的名称

console=ttyS2,115200：设置串口 2，115200 波特率。

（3）重启，进入网络文件系统。

在开发板上挂载 NFS 网络文件系统(Linux 中最常用的方法就是采用 NFS 来执行各种程序,这样可以不必花费很多时间下载程序。

高级篇驱动

10． 配置 Eclipse 编程

10.1 用 eclipse 开发应用程序

安装程序：

```
$sudo apt-get install eclipse-platform
$sudo apt-get install eclipse-cdt
```

运行 eclipse 启动程序 eclipse3.8 ：

```
$eclipse
```

启动后选择工程目录：

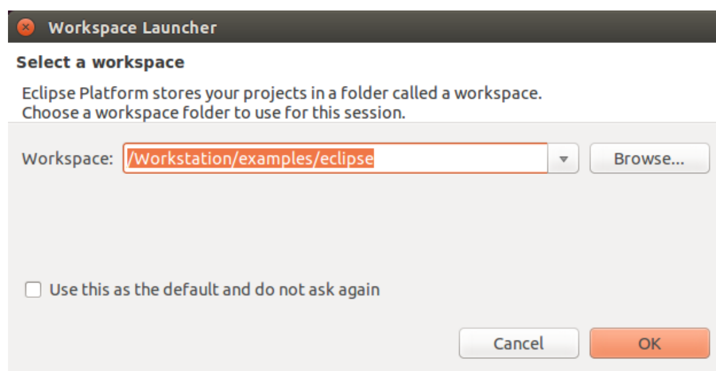


图 10.1 安装 eclipse-选择工程目录

经过启动界面后，新建 C 工程：

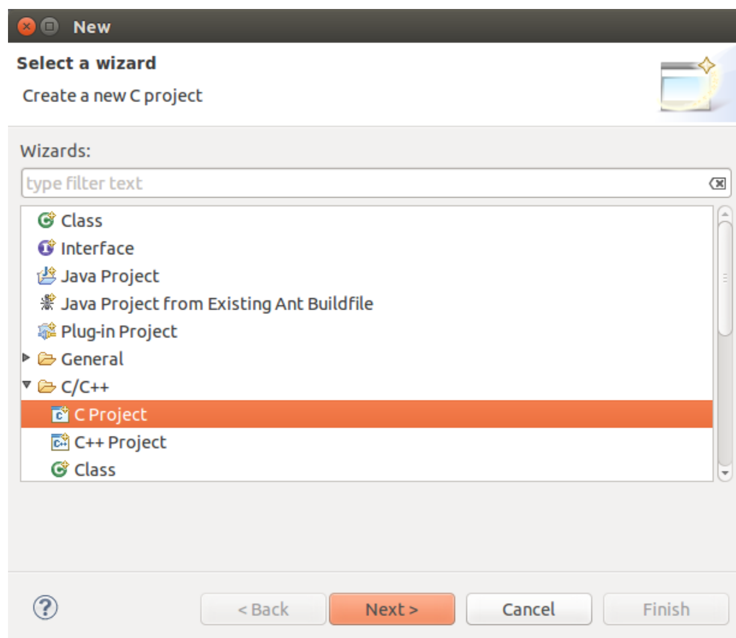


图 10.2 使用 eclipse-新建 C 工程

新建空工程，采用 CrossGCC，重设交叉编译器：

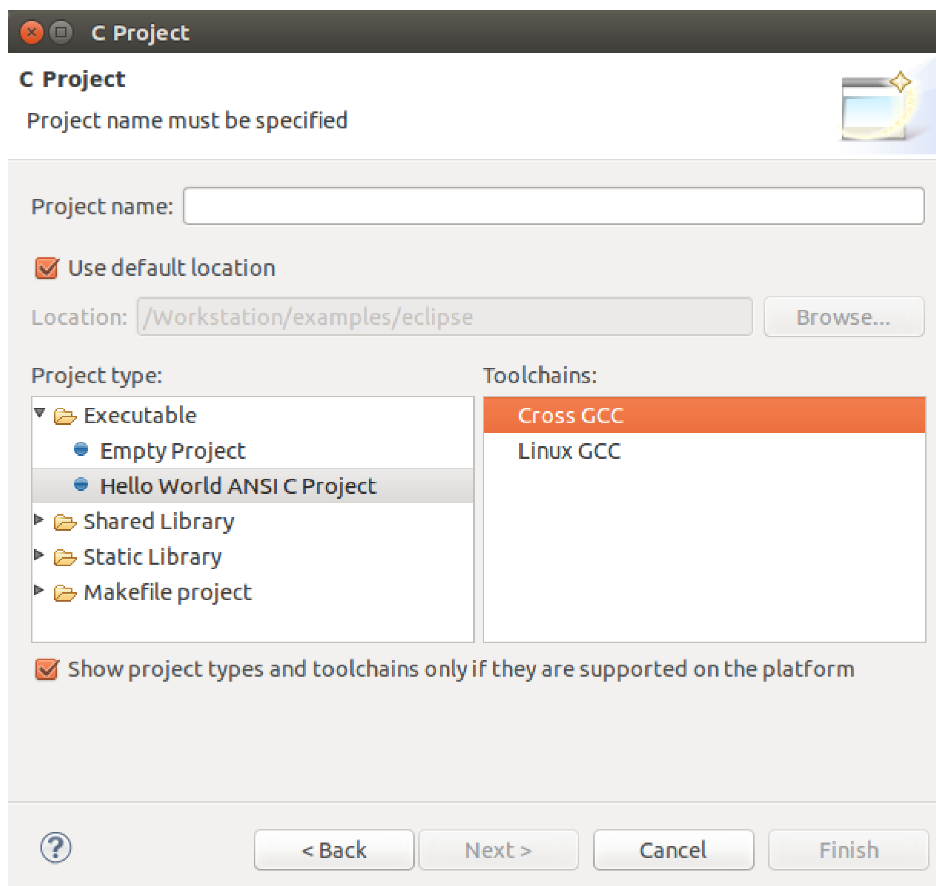


图 10.3 使用 eclipse-重设交叉编译器

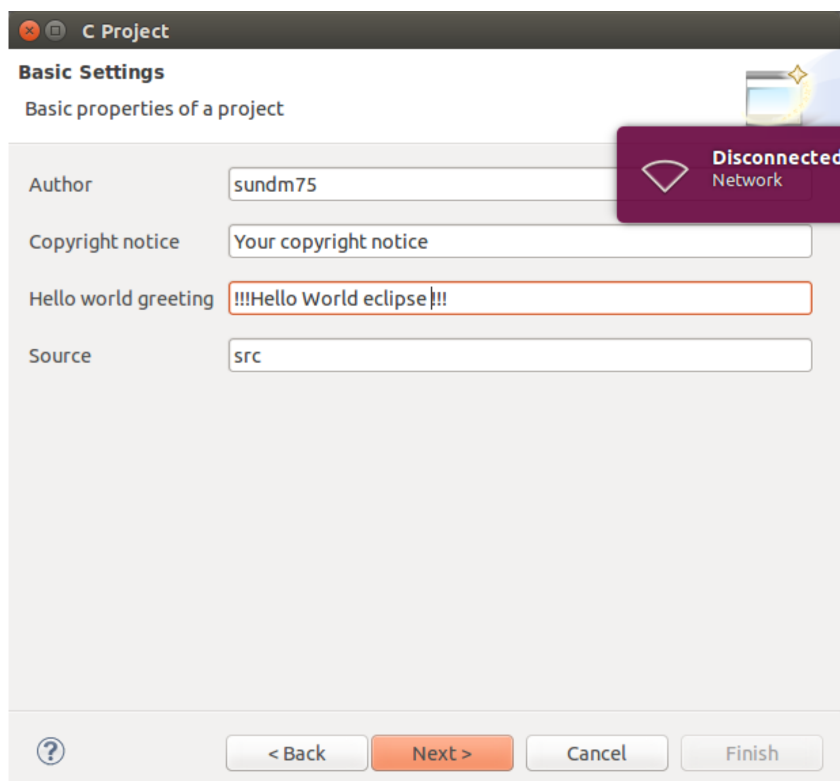


图 10.4 使用 eclipse-新建工程

Next 后，选择工程配置，

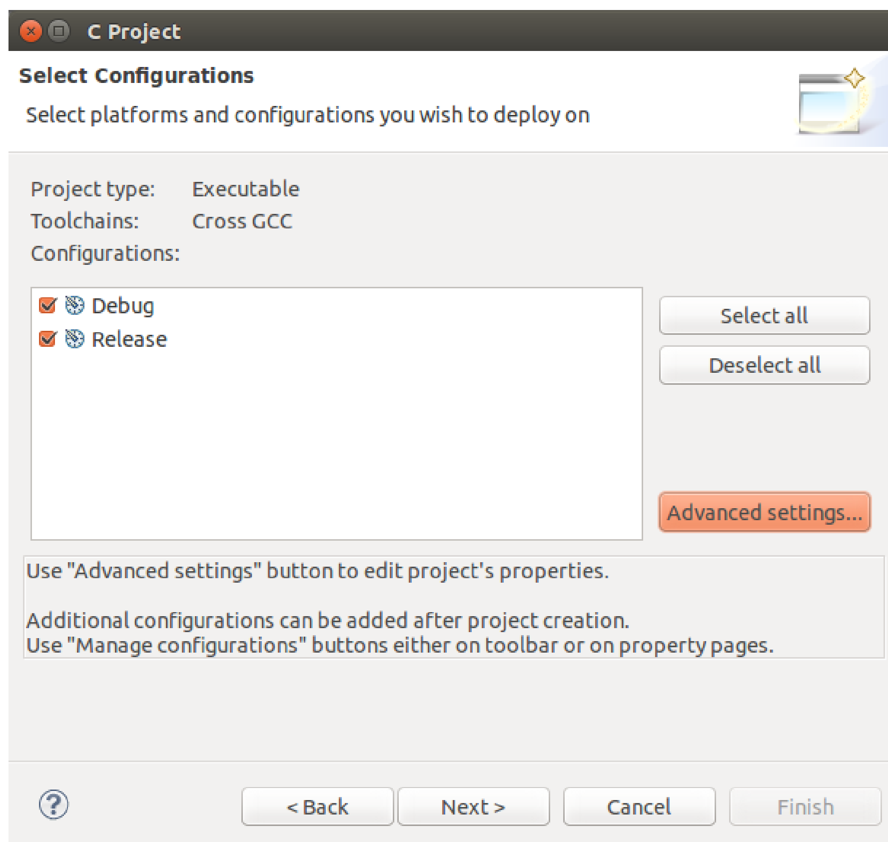


图 10.5 使用 eclipse-进行工程配置

点击 Advanced settings, 点击 Project→Properties，在弹出的工程属性界面，点击 C/C++ Build 的 Toolchain Editor，在 Current toolchain 栏将编译器类型设置为 Corss GCC。

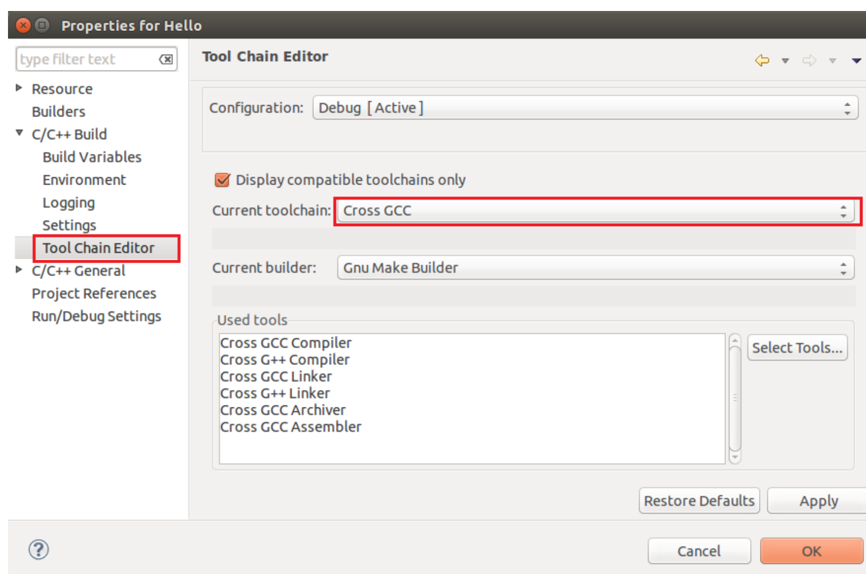


图 10.6 使用 eclipse-配置交叉编译器 1

在点击 C/C++ Build 的 Settings 栏，在 Prefix 栏填写交叉编译器的前缀（如 mipsel-linux-），在 Path 中填写交叉编译器的实际路径 /opt/gcc-4.3-ls232/bin。

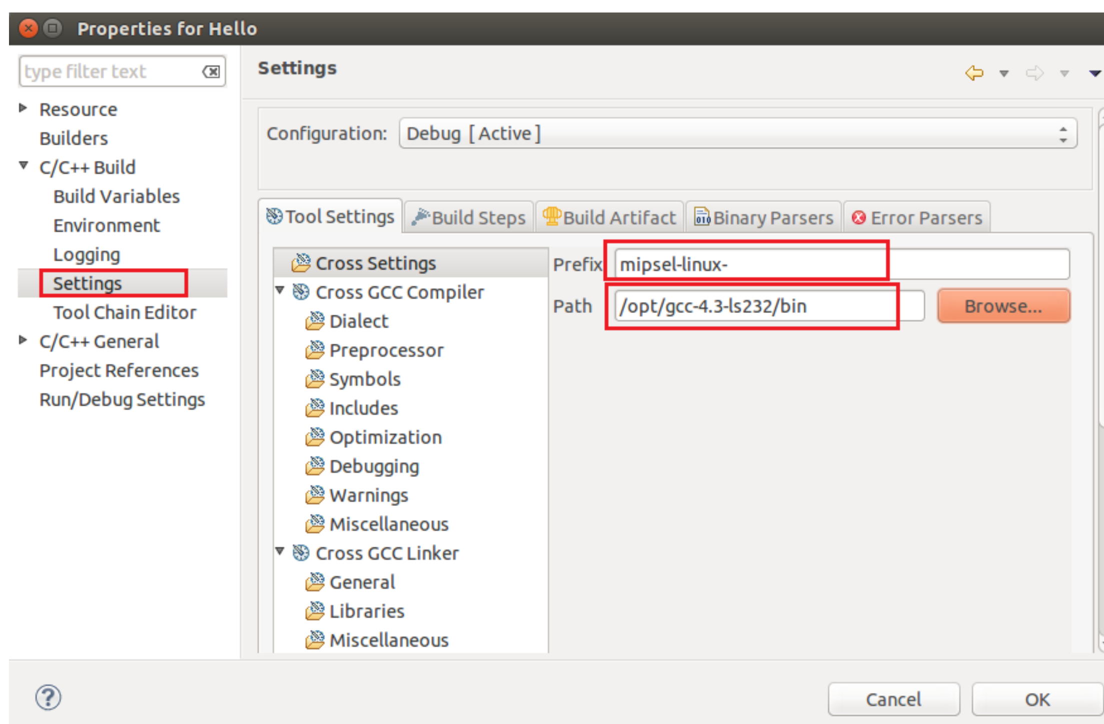


图 10.7 使用 eclipse-配置交叉编译器 2

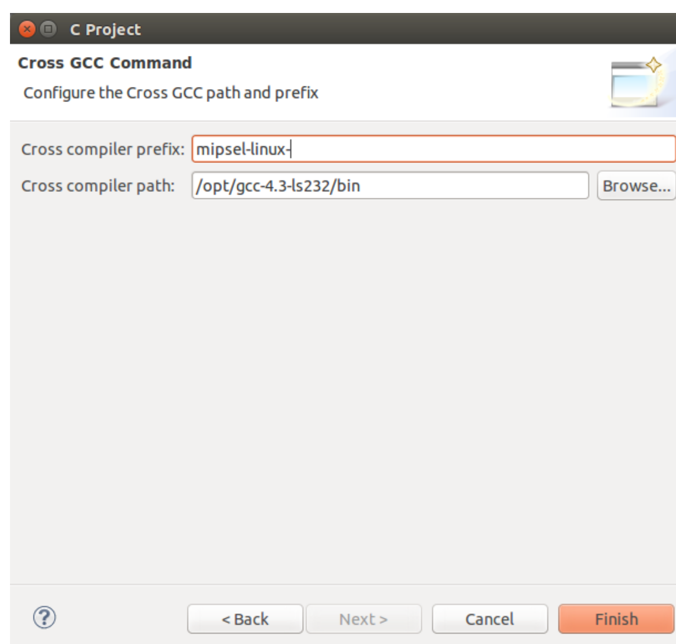


图 10.8 使用 eclipse-配置交叉编译器 3

如果使用 eclipse 交叉编译和远程调试，要装 gdbserver。

下载 gdb-7.11.tar.gz 地址：ftp://ftp.gnu.org/gnu/gdb 。

10.2 用 eclipse 开发 内核模块

参考以下网址资料：

http://blog.csdn.net/cp1300/article/details/8266806

http://blog.chinaunix.net/uid-24512513-id-3183457.html

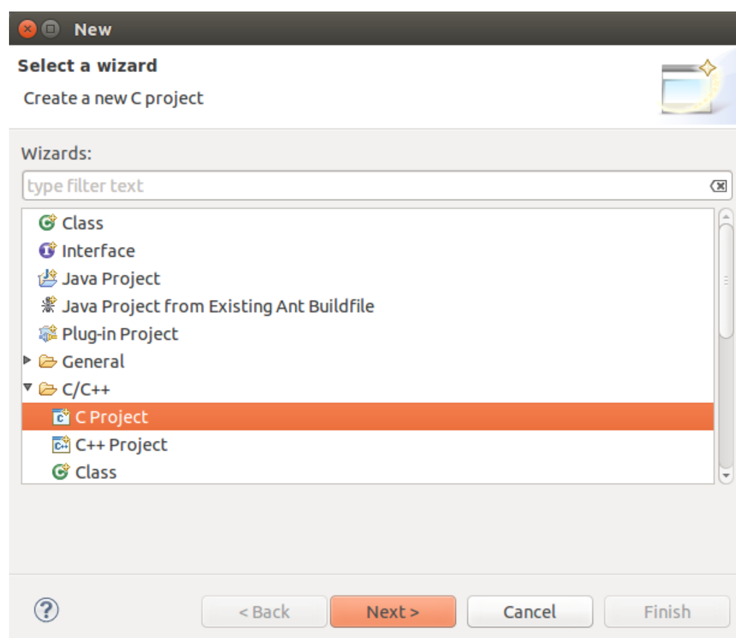


图 10.9 eclipse 开发内核模块新建 C 工程

一直下一步，直到下图位置，填好自己的 arm-linux-gcc 的路径。

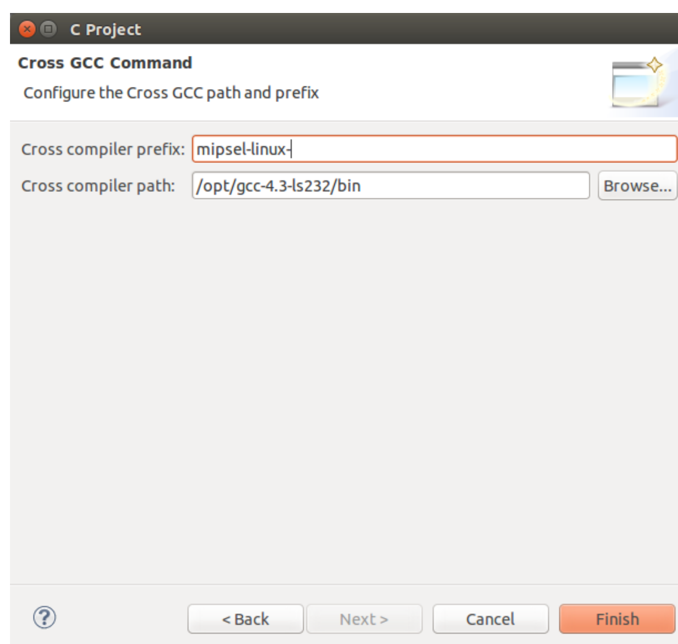


图 10.10 eclipse 设置交叉编译工具链

交叉编译环境下的 4 个头文件已经出现在 include 文件夹中：

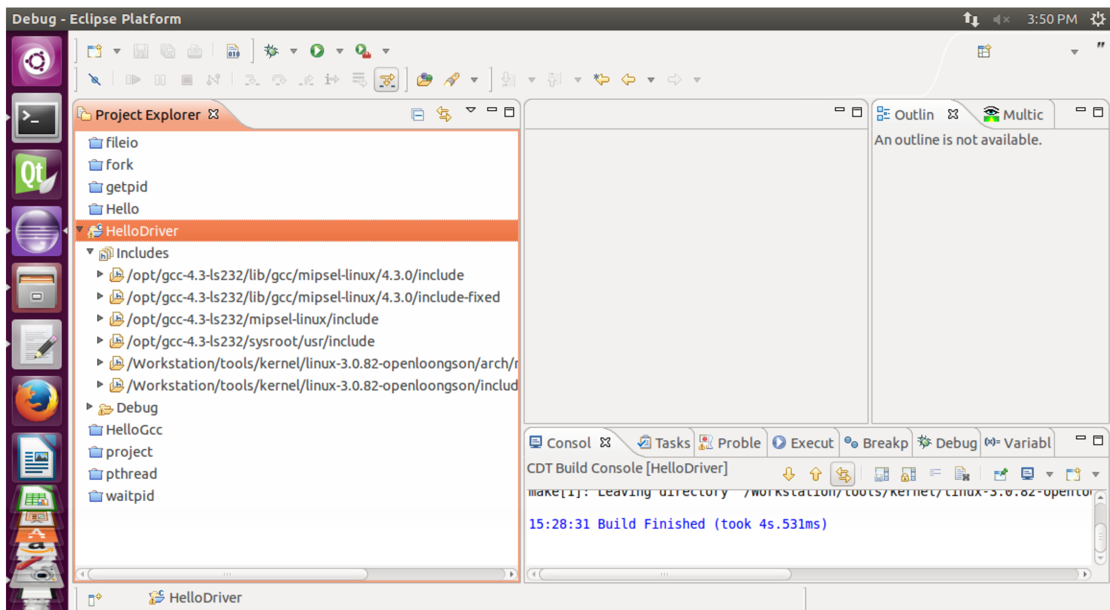


图 10.11 eclipse 配置头文件

下面添加路径和符号，首先要生成符号表（xml）文件。先到内核目录下（/Workstation/tools/kernel/linux-3.0.82-openloongson/include/generated），运行以下命令，将 autoconf.h 文件中定义的宏定义转换成 xml 文件。

```
cat autoconf.h |grep define |awk '{print "<macro><name>" $2 "</name><value>" $3 "</value></macro>"}' > symbol.xml
```

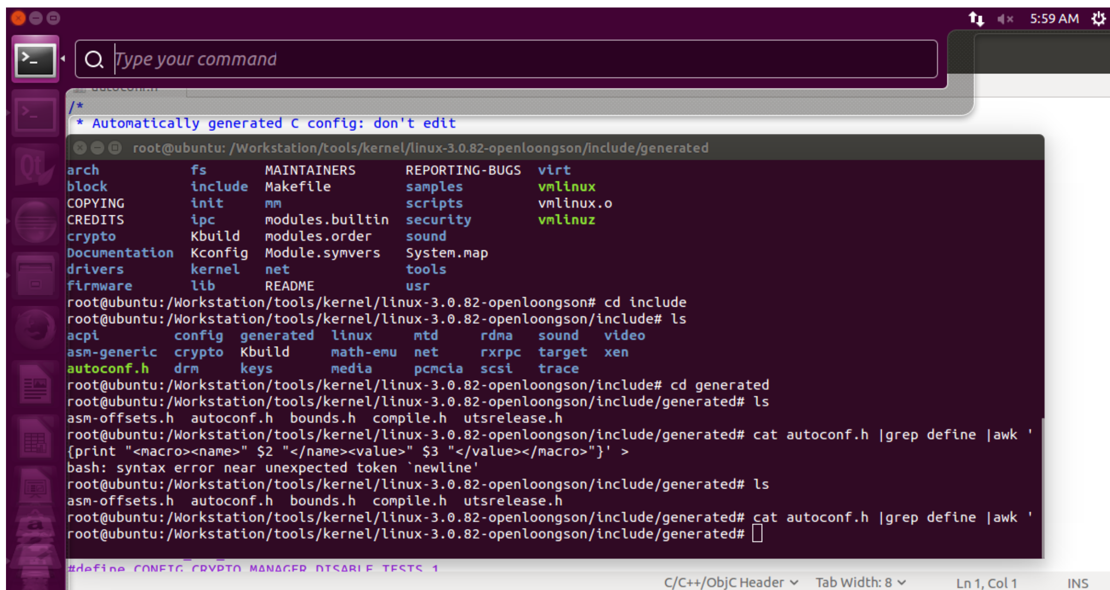


图 10.12 虚拟机中宏定义转换成 xml

生成宏定义文件：symbol.xml：

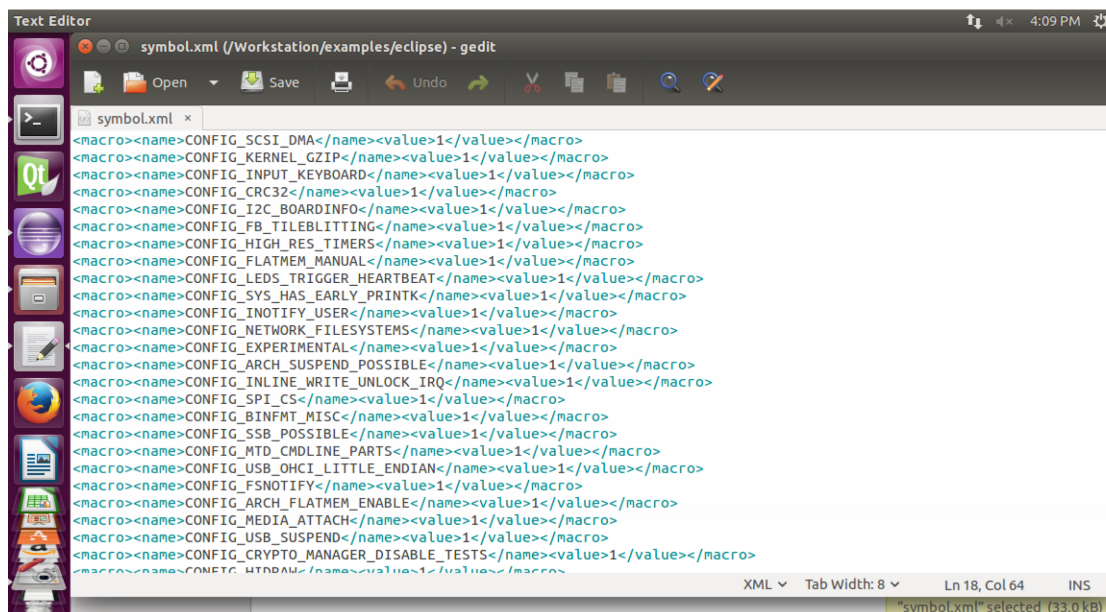


图 10.13 虚拟中显示 symbol.xml 文件

导出符号表 LS1C_ECLIPSE_CONFIG.xml：

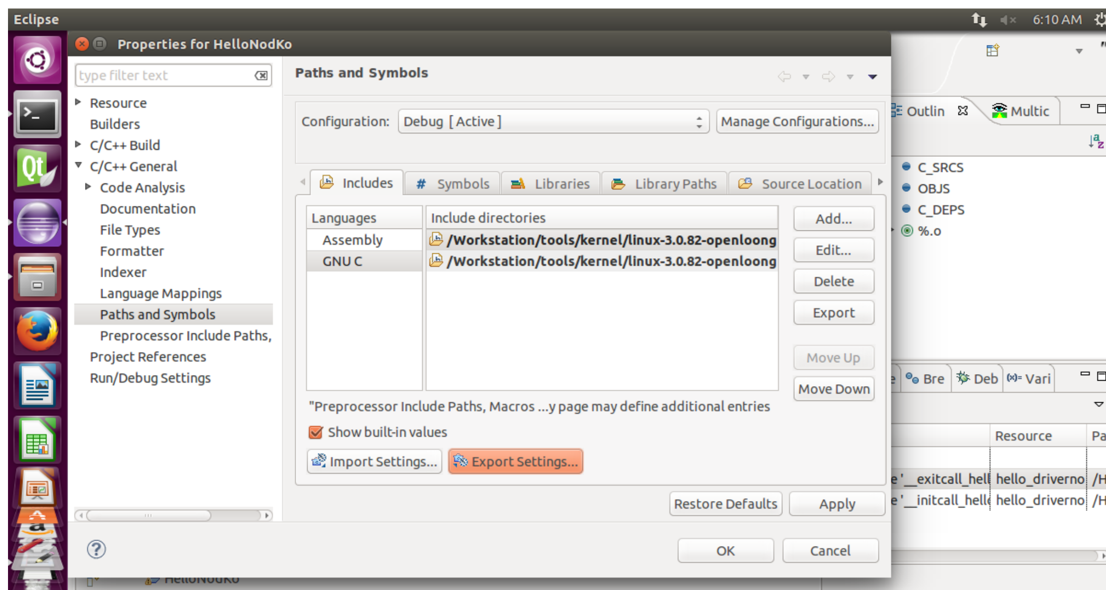


图 10.14 虚拟中显示符号表

在导出的文件中添加刚才生成的 symbo.xml 文件中的所有内容。

```
<language name="C Source File">
<macro>
<name>__KERNEL__</name><value>1${ArchType}</value>
</macro>
/*添加的宏定义*/
<macro><name>CONFIG_SCSI_DMA</name><value>1</value></macro>
<macro><name>CONFIG_KERNEL_GZIP</name><value>1</value></macro>
<macro><name>CONFIG_INPUT_KEYBOARD</name><value>1</value></macro>
<macro><name>CONFIG_CRC32</name><value>1</value></macro>
…………
</language>
```

存盘后，再将此符号表导入工程。

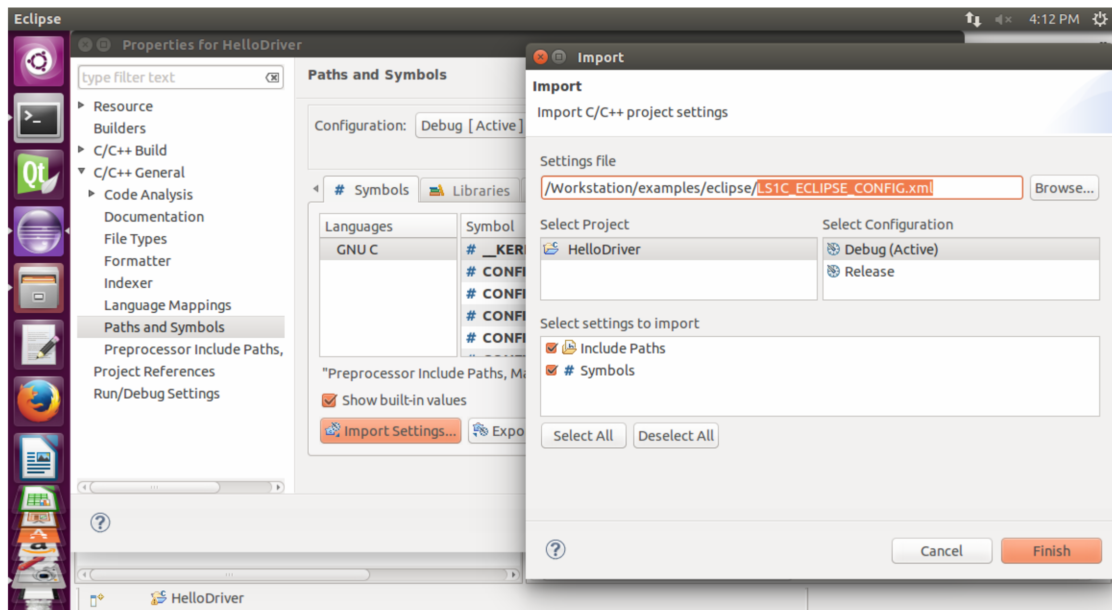


图 10.15 虚拟机中导入工程

再添加 2 个目标： all 、 clean。

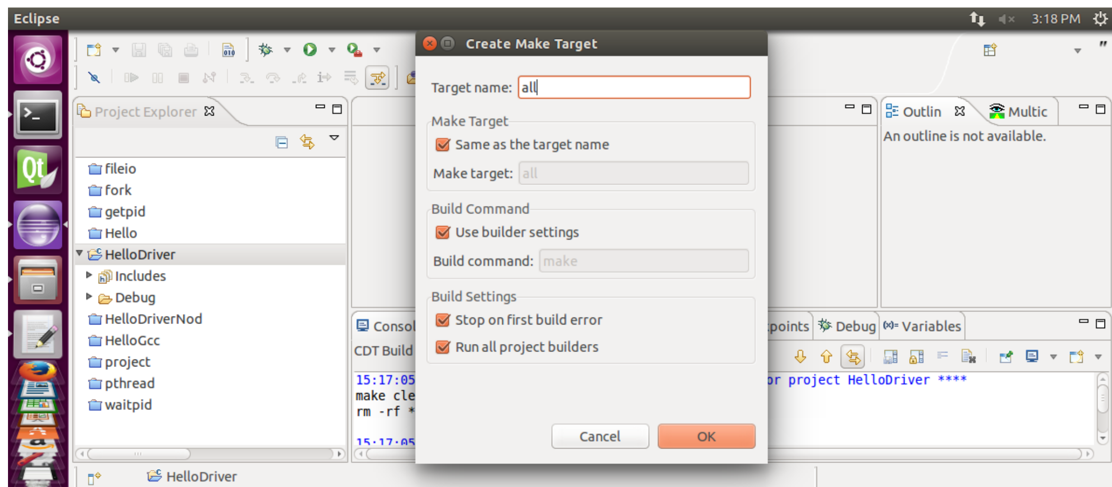


图 10.16 虚拟机中添加编译目标

在 debug 文件夹下添加两文件 hello_drivernod.c 和 Makefile。

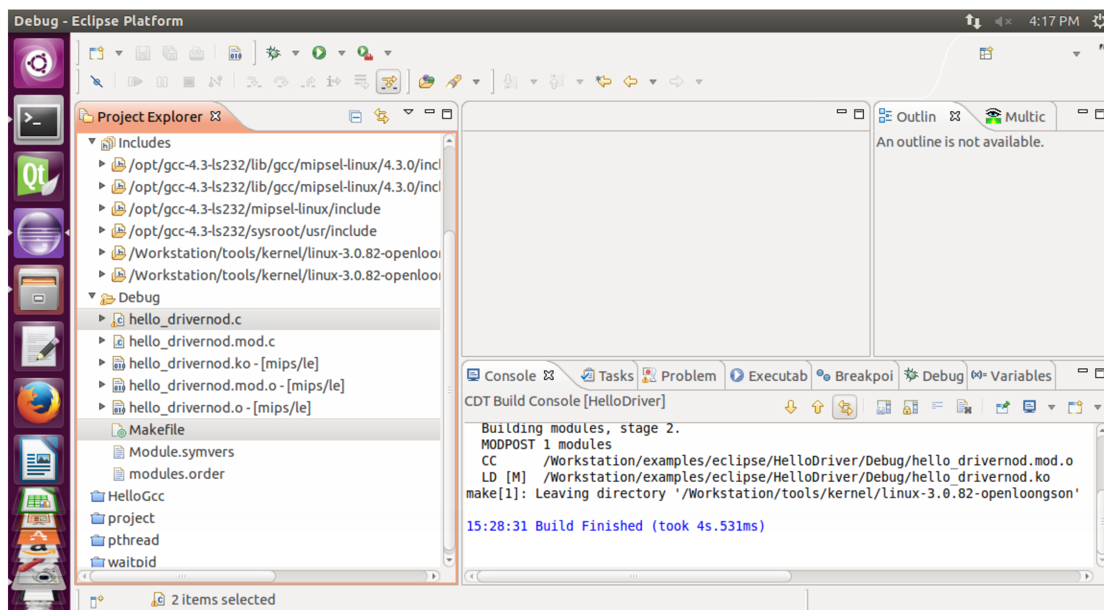


图 10.17 虚拟机中在 debug 中下添加节点和 makefile

将自动生成 makefile 文件的选项去除:

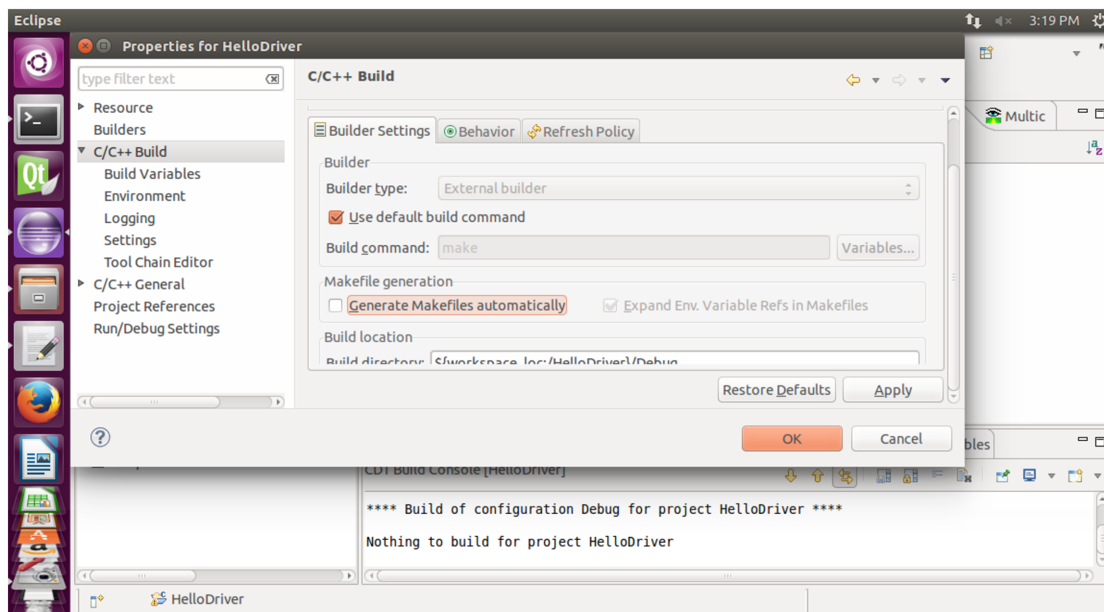


图 10.18 虚拟机中自动生成 makefile 文件的选项去除

将 build 过程中的 all clean 选项选中。

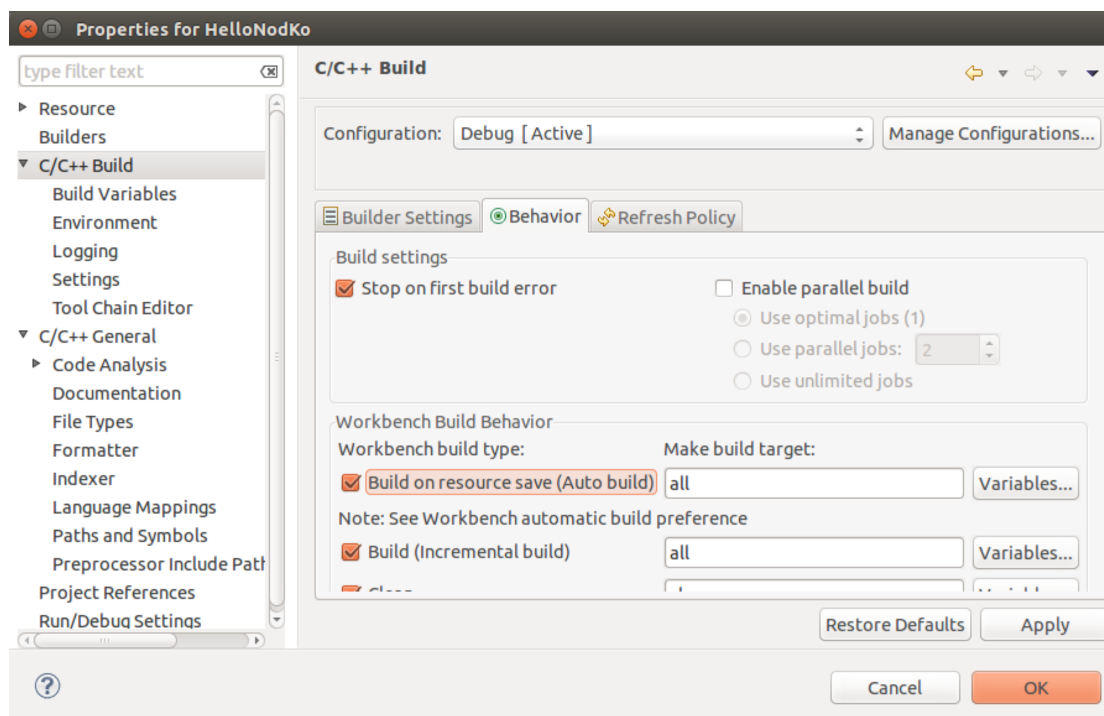


图 10.19 虚拟机中将 build 过程中的 all clean 选项选中

最后， ctrl+b 或者 build all:

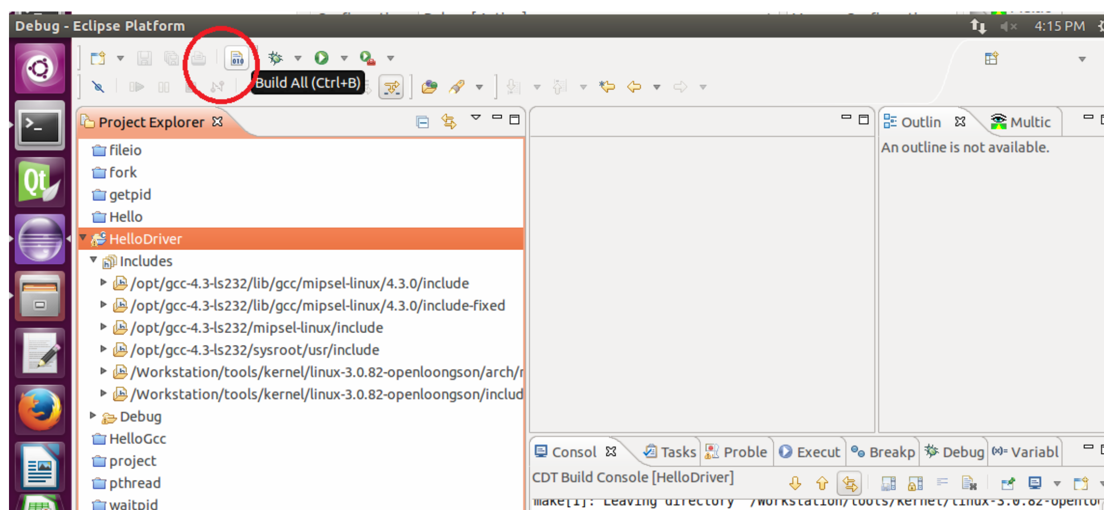


图 10.20 虚拟机中编译

最后,在右下角的 Build Console 中出现：

```
15:28:26 **** Build of configuration Debug for project HelloDriver ****
make all
make -C /Workstation/tools/kernel/linux-3.0.82-openloongson
M=/Workstation/examples/eclipse/HelloDriver/Debug modules ARCH=mips CROSS_COMPILE=mipsel-linux-
make[1]: Entering directory '/Workstation/tools/kernel/linux-3.0.82-openloongson'
  CC [M]  /Workstation/examples/eclipse/HelloDriver/Debug/hello_drivernod.o
/Workstation/examples/eclipse/HelloDriver/Debug/hello_drivernod.c: In function 'hello_init':
/Workstation/examples/eclipse/HelloDriver/Debug/hello_drivernod.c:87: warning: passing argument 1 of
'cdev_del' makes pointer from integer without a cast
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /Workstation/examples/eclipse/HelloDriver/Debug/hello_drivernod.mod.o
  LD [M]  /Workstation/examples/eclipse/HelloDriver/Debug/hello_drivernod.ko
```

```
make[1]: Leaving directory '/Workstation/tools/kernel/linux-3.0.82-openloongson'
```

已经生成了驱动模块。

11. 一个简单的字符设备驱动

Linux 是 Unix 操作系统的一种变种，在 Linux 下编写驱动程序的原理和思想完全类似于其他的 Unix 系统，但它 dos 或 window 环境下的驱动程序有很大的区别。

在 Linux 操作系统下有两类主要的设备文件类型，一种是字符设备，另一种是块设备。字符设备和块设备的主要区别是：在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了，块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的 CPU 时间来等待。

本节简单介绍如何写一个简单的字符设备驱动，实现一个与硬件设备无关的字符设备驱动，仅仅操作从内核中分配的一些内存。

11.1 主设备号和次设备号

用户进程是通过设备文件来与实际的硬件打交道。每个设备文件都都有其文件属性(c/b)，表示是字符设备还是块设备。另外每个文件都有两个设备号，第一个是主设备号，标识驱动程序，第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备，比如有两个软盘，就可以用从设备号来区分他们，也就是多个驱动程序共享主设备号的情况。而次设备号有内核使用，用于确定/dev 下的设备文件对应的具体设备。设备文件的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

对于字符设备的访问是通过文件系统中的设备名称进行的。他们通常位于/dev 目录下。

```
[root@Loongson:/dev]#ls /dev -l
total 0
crw-rw----    1 root     root     251,   0 Jan  1 00:03 cdevdemo
crw-rw----    1 root     root       5,   1 Jan  1 00:00 console
crw-rw----    1 root     root      10,  61 Jan  1 00:00 cpu_dma_latency
crw-rw----    1 root     root      14,   3 Jan  1 00:00 dsp
crw-rw----    1 root     root      29,   0 Jan  1 00:00 fb0
crw-rw----    1 root     root       1,   7 Jan  1 00:00 full
crw-rw----    1 root     root      89,   0 Jan  1 00:00 i2c-0
crw-rw----    1 root     root      89,   1 Jan  1 00:00 i2c-1
crw-rw----    1 root     root      89,   2 Jan  1 00:00 i2c-2
drwxr-xr-x    2 root     root            60 Jan  1 00:00 input
crw-rw----    1 root     root       1,  11 Jan  1 00:00 kmsg
crw-rw----    1 root     root      10,  63 Jan  1 00:00 ls1f-pwm
```

其中首字母 b 代表块设备，c 代表字符设备。对于普通文件来说，ls -l 会列出文件的长度，而对于设备文件来说，上面的 251 等代表的是对应设备的主设备号，而后面的 0,1,2,61 等则是对应设备的次设备号。举一个例子，虚拟控制台和串口终端有驱动程序管理，而不同的终端分别有不同的次设备号。

1)设备编号的表达

在内核中，dev_t 用来保存设备编号，包括主设备号和次设备号。在内核中，dev_t 是一个 32 位的数，其中 12 位用来表示主设备号，其余 20 位用来标识次设备号。

通过 dev_t 获取主设备号和次设备号使用下面的宏：

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

相反，通过主设备号和次设备号转换为 dev_t 类型使用：

```
MKDEV(int major, int minor);
```

2)分配和释放设备编号

在构建一个字符设备之前，驱动程序首先要获得一个或者多个设备编号，这类似一个营业执照，有了营业执照才在内核中正常工作营业。完成此工作的函数是：

```
int register_chrdev_region(dev_t first, unsigned int count, const char *name);
```

first 是要分配的设备编号范围的起始值。count 是连续设备的编号的个数。name 是和该设备编号范围关联的设备名称，他将出现在/proc/devices 和 sysfs 中。此函数成功返回 0，失败返回负的错误码。/proc/devices 存放着系统中所有的设备编号。如果加载成功，就会在/proc/devices 中看到该设备。

```
[root@Loongson:/proc]#cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
 81 video4linux
 89 i2c
 90 mtd
128 ptm
136 pts
153 spi
180 usb
189 usb_device
251 cdevdemo
252 hidraw
253 bsg
254 rtc

Block devices:
259 blkext
  8 sd
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
```

register_chrdev_region 函数是在已知主设备号的情况下使用，在未知主设备号的情况下，使用下面的函数：

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, const char *name);
```

dev 用于输出申请到的设备编号，firstminor 要使用的第一个此设备编号。最好不要随机选择一个当前未使用的设备号，而应该用动态分配机制去获取主设备号。

在不使用时需要释放这些设备编号，已提供其他设备程序使用：

```
void unregister_chrdev_region(dev_t dev, unsigned int count);
```

此函数多在模块的清除函数中调用。

分配到设备编号之后，只是拿到了营业执照，虽说现在已经准备的差不多了,但是只是从内核中申请到了设备号,应用程序还是不能对此设备作任何事情,需要一个简单的函数来把设备编号和此设备能实现的功能连接起来,这样模块才能提供具体的功能.这个操作很简单，稍后就会提到，在此之前先介绍几个重要的数据结构。

11.2 重要的数据结构

注册设备编号仅仅是完成一个字符设备驱动的第一步。下面介绍大部分驱动都会包含的三个重要的内核的数据结构。

Linux 系统中，设备驱动程序是操作系统内核的重要组成部分，它与硬件设备之间建立了标准的抽象接口。通过这个接口，用户可以像处理普通文件一样，对硬件设备进行打开(open)、关闭(close)、读写(read/write)、控制（ioctl）等操作。

设备驱动程序接口是由结构 file_operations 结构体向系统说明的，它定义在include/linux/fs.h 中。

1)文件操作 file_operations

file_operations 是第一个重要的结构，定义在 <linux/fs.h>, 是一个函数指针的集合，设备所能提供的功能大部分都由此结构提供。这些操作也是设备相关的系统调用的具体实现。此结构的具体实现如下所示：

```
struct file_operations {
	//它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为 THIS_MODULE
	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);
	ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
	ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
	int (*iterate) (struct file *, struct dir_context *);
	unsigned int (*poll) (struct file *, struct poll_table_struct *);
	long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
	long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
	int (*mmap) (struct file *, struct vm_area_struct *);
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *, fl_owner_t id);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, loff_t, loff_t, int datasync);
	int (*aio_fsync) (struct kiocb *, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
	unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
	int (*check_flags)(int);
	int (*flock) (struct file *, int, struct file_lock *);
	ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
	ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
	int (*setlease)(struct file *, long, struct file_lock **);
	long (*fallocate)(struct file *file, int mode, loff_t offset,
			  loff_t len);
	int (*show_fdinfo)(struct seq_file *m, struct file *f);
```

```
};
```

需要说明的是这里面的函数在驱动中不用全部实现，不支持的操作留置为 NULL。

几个主要的入口函数打开(open)、关闭(close)、读写(read/write)、控制（ioctl），编写驱动时需要填充。

⑴ open 入口点：

open 函数负责打开设备、准备 I/O。任何时候对设备文件进行打开操作，都会调用设备的 open 入口点。所以，open 函数必须对将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的。则 open 函数必须将设备标记成忙状态。

⑵ close 入口点

close 函数负责关闭设备的操作，当最后一次使用设备完成后，调用 close 函数，关闭设备文件。独占设备必须标记为可再次使用。

close()函数作用是关闭打开的文件。

⑶ read 入口点

read 函数负责从设备上读数据和命令，有缓冲区的 I/O 设备操作一般是从缓冲区里读数据。

⑷ write 入口点

write 函数负责往设备上写数据。对于有缓冲区的 I/O 设备操作，一般是把数据写入缓冲区里。对字符设备文件进行写操作将调用 write 函数。

⑸ ioctl 入口点

ioctl 函数执行读、写之外的操作，主要实现对设备的控制。

2)文件结构 struct file

struct file，定义于 <linux/fs.h>，是设备驱动中第二个最重要的数据结构。文件结构代表一个打开的文件. (它不特定给设备驱动; 系统中每个打开的文件有一个关联的 struct file 在内核空间). 它由内核在 open 时创建, 并传递给在文件上操作的任何函数, 直到最后的关闭. 在文件的所有实例都关闭后, 内核释放这个数据结构。file 结构的详细可参考 fs.h，这里列出来几个重要的成员。

```
struct file_operations *f_op：就是上面刚刚介绍的文件操作的集合结构。
mode_t f_mode：文件模式确定文件是可读的或者是可写的(或者都是), 通过位 FMODE_READ 和 FMODE_WRITE. 你可能想在你的 open 或者 ioctl 函数中检查这个成员的读写许可, 但是你不需要检查读写许可, 因为内核在调用你的方法之前检查. 当文件还没有为那种存取而打开时读或写的企图被拒绝, 驱动甚至不知道这个情况
loff_t f_pos：当前读写位置. loff_t 在所有平台都是 64 位。驱动可以读这个值, 如果它需要知道文件中的当前位置, 但是正常地不应该改变它。
unsigned int f_flags：这些是文件标志, 例如 O_RDONLY, O_NONBLOCK, 和 O_SYNC. 驱动应当检查 O_NONBLOCK 标志来看是否是请求非阻塞操作。
void *private_data：open 系统调用设置这个指针为 NULL, 在为驱动调用 open 方法之前. 你可自由使用这个成员或者忽略它; 你可以使用这个成员来指向分配的数据, 但是接着你必须记住在内核销毁文件结构之前, 在 release 方法中释放那个内存. private_data 是一个有用的资源, 在系统调用间保留状态信息, 我们大部分例子模块都使用它
```

3)inode 结构

inode 结构由内核在内部用来表示文件。因此, 它和代表打开文件描述符的文件结构是不同的。 可能有代表单个文件的多个打开描述符的许多文件结构, 但是它们都指向一个单个 inode 结构。

inode 结构包含大量关于文件的信息比如文件的创建者、文件的创建日期、文件的大小等等。中文译名为"索引节点"。

inode 包含文件的元信息，具体来说有以下内容：

* 文件的字节数;

* 文件拥有者的 User ID;

* 文件的 Group ID;

* 文件的读、写、执行权限;

* 文件的时间戳，共有三个：ctime 指 inode 上一次变动的时间，mtime 指文件内容上一次变动的时间，atime 指文件上一次打开的时间;

* 链接数，即有多少文件名指向这个 inode;

* 文件数据 block 的位置 ；

可以用 stat 命令，查看某个文件的 inode 信息：

```
[root@Loongson:/]#cat hello.c
hello,I'm Loongson,This is    file io test!
[root@Loongson:/]#stat hello.c
   File: hello.c
   Size: 41                Blocks: 1            IO Block: 4096    regular file
Device: 1f01h/7937d        Inode: 779           Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 1970-01-01 00:02:45.000000000
Modify: 1970-01-01 00:02:45.000000000
Change: 1970-01-01 00:02:45.000000000
```

查看每个硬盘分区的 inode 总数和已经使用的数量，可以使用 df 命令。

```
[root@Loongson:/]#df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/root                51200     16604     34596  32% /
devtmpfs                    64         0        64   0% /dev
tmpfs                       64         0        64   0% /dev
tmpfs                    11656         0     11656   0% /tmp
[root@Loongson:/]#df /dev/
Filesystem           1K-blocks      Used Available Use% Mounted on
devtmpfs                    64         0        64   0% /dev
```

由于每个文件都必须有一个 inode，因此有可能发生 inode 已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件。

这里值得重复一遍，Unix/Linux 系统内部不使用文件名，而使用 inode 号码来识别文件。对于系统来说，文件名只是 inode 号码便于识别的别称或者绰号。表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的 inode 号码；其次，通过 inode 号码，获取 inode 信息；最后，根据 inode 信息，找到文件数据所在的 block，读出数据。

使用 ls -i 命令，可以看到文件名对应的 inode 号码：

```
[root@Loongson:/]#ls -i hello.c
    779 hello.c
```

有关 inode 的实际问题：

在一台配置较低的 Linux 服务器（内存、硬盘比较小）的/data 分区内创建文件时，系统提示磁盘空间不足，用 df -h 命令查看了一下磁盘使用情况，发现/data 分区只使用了 66%，还有 12G 的剩余空间，按理说不会出现这种问题。 后来用 df -i 查看了一下/data 分区的索引节点(inode)，发现已经用满(IUsed=100%)，导致系统无法创建新目录和文件。

查找原因：

/data/cache 目录中存在数量非常多的小字节缓存文件，占用的 Block 不多，但是占用了大量的 inode。

解决方案：

1、删除/data/cache 目录中的部分文件，释放出/data 分区的一部分 inode。

2、用软连接将空闲分区/opt 中的 newcache 目录连接到/data/cache，使用/opt 分区的 inode 来缓解/data 分区 inode 不足的问题：

```
ln -s /opt/newcache /data/cache
```

11.3 字符设备的注册

内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用设备操作前, 编写分配并注册一个或几个这些结构。

有 2 种方法来分配和初始化一个这些结构. 如果想在运行时获得一个独立的 cdev 结构, 可使用这样的代码:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

更多的情况是把 cdv 结构嵌入到你自己封装的设备结构中，这时需要使用下面的方法来分配和初始化：

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

后面的例子程序就是这么做的。一旦 cdev 结构建立, 最后的步骤是把它告诉内核：

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count)
```

这里, dev 是 cdev 结构, num 是这个设备响应的第一个设备号, count 是应当关联到设备的设备号的数目. 常常 count 是 1。

从系统去除一个字符设备, 调用:

```
void cdev_del(struct cdev *dev);
```

内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用你的设备操作前, 你编写分配并注册一个或几个这些结构。

11.4 具体实例

上面大致介绍了实现一个字符设备所要做的工作，下面就来一个真实的例子来总结上面介绍的内容。设备驱动程序是 I/O 进程与设备控制器之间的通信程序。

本驱动程序的功能：

⑴ 接收由设备独立性软件发来的命令和参数，并将命令中的抽象要求转换为具体的要求。

⑵ 检查用户 I/O 请求的合法性，了解 I/O 设备的状态，传递有关参数，设置设备的工作方式。

⑶ 发出 I/O 命令。

⑷ 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。

⑸ 对于设置有通道的计算机系统，驱动程序还应能够根据用户的 I/O 请求，自动地构建通道程序。

设备驱动程序的处理过程：

⑴ 将抽象要求转换为具体要求

⑵ 检查 I/O 设备请求的合法性

⑶ 读出和检查设备的状态

⑷ 传送必要的参数

⑸ 工作方式的设置

⑹ 启动 I/O 设备

源码中的关键地方已经作了注释。

```
/*devdemo.c*/
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <linux/timer.h>
#include <asm/atomic.h>
#include <linux/slab.h>
#include <linux/device.h>

#define CDEVDEMO_MAJOR 255    /*预设 cdevdemo 的主设备号*/

static int cdevdemo_major = CDEVDEMO_MAJOR;

/*设备结构体,此结构体可以封装设备相关的一些信息等
   信号量等也可以封装在此结构中，后续的设备模块一般都
   应该封装一个这样的结构体，但此结构体中必须包含某些
   成员，对于字符设备来说，我们必须包含 struct cdev cdev*/
struct cdevdemo_dev
{
	struct cdev cdev;
};

struct cdevdemo_dev *cdevdemo_devp; /*设备结构体指针*/

/*文件打开函数，上层对此设备调用 open 时会执行*/
int cdevdemo_open(struct inode *inode, struct file *filp)
{
	printk(KERN_NOTICE "======== cdevdemo_open ");
	return 0;
}

/*文件释放，上层对此设备调用 close 时会执行*/
int cdevdemo_release(struct inode *inode, struct file *filp)
{
	printk(KERN_NOTICE "======== cdevdemo_release ");
	return 0;
}

/*文件的读操作，上层对此设备调用 read 时会执行*/
static ssize_t cdevdemo_read(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
{
	printk(KERN_NOTICE "======== cdevdemo_read ");
}

/*  文件操作结构体，文中已经讲过这个结构*/
static const struct file_operations cdevdemo_fops =
{
	.owner = THIS_MODULE,
	.open = cdevdemo_open,
	.release = cdevdemo_release,
	.read = cdevdemo_read,
};

/*初始化并注册 cdev*/
static void cdevdemo_setup_cdev(struct cdevdemo_dev *dev, int index)
{
	printk(KERN_NOTICE "======== cdevdemo_setup_cdev 1");
	int err, devno = MKDEV(cdevdemo_major, index);
	printk(KERN_NOTICE "======== cdevdemo_setup_cdev 2");
```

```
    /*初始化一个字符设备，设备所支持的操作在 cdevdemo_fops 中*/
    cdev_init(&dev->cdev, &cdevdemo_fops);
    printk(KERN_NOTICE "======== cdevdemo_setup_cdev 3");
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &cdevdemo_fops;
    printk(KERN_NOTICE "======== cdevdemo_setup_cdev 4");
    err = cdev_add(&dev->cdev, devno, 1);
    printk(KERN_NOTICE "======== cdevdemo_setup_cdev 5");
    if(err)
    {
        printk(KERN_NOTICE "Error %d add cdevdemo %d", err, index);
    }
}

int cdevdemo_init(void)
{
    printk(KERN_NOTICE "======== cdevdemo_init ");
    int ret;
    dev_t devno = MKDEV(cdevdemo_major, 0);

    struct class *cdevdemo_class;
    /*申请设备号，如果申请失败采用动态申请方式*/
    if(cdevdemo_major)
    {
        printk(KERN_NOTICE "======== cdevdemo_init 1");
        ret = register_chrdev_region(devno, 1, "cdevdemo");
    }else
    {
        printk(KERN_NOTICE "======== cdevdemo_init 2");
        ret = alloc_chrdev_region(&devno,0,1,"cdevdemo");
        cdevdemo_major = MAJOR(devno);
    }
    if(ret < 0)
    {
        printk(KERN_NOTICE "======== cdevdemo_init 3");
        return ret;
    }
    /*动态申请设备结构体内存*/
    cdevdemo_devp = kmalloc(sizeof(struct cdevdemo_dev), GFP_KERNEL);
    if(!cdevdemo_devp)   /*申请失败*/
    {
        ret = -ENOMEM;
        printk(KERN_NOTICE "Error add cdevdemo");
        goto fail_malloc;
    }

    memset(cdevdemo_devp,0,sizeof(struct cdevdemo_dev));
    printk(KERN_NOTICE "======== cdevdemo_init 3");
    cdevdemo_setup_cdev(cdevdemo_devp, 0);

    /*下面两行是创建了一个总线类型，会在/sys/class 下生成 cdevdemo 目录
      这里的还有一个主要作用是执行 device_create 后会在/dev/下自动生成
      cdevdemo 设备节点。而如果不调用此函数，如果想通过设备节点访问设备
      需要手动 mknod 来创建设备节点后再访问。*/
    cdevdemo_class = class_create(THIS_MODULE, "cdevdemo");
    device_create(cdevdemo_class, NULL, MKDEV(cdevdemo_major, 0), NULL, "cdevdemo");

    printk(KERN_NOTICE "======== cdevdemo_init 4");
    return 0;

    fail_malloc:
        unregister_chrdev_region(devno,1);
}

void cdevdemo_exit(void)     /*模块卸载*/
```

```
{
	printk(KERN_NOTICE "End cdevdemo");
	cdev_del(&cdevdemo_devp->cdev);   /*注销 cdev*/
	kfree(cdevdemo_devp);          /*释放设备结构体内存*/
	unregister_chrdev_region(MKDEV(cdevdemo_major,0),1);     //释放设备号
}

MODULE_LICENSE("Dual BSD/GPL");
module_param(cdevdemo_major, int, S_IRUGO);
module_init(cdevdemo_init);
module_exit(cdevdemo_exit);
```

具体步骤总结：

1) file_operations 结构体设计

```
/* 文件操作结构体，文中已经讲过这个结构*/
static const struct file_operations cdevdemo_fops =
{
	.owner = THIS_MODULE,
	.open = cdevdemo_open,
	.release = cdevdemo_release,
	.read = cdevdemo_read,
};
```

2) 模块初始化、模块卸载函数实现

(1) 注册设备号

分配设备编号，注册设备与注销设备的函数均在 fs.h 中声明，如下：

```
extern int register_chrdev_region(dev_t,unsigned int,const char*);//表示静态的申请和注册设备号
extern int alloc_chrdev_region(dev_t,unsigned int,const char*);//表示动态的申请和注册设备号
extern int register_chrdev(unsigned int,const char*,struct file_operations*);//表示 int 为 0 时动态注册，非零静态注册。
```

在 linux2.6 版本里面，register_chrdev_region 是 register_chrdev 的升级版。

使用 register_chrdev_region 函数时，首先要定义一个 dev_t 变量来作为一个设备号，dev_t dev_num；如果想静态申请，那么

```
dev_num=MKDEV(major_no,0);//major_no 表示设备号的变量，然后便可以使用
register_chrdev_region(dev_num,2,"my_dev");//第二个参数表示注册的设备数量，第三个表示驱动名
```

如果要动态的注册设备号，使用下面 alloc_chrdev_region(&dev_num，0,2，"memdev"); 次设备号从 0 开始，注册两个设备，设备名为 memdev。

(2) 添加设备

前面只是注册了设备号，后面要向内核添加设备了；

```
struct cdev devno;
cdev_init(&devno,&file_operations) // 初始化设备
devno.owner=THIS_MODULE;
devno.ops=&mem_fops
```

对于已经知道了主设备号，就用

```
cdev_add(&devno,dev_num,MEMDEV_NR_DEVS);//添加设备
```

如果是动态申请的设备号，就用

```
cdev_add(&devno,MKDEV(mem_major,0),MEMDEV_NR_DEVS);
```

由此可见，使用 register_chrdev_region()比 register_chrdev()多了一步，就是向内核注册添加 cdev 设备的步骤。

(3) 添加设备结点

自动添加、删除设备节点的方法

以下创建了一个总线类型，在/sys/class 下生成 cdevdemo 目录 后，在执行 device_create

后会在/dev/下自动生成 cdevdemo 设备节点。而如果不调用此函数，如果想通过设备节点访问设备 ， 需要手动 mknod 来创建设备节点后再访问。

```
    cdevdemo_class = class_create(THIS_MODULE, "cdevdemo");
    device_create(cdevdemo_class, NULL, MKDEV(cdevdemo_major, 0), NULL, "cdevdemo");
```

本实例使用的是这种方法。

另外一种手动添加、删除设备节点的方法

创建设备文件：

```
    #mknod  /dev/ devdemo c  major  minor
```

c 是指字符设备，major 是主设备号，就是在/proc/devices 里看到的。

用 shell 命令：

```
    $ cat /proc/devices
```

就可以获得主设备号。 minor 是从设备号，设置成 0 就可以了。

mknod 命令创建的设备节点，可以使用 rm -f 设备文件名称（注意这个地方不是跟全路径，就是一个名字）可以删掉，重启电脑或者删除对应的文件都是无效的。

3) 读写函数的实现

```
/*文件打开函数，上层对此设备调用 open 时会执行*/
int cdevdemo_open(struct inode *inode, struct file *filp)
{
    printk(KERN_NOTICE "======== cdevdemo_open ");
    return 0;
}

/*文件释放，上层对此设备调用 close 时会执行*/
int cdevdemo_release(struct inode *inode, struct file *filp)
{
    printk(KERN_NOTICE "======== cdevdemo_release ");
    return 0;
}

/*文件的读操作，上层对此设备调用 read 时会执行*/
static ssize_t cdevdemo_read(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
{
    printk(KERN_NOTICE "======== cdevdemo_read ");
}
```

4) 驱动程序编译

编写 Makefile，后在虚拟机中编译。

```
obj-m := devdemo1.o
#定义目录变量
KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson
PWD := $(shell pwd)
all:
# make 文件
	make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS_COMPILE=mipsel-linux-
clean:
	rm -rf *.o   *.mod.c *.ko
```

5) 驱动程序编译和加载

在用 insmod 命令将编译好的模块调入内存时，init_module 函数被调用。在这里，编写 makefile 文件编译该设备驱动程序，编译结束后产生 devdemo.ko 文件。

驱动程序已经编译好了，现在把它安装到系统中去：

```
[root@Loongson:/]#insmod     devdemo.ko
======== cdevdemo_init
======== cdevdemo_init 1
```

```
======== cdevdemo_init 3
======== cdevdemo_setup_cdev 1
======== cdevdemo_setup_cdev 2
======== cdevdemo_setup_cdev 3
======== cdevdemo_setup_cdev 4
======== cdevdemo_setup_cdev 5
======== cdevdemo_init 4
```

如果安装成功，在/proc/devices 文件中就可以看到设备 devdemo，并可以看到它的主设备号。

```
[root@Loongson:/]#cat /proc/devices | grep demo
255 cdevdemo
```

要卸载的话，运行命令：

```
[root@Loongson:/]#rmmod devdemo

End cdevdemo
```

6) 驱动程序测试

现在可以通过设备文件来访问我们的驱动程序。可以写一个小小的测试程序：

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>


int main(void)
{
	int fd;
	int i;
	char buf[10] ;

	fd = open("/dev/cdevdemo", O_RDWR);
	if (fd < 0)
	{
		perror("open error");
		return -1;
	}

	read(fd，buf，10);
	for (i = 0; i < 10;i++)

		printf("%d\n"，buf[i]);
	 close(fd);
	return 0;
}
```

编译运行，运行结果：

```
[root@Loongson:/]#insmod devdemo.ko
======== cdevdemo_init
======== cdevdemo_init 1
======== cdevdemo_init 3
======== cdevdemo_setup_cdev 1
======== cdevdemo_setup_cdev 2
======== cdevdemo_setup_cdev 3
======== cdevdemo_setup_cdev 4
======== cdevdemo_setup_cdev 5
======== cdevdemo_init 4[root@Loongson:/]#test_devdemo
-/bin/sh: test_devdemo: not found
[root@Loongson:/]#chmod u+x test_devdemo
[root@Loongson:/]#./test_devdemo

======== cdevdemo_open
```

```
======== cdevdemo_read the string is :
======== cdevdemo_release hello
```

以上只是一个简单的演示。真正实用的驱动程序要复杂的多，要处理如中断，DMA，I/O port 等问题。

11.5 一些有用的资料

Linux 驱动之模块化编程
http://blog.chinaunix.net/uid-26833883-id-4366882.html

Linux 驱动 之 模块化编程
http://blog.chinaunix.net/uid-26833883-id-4366909.html

Linux 设备驱动之字符设备(一)
http://blog.chinaunix.net/uid-26833883-id-4369060.html

Linux 设备驱动之字符设备(二)
http://blog.chinaunix.net/uid-26833883-id-4369117.html

Linux 设备驱动之字符设备(三)
http://blog.chinaunix.net/uid-26833883-id-4371047.html

11.6 例子修改成模块注销自动删除设备节点

实例文件：devdemo.c。

添加设备节点时，添加的类设置其属性为 666。即将以下两句

```
    cdevdemo_class = class_create(THIS_MODULE, "cdevdemo");
    device_create(cdevdemo_class, NULL, MKDEV(cdevdemo_major, 0), NULL, "cdevdemo");
```

修改成：

```
    cdevdemo_class = class_create(THIS_MODULE, "cdevdemo");
    if (IS_ERR(cdevdemo_class))
    {
        ret = PTR_ERR(cdevdemo_class);
        printk(KERN_ERR "class create error %d\n", ret);
        goto fail_malloc;
    }
    cdevdemo_class->devnode = chardev_devnode;
    dev = device_create(cdevdemo_class, NULL, MKDEV(cdevdemo_major, 0), NULL, "cdevdemo");
```

添加两个定义：

```
struct class *cdevdemo_class;
static char *chardev_devnode(struct device *dev, umode_t *mode)
{
    if (mode)
        *mode = 0666;

    return NULL;
}
```

模块注销卸载中添加两句：

```
void cdevdemo_exit(void)    /*模块卸载*/
{
    device_destroy(cdevdemo_class, MKDEV(cdevdemo_major,0));
    class_destroy(cdevdemo_class);
```

```
    ……
}
```

12. misc 杂项设备驱动

在 Linux 驱动中把无法归类的五花八门的设备定义为混杂设备(用 miscdevice 结构体表述)。misc 设备其实也就是特殊的字符设备。miscdevice 共享一个主设备号 MISC_MAJOR(即10)，但次设备号不同。 所有的 miscdevice 设备形成了一个链表，对设备访问时内核根据次设备号查找对应的 miscdevice 设备，然后调用其 file_operations 结构中注册的文件操作接口进行操作。 在内核中用 struct miscdevice 表示 miscdevice 设备，然后调用其 file_operations 结构中注册的文件操作接口进行操作。miscdevice 的 API 实现在 drivers/char/misc.c 中。

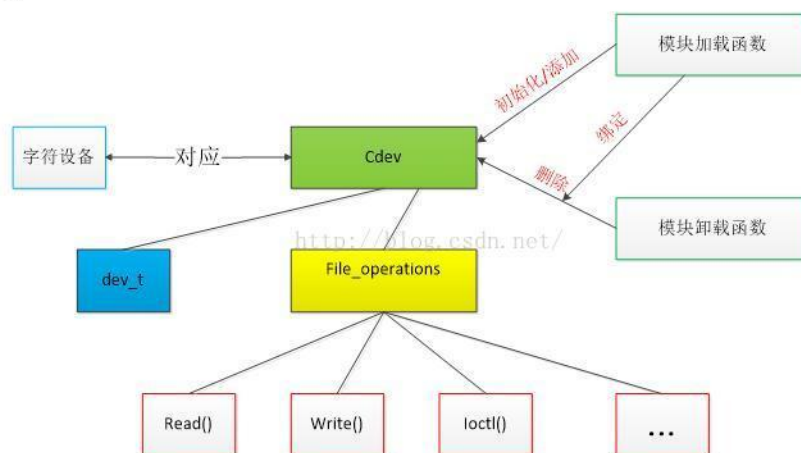


图 12.1 linux 设备类结构图

12.1misc 使用的结构体和函数

misc 设备其实也是字符设备，只不过 misc 设备驱动在字符设备的基础上又进行了一次封装，使用户可以更方便的使用。

```
struct miscdevice {
        int minor;
        const charchar *name;
        const struct file_operations *fops;//还是字符设备中的文件操作结构体，只不过 misc 结构体对其又封装了一次
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const charchar *nodename;
        mode_t mode;
};
int misc_register(struct miscdevice * misc)
```

用户在注册 misc 设备的时候只需要：初始化 file_operations 结构体->初始化 miscdevice 结构体->调用 misc_register 将 miscdevice 注册到系统中就 ok 了。

```
static const struct file_operations mmtimer_fops = {
.owner = THIS_MODULE,
.mmap =mmtimer_mmap,
.unlocked_ioctl = mmtimer_ioctl,
.llseek = noop_llseek,
};
static struct miscdevice mmtimer_miscdev = {
SGI_MMTIMER,
MMTIMER_NAME,
```

```
&mmtimer_fops
};
static  int  __init  mmtimer_init(void)
{
        cnodeid_t  node,  maxn  =  -1;

......
        if       (request_irq(SGI_MMTIMER_VECTOR,        mmtimer_interrupt,        IRQF_PERCPU,
MMTIMER_NAME,  NULL))  {
                printk(KERN_WARNING  "%s:  unable  to  allocate  interrupt.",
                        MMTIMER_NAME);
                goto  out1;
        }

        if  (misc_register(&mmtimer_miscdev)){
                printk(KERN_ERR  "%s:  failed  to  register  device\n",
                              MMTIMER_NAME);
                goto  out2;
        }
......

        return  0;

out3:
        kfree(timers);
        misc_deregister(&mmtimer_miscdev);
out2:
        free_irq(SGI_MMTIMER_VECTOR,  NULL);
out1:
        return  -1;
}
```

那么到了这里就会有疑问，为什么 linux 还费劲的又造了一个 misc 设备呢？为什么不直接都使用字符设备驱动呢？

12.2 为什么要有 misc 设备

优点：

1)节省主设备号：使用普通字符设备驱动框架，不管该设备的主设备号是静态或者是动态分布的都会使用一个主设备号。而 miscdevice 结构体的**主设备号是固定的，MISC_MAJOR 定义为 10**，在 linux 内核中，大概可以找到 200 多处使用 miscdevice 框架结构的驱动。

2)使用方便：Misc 驱动不再直接采用 registe_chrdev_region()或者 alloc_chrdev_region()、cdev_add()之类的原始方法申请设备号、注册，而是才用 miscdevice 的注册方法 misc_register(struct miscdevice * misc)。因为它已经封装和优化的很好，能很大程度上简化我们的工作量。

3)利于 linux 驱动的分层设计思想：由于 Linux 驱动倾向于分层设计，各个具体的设备都可以找到它归属的类型，从而嵌套它相应的架构上面，并且只需要实现最底层的那一部分。因为有些设备不知道它属于什么类型，Misc 驱动的引入，很好的解决了这个问题，同时它使用起来也更加的方便。

总的来讲，如果使用 misc 驱动可以满足要求的话，那么这可以为开发人员剩下不少麻烦。

主要使用到的函数有：

```
int  misc_register(struct  miscdevice  *  misc);
int  misc_deregister(struct  miscdevice  *misc);
```

12.3 内核源码

1) miscdevice 结构体

```
include\linux\Miscdevice.h        内核版本：2.6.11.12

struct miscdevice   {
        int minor;
        const char *name;
        struct file_operations *fops;
        struct list_head list;
        struct device *dev;
        struct class_device *class;
        char devfs_name[64];
};
extern int misc_register(struct miscdevice * misc);
extern int misc_deregister(struct miscdevice * misc);
```

2) misc_register 函数

```
/**
 *    misc_register    -    register a miscellaneous device                    <span style="font-family: monospace; white-space: pre; background-color: rgb(240, 240, 240);">linux/drivers/char/misc.c</span>
 *    @misc: device structure
 *
 *    Register a miscellaneous device with the kernel. If the minor
 *    number is set to %MISC_DYNAMIC_MINOR a minor number is assigned
 *    and placed in the minor field of the structure. For other cases
 *    the minor number requested is used.
 *
 *    The structure passed is linked into the kernel and may not be
 *    destroyed until it has been unregistered.
 *
 *    A zero is returned on success and a negative errno code for
 *    failure.
 */

int misc_register(struct miscdevice * misc)
{
        struct miscdevice *c;
        dev_t dev;
        int err;

        down(&misc_sem);
        list_for_each_entry(c, &misc_list, list) {
                if (c->minor == misc->minor) {
                        up(&misc_sem);
                        return -EBUSY;
                }
        }

        if (misc->minor == MISC_DYNAMIC_MINOR) {
                int i = DYNAMIC_MINORS;
                while (--i >= 0)
                        if ( (misc_minors[i>>3] & (1 << (i&7))) == 0)
                                break;
                if (i<0) {
                        up(&misc_sem);
                        return -EBUSY;
                }
                misc->minor = i;
        }

        if (misc->minor < DYNAMIC_MINORS)
                misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
```

```
        if (misc->devfs_name[0] == '\0') {
                snprintf(misc->devfs_name, sizeof(misc->devfs_name),
                                "misc/%s", misc->name);
        }
        //由主、次设备号生产设备号
        dev = MKDEV(MISC_MAJOR, misc->minor);
*/
/**
  *struct class_device *class_simple_device_add(struct class_simple *cs, dev_t dev,
  * struct device *device, const char *fmt, ...)
  * 为一个简单设备类添加设备。
  *         cs:             将设备添加到此简单设备类。
  *         dev:            分配的设备号。
  *         device:         要添加的设备。
  *         fmt:        用于格式化名称。
*/
        misc->class = class_simple_device_add(misc_class, dev,
                                              misc->dev, misc->name);

        if (IS_ERR(misc->class)) {
                err = PTR_ERR(misc->class);
                goto out;
        }

        err = devfs_mk_cdev(dev, S_IFCHR|S_IRUSR|S_IWUSR|S_IRGRP,
                            misc->devfs_name);
        if (err) {
                class_simple_device_remove(dev);
                goto out;
        }

        /*
         * Add it to the front, so that later devices can "override"
         * earlier defaults
         */
        list_add(&misc->list, &misc_list);
 out:
        up(&misc_sem);
        return err;
}
```

3）misc_deregister 函数

```
/**
  *   misc_deregister - unregister a miscellaneous device                    <span style="font-family: monospace; white-space: pre; background-color: rgb(240, 240, 240);">linux/drivers/char/misc.c</span>
  *    @misc: device to unregister
  *
  *    Unregister a miscellaneous device that was previously
  *    successfully registered with misc_register(). Success
  *    is indicated by a zero return, a negative errno code
  *    indicates an error.
  */

int misc_deregister(struct miscdevice * misc)
{
        int i = misc->minor;

        if (list_empty(&misc->list))
                return -EINVAL;

        down(&misc_sem);
        list_del(&misc->list);
*/
/**
  * 当拨除设备时，使用下面的 class_simple_device_remove 函数删除类入口。
```

```
 */
        class_simple_device_remove(MKDEV(MISC_MAJOR, misc->minor));
        devfs_remove(misc->devfs_name);
        if (i < DYNAMIC_MINORS && i>0) {
                misc_minors[i>>3] &= ~(1 << (misc->minor & 7));
        }
        up(&misc_sem);
        return 0;
}

EXPORT_SYMBOL(misc_register);
EXPORT_SYMBOL(misc_deregister);
```

12.4　具体实例

实例代码：miscdemo.c，杂项设备的驱动。

```
/*miscdemo.c*/
#include    <linux/init.h>
#include    <linux/module.h>
#include    <linux/fs.h>
#include    <linux/miscdevice.h>

int    open_state    =    0;
//////////////////////////////////////////////////
int    miscdemo_open(struct    inode    *inode,    struct    file    *filp)
{
        if    (open_state    ==    0)
        {
                open_state    =    1;
                printk("miscdemo open!\n");
                return    0;
        }
        printk("miscdemo    has    been    open!\n");
        return    -1;
}

int    miscdemo_release(struct    inode    *inode,    struct    file    *filp)
{
        if    (open_state    ==    1)
        {
                open_state    =    0;
                printk("miscdemo    release!\n");
                return    0;
        }
        printk("miscdemo    has    not    been    open    yet!\n");
        return    -1;
}

ssize_t    miscdemo_read(struct    file    *filp,    char    *buf,
                size_t    count,    loff_t    fpos)
{
        printk("miscdemo    read!\n");
        return    0;
}

ssize_t    miscdemo_write(struct    file    *filp,    char    *buf,
                size_t    count,    loff_t    fpos)
{
        printk("miscdemo    write!\n");
        return    0;
}

int    miscdemo_ioctl(struct    inode    *inode,    struct    file    *filp,
                unsigned    int    cmd,    unsigned    long    arg)
```

```
{
	printk("ioctl is called!\n");
	printk("cmd:%d arg:%d\n", cmd, arg);
	return 0;
}
//////////////////////////////////////////////////
struct file_operations fops =
{
	.owner		=	THIS_MODULE,
	.open		=	miscdemo_open,
	.release	=	miscdemo_release,
	.write		=	miscdemo_write,
	.read		=	miscdemo_read,
	. unlocked_ioctl	=	miscdemo_ioctl
};

struct miscdevice dev =
{
	.minor	=	MISC_DYNAMIC_MINOR,
	.fops	=	&fops,
	.name	=	"miscdemo",
	.nodename = "miscdemo_node"
};

int setup_miscdemo(void)
{

	return misc_register(&dev);
}
//////////////////////////////////////////////////
static int __init miscdemo_init(void)
{
	printk("miscdemo init!\n");
	return setup_miscdemo();
}

static void __exit miscdemo_exit(void)
{
	printk("miscdemo exit!\n");
	misc_deregister(&dev);
}

MODULE_AUTHOR("sundm75");
MODULE_LICENSE("GPL");
module_init(miscdemo_init);
module_exit(miscdemo_exit);
```

测试函数 test_miscdemo.c。

```
/*test_miscdemo.c*/
#include <stdio.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
//////////////////////////////////////////////////
int main(int argc, char **argv)
{
	int fd;
	fd = open("/dev/miscdemo_node", O_RDONLY);
	if (fd < 0)
	{
		printf("open /dev/miscdemo_node failed!\n");
		printf("%s\n", strerror(errno));
		return -1;
	}
```

```
        printf("open  /dev/miscdemo_node  ok!\n");
        if  (ioctl(fd,  6)  !=  0)
        {
                printf("ioctl  failed!\n");
                printf("%s\n",  strerror(errno));
        }
        else
                printf("ioctl  ok!\n");
        close(fd);
        return  0;
}
```

运行过程：首先载入模块，显示模块，运行测试程序。运行结果：

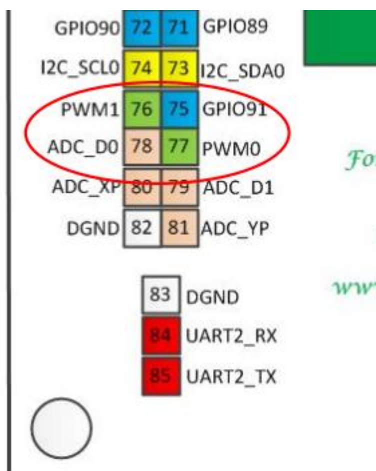
```
[root@Loongson:/]#insmod miscdemo.ko
miscdemo  init!
[root@Loongson:/]#lsmod
Module                  Size  Used by     Not tainted
miscdemo                1481  0
[root@Loongson:/]#./test_miscdemo
miscdemo open!
open  /dev/miscdioctl  is  called!
emo_node  ok!
cmd:30  arg:30
ioctl  ok!miscdemo  release!

[root@Loongson:/]#cat /dev/miscdemo_node
miscdemo open!
miscdemo  read!
miscdemo  release!
```

13. PWM 控制输出

13.1 利用 LED_PWM

1) PWM0 PWM1 说明

原始功能没有复用 06 和 92 引脚：



URT0_RI	84(U)	–	GPIO45	SRAM_A23	UART3_TX	CAN
PWM1	77(A) 85(U)	–	GPIO92	ADC_YN		
PWM0/ CAMCLKOUT	78(A) 86(U)	–	GPIO06	ADC_XN		
SPI0_CS2	79(A)	–	GPIO83	Sdio_Dat3		
CORE_VSS	80(A)	50				

图 13.1 开发板 PWM 位置及复用关系图

```
55 #define SPI0_CS3            3
56
57 #define SPI1_CS0            0
58 #define SPI1_CS1            1
59 #define SPI1_CS2            2
60
61 /* pwm regs */
62 #define LS1X_PWM0_BASE  0x1fe5c000
63 #define LS1X_PWM1_BASE  0x1fe5c010
64 #define LS1X_PWM2_BASE  0x1fe5c020
65 #define LS1X_PWM3_BASE  0x1fe5c030
66
67 /* gmac regs */
68 #define LS1X_GMAC0_BASE     0x1fe10000
69 #define LS1X_GMAC1_BASE     0x1fe20000
70
71 /* UART regs */
72 #define LS1X_UART0_BASE     0x1fe40000
```

图 13.2 头文件中 PWM 寄存器定义

在寄存器定义中，LS1X_PWM0_BASE 定义为 0x1fe5c000，此处是物理地址；0xbfefc0000 为核心虚地址空间的程序空间地址。

使用的平台文件 的 Makefile 中，定义了 pwm.o 的目标文件。

P\linux\linux-3.0.82-openloongson\linux-3.0.82-openloongson\arch\mips\loongson\ls1x\Makefile - Notepad++ [Administrator]

```
1  #
2  # Makefile for LST ls232 board.
3  #
4
5  obj-y          += clock.o setup.o prom.o reset.o irq.o mipsdha.o gpio.o mem.o
6  obj-$(CONFIG_HAVE_PWM)  += pwm.o
7  obj-$(CONFIG_LS1X_TIMER)  += time.o
8  obj-$(CONFIG_LS1X_HPET)  += hpet-time.o
9  obj-$(CONFIG_PCI)  += pci.o
10
11 ifdef CONFIG_LS1A_MACH
12 obj-$(CONFIG_SUSPEND)    += pm.o
13 endif
14
15 # 1A 板型定义，如：云终端、方案核心板
16 obj-$(CONFIG_LS1A_CORE_BOARD)  += ls1a-core/
17 obj-$(CONFIG_LS1A_CLOUD_TERMIAL)  += ls1a-cloud/
18 # 1B 板型定义，如：开发板、核心板
19 obj-$(CONFIG_LS1B_BOARD)  += ls1b-board/
20 obj-$(CONFIG_LS1B_CORE_BOARD)  += ls1b-core/
21
```

图 13.3 PWM 的目标文件定义

```
188     depends on LS1A_MACH
189
190 config HAVE_PWM
191     bool "Enable PWM"
192     default n
193     depends on LS1A_MACH || LS1B_MACH || LS1C_MACH
194     help
195       This option enable loongson 1a/1b cpu's pwm controller.
196       Be careful, some module's Pin multiplexing with the pwm,
197       as NAND GMAC and SPI.
198
199 config LS1X_EXTERN_TIMER
```

图 13.4 内核关联的配置

这就需要在内核配置时，添加以下选项：

```
config - Linux/mips 3.0.82 Kernel Configuration
                              Machine selection
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend:
  [*] built-in  [ ] excluded  <M> module  < > module capable

                  System type (Loongson family of machines)  --->
                  Machine Type (Loongson 1C board)  --->
              [ ] loongson 1c v2
              [ ] Enable PWM
```

图 13.5 内核配置 PWM 的选项

添加了 HAVE_PWM 后，就有了平台设备 pwm_device 的定义　ls1x_pwm_list：

```
31
32 #include <loongson1.h>
33 #include <irq.h>
34 #include <asm/gpio.h>
35 #include <asm-generic/sizes.h>
36 #include <media/soc_camera.h>
37 #include <media/soc_camera_platform.h>
38
39 #if defined(CONFIG_HAVE_PWM)
40 struct pwm_device ls1x_pwm_list[] = {
41     { 0, 06, false },
42     { 1, 92, false },
43     { 2, 93, false },
44     { 3, 37, false },
45 };
46 #endif
47
48 #ifdef CONFIG_MTD_NAND_LS1X
49 #include <ls1x_nand.h>
50 static struct mtd_partition ls1x_nand_partitions[] = {
51     {
52         .name   = "bootloader",
53         .offset = MTDPART_OFS_APPEND,
54         .size   = 1024*1024,
```

图 13.6　平台文件中配置 PWM 相关内容

(2)内核中选择 Loongson 1C board 的 CPU：

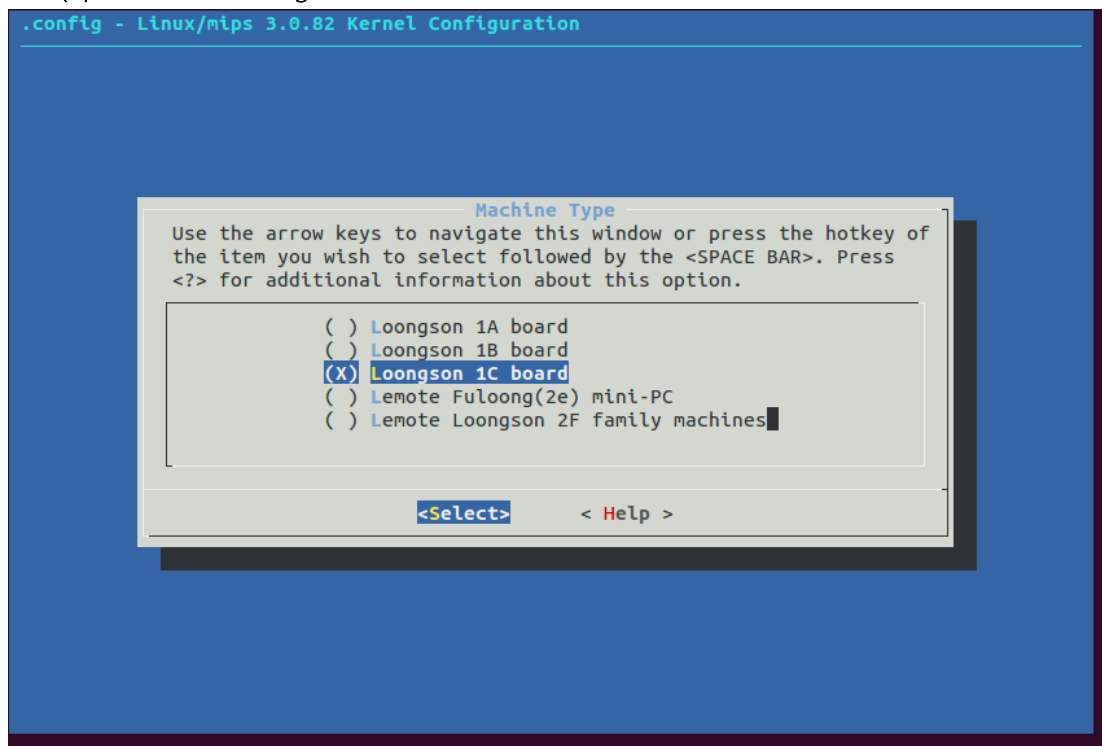


图 13.7　内核配置时 CPU 选项

```
\linux\linux-3.0.82-openloongson\linux-3.0.82-openloongson\arch\mips\Kconfig - Notepad++ [Administrator]

1209        release 2 instruction set.
1210
1211  config CPU_LOONGSON1C
1212      bool "Loongson 1C"
1213      depends on SYS_HAS_CPU_LOONGSON1C
1214      select CPU_LS232
1215      select GENERIC_GPIO
1216      select ARCH_REQUIRE_GPIOLIB
1217      help
1218        The Loongson 1C is a 32-bit SoC, which implements the MIPS32
1219        release 2 instruction set.
1220
1221  config CPU_MIPS32_R1
1222      bool "MIPS32 Release 1"
```

图 13.8 内核配置的 config 文件中 CPU 相关内容

则在 pwm.c 中定义了外部的 ls1x_pwm_list，分析驱动 pwm.c：

```
F:\Acer笔记本电脑\智龙2\1.智龙主板BSP\linux\linux-3.0.82-openloongson\linux-3.0.82-openloongson\arch\mips\loongson\ls1x\pwm.c - Notepad++ [Administrator]

52  DEFINE_MUTEX(ls1x_pwm_mutex);
53
54  #if defined(CONFIG_LS1A_MACH) || defined(CONFIG_LS1B_MACH)
55  static struct pwm_device ls1x_pwm_list[] = {
56      { 0, LS_GPIO_PWM0, false },
57      { 1, LS_GPIO_PWM1, false },
58      { 2, LS_GPIO_PWM2, false },
59      { 3, LS_GPIO_PWM3, false },
60  };
61  #elif defined(CONFIG_LS1C_MACH)
62  extern struct pwm_device ls1x_pwm_list[];
63  #endif
64
65  struct pwm_device *pwm_request(int id, const char *label)
66  {
67      int ret = 0;
68      u32 x;
69      struct pwm_device *pwm;
70
```

图 13.9 pwm.c 文件中定义的结构体

这个变量在 平台文件中定义，分析 ls1c300a_openloongson_v2.0_platform.c：

```
F:\Acer笔记本电脑\智龙2\1.智龙主板BSP\linux\linux-3.0.82-openloongson\linux-3.0.82-openloongson\arch\mips\loongson\ls1x\ls1c\ls1c300a_openloon...

30  #include <video/ls1xfb.h>
31
32  #include <loongson1.h>
33  #include <irq.h>
34  #include <asm/gpio.h>
35  #include <asm-generic/sizes.h>
36  #include <media/soc_camera.h>
37  #include <media/soc_camera_platform.h>
38
39  #if defined(CONFIG_HAVE_PWM)
40  struct pwm_device ls1x_pwm_list[] = {
41      { 0, 06, false },
42      { 1, 92, false },
43      { 2, 93, false },
44      { 3, 37, false },
45  };
```

图 13.10 平台文件中定义的结构体

下面看 leds_pwm：

在 drivers/leds/ 下的 makefile, leds_pwm 的编译依赖于 CONFIG_LEDS_PWM：

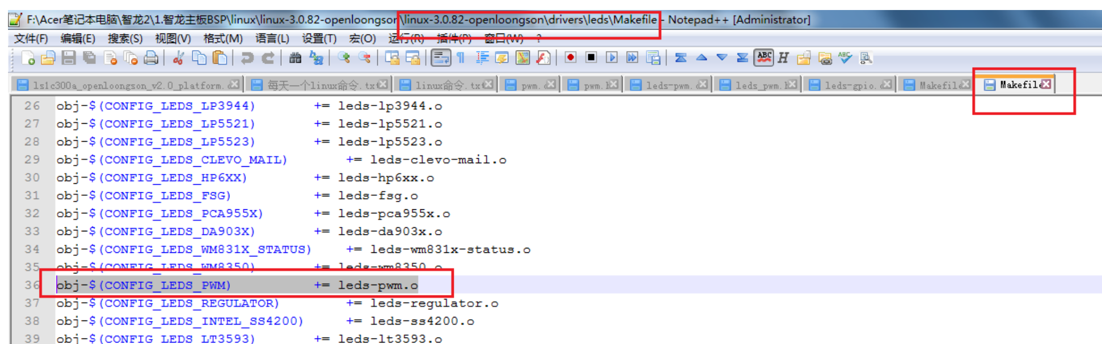


图 13.11 内核 led 目录下的 Makefile 文件中的目标文件

CONFIG_LEDS_PWM 在内核配置中如下配置：

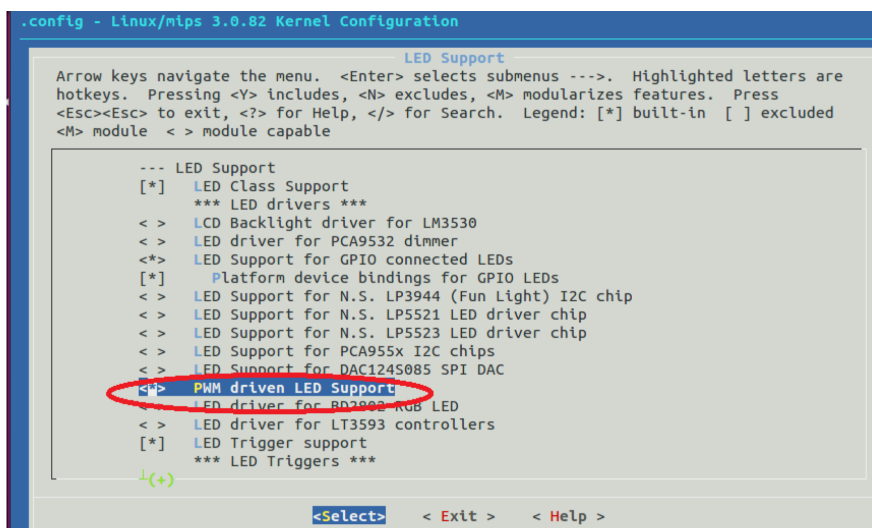


图 13.12 内核配置时 LED-PWM 选项

该配置选项的出现依赖于前图中的 HAVE_PWM 配置项：

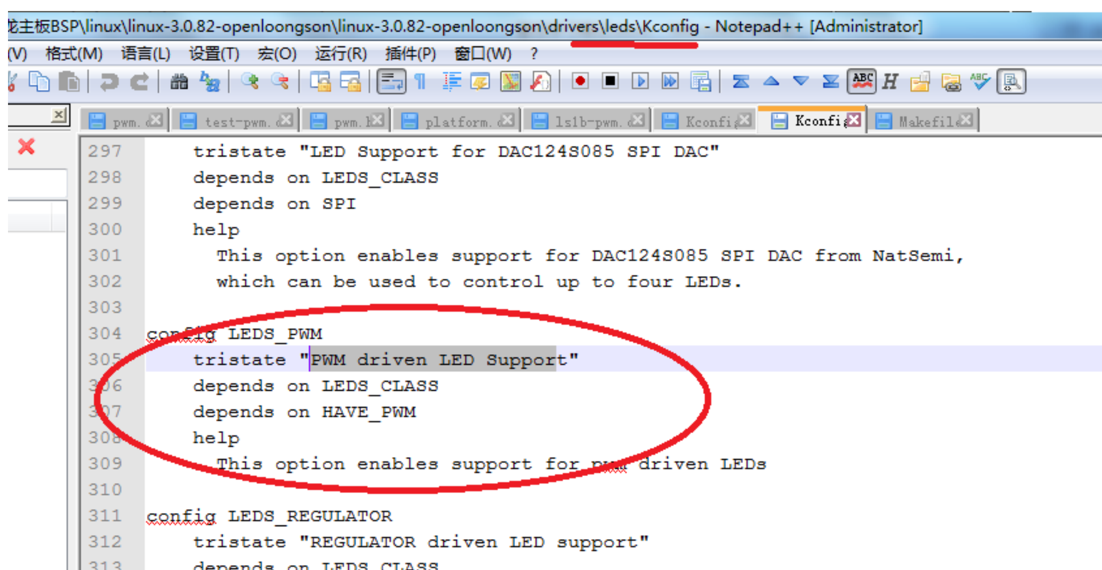


图 12.13 内核配置文件 led-pwm 选项

其中， struct pwm_device 在文件

linux-3.0.82-openloongson\linux-3.0.82-openloongson\arch\mips\include\asm\mach-loongson\ls1x\ls1x_pwm.h 中：

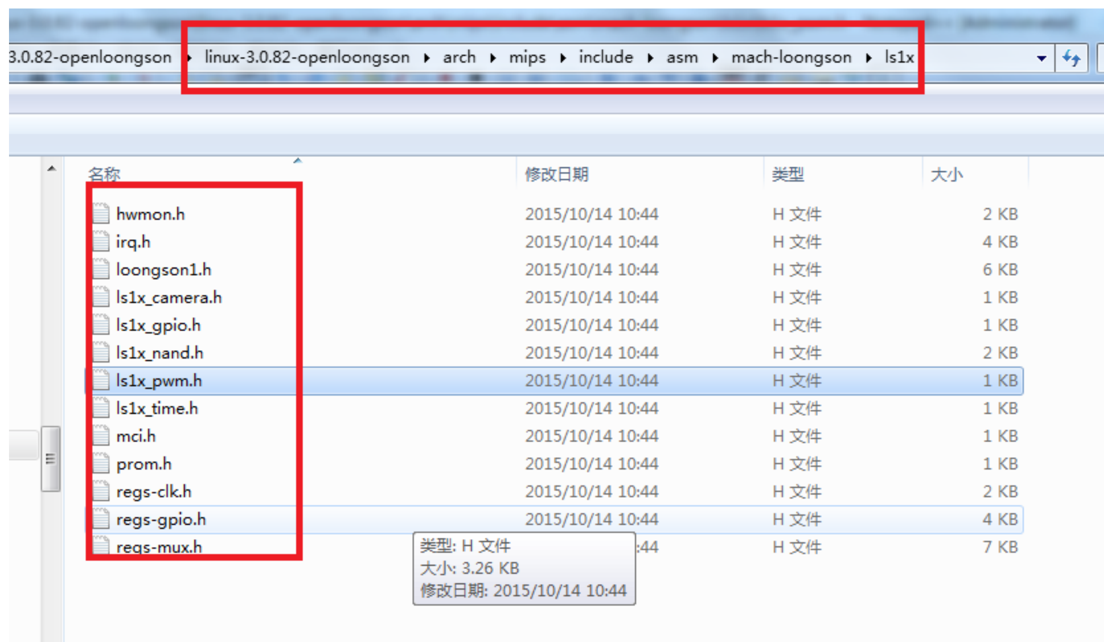


图 12.14 ls1x_pwm.h 文件中

```
#ifndef __ASM_ARCH_LS1X_PWM_H
#define __ASM_ARCH_LS1X_PWM_H

struct pwm_device {
        unsigned int id;
        unsigned int gpio;
        bool used;
};

#endif /* __ASM_ARCH_LS1X_PWM_H */
```

（3）Linux 内核配置选中 pwm 驱动

```
Machine selection    --->
   System type(Loongson family of machines)    --->
   Macine Type(Loongson 1C board)    --->
[ ] loongson 1c v2
[*] Enable PWM
```

在 LED 类中配置 PWM

```
Device Drivers    --->
[*] LED Support    --->
     <*> PWM driven LED Support
```

在平台文件 ls1c300a_openloongson_v2.0_platform.c 中添加：

```
#ifdef CONFIG_LEDS_PWM
static struct led_pwm ls1x_pwm_leds[] = {
        {
                .name              = "ls1x_pwm_led1",
                .pwm_id            = 0,
                .max_brightness  = 255,
                .pwm_period_ns = 7812500,
```

```
        },
        {
                .name           = "ls1x_pwm_led2",
                .pwm_id         = 1,
                .max_brightness = 255,
                .pwm_period_ns = 7812500,
        },
};

static struct led_pwm_platform_data ls1x_pwm_data = {
        .num_leds  = ARRAY_SIZE(ls1x_pwm_leds),
        .leds      = ls1x_pwm_leds,
};

static struct platform_device ls1x_leds_pwm = {
        .name      = "leds_pwm",
        .id        = -1,
        .dev  = {
                .platform_data = &ls1x_pwm_data,
        },
};


#endif //#ifdef CONFIG_LEDS_PWM
```

上一个平台文件中，函数 static struct platform_device *ls1b_platform_devices[] __initdata 中添加：

```
#ifdef CONFIG_LEDS_PWM
        &ls1x_leds_pwm,
#endif
```

平台文件增加设备成功 leds_pwm：

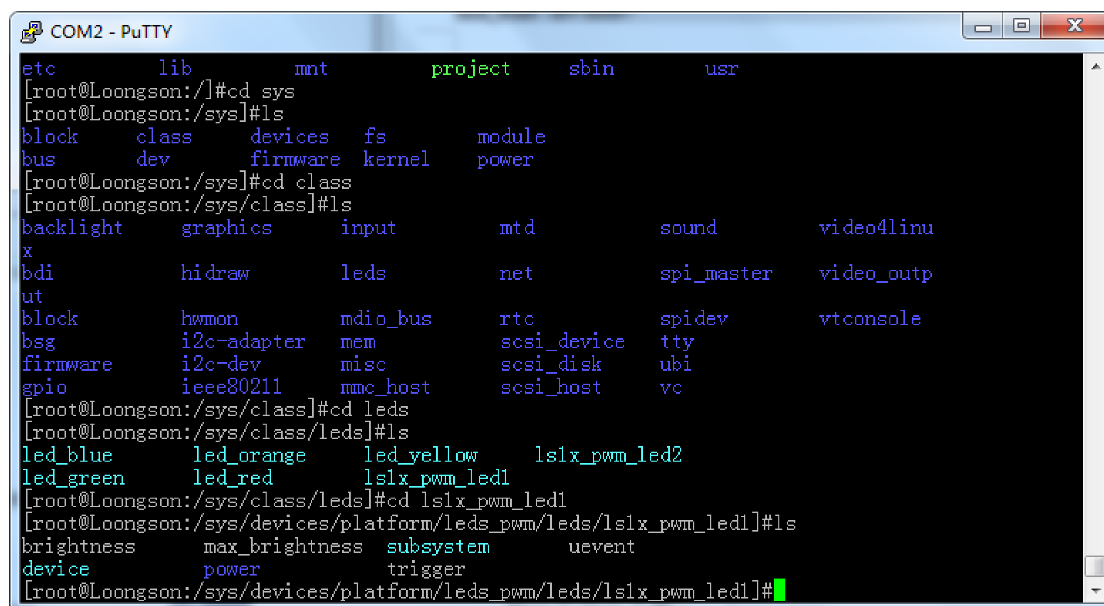


图 12.15 控制台中查看增加的设备

将 PWM0 占比调到最大(高电平占 100%)：

```
echo 255 > brightness
```

将 PWM0 占比调到最小(低电平占 100%)：

```
echo 0 > brightness
```

13.2 自己编写驱动文件

首先在平台文件中添加资源：

平台文件\arch\mips\loongson\ls1x\ls1c\ls1c300a_openloongson_v2.0_platform.c 添加：

```
#ifdef CONFIG_LS1C_PWM_DRIVER
static struct resource ls1c_pwm0_resource[] = {
	[0]={
		.start	= LS1X_PWM0_BASE,
		.end	= (LS1X_PWM0_BASE + 0x0f),
		.flags	= IORESOURCE_MEM,
	},
	[1]={
		.start	= LS1X_PWM1_BASE,
		.end	= (LS1X_PWM1_BASE + 0x0f),
		.flags	= IORESOURCE_MEM,
	},
	[2]={
		.start	= LS1X_PWM2_BASE,
		.end	= (LS1X_PWM2_BASE + 0x0f),
		.flags	= IORESOURCE_MEM,
	},
	[3]={
		.start	= LS1X_PWM3_BASE,
		.end	= (LS1X_PWM3_BASE + 0x0f),
		.flags	= IORESOURCE_MEM,
	},
};

static struct platform_device ls1c_pwm_device = {
	.name		= "ls1c-pwm",
	.id	= -1,
	.num_resources	= ARRAY_SIZE(ls1c_pwm0_resource),
	.resource	= ls1c_pwm0_resource,
};
#endif //#ifdef CONFIG_LS1C_PWM_DRIVER
```

上一个文件中，函数 static struct platform_device *ls1b_platform_devices[] __initdata 中添加：

```
#ifdef CONFIG_LS1C_PWM_DRIVER
	&ls1c_pwm_device,
#endif
```

在 \drivers\char\Kconfig 中添加：

```
config LS1C_PWM_DRIVER
	bool "ls1c pwm driver"
	default	n
	help
	  pwm.
```

在 \drivers\char\ 下的 Makefile 文件中添加语句：

```
obj-$(CONFIG_LS1C_PWM_DRIVER)		+= ls1c-pwm.o
```

平台设备增加成功：ls1c-pwm

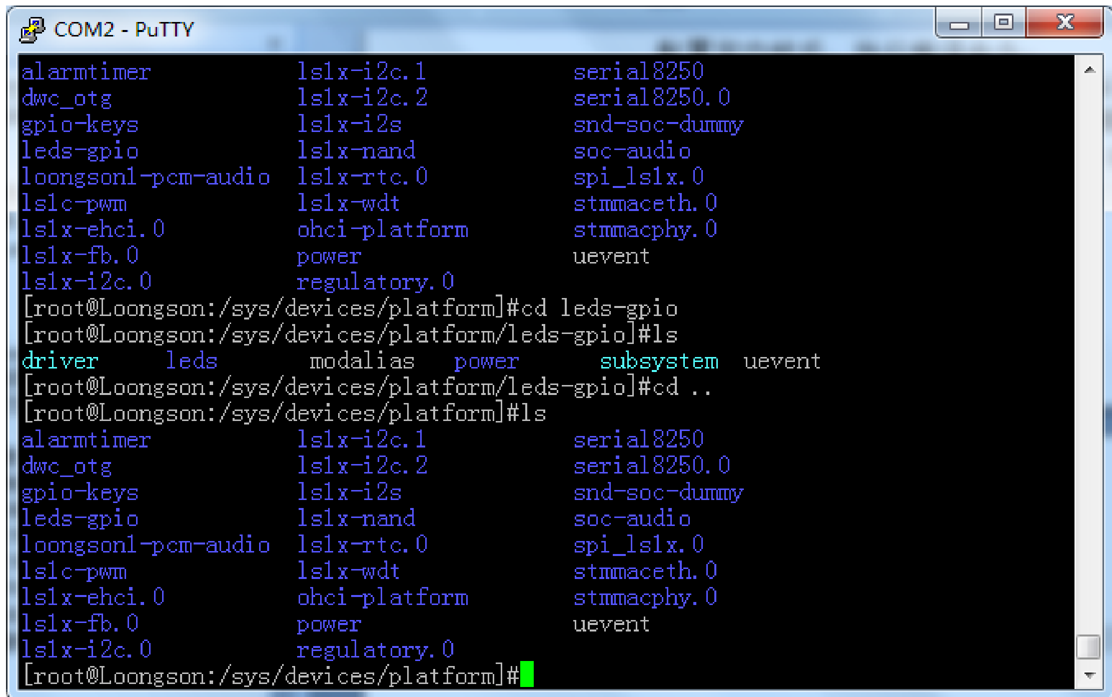


图 13.16 控制台中查看平台设备

编写驱动文件 ls1c_pwm.c（根据 1B 中的资料修改），放到内核代码 driver/char 下，

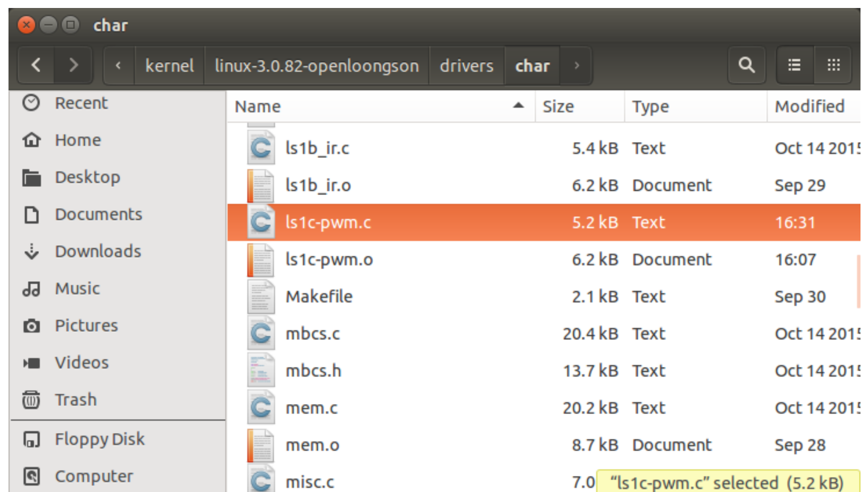


图 13.17 驱动文件在内核中位置

修改 Makefile 文件，添加语句：obj-$(CONFIG_LS1C_PWM_DRIVER) += ls1c-pwm.o

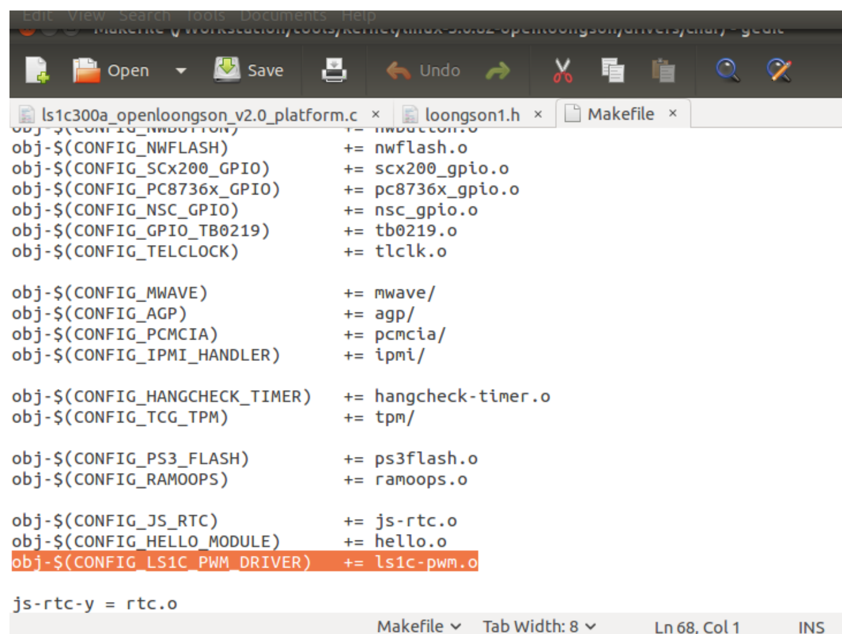


图 13.18 内核中 Makefile 生成目标文件

以下为设备驱动 ls1c_pwm.c 代码：

```
/*ls1c_pwm.c*/
#include <linux/miscdevice.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/ioctl.h>
#include <linux/platform_device.h>
#include <linux/timer.h>

#define CMD_PWM_GET          _IO('c',0x01)
#define CMD_PWM_START        _IO('c',0x02)
#define CMD_PWM_STOP         _IO('c',0x03)

#define REG_PWM_CNTR  0x00 // 0xBFE5C010
#define REG_PWM_HRC   0x04 // 0xBFE5C014
#define REG_PWM_LRC   0x08 // 0xBFE5C018
#define REG_PWM_CTRL  0x0c    // 0xBFE5C01C
#define REG_GPIO_CFG0  0xbfd010c0//GPIO GPIO[31:0] 配置寄存器 1 表示配置为 GPIO， 0 表示无效
#define REG_GPIO_CFG2  0xbfd010c8//GPIO[95:64] 配置寄存器 1 表示配置为 GPIO， 0 表示无效
#define SB2F_GPIO_MUX_CTRL1 0xbfd00424//功能复用寄存器

static unsigned char __iomem *pwm_base = NULL;
struct resource *res;
static int ls1f_pwm_probe(struct platform_device *pdev);
struct platform_device *pwm_dev;
struct resource *res1 = NULL;

static int ls1f_pwm_getResourse(struct platform_device *pdev, unsigned int index);

static struct platform_driver ls1f_pwm_driver = {
        .probe = ls1f_pwm_probe,
        .driver = {
                .name = "ls1c-pwm",
        },
};

static int ls1f_pwm_open(struct inode *inode, struct file *file)
{
```

```
        //配置 PWM0 PWM1  原始功能没有复用
        //long val = readl(SB2F_GPIO_MUX_CTRL1);
        //val &= 0xfffffffc;
        //writel(val, SB2F_GPIO_MUX_CTRL1);

        long val = readl(REG_GPIO_CFG0);
        //配置 GPIO6 引脚为普通功能，而非 GPIO 功能
        val &= 0xffffffbf;
        writel(val, REG_GPIO_CFG0);

        val = readl(REG_GPIO_CFG0);
        //配置 GPIO92 引脚为普通功能，而非 GPIO 功能
        val &= 0xefffffff;
        writel(val, REG_GPIO_CFG2);


        return 0;
}

static int ls1f_pwm_close(struct inode *inode, struct file *file)
{
        writel(0x0, pwm_base + REG_PWM_CTRL);
        return 0;
}

static ssize_t ls1f_pwm_read(struct file *file, char __user *buf, size_t count, loff_t *ptr)
{
        unsigned int pwm_val;
        pwm_val = readl(pwm_base);

        if (copy_to_user(buf, &pwm_val, sizeof(unsigned int)))
                return -EFAULT;
        return 4;
}

static ssize_t ls1f_pwm_write(struct file *file, const char __user *buf, size_t count, loff_t *ptr)
{
        unsigned int hrc_val, lrc_val;
        unsigned int data[2] = {0x0};

        if (copy_from_user(data, buf, sizeof(data)))
        {
                printk("Write error!\n");
                return -EIO;
        }

        hrc_val = data[1] - 1;
        lrc_val = data[0] + data[1] -1;

        //设置占空比
        writel(hrc_val, pwm_base + REG_PWM_HRC);
        writel(lrc_val, pwm_base + REG_PWM_LRC);
        printk("hrc:%i ; lrc:%i\n",hrc_val,lrc_val);
        return 0;
}

static int pwm_start_stop(unsigned int cmd, unsigned long arg)
{
        printk("into: %s\n", __FUNCTION__);
        printk("arg: %ld\n", arg);

        if (arg > 3)
                return -1;
        //从 platform 中获取指定 PWM 的寄存器基址  pwm_base
        //通过改变 arg，实现在 pwm0、pwm1、pwm2、pwm3 之间切换
        ls1f_pwm_getResourse(pwm_dev, arg);
```

```
	switch (cmd) {
	//启动 PWM
	case CMD_PWM_START:
		printk("CMD_PWM_START\n");
		writel(0x0, pwm_base + REG_PWM_CNTR);
		writel(0x01, pwm_base + REG_PWM_CTRL);
		break;
	//停止 PWM
	case CMD_PWM_STOP:
		printk("CMD_PWM_STOP\n");
		writel(0x0, pwm_base + REG_PWM_CTRL);
		break;
	default:
		break;
	}
	return 0;
}

static long ls1f_pwm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
	printk("into %s\n", __FUNCTION__);
	printk("cmd: %d\n", cmd);
	printk("arg: %ld\n", arg);

	if (arg > 3)
		return -1;

	switch (cmd) {
	case CMD_PWM_GET:
		ls1f_pwm_getResourse(pwm_dev, arg);
		break;
	case CMD_PWM_START:
	case CMD_PWM_STOP:
		return pwm_start_stop(cmd, arg);
	default:
		break;
	}
	return 0;
}


static const struct file_operations ls1f_pwm_ops = {
		.owner = THIS_MODULE,
		.open = ls1f_pwm_open,
		.release = ls1f_pwm_close,
		.read = ls1f_pwm_read,
		.write = ls1f_pwm_write,
		.unlocked_ioctl = ls1f_pwm_ioctl,
};

static struct miscdevice ls1f_pwm_miscdev = {
		MISC_DYNAMIC_MINOR,
		"ls1f-pwm",
		&ls1f_pwm_ops,
};

static int ls1f_pwm_getResourse(struct platform_device *pdev, unsigned int index)
{
	res = platform_get_resource(pdev, IORESOURCE_MEM, index);

	if (res == NULL)
	{
	   printk("Fail to get ls1f_pwm_resource!\n");
	   return -ENOENT;
	}
	printk("Resource start=0x%x, end = 0x%x\n", res->start, res->end);
	if (res1 != NULL)
```

```
        {
                release_mem_region(res->start, 0x0f);
        }
        res1 = request_mem_region(res->start, 0x0f, "ls1f-pwm");
        if (res1 == NULL)
        {
           printk("Fail to request ls1f_pwm region!\n");
           return -ENOENT;
        }
        pwm_base = ioremap(res->start, res->end - res->start + 1);
        if (pwm_base == NULL)
        {
           printk("Fail to ioremap ls1f_pwm resource!\n");
           return -EINVAL;
        }
        return 0;
}

static int __devinit ls1f_pwm_probe(struct platform_device *pdev)
{
        pwm_dev = pdev;
        return ls1f_pwm_getResourse(pdev, 1);
}

static int __init ls1f_pwm_init(void) {
        if (misc_register(&ls1f_pwm_miscdev))
        {
           printk(KERN_WARNING "pwm: Couldn't register device 10, %d.\n", 255);
           return -EBUSY;
        }
        return platform_driver_register(&ls1f_pwm_driver);
}

static void __exit ls1f_pwm_exit(void)
{
        misc_deregister(&ls1f_pwm_miscdev);
        release_mem_region(res->start, 0x20);
        platform_driver_unregister(&ls1f_pwm_driver);
}

module_init(ls1f_pwm_init);
module_exit(ls1f_pwm_exit);
MODULE_AUTHOR("sundm");
MODULE_DESCRIPTION("loongson 1C PWM driver");
MODULE_LICENSE("GPL");
```

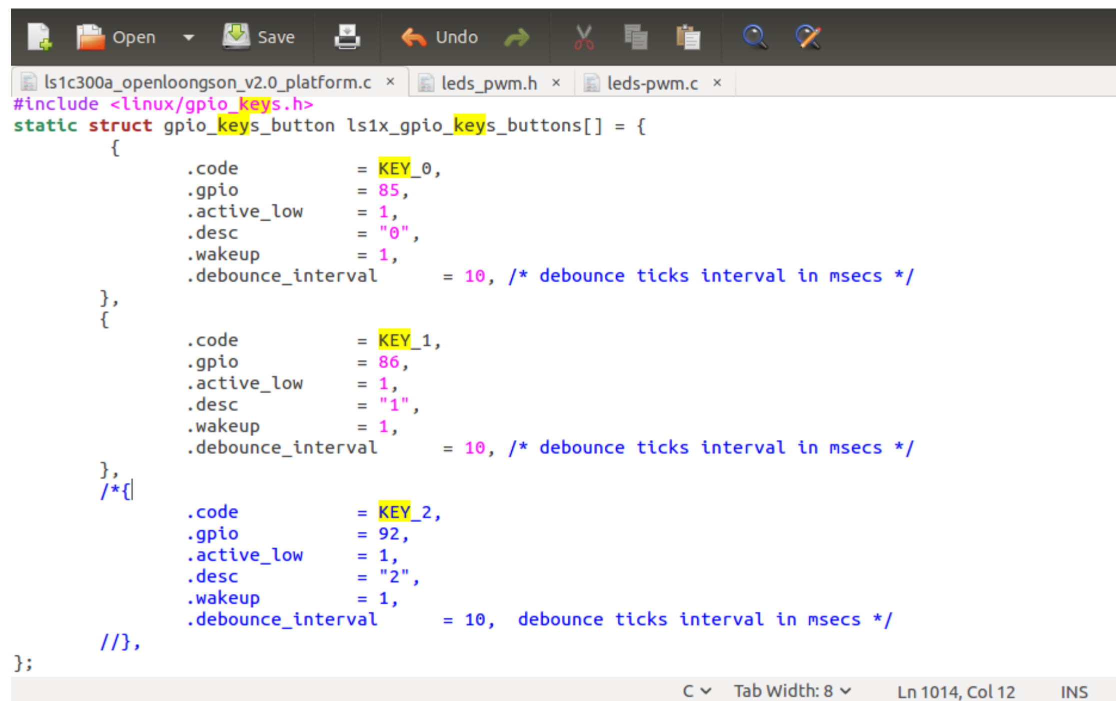
编译成功后，编写文件 pwm.c，包含了对 PWM 设备的操作函数：

```
/*pwm.c*/
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define CMD_PWM_GET         _IO('c',0x01)
#define CMD_PWM_START       _IO('c',0x02)
#define CMD_PWM_STOP        _IO('c',0x03)
//sel 为选择 PWM 通道(0 或者 1);value 为占空比(0-100);    freq 为频率().
int pwmset(unsigned int sel, unsigned int value, unsigned int freq) {
        int fd;
       //data[0]的值代表的是高电平脉冲所占的时钟数，data[1]的值代表的是低电平脉冲所占的时钟数
        unsigned int data[2] = {0x7ffffff,0x7ffffff};

        if(value>100)
                value = 100;
        data[0] = freq / 100 * value;
        data[1] = freq / 100 * (100-value);
```

```
        printf("cys: ready to open!\n");
        fd = open("/dev/ls1f-pwm",O_RDWR);
        sleep(2);
        if (fd != -1)
        {
           printf("open ok!\n");
           //共有四路 pwm(pwm0...pwm3)，四路 pwm 可同时工作，也可选择使用其中任何一路

               if (sel <0 || sel > 3)
                  return -1;
               switch(sel) {
                  case 0:
                       //在驱动中没有用到 ioctl 的第三个参数，故直接赋予 0 就可以啦
                     ioctl(fd, CMD_PWM_START , 0);
                     write(fd, data, sizeof(data) );
                     break;
                  case 1:
                   ioctl(fd, CMD_PWM_START , 1);
                     write(fd, data, sizeof(data) );
                   break;
                  case 2:
                   ioctl(fd, CMD_PWM_START , 2);
                     write(fd, data, sizeof(data) );
                   break;
                  case 3:
                   ioctl(fd, CMD_PWM_START , 3);
                     write(fd, data, sizeof(data) );
                   break;
                  default:
                     break;
               }
        }
        else
        {
               printf("Device open failure\n");
        }
        sleep(10);
        close(fd);
        return 0;
}
```

举例：调用函数 pwmset(0,50,10000) ，意思是在 PWM0 引脚产生占空比 50%, 周期 10000 时钟周期的波形。

另外 GPIO92 与其中一个 KEY 冲突，需要在平台文件中注释掉 KEY 92 的定义。



```
#include <linux/gpio_keys.h>
static struct gpio_keys_button ls1x_gpio_keys_buttons[] = {
	 {
		.code		= KEY_0,
		.gpio		= 85,
		.active_low	= 1,
		.desc		= "0",
		.wakeup		= 1,
		.debounce_interval	= 10, /* debounce ticks interval in msecs */
	},
	{
		.code		= KEY_1,
		.gpio		= 86,
		.active_low	= 1,
		.desc		= "1",
		.wakeup		= 1,
		.debounce_interval	= 10, /* debounce ticks interval in msecs */
	},
	/*{
		.code		= KEY_2,
		.gpio		= 92,
		.active_low	= 1,
		.desc		= "2",
		.wakeup		= 1,
		.debounce_interval	= 10,  debounce ticks interval in msecs */
	//},
};
```

图 13.19 内核在平台文件中注释掉 KEY 92 的定义

14. I2C 驱动

14.1 Linux I2C 设备驱动编写

摘自 http://blog.csdn.net/airk000/article/details/21345457。

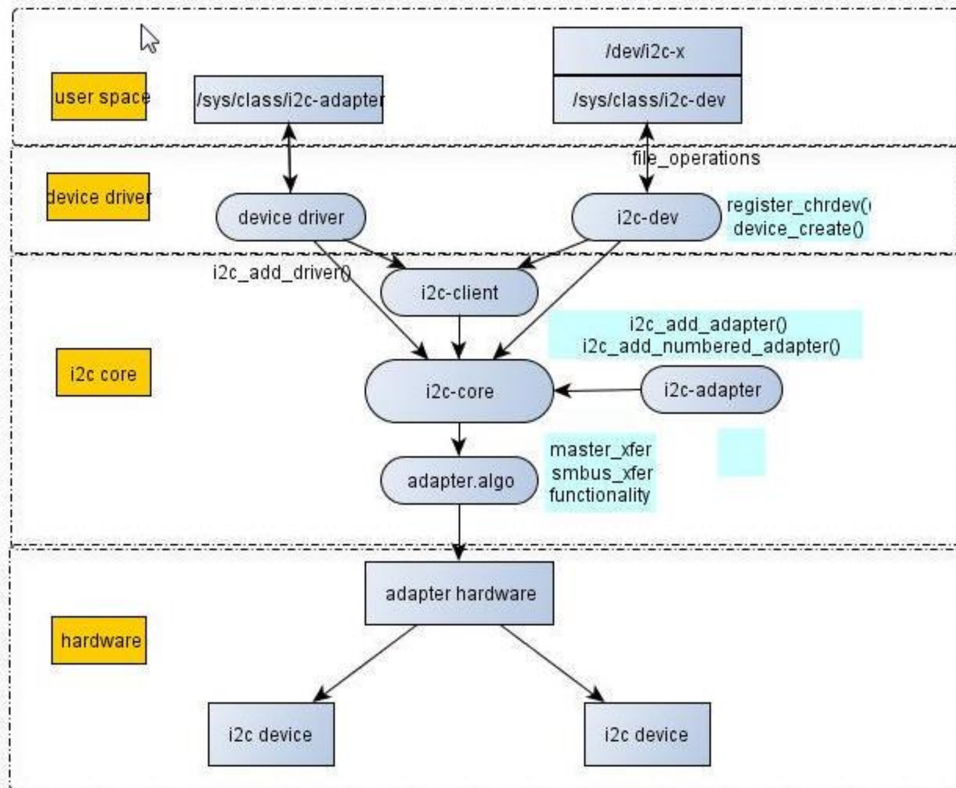


图 14.1 I2C 驱动示意图

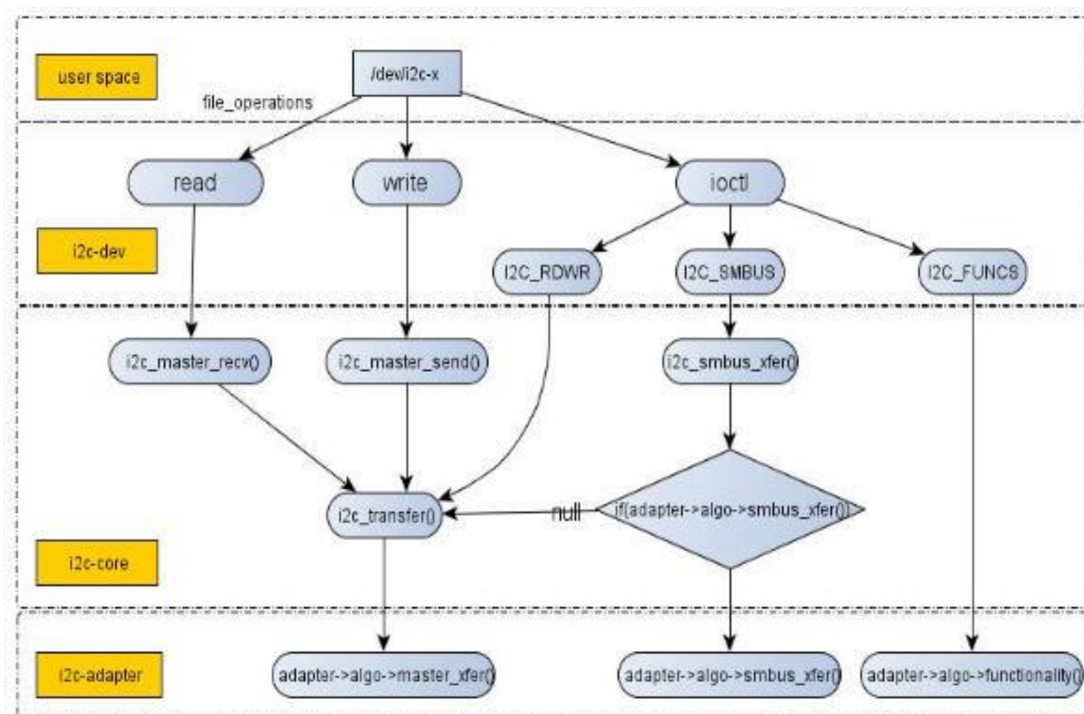


图 14.2 I2C 设备示意图

14.1.1 I2C adapter

是 CPU 集成或外接的 I2C 适配器，用来控制各种 I2C 从设备，其驱动需要完成对适配器的完整描述，最主要的工作是需要完成 i2c_algorithm 结构体。这个结构体包含了此 I2C 控制器的数据传输具体实现，以及对外上报此设备所支持的功能类型。i2c_algorithm 结构体如下：

```
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 *);
};
```

如果一个 I2C 适配器不支持 I2C 通道，那么就将 master_xfer 成员设为 NULL。如果适配器支持 SMBUS 协议，那么需要去实现 smbus_xfer，如果 smbus_xfer 指针被设为 NULL，那么当使用 SMBUS 协议的时候将会通过 I2C 通道进行仿真。master_xfer 指向的函数的返回值应该是已经成功处理的消息数，或者返回负数表示出错了。functionality 指针很简单，告诉询问着这个 I2C 主控器都支持什么功能。

在内核的 drivers/i2c/i2c-stub.c 中实现了一个 i2c adapter 的例子，其中实现的是更为复杂的 SMBUS。

14.1.1.1 SMBus 与 I2C 的区别

通常情况下，I2C 和 SMBus 是兼容的，但是还是有些微妙的区别的。

时钟速度对比：

	I2C	SMBus
最小	无	10kHz
最大	100kHZ（标准）400kHz（快速模式）2MHz（高速模式）	100kHz
超时	无	35ms

图 14.3 I2C 总线与 SMBUS 对比

在电气特性上 SMBus 要求的电压范围更低。

14.1.1.2 I2C driver

具体的 I2C 设备驱动，如相机、传感器、触摸屏、背光控制器常见硬件设备大多都有或都是通过 I2C 协议与主机进行数据传输、控制。结构体如下：

```
struct i2c_driver {
	unsigned int class;

	/* Notifies the driver that a new bus has appeared or is about to be
	 * removed. You should avoid using this, it will be removed in a
	 * near future.
	 */
	int (*attach_adapter)(struct i2c_adapter *) __deprecated;   //旧的与设备进行绑定的接口函数
	int (*detach_adapter)(struct i2c_adapter *) __deprecated;   //旧的与设备进行解绑的接口函数

	/* Standard driver model interfaces */
	int (*probe)(struct i2c_client *, const struct i2c_device_id *); //现行通用的与对应设备进行绑定的接口函
数
	int (*remove)(struct i2c_client *);   //现行通用与对应设备进行解绑的接口函数

	/* driver model interfaces that don't relate to enumeration   */
	void (*shutdown)(struct i2c_client *);   //关闭设备
	int (*suspend)(struct i2c_client *, pm_message_t mesg); //挂起设备，与电源管理有关，为省电
	int (*resume)(struct i2c_client *); //从挂起状态恢复

	/* Alert callback, for example for the SMBus alert protocol.
	 * The format and meaning of the data value depends on the protocol.
	 * For the SMBus alert protocol, there is a single bit of data passed
	 * as the alert response's low bit ("event flag").
	 */
	void (*alert)(struct i2c_client *, unsigned int data);

	/* a ioctl like command that can be used to perform specific functions
	 * with the device.
	 */
	int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

	struct device_driver driver;   //I2C 设备的驱动模型
	const struct i2c_device_id *id_table;   //匹配设备列表

	/* Device detection callback for automatic device creation */
	int (*detect)(struct i2c_client *, struct i2c_board_info *);
	const unsigned short *address_list;
	struct list_head clients;
};
#define to_i2c_driver(d) container_of(d, struct i2c_driver, driver)   //一般编写驱动过程中对象常是 driver 类型，
可以通过 to_i2c_driver 找到其父类型 i2c_driver
```

如同普通设备的驱动能够驱动多个设备一样，一个 I2C driver 也可以对应多个 I2C client。

以重力传感器 AXLL34X 为例，其实现的 I2C 驱动为：

```
static const struct i2c_device_id adxl34x_id[] = {
	{ "adxl34x", 0 },   //匹配 i2c client 名为 adxl34x 的设备
```

```
        { }
};

MODULE_DEVICE_TABLE(i2c, adxl34x_id);

static struct i2c_driver adxl34x_driver = {
        .driver = {
                .name = "adxl34x",
                .owner = THIS_MODULE,
                .pm = &adxl34x_i2c_pm,    //指定设备驱动的电源管理接口，包含 suspend、resume
        },
        .probe      = adxl34x_i2c_probe,    //组装设备匹配时候的匹配动作
        .remove     = adxl34x_i2c_remove,    //组装设备移除接口
        .id_table = adxl34x_id,    //制定匹配设备列表
};

module_i2c_driver(adxl34x_driver);
```

这里要说明一下 module_i2c_driver 宏定义(i2c.h)：

```
#define module_i2c_driver(__i2c_driver) \
        module_driver(__i2c_driver, i2c_add_driver, \
                                i2c_del_driver)

#define i2c_add_driver(driver) \
                i2c_register_driver(THIS_MODULE, driver)
```

module_driver():

```
#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
                return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init); \
static void __exit __driver##_exit(void) \
{ \
                __unregister(&(__driver) , ##__VA_ARGS__); \
} \
module_exit(__driver##_exit);
```

理解上述宏定义后，将 module_i2c_driver(adxl34x_driver)展开就可以得到：

```
static int __int adxl34x_driver_init(void)
{
        return i2c_register_driver(&adxl34x_driver);
}
module_init(adxl34x_driver_init);
static void __exit adxl34x_driver_exit(void)
{
        return i2c_del_driver(&adxl34x_driver);
}
module_exit(adxl34x_driver_exit);
```

这一句宏就解决了模块 module 安装卸载的复杂代码。这样驱动开发者在实现 I2C 驱动时只要将 i2c_driver 结构体填充进来就可以了，无需关心设备的注册与反注册过程。

14.1.1.3 I2C client

即 I2C 设备。I2C 设备的注册一般在板级代码中，在解析实例前还是先熟悉几个定义：

```
struct i2c_client {
        unsigned  short  flags;              //I2C_CLIENT_TEN 表示设备使用 10bit 从地址，I2C_CLIENT_PEC 表示
设备使用 SMBus 检错
        unsigned  short  addr;              //设备从地址，7bit。这里说一下为什么是 7 位，因为最后以为 0 表示写，
1 表示读，通过对这个 7bit 地址移位处理即可。addr<<1 & 0x0 即写，addr<<1 | 0x01 即读。
        char name[I2C_NAME_SIZE];    //从设备名称
        struct i2c_adapter *adapter;       //此从设备依附于哪个 adapter 上
```

```
	struct i2c_driver *driver;	// 此设备对应的 I2C 驱动指针
	struct device dev;		// 设备模型
	int irq;			// 设备使用的中断号
	struct list_head detected;	//用于链表操作
};
#define to_i2c_client(d) container_of(d, struct i2c_client, dev)	//通常使用 device 设备模型进行操作，可以通过 to_i2c_client 找到对应 client 指针

struct i2c_board_info {
	char		type[I2C_NAME_SIZE];	//设备名，最长 20 个字符，最终安装到 client 的 name 上
	unsigned short	flags;	//最终安装到 client.flags
	unsigned short	addr;	//设备从地址 slave address，最终安装到 client.addr 上
	void		*platform_data;	//设备数据，最终存储到 i2c_client.dev.platform_data 上
	struct dev_archdata	*archdata;
	struct device_node *of_node;	//OpenFirmware 设备节点指针
	struct acpi_dev_node acpi_node;
	int		irq;	//设备采用的中断号，最终存储到 i2c_client.irq 上
};
//可以看到，i2c_board_info 基本是与 i2c_client 对应的。
#define I2C_BOARD_INFO(dev_type, dev_addr) \
	.type = dev_type, .addr = (dev_addr)
//通过这个宏定义可以方便的定义 I2C 设备的名称和从地址（别忘了是 7bit 的）
```

下面还是以 adxl34x 为例：

```
static struct i2c_board_info i2c0_devices[] = {
	{
		I2C_BOARD_INFO("ak4648", 0x12),
	},
	{
		I2C_BOARD_INFO("r2025sd", 0x32),
	},
	{
		I2C_BOARD_INFO("ak8975", 0x0c),
		.irq = intcs_evt2irq(0x3380), /* IRQ28 */
	},
	{
		I2C_BOARD_INFO("adxl34x", 0x1d),
		.irq = intcs_evt2irq(0x3340), /* IRQ26 */
	},
};
...
i2c_register_board_info(0, i2c0_devices, ARRAY_SIZE(i2c0_devices));
```

这样 ADXL34X 的 i2c 设备就被注册到了系统中，当名字与 i2c_driver 中的 id_table 中的成员匹配时就能够出发 probe 匹配函数了。

带着问题去分析可能会更有帮助吧，通过对 1. 的了解后，可能会产生以下的几点疑问：

- i2c_adapter 驱动如何添加？
- i2c_client 与 i2c_board_info 究竟是什么关系？

14.1.2 I2C 子系统驱动模块的 API

Linux 内核的 I2C 子系统对驱动模块的 API 有以下：

```
// 对外数据结构
struct i2c_driver — 代表一个 I2C 设备驱动
struct i2c_client — 代表一个 I2C 从设备
struct i2c_board_info — 从设备创建的模版
I2C_BOARD_INFO — 创建 I2C 设备的宏，包含名字和地址
struct i2c_algorithm — 代表 I2C 传输方法
struct i2c_bus_recovery_info — I2C 总线恢复信息？内核新加入的结构，不是很清楚。
//对外函数操作
module_i2c_driver — 注册 I2C 设备驱动的宏定义
```

```
i2c_register_board_info — 静态声明（注册）I2C 设备，可多个
i2c_verify_client — 如果设备是 i2c_client 的 dev 成员则返回其父指针，否则返回 NULL。用来校验设备是否为 I2C 设备
i2c_lock_adapter — I2C 总线持锁操作，会找到最根源的那个 i2c_adapter。说明你的模块必须符合 GPL 协议才可以使用这个接口。后边以 GPL 代表。
i2c_unlock_adapter — 上一个的反操作，GPL
i2c_new_device — 由 i2c_board_info 信息声明一个 i2c 设备（client），GPL
i2c_unregister_device — 上一个的反操作，GPL。
i2c_new_dummy — 声明一个名为 dummy（指定地址）的 I2C 设备，GPL
i2c_verify_adapter — 验证是否是 i2c_adapter
i2c_add_adapter — 声明 I2C 适配器，系统动态分配总线号。
i2c_add_numbered_adapter — 同样是声明 I2C 适配器，但是指定了总线号，GPL
i2c_del_adapter — 卸载 I2C 适配器
i2c_del_driver — 卸载 I2C 设备驱动
i2c_use_client — i2c_client 引用数+1
i2c_release_client — i2c_client 引用数-1
__i2c_transfer — 没有自动持锁(adapter lock)的 I2C 传输接口
i2c_transfer — 自动持锁的 I2C 传输接口
i2c_master_send — 单条消息发送
i2c_master_recv — 单条消息接收
i2c_smbus_read_byte — SMBus “receive byte” protocol
i2c_smbus_write_byte — SMBus “send byte” protocol
i2c_smbus_read_byte_data — SMBus “read byte” protocol
i2c_smbus_write_byte_data — SMBus “write byte” protocol
i2c_smbus_read_word_data — SMBus “read word” protocol
i2c_smbus_write_word_data — SMBus “write word” protocol
i2c_smbus_read_block_data — SMBus “block read” protocol
i2c_smbus_write_block_data — SMBus “block write” protocol
i2c_smbus_xfer — execute SMBus protocol operations
```

1. 对几个基本的结构体和宏定义也有了大概的解释，相信结合 I2C 的理论基础不难理解。对以上一些 I2C 的 API 进行分类：

表 14.1 I2C 的 API 分类表

No.	Adapter	Driver	Device(client)	Transfer
1	i2c_add_adapter	module_i2c_driver	i2c_register_board_info	__i2c_transfer
2	i2c_add_numbered_adapter	i2c_del_driver	i2c_new_device	i2c_transfer
3	i2c_del_adapter		i2c_new_dummy	i2c_master_send
4	i2c_lock_adapter		i2c_verify_client	i2c_master_recv
5	i2c_unlock_adapter		i2c_unregister_device	i2c_smbus_read_byte
6	i2c_verify_adapter		i2c_use_client	i2c_smbus_write_byte
7			i2c_release_client	i2c_smbus_read_byte_data
8				i2c_smbus_write_byte_data
9				i2c_smbus_read_word_data
10				i2c_smbus_write_word_data
11				i2c_smbus_read_block_data
12				i2c_smbus_write_block_data
13				i2c_smbus_xfer

发现在 Linux I2C 子系统中，最重要的要数 i2c_client，而最多样化的就是数据的传输。下边的顺序是由 client 到 driver，再到 adapter。

14.1.3 I2C client 的注册

i2c_client 即 I2C 设备的注册接口有三个：

```
i2c_register_board_info
i2c_new_device
i2c_new_dummy
```

而 i2c_new_dummy 在内部其实也就是将 client 的 name 指定为 dummy 后依旧执行的是 i2c_new_device，所以就只分析前两个就可以了。首先看这两个函数的原型：

```
i2c_register_board_info(int busnum, struct i2c_board_info const *info, unsigned len)
//busnum 通过总线号指定这个（些）设备属于哪个总线
//info i2c 设备的数组集合 i2c_board_info 格式
//len 数组个数 ARRAY_SIZE(info)

i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
//adap 此设备所依附的 I2C 适配器指针
//info 此设备描述，i2c_board_info 格式，bus_num 成员是被忽略的
```

14.1.3.1 使用总线号声明设备 i2c_register_board_info

在内核的初始化中定义设备的信息。前提是内核编译的时候已经确定有哪些 i2c 设备和它们的地址，还要知道连接的总线的编号。

```
int __init
i2c_register_board_info(int busnum,
	struct i2c_board_info const *info, unsigned len)
{
	int status;

	down_write(&__i2c_board_lock);	//i2c 设备信息读写锁，锁写操作，其他只读

	/* dynamic bus numbers will be assigned after the last static one */
	if (busnum >= __i2c_first_dynamic_bus_num)	//与动态分配的总线号相关，动态分配的总线号应该是从已经现有最大总线号基础上+1 的，这样能够保证动态分配出的总线号与板级总线号不会产生冲突
		__i2c_first_dynamic_bus_num = busnum + 1;

	for (status = 0; len; len--, info++) {	//处理 info 数组中每个成员
		struct i2c_devinfo	*devinfo;

		devinfo = kzalloc(sizeof(*devinfo), GFP_KERNEL);
		if (!devinfo) {
			pr_debug("i2c-core: can't register boardinfo!\n");
			status = -ENOMEM;
			break;
		}

		devinfo->busnum = busnum;	//组装总线号
		devinfo->board_info = *info;	//组装设备信息
		list_add_tail(&devinfo->list, &__i2c_board_list);	//加入到__i2c_board_list 链表中（尾部）
	}

	up_write(&__i2c_board_lock);	//释放读锁，其他可读可写

	return status;
}
```

怎么将相关信息放到链表中就算完事了吗？不着急，来看下内核中已经给出的解释：

```
* Systems using the Linux I2C driver stack can declare tables of board info
 * while they initialize.  This should be done in board-specific init code
 * near arch_initcall() time, or equivalent, before any I2C adapter driver is
 * registered.  For example, mainboard init code could define several devices,
 * as could the init code for each daughtercard in a board stack.
 *
 * The I2C devices will be created later, after the adapter for the relevant
 * bus has been registered.  After that moment, standard driver model tools
 * are used to bind "new style" I2C drivers to the devices.  The bus number
 * for any device declared using this routine is not available for dynamic
 * allocation.
```

就是说关于集成的 I2C 设备注册过程应该在板级代码初始化期间，也就是 arch_initcall

前后的时间，或者就在这个时候（board-xxx-yyy.c 中），切记切记！！！一定要在 I2C 适配器驱动注册前完成！！！为什么说是静态注册，是因为真实的 I2C 设备是在适配器成功注册后才被生成的。如果在 I2C 适配器注册完后还想要添加 I2C 设备的话，就要通过新方式！（即 i2c_new_device）

对于__i2c_board_list 链表中的信息是如何变成实际的 i2c 设备信息的过程放在之后 adapter 注册过程的分析中。记得，重点是 i2c_register_board_info 方式一定要赶在 I2C 适配器的注册前，这样就没有问题。

14.1.3.2 枚举设备 i2c_new_device 或者 i2c_new_probed_device

方法 1 有诸多限制，必须必须在编译内核的时候知道 i2c 的总线编号和物理的连接。有时开发者面对的是一个已经存在的系统，无法修改内核。

或者内核开发者移植系统的时候也不知道有哪些 i2c 设备或者到底有多少 i2c 总线。

在这种情况下就需要用到 i2c_new_device()了。它的原型是：

```
struct i2c_client * i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info);
```

这个函数将会使用 info 提供的信息建立一个 i2c_client 并与第一个参数指向的 i2c_adapter 绑定。返回的参数是一个 i2c_client 指针。

驱动中可以直接使用 i2c_client 指针和设备通信了。这个方法是一个比较简单的方法。

```
struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
{
	struct i2c_client	*client;
	int			status;

	client = kzalloc(sizeof *client, GFP_KERNEL);	//为即将注册的 client 申请内存
	if (!client)
		return NULL;

	client->adapter = adap;	//绑定指定的 adapter 适配器

	client->dev.platform_data = info->platform_data;	//保存设备数据

	if (info->archdata)	//代码上看是 DMA 相关操作数据
		client->dev.archdata = *info->archdata;

	client->flags = info->flags;	//类型，（一）中说过，或是 10 位地址，或是使用 SMBus 检错
	client->addr = info->addr;	//设备从地址
	client->irq = info->irq;	//设备终端

	strlcpy(client->name, info->type, sizeof(client->name));	//从设备名
			//（1.）中说过 i2c_board_info 中的信息是与 i2c_client 有对应关系的

	/* Check for address validity */
	status = i2c_check_client_addr_validity(client);	//检测地址是否有效，10 位地址是否大于 0x3ff，7 位地址是否大于 0x7f 或为 0
	if (status) {	//非零（实际上为-22，无效参数 Invalid argument
		dev_err(&adap->dev, "Invalid %d-bit I2C address 0x%02hx\n",
			client->flags & I2C_CLIENT_TEN ? 10 : 7, client->addr);
		goto out_err_silent;
	}

	/* Check for address business */
	status = i2c_check_addr_busy(adap, client->addr);	//检测指定适配器上该地址状态
	if (status)
		goto out_err;
```

```
	client->dev.parent = &client->adapter->dev;	//建立从设备与适配器的父子关系
	client->dev.bus = &i2c_bus_type;
	client->dev.type = &i2c_client_type;
	client->dev.of_node = info->of_node;
	ACPI_HANDLE_SET(&client->dev, info->acpi_node.handle);

	/* For 10-bit clients, add an arbitrary offset to avoid collisions */
	dev_set_name(&client->dev, "%d-%04x", i2c_adapter_id(adap),
		     client->addr | ((client->flags & I2C_CLIENT_TEN)
				     ? 0xa000 : 0));	//如果是 10 位地址设备，那名字格式与 7bit 的会有不同
	status = device_register(&client->dev);	//注册了！注册了！！！
	if (status)
		goto out_err;

	dev_dbg(&adap->dev, "client [%s] registered with bus id %s\n",
		client->name, dev_name(&client->dev));

	return client;

out_err:
	dev_err(&adap->dev, "Failed to register i2c client %s at 0x%02x "
		"(%d)\n", client->name, client->addr, status);
out_err_silent:
	kfree(client);
	return NULL;
}
```

i2d_new_device 没什么好多说的，由于有 i2c_register_board_info 的铺垫，相信也很好理解了。而 i2c_new_device 不但印证了 i2c_client 与 i2c_board_info 的对应关系，还顺便体现了 10bit 地址设备与 7bit 地址设备的略微不同。通过两者的对比，可以再总结出几点区别，从而更好的理解 I2C 设备的注册方法：

- i2c_register_board_info 的形参需要的是总线号
- i2c_new_device 的形参需要的直接是适配器的指针

我想这也正好能完美的说明两者的根本区别，对于板级设备更在乎适配器的总线号，因为这是固定的，没有异议的。而对于可插拔设备，因为其适配器可能非板级集成的，所以不能在乎其总线号，反而只要寻求其适配器指针进行绑定即可。后边 adapter 注册的分析能更好的证明这一点。

- i2c_register_board_info 可以同时注册多个 I2C 设备
- i2c_new_device 只能一次注册一个 I2C 设备

这也是其根本区别决定的，板级代码中常包含有许多 I2C 设备，所以 i2c_register_board_info 需要有同时注册多个 I2C 设备的能力也可以说是刚需。而 i2c_new_device 既然是用来给可插拔设备用的，想必设备数量并不多，而常可能只是一个两个而已，所以一次一个就够了，需求量并不大。使用实例如下：

```
static struct i2c_board_info at24cxx_info = {
	I2C_BOARD_INFO("at24c08", 0x50),
};

static struct i2c_client *at24cxx_client;
static int at24cxx_dev_init(void)
{
	struct i2c_adapter *i2c_adap;

	i2c_adap = i2c_get_adapter(2);
	at24cxx_client = i2c_new_device(i2c_adap, &at24cxx_info);
	i2c_put_adapter(i2c_adap);

	return 0;
}
```

如果连 i2c 设备的地址都是不固定的，甚至在不同的板子上有不同的地址，可以提供一个地址列表供系统探测。

此时应该使用的函数是 i2c_new_probe_device.。用法如下：

```
static const unsigned short addr_list[] = { 0x50, 0x57, I2C_CLIENT_END };

static int at24cxx_dev_init(void)
{
	struct i2c_adapter *i2c_adap;
	struct i2c_board_info at24cxx_info;

	memset(&at24cxx_info, 0, sizeof(struct i2c_board_info));
	strlcpy(at24cxx_info.type, "at24c08", I2C_NAME_SIZE);

	i2c_adap = i2c_get_adapter(2);
	at24cxx_client = i2c_new_probed_device(i2c_adap, &at24cxx_info, addr_list, NULL);
	i2c_put_adapter(i2c_adap);

	if (at24cxx_client)
		return 0;
	else
		return -ENODEV;
}
```

说明：获取 i2c_adapter 指针的函数是：

```
struct i2c_adapter* i2c_get_adapter(int id)；//它的参数是 i2c 总线编号。
```

获得总线号后，要将注册的 i2c 设备挂在总线 上。

使用完要释放：

```
void i2c_put_adapter(struct i2c_adapter *adap)；
```

14.1.3.3 从用户空间初始化 I2C 设备

这种方法添加 i2c 设备又称之为“i2c 设备的实例化”。

在一般情况下，内核应该知道哪些 I2C 设备被连接以及他们的地址是什么。然而，在某些情况下，它没有，所以 sysfs 接口用作让用户提供信息。这接口是由属性组成，在每一个 I2C 总线上创建的文件目录：new_device 和 delete_device。这两个文件只写而且为了完成初始化、分别删除一个 I2C 设备你必须写正确的参数，给他们以正确的实例。

文件 new_device 需要 2 个参数：I2C 设备的名称和 I2C 器件的地址。

文件 delete_device 需要一个参数：I2C 设备地址。由于没有两个设备可以定义在同一地址上，地址是足以唯一标识一个需要删除的设备。

例如：

创建设备：

```
#echo at24c08 0x50 > /sys/bus/i2c/devices/i2c-2/new_device
```

删除设备：

```
# echo 0x50 > /sys/bus/i2c/devices/i2c-2/delete_device
```

虽然这个接口只用于当内核中的设备声明不能生效时，有各种不同的情况下它是非常有用的：

- I2C 驱动程序通常会检测设备（方法 3），但总线的设备存在不适当的位组， 因此检测不会触发。
- I2C 驱动程序通常会检测设备，但您的设备在一个未知的地址。
- I2C 驱动程序通常会检测设备，但您的设备没有被检测到。 要么因为过于严格的检测程序，或者是因为您 设备不能正常支持，但你知道它是兼容的。

- *你正在开发一个自己焊接在测试板上的 I2C 设备驱动程序。

此接口是 I2C 驱动工具 FORCE_ *模块的一些参数的替代品。在 i2c-core 中执行胜过每个设备驱动单独执行，更改设置时你不必重新加载驱动程序，这样做更有效，也更有优势。还可以在设备加载或可用之前初始化的设备的驱动程序，你不需要知道驱动设备的需求是什么。

总结：

添加 i2c 设备的方法很灵活。根据 Linux 的官方文档添加 i2c 设备的方法总结有 4 种：

1） i2c_register_board_info：根据总线编号、设备名字（“at24c08”）、设备地址（0x50）注册一个字符驱动。这种方法最简单、最粗暴，最贴近平时在开片机上开发 i2c 器件的。

2）. i2c_new_device：根据 i2c 总线的编号，声明一个 i2c 设备：这种方法就是上面例子用的方法。这种方法也简单，但是需要事先知道器件挂接在哪条总线上。对于设备，还实现知道了设备地址 0x50，总线适配器也支持名字为“at24c08”的设备

3） i2c_new_probed_device：根据地址列表通过系统探测出设备地址，这种方法最灵活通用。

4）从用户空间实例化一个器件：这个方法相当智能快速，如下输入指令，即可增加一个 i2c 设备，同时增加了对应的设备文件。

14.1.4 I2C driver

14.1.4.1 I2C 设备驱动

Linux 内核给出的接口只有两个，一个是注册，另一个就是卸载。在（一）也分析过 module_i2c_driver 这个宏定义，因为有它的存在，I2C 设备驱动的开发可以不用在意你的 I2C 驱动需要如何注册以及如何卸载的，全部的精力都放在 i2c_driver 的完善上就可以了。

通过最开始的表单能明显察觉到，I2C 子系统中 I2C driver 的开放接口最少，说白了就是需要驱动编写者完成完了 i2c_driver 放入 module_i2c_driver 宏中即可，而正因为如此，也恰恰说明，i2c_driver 的灵活性是最高的。通常驱动会首先在意在用户空间的打开、关闭、读写等接口，但是对于 i2c_driver 来说，这些工作是 I2C 子系统已经做好的，关于常用的读写最终也是通过 adapter 实现的 i2c_algorithm 达到目的。好吧，再次说明了 I2C 子系统的完善程度，对于 I2C 设备及驱动开发来说是极其方便的。那么 I2C 驱动要实现什么呢？

再次回顾一下 i2c_driver 结构体，不过现在要剔除一些不常用的成员：

```
struct i2c_driver {
	int (*probe)(struct i2c_client *, const struct i2c_device_id *); //现行通用的与对应设备进行绑定的接口函
数
	int (*remove)(struct i2c_client *);	//现行通用与对应设备进行解绑的接口函数

	void (*shutdown)(struct i2c_client *);	//关闭设备
	int (*suspend)(struct i2c_client *, pm_message_t mesg); //挂起设备，与电源管理有关，为省电
	int (*resume)(struct i2c_client *); //从挂起状态恢复

	struct device_driver driver;	//I2C 设备的驱动模型
	const struct i2c_device_id *id_table;	//匹配设备列表
	...
};
```

如果有可能的话，我还想再精简一下：

```
struct i2c_driver {
	int (*probe)(struct i2c_client *, const struct i2c_device_id *); //现行通用的与对应设备进行绑定的接口函
```

```
数
	int (*remove)(struct i2c_client *);	//现行通用与对应设备进行解绑的接口函数

	struct device_driver driver;	//I2C 设备的驱动模型
	const struct i2c_device_id *id_table;	//匹配设备列表
	...
};
```

好了，精简到这种程度，为什么把电源管理相关也干掉了呢？实际上没有，通常实际的 I2C 驱动喜欢在 drivers 中完成这个动作（以 mpu3050 为例）：

```
static UNIVERSAL_DEV_PM_OPS(mpu3050_pm, mpu3050_suspend, mpu3050_resume, NULL);

static const struct i2c_device_id mpu3050_ids[] = {
	{ "mpu3050", 0 },
	{ }
};
MODULE_DEVICE_TABLE(i2c, mpu3050_ids);

static const struct of_device_id mpu3050_of_match[] = {
	{ .compatible = "invn,mpu3050", },
	{ },
};
MODULE_DEVICE_TABLE(of, mpu3050_of_match);

static struct i2c_driver mpu3050_i2c_driver = {
	.driver	= {
		.name	= "mpu3050",
		.owner	= THIS_MODULE,
		.pm	= &mpu3050_pm,
		.of_match_table = mpu3050_of_match,
	},
	.probe		= mpu3050_probe,
	.remove		= mpu3050_remove,
	.id_table	= mpu3050_ids,
};

module_i2c_driver(mpu3050_i2c_driver);
```

可以看到，实际驱动中喜欢将电源管理集成在 i2c_driver 的 driver 成员中。

UNIVERSAL_DEV_PM_OPS 这个名字很犀利，貌似是“宇宙终极驱动电源管理大法”的样子：

```
#define UNIVERSAL_DEV_PM_OPS(name, suspend_fn, resume_fn, idle_fn) \
const struct dev_pm_ops name = { \
	SET_SYSTEM_SLEEP_PM_OPS(suspend_fn, resume_fn) \
	SET_RUNTIME_PM_OPS(suspend_fn, resume_fn, idle_fn) \
}

#define SET_SYSTEM_SLEEP_PM_OPS(suspend_fn, resume_fn) \
	.suspend = suspend_fn, \
	.resume = resume_fn, \
	.freeze = suspend_fn, \
	.thaw = resume_fn, \
	.poweroff = suspend_fn, \
	.restore = resume_fn,

#define SET_RUNTIME_PM_OPS(suspend_fn, resume_fn, idle_fn) \
	.runtime_suspend = suspend_fn, \
	.runtime_resume = resume_fn, \
	.runtime_idle = idle_fn,
```

结合 MPU3050 的驱动将其完整展开可以得到：

```
static const struct dev_pm_ops mpu3050_pm = {
	.suspend = mpu3050_suspend,
	.resume = mpu3050_resume,
	.freeze = mpu3050_suspend,
	.thaw = mpu3050_resume,
```

```
        .poweroff = mpu3050_suspend,
        .restore = mpu3050_resume,
        .runtime_suspend = mpu3050_suspend,
        .runtime_resume = mpu3050_resume,
        .runtime_idle = NULL,
}
```

对电源管理有兴趣的可以去查阅 pm.h，其中对电源管理有详尽的说明了，这里不做分析。可以看到，在电源管理中，有很多成员实际上是一样的，在现实驱动中这样的情况也经常出现，所以会有“终极电源管理大法”宏的出现了。

上边说过，i2c_driver 的多样化最多，从 mpu3050 的驱动注册中也可以发现，其注重实现的为 probe 与电源管理，其中 probe 最为重要（好像是废话，哪个驱动这个都是最重要的-。-）。因为主要是从驱动的角度看待 I2C 子系统，所以这里不详尽分析 mpu3050 的代码，只以其为例说明 I2C 驱动大体框架。在 mpu3050 的 probe 主要对此传感器进行上电、工作模式初始化、注册 INPUT 子系统接口、关联中断处理程序（在中断处理线程中上报三轴参数）等工作。

14.1.4.2 关于 I2C 设备驱动的小总结

I2C 设备驱动通常只是需要挂载在 I2C 总线（即依附于 I2C 子系统），I2C 子系统对于设备驱动来说只是一个载体、基石。许多设备的主要核心是建立在其他子系统上，如重力传感器、三轴传感器、触摸屏等通常主要工作集中在 INPUT 子系统中，而相机模块、FM 模块、GPS 模块大多主要依附于 V4L2 子系统。这也能通过 I2C 设计理念证明，I2C 的产生正是为了节省外围电路复杂度，让 CPU 使用有限的 IO 口挂载更多的外部模块。假设 CPU 的扩展 IO 口足够多，我想 I2C 也没什么必要存在了，毕竟直接操作 IO 口驱动设备比 I2C 来的更简单。

14.1.5 I2C adapter 的注册

14.1.5.1 注册方法

如上表所示，对于 I2C adapter 的注册有两种途径：i2c_add_adapter 或 i2c_add_numbered_adapter，两者的区别是后者在注册时已经指定了此 I2C 适配器的总线号，而前者的总线号将由系统自动分配。

其各自的声明格式为：

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
```

在 i2c_add_numberd_adapter 使用前必须制定 adap->nr，如果给-1，说明还是叫系统去自动生成总线号的。

14.1.5.2 使用场景

之所以区分开两种 I2C adapter 的注册方式，是因为他们的使用场景有所不同。

- i2c_add_adapter 的使用经常是用来注册那些可插拔设备，如 USB PCI 设备等。主板上的其他模块与其没有直接联系，说白了就是现有模块不在乎新加入的 I2C 适配器的总线号是多少，因为他们不需要。反而这个可

插拔设备上的一些模块会需要其注册成功的适配器指针。回看一开始就分析的 i2c_client，会发现不同场景的设备与其匹配的适配器有着这样的对应关系：

1. i2c_register_board_info 需要指定已有的 busnum，而 i2c_add_numbered_adapter 注册前已经指定总线号；

2. i2c_new_device 需要指定 adapter 指针，而 i2c_add_adapter 注册成功后恰好这个指针就有了。

- i2c_add_numbered_adapter 用来注册 CPU 自带的 I2C 适配器，或是集成在主板上的 I2C 适配器。主板上的其他 I2C 从设备(client)在注册时候需要这个总线号。

通过简短的代码分析看一看他们的区别究竟如何，以及为什么静态注册的 i2c_client 必须要在 adapter 注册前（此处会精简部分代码，只留重要部分）：

```
int i2c_add_adapter(struct i2c_adapter *adapter)
{
	int	id, res = 0;
	res = idr_get_new_above(&i2c_adapter_idr, adapter,
				__i2c_first_dynamic_bus_num, &id);	//动态获取总线号
	adapter->nr = id;
	return i2c_register_adapter(adapter);	//注册 adapter
}

int i2c_add_numbered_adapter(struct i2c_adapter *adap)
{
	int	id;
	int	status;
	if (adap->nr == -1) /* -1 means dynamically assign bus id */
		return i2c_add_adapter(adap);
	status = i2c_register_adapter(adap);
	return status;
}
```

最终他们都是通过 i2c_register_adapter 注册适配器：

```
static int i2c_register_adapter(struct i2c_adapter *adap)
{
	int res = 0;

	/* Can't register until after driver model init */	//时序检查
	if (unlikely(WARN_ON(!i2c_bus_type.p))) {
		res = -EAGAIN;
		goto out_list;
	}

	/* Sanity checks */
	if (unlikely(adap->name[0] == '\0')) {	//防御型代码，检查适配器名称
		pr_err("i2c-core: Attempt to register an adapter with "
		       "no name!\n");
		return -EINVAL;
	}
	if (unlikely(!adap->algo)) {	//适配器是否已经完成了通信方法的实现
		pr_err("i2c-core: Attempt to register adapter '%s' with "
		       "no algo!\n", adap->name);
		return -EINVAL;
	}

	rt_mutex_init(&adap->bus_lock);
	mutex_init(&adap->userspace_clients_lock);
	INIT_LIST_HEAD(&adap->userspace_clients);

	/* Set default timeout to 1 second if not already set */
	if (adap->timeout == 0)
		adap->timeout = HZ;

	dev_set_name(&adap->dev, "i2c-%d", adap->nr);
	adap->dev.bus = &i2c_bus_type;
```

```
	adap->dev.type = &i2c_adapter_type;
	res = device_register(&adap->dev);	//注册设备节点
	if (res)
		goto out_list;

	/* create pre-declared device nodes */ //创建预-声明的 I2C 设备节点
	if (adap->nr < __i2c_first_dynamic_bus_num)
		i2c_scan_static_board_info(adap);
		//如果 adapter 的总线号小于动态分配的总线号的最小那个，说明是板级 adapter。
		//因为通过 i2c_add_adapter 加入的适配器所分配的总线号一定是比__i2c_first_dynamic_bus_num 大的。
	...
}
```

对于 i2c_add_numbered_adapter 来说会触发 i2c_scan_static_board_info：

```
static void i2c_scan_static_board_info(struct i2c_adapter *adapter)
{
	struct i2c_devinfo	*devinfo;

	down_read(&__i2c_board_lock);	//持有读写锁的读，有用户读的时候不允许写入
	list_for_each_entry(devinfo, &__i2c_board_list, list) {	// 又见 __i2c_board_list ， 这 不 是 通 过 i2c_register_board_info 组建起来的那个链表吗！
		if (devinfo->busnum == adapter->nr
				&& !i2c_new_device(adapter,
						&devinfo->board_info))	//找到总线号与刚注册的这个 adapter 相同的并通过 i2c_new_device 进行注册
			dev_err(&adapter->dev,
				"Can't create device at 0x%02x\n",
				devinfo->board_info.addr);
	}
	up_read(&__i2c_board_lock);	//释放读写锁
}
```

而 i2c_board_info 成员与 i2c_client 的对应动作也是在 i2c_new_device 中进行的，这一点在上边已经分析过了。看到这里，对 adapter 与 client 的微妙关系应该了解程度就比较深了，为什么说 i2c_register_board_info 与 i2c_add_numbered_adapter 对应而不是 i2c_add_adapter 也可以说得通。

那么，最终回答开篇提出的那两个问题：

- i2c_adapter 驱动如何添加？

板级适配器（CPU 自带、主板集成）要通过 i2c_add_numbered_adapter 注册，注册前要指定总线号，从 0 开始。假如板级 I2C 适配器注册了 3 个，那么第一个动态总线号一定是 3，也就是说可插拔设备所带有的 I2C 适配器需要通过 i2c_add_adapter 进行注册，其总线号由系统指定。

- i2c_client 与 i2c_board_info 究竟是什么关系？

i2c_client 与 i2c_board_info 的对应关系在 i2c_new_device 中有完整体现。

```
i2c_client->dev.platform_data = i2c_board_info->platform_data;
i2c_client->dev.archdata = i2c_board_info->archdata;
i2c_client->flags = i2c_board_info->flags;
i2c_client->addr = i2c_board_info->addr;
i2c_client->irq = i2c_board_info->irq;
```

14.1.5.3 物理 i2c 总线的编号查询

```
[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-1/name
loongson1
[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-0/name
loongson1
[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-2/name
loongson1
```

14.1.6 I2C tools 使用

14.1.6.1 下载安装

下载网址：http://packages.debian.org/search?keywords=i2c-tools

在虚拟机中解压源码包：

```
tar -xvf i2c-tools-3.1.0.tar.bz2
```

解压后，修改 Makefile 文件中编译工具链后编译：

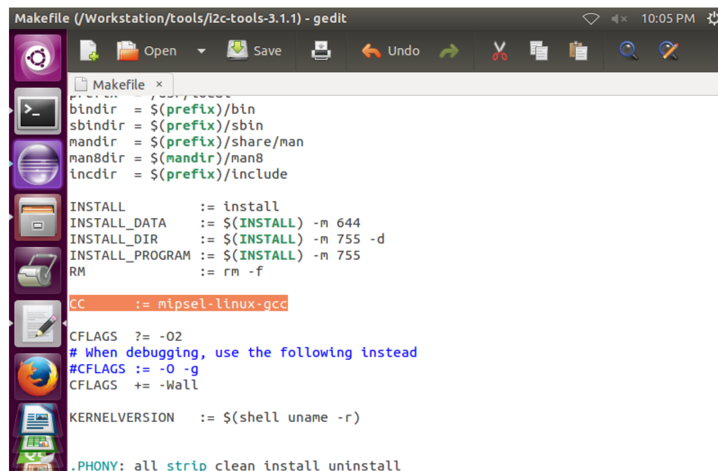


图 14.4 修改 I2Ctools 工具包中 Makefile 文件中编译工具链

最后在 tools 下生成 i2cdetect i2cdump i2cset i2cget：

```
root@ubuntu:/Workstation/tools/i2c-tools-3.1.1# make
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cdetect.c -o tools/i2cdetect.o
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cbusses.c -o tools/i2cbusses.o
mipsel-linux-gcc -o tools/i2cdetect tools/i2cdetect.o tools/i2cbusses.o
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cdump.c -o tools/i2cdump.o
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/util.c -o tools/util.o
mipsel-linux-gcc -o tools/i2cdump tools/i2cdump.o tools/i2cbusses.o tools/util.o
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cset.c -o tools/i2cset.o
mipsel-linux-gcc -o tools/i2cset tools/i2cset.o tools/i2cbusses.o tools/util.o
mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cget.c -o tools/i2cget.o
mipsel-linux-gcc -o tools/i2cget tools/i2cget.o tools/i2cbusses.o tools/util.o
root@ubuntu:/Workstation/tools/i2c-tools-3.1.1#
```

14.1.6.2 I2C 总线扫描

将《14.1.6.1》编译好的 i2cdetect , i2cdump , i2cget , i2cset 拷备到开发板的新建的一个目录下：

```
[root@Loongson:/i2ctools]#ls
i2cdetect    i2cdump     i2cget      i2cset
```

通过 i2cdetect -l 指令可以查看设备上的 I2C 总线。

```
[root@Loongson:/i2ctools]#./i2cdetect -l
i2c-0    i2c          i2c-ls1x                              I2C adapter
i2c-1    i2c          i2c-ls1x                              I2C adapter
i2c-2    i2c          i2c-ls1x                              I2C adapter
```

14.1.6.3 I2C 设备查询

将设备的 SDA 连接到 GPIO50，SCL 连接到 GPIO51，若总线上挂载 I2C 从设备，可通过 i2cdetect 扫描某个 I2C 总线上的所有设备。可通过控制台输入 i2cdetect -y 2，结果如下所示。

```
[root@Loongson:/i2ctools]#./i2cdetect -y 2
   0 1 2 3 4 5 6 7 8 9 a b c d e f
00:        -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

说明 1：-y 为一个可选参数，如果有-y 参数的存在则会有一个用户交互过程 z，意思是希望用户停止使用该 I2C 总线。如果写入该参数，则没有这个交互过程，一般该参数在脚本中使用。

说明 2：此处 I2C 总线共挂载两个设备——DS3231 和 AT24C02，从机地址 0x68 为 DS3231，从机地址 0x50 为 AT24C02。

14.1.6.4 寄存器内容导出

通过 i2cdump 指令可导出 I2C 设备中的所有寄存器内容，例如输入 i2cdump -y 1 0x68，可获得以下内容:

```
[root@Loongson:/i2ctools]#./i2cdump -y 2 0x68
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 21 45 12 02 07 04 17 00 00 00 00 00 00 00 1c 88    !E?????.......??
10: 00 1c 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .?..............
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

14.1.6.5 寄存器内容写入

如果向 I2C 设备中写入某字节，可输入指令/i2cset -y 2 0x50 0x01 0x15

-y　代表曲线用户交互过程，直接执行指令

2　代表 I2C 总线编号

0x50　代表 I2C 设备地址，此处选择 AT24C02

0x01　代表存储器地址

0x15　代表存储器地址中的具体内容

```
[root@Loongson:/i2ctools]#./i2cset -y 2 0x50 0x01 0x15
[root@Loongson:/i2ctools]#./i2cdump -y 2 0x50
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: ff 15 ff ff ff ff ff ff ff ff ff ff ff ff ff ff    .?..............
10: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
20: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
30: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
40: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
50: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
60: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
70: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
90: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
b0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
d0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
f0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
```

14.1.6.6 寄存器内容读出

如果从 I2C 从设备中读出某字节，可输入执行 i2cget -y 1 0x50 0x00，可得到以下反馈结果

-y　代表曲线用户交互过程，直接执行指令

1　代表 I2C 总线编号

0x50　代表 I2C 设备地址，此处选择 AT24C04 的低 256 字节内容

0x00　代表存储器地址

```
[root@Loongson:/i2ctools]#./i2cget -y 2 0x50 0x01
0x15
```

i2ctools 是一个简单好用的工具，该工具使得 I2C 设备的调试更加方便。

14.1.7 内核模块分析

内核中配置相关 I2C 的选项已经打开如下：

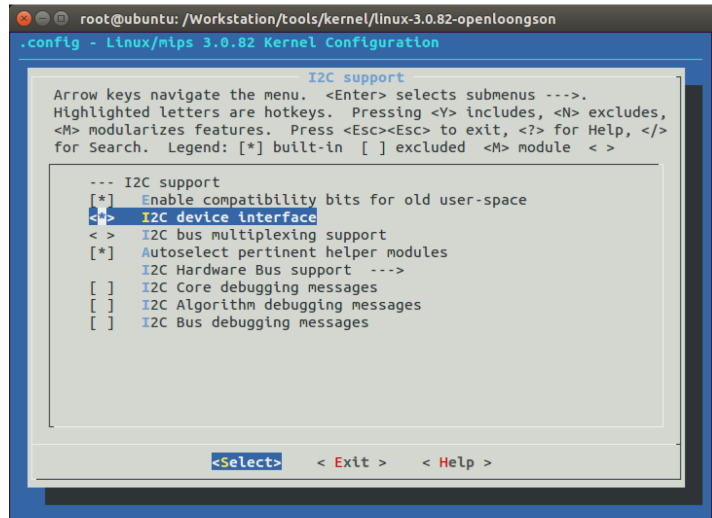


图 14.5 内核中配置相关 I2C 的选项 1

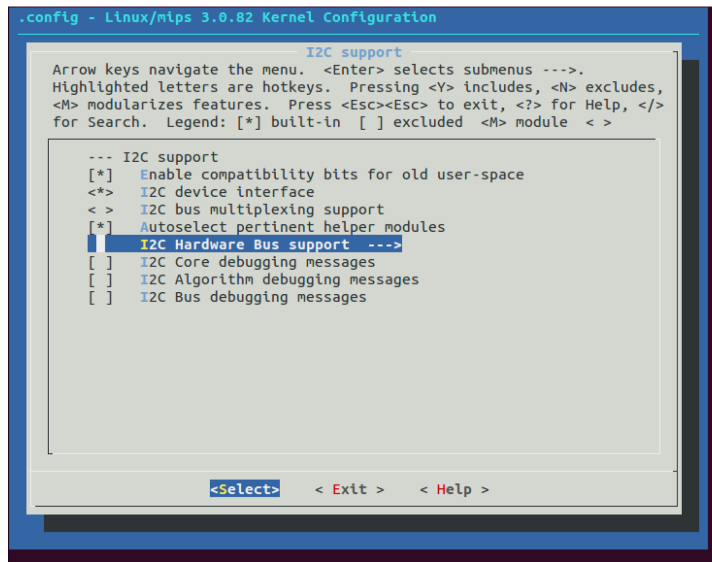


图 14.6 内核中配置相关 I2C 的选项 2

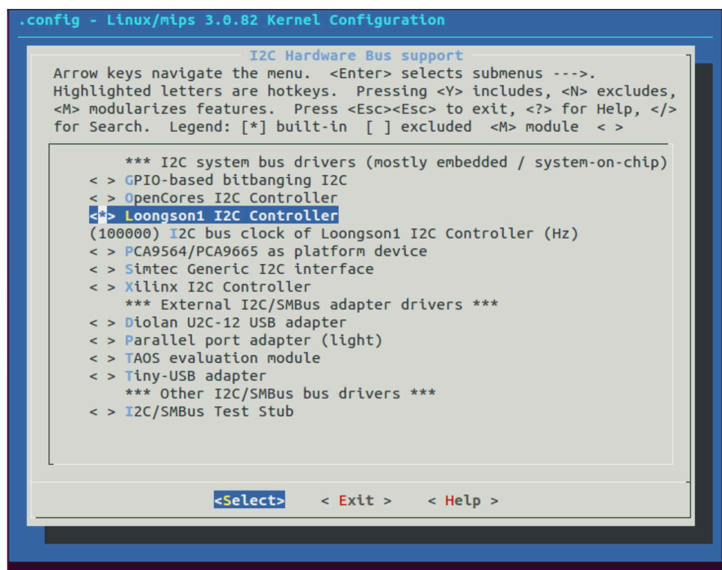


图 14.7 内核中配置相关 I2C 的选项 3

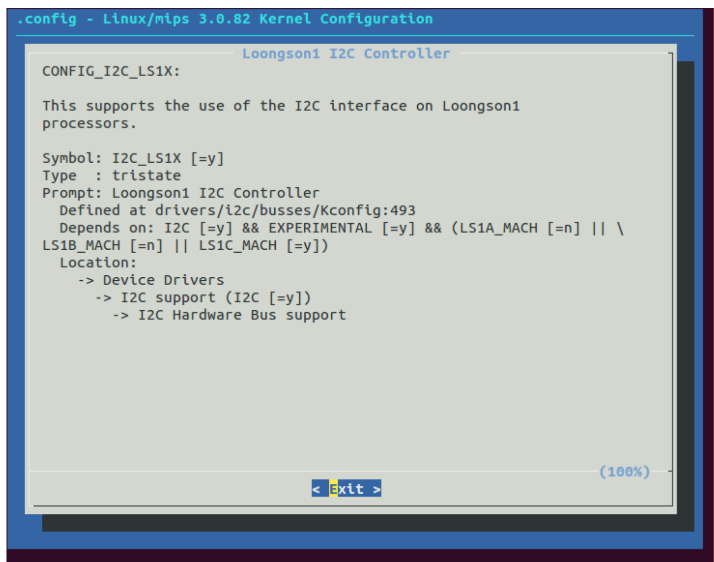


图 14.8 内核中配置相关 I2C 的选项 4

以上编译总共得到了 3 个驱动文件 i2c-core.ko，i2c-dev.ko，i2c_LS1X.ko, 这三个模块已经编译到内核中，会自动创建/dev/i2c-0 设备节点，然后就可以直接调用/dev/i2c-0 文件节点进行访问设备了。之后应用程序员可以利用下面两种 ioctl 函数 ioctl(fd, I2C_RDWR, (unsigned long)&work_queue);或者 octl(file,I2C_SMBUS,&args);进行与 i2c 设备通信了。

一句话概括之：platform 设备是针对硬件资源的分配，platform 驱动是软件对已分配的硬件资源的使用。

14.2 实例分析 at24cxx

硬件连接：使用 I2C2 ，

11	CAMDATA1	109	51		LCD_B1	SPI1_CS2	I2C_SCL2	UART10_RX/UART8_DSR	
12	CAMDATA0	110	50		LCD_B0	SPI1_CS1	I2C_SDA2	UART10_TX/UART8_DTR	

图 14.9 开发板上 I2C 管脚复用

引脚 GPIO85 86 I2C0，用在了按键上：

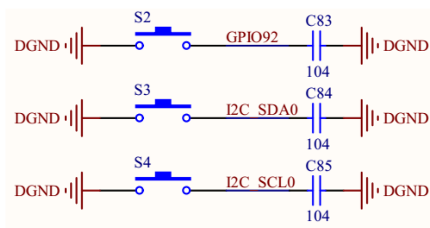


图 14.10 开发板上按键管脚

引脚 GPIO55,54 I2C1，用在了 CAN 上

15	CAMDATA5	105	55		LCD_R0	CAN0_TX	I2C_SCL1	UART8_RX	
16	CAMDATA4	106	54		LCD_G1	CAN0_RX	I2C_SDA1	UART8_TX	

图 14.11 开发板上 CAN 总线管脚复用

步骤如下：

1、注册一个设备，里面要有设备名，以及该设备 id，并挂载到这条 i2c 总线设备链表上

2、注册一个驱动，该设备驱动需要有设备名，probe 函数、id_table 设备地址，然后挂载到驱动链表上

3、比较两条链表上的设备名字是否相同，如果有相同的话，就去到相关的驱动设备上的 probe 函数，进行操作

4、probe 函数里面就是正常的驱动程序的编程程序了，主入口、出口、file_operation 结构体的构造等等操作；最终应用程序会进入这里操作硬件对象

14.2.1 注册新设备

文件 at24cxx_dev.c:注册新设备， 采用方法 i2c_new _device。

文件 at24cxx_dev1.c:注册新设备， 采用方法 i2c_new_probed_device。

14.2.2 注册新驱动

at24cxx_drv.c: 注册一个新驱动

14.2.3 对 i2c 驱动的操作

完成设备的打开、读、写、关闭等。填充 at24cxx_drv.c 中的函数。

14.2.4 编译用的 Makefile

```
KERN_DIR = /Workstation/tools/kernel/linux-3.0.82-openloongson

all:
	make -C $(KERN_DIR) M=`pwd` modules ARCH=mips CROSS_COMPILE=mipsel-linux-

clean:
	make -C $(KERN_DIR) M=`pwd` modules clean
	rm -rf modules.order

#obj-m	+= at24cxx_dev.o
#obj-m	+= at24cxx_dev1.o
obj-m	+= at24cxx_drv.o
```

14.2.5 测试应用编程 test_at24cxx.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


/* i2c_test r addr
 * i2c_test w addr val
 */

void print_usage(char *file)
{
	printf("%s r addr\n", file);
	printf("%s w addr val\n", file);
}

int main(int argc, char **argv)
```

```
{
	int fd;
	unsigned char buf[2];

	if ((argc != 3) && (argc != 4))
	{
		print_usage(argv[0]);
		return -1;
	}

	fd = open("/dev/at24cxx", O_RDWR);
	if (fd < 0)
	{
		printf("can't open /dev/at24cxx\n");
		return -1;
	}

	if (strcmp(argv[1], "r") == 0)
	{
		buf[0] = strtoul(argv[2], NULL, 0);
		read(fd, buf, 1);
		printf("data: %c, %d, 0x%2x\n", buf[0], buf[0], buf[0]);
	}
	else if ((strcmp(argv[1], "w") == 0) && (argc == 4))
	{
		buf[0] = strtoul(argv[2], NULL, 0);
		buf[1] = strtoul(argv[3], NULL, 0);
		if (write(fd, buf, 2) != 2)
			printf("write err, addr = 0x%02x, data = 0x%02x\n", buf[0], buf[1]);
	}
	else
	{
		print_usage(argv[0]);
		return -1;
	}

	return 0;
}
```

14.3 实例分析 DS3231

参考从长计议的《智龙 V2 通过 I2C 连接 DS3231 时钟模块》

http://www.openloongson.org/forum.php?mod=viewthread&tid=149&extra=page%3D1

作了改动，将静态注册修改成了动态注册，这样就不需要重新编译内核，原来是：

```
/**
在 platform.c 中入加

ls1x_i2c0_devs 加入成员

{	I2C_BOARD_INFO("ds3231", 0x68),},



*/
```

修改成：

```
static struct i2c_board_info ds3231_info = {
	I2C_BOARD_INFO("ds3231", 0x68),
};
。。。。。。
。。。。。。
static int __init ds3231_drv_init(void)
{
```

```
    。。。。。。
    ds3231_client = i2c_new_device(i2c_adap, &ds3231_info);
    i2c_put_adapter(i2c_adap);
     。。。。。。
}
```

15. SPI 总线和设备驱动架构

本节来自于网址：http://blog.csdn.net/DroidPhone/article/details/23367051。

15.1 SPI 概述

SPI 是"Serial Peripheral Interface" 的缩写，是一种四线制的同步串行通信接口，用来连接微控制器、传感器、存储设备，SPI 设备分为主设备和从设备两种，用于通信和控制的四根线分别是：

CS　片选信号

SCK　时钟信号

MISO　主设备的数据输入、从设备的数据输出脚

MOSI　主设备的数据输出、从设备的数据输入脚

因为在大多数情况下，CPU 或 SOC 一侧通常都是工作在主设备模式，所以，目前的 Linux 内核版本中，只实现了主模式的驱动框架。

15.1.1 硬件结构

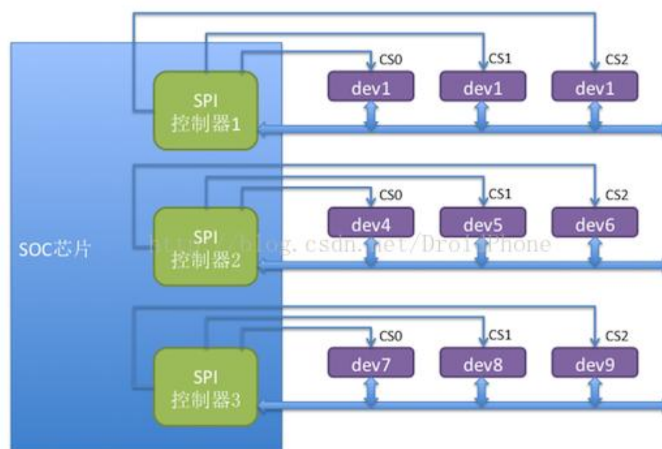


图 15.1 SPI 总线硬件结构图

主设备对应 SOC 芯片中的 SPI 控制器，通常，一个 SOC 中可能存在多个 SPI 控制器，像上面的例子所示，SOC 芯片中有 3 个 SPI 控制器。每个控制器下可以连接多个 SPI 从设备，每个从设备有各自独立的 CS 引脚。每个从设备共享另外 3 个信号引脚：SCK、MISO、MOSI。任何时刻，只有一个 CS 引脚处于有效状态，与该有效 CS 引脚连接的设备此时可以与主设备（SPI 控制器）通信，其它的从设备处于等待状态，并且它们的 3 个引脚必须处于高阻状态。

通常，负责发出时钟信号的设备我们称之为主设备，另一方则作为从设备.

15.1.1.1 工作时序

按照时钟信号和数据信号之间的相位关系，SPI 有 4 种工作时序模式：

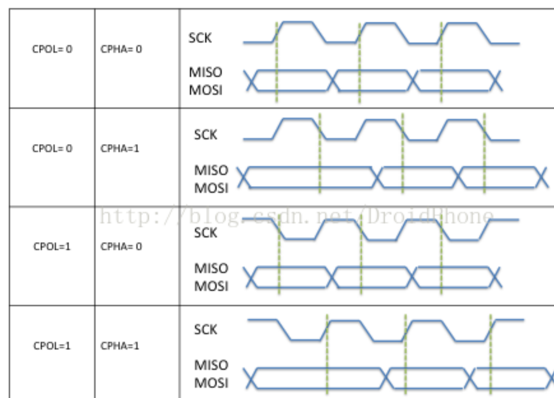


图 15.2 SPI 总线 4 种工作时序模式

用 CPOL 表示时钟信号的初始电平的状态，CPOL 为 0 表示时钟信号初始状态为低电平，为 1 表示时钟信号的初始电平是高电平。另外，我们用 CPHA 来表示在那个时钟沿采样数据，CPHA 为 0 表示在首个时钟变化沿采样数据，而 CPHA 为 1 则表示要在第二个时钟变化沿来采样数据。内核用 CPOL 和 CPHA 的组合来表示当前 SPI 需要的工作模式：

- CPOL＝0，CPHA＝1　　模式 0
- CPOL＝0，CPHA＝1　　模式 1
- CPOL＝1，CPHA＝0　　模式 2
- CPOL＝1，CPHA＝1　　模式 3

15.1.2 软件架构

在内核的 SPI 驱动的软件架构中，进行了合理的分层和抽象，如图 15.3 所示：

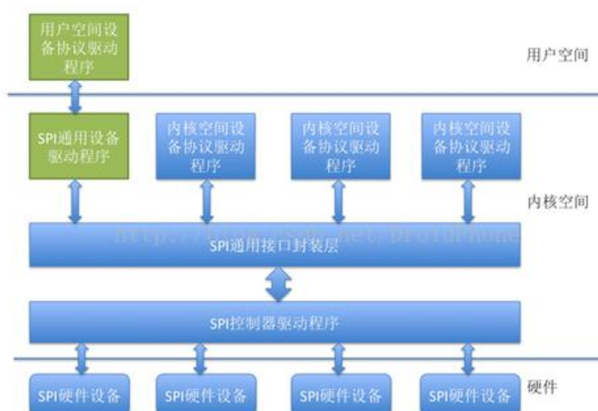


图 15.3 内核 SPI 驱动软件架构图

5.1.2.1 SPI 控制器驱动程序

SPI 控制器不用关心设备的具体功能，它只负责把上层协议驱动准备好的数据按 SPI 总线的时序要求发送给 SPI 设备，同时把从设备收到的数据返回给上层的协议驱动，因此，内核把 SPI 控制器的驱动程序独立出来。SPI 控制器驱动负责控制具体的控制器硬件，诸如 DMA 和中断操作等等，因为多个上层的协议驱动可能会通过控制器请求数据传输操作，所以，SPI 控制器驱动同时也要负责对这些请求进行队列管理，保证先进先出的原则。

5.1.2.2 SPI 通用接口封装层

为了简化 SPI 驱动程序的编程工作，同时也为了降低协议驱动程序和控制器驱动程序的耦合程度，内核把控制器驱动和协议驱动的一些通用操作封装成标准的接口，加上一些通用的逻辑处理操作，组成了 SPI 通用接口封装层。这样的好处是，对于控制器驱动程序，只要实现标准的接口回调 API，并把它注册到通用接口层即可，无需直接和协议层驱动程序进行交互。而对于协议层驱动来说，只需通过通用接口层提供的 API 即可完成设备和驱动的注册，并通过通用接口层的 API 完成数据的传输，无需关注 SPI 控制器驱动的实现细节。

5.1.2.3 SPI 协议驱动程序

上面我们提到，控制器驱动程序并不清楚和关注设备的具体功能，SPI 设备的具体功能是由 SPI 协议驱动程序完成的，SPI 协议驱动程序了解设备的功能和通信数据的协议格式。向下，协议驱动通过通用接口层和控制器交换数据，向上，协议驱动通常会根据设备具体的功能和内核的其它子系统进行交互，例如，和 MTD 层交互以便把 SPI 接口的存储设备实现为某个文件系统，和 TTY 子系统交互把 SPI 设备实现为一个 TTY 设备，和网络子系统交互以便把一个 SPI 设备实现为一个网络设备，等等。当然，如果是一个专有的 SPI 设备，我们也可以按设备的协议要求，实现自己的专有协议驱动。

5.1.2.4 SPI 通用设备驱动程序

有时候，考虑到连接在 SPI 控制器上的设备的可变性，在内核没有配备相应的协议驱动程序，对于这种情况，内核为我们准备了通用的 SPI 设备驱动程序，该通用设备驱动程序向用户空间提供了控制 SPI 控制的控制接口，具体的协议控制和数据传输工作交由用户空间根据具体的设备来完成，在这种方式中，只能采用同步的方式和 SPI 设备进行通信，所以通常用于一些数据量较少的简单 SPI 设备。

15.2 SPI 通用接口层

SPI 通用接口层用于把具体 SPI 设备的协议驱动和 SPI 控制器驱动联接在一起，通用接口层除了为协议驱动和控制器驱动提供一系列的标准接口 API，同时还为这些接口 API 定义了相应的数据结构，这些数据结构一部分是 SPI 设备、SPI 协议驱动和 SPI 控制器的数据抽象，一部分是为了协助数据传输而定义的数据结构。另外，通用接口层还负责 SPI 系统与 Linux 设备模型相关的初始化工作。本章的我们就通过这些数据结构和 API 的讨论来对整个通用接口层进行深入的了解。

SPI 通用接口层的代码集中在：/drivers/spi/spi.c 中。

15.2.1 SPI 设备模型的初始化

通常地，根据 linux 设备模型的组织方式，各种设备会挂在合适的总线上，设备驱动和设备通过总线互相进行匹配，使得设备能够找到正确的驱动程序进行控制和驱动。同时，性质相似的设备可以归为某一个类的设备，它们具有某些共同的设备属性，在设备模型上就是所谓的 class。SPI 设备也不例外，它们也遵循 linux 的设备模型的规则：

```
struct bus_type spi_bus_type = {
        .name           = "spi",
        .dev_attrs      = spi_dev_attrs,
        .match          = spi_match_device,
        .uevent         = spi_uevent,
        .pm             = &spi_pm,
```

```
};

static struct class spi_master_class = {
	.name		= "spi_master",
	.owner		= THIS_MODULE,
	.dev_release	= spi_master_release,
};

static int __init spi_init(void)
{
	int	status;

	buf = kmalloc(SPI_BUFSIZ, GFP_KERNEL);
	......
	status = bus_register(&spi_bus_type);
	......
	status = class_register(&spi_master_class);
	......
	return 0;
	......
}


postcore_initcall(spi_init);
```

可见，在初始化阶段，spi_init 函数向系统注册了一个名为 spi 的总线类型，同时也为 SPI 控制器注册了一个名为 spi_master 的设备类。这样，以后在 sysfs 中就会出现以下两个文件节点：

```
sys/bus/spi
sys/class/spi_master
```

代表 spi 总线的 spi_bus_type 结构的 match 字段指向了 spi_match_device 函数，该函数用于匹配 spi 总线上的设备和驱动，具体的代码这里就不贴了，各位可以自行查看内核的代码树。

15.2.2 spi_master 结构

SPI 控制器负责按照设定的物理信号格式在主控和 spi 设备之间交换数据，SPI 控制器数据是如何被传输的，而不关心数据的内容。SPI 通用接口层用 spi_master 结构来表示一个 spi 控制器，它的主要字段的意义如表 15.1 所示。

表 15.1 spi_master 结构体字段意义

字段名称	描述
struct device　dev	spi 控制器对应的 device 结构
struct list_head list	系统可能有多个控制器，用该链表链接在一个全局链表变量上
s16　bus_num	该控制器对应的 spi 总线编号，从 0 开始，通常由板级代码设定
u16　num_chipselect	连接到该 spi 控制器的片选信号的个数
u16　mode_bits	工作模式，由驱动解释该模式的意义
u32　min_speed_hz	最低工作时钟
u32　max_speed_hz	最高工作时钟
u16　flags	用于设定某些限制条件的标志位
int　(*setup)(struct spi_device *spi)	回调函数，用于设置某个 spi 设备在该控制器上的工作参数
int　(*transfer)(......）	回调函数，用于把包含数据信息的 mesg 结构加入控制器的消息链表中

void (*cleanup)(struct spi_device *spi)	回调函数，当 spi_master 被释放时，该函数被调用
struct kthread_worker kworker	用于管理数据传输消息队列的工作队列线程
struct kthread_work pump_messages	具体实现数据传输队列的工作队列
struct list_head queue	该控制器的消息队列，所有等待传输的消息队列挂在该链表下
struct spi_message *cur_msg	当前正带处理的消息队列
int (*prepare_transfer_hardware)(......)	回调函数，正式发起传送前会被调用，用于准备硬件资源
int (*transfer_one_message)(......)	单个消息的原子传送回调函数，队列中的每个消息都会调用一次该回调来完成传输工作
int (*unprepare_transfer_hardware)(......)	清理回调函数
int *cs_gpios	片选信号所用到的 gpio

spi_master 结构通常由控制器驱动定义，然后通过以下通用接口层的 API 注册到系统中：

```
int spi_register_master(struct spi_master *master);
```

15.2.3 spi_device 结构

SPI 通用接口层用 spi_device 结构来表示一个 spi 设备，它的各个字段的意义如表 15.2 所示。

表 15.2 spi_device 结构体字段意义

struct device dev	代表该 spi 设备的 device 结构
struct spi_master *master	指向该 spi 设备所使用的控制器
u32 max_speed_hz	该设备的最大工作时钟频率
u8 chip_select	在控制器中的片选引脚编号索引
u16 mode	设备的工作模式，包括时钟格式，片选信号的有效电平等等
u8 bits_per_word	设备每个单位数据所需要的比特数
int irq	设备使用的 irq 编号
char modalias[SPI_NAME_SIZE]	该设备的名字，用于 spi 总线和驱动进行配对
int cs_gpio	片选信号的 gpio 编号，通常不用我们自己设置，接口层会根据上面的 chip_select 字段在 spi_master 结构中进行查找并赋值

要完成向系统增加并注册一个 SPI 设备，我们还需要另一个数据结构：

```
struct spi_board_info {
        char            modalias[SPI_NAME_SIZE];
        const void      *platform_data;
        void            *controller_data;
        int             irq;
        u32             max_speed_hz;
        u16             bus_num;
        u16             chip_select;
        u16             mode;
};
```

spi_board_info 结构大部分字段和 spi_device 结构相对应，bus_num 字段则用来指定所属的控制器编号，通过 spi_board_info 结构，我们可以有两种方式向系统增加 spi 设备。第一种方式是在 SPI 控制器驱动已经被加载后，我们使用通用接口层提供的如下 API 来完成：

```
struct spi_device *spi_new_device(struct spi_master *master, struct spi_board_info *chip);
```

第二种方式是在板子的初始化代码中，定义一个 spi_board_info 数组，然后通过以下 API

注册 spi_board_info：

```
int spi_register_board_info(struct spi_board_info const *info, unsigned n);
```

上面这个 API 会把每个 spi_board_info 挂在全局链表变量 board_list 上，并且遍历已经在系统中注册了的控制器，匹配上相应的控制器并取得它们的 spi_master 结构指针，最终也会通过 spi_new_device 函数添加 SPI 设备。因为 spi_register_board_info 可以在板子的初始化代码中调用，可能这时控制器驱动尚未加载，此刻无法取得相应的 spi_master 指针，不过不要担心，控制器驱动被加载时，一定会调用 spi_register_master 函数来注册 spi_master 结构，而 spi_register_master 函数会反过来遍历全局链表 board_list 上的 spi_board_info，然后通过 spi_new_device 函数添加 SPI 设备。

```
int __init

spi_register_board_info(struct spi_board_info const *info, unsigned n)
{
   struct boardinfo *bi;
   int i;

   bi = kzalloc(n * sizeof(*bi), GFP_KERNEL);
   if (!bi)
      return -ENOMEM;

   for (i = 0; i < n; i++, bi++, info++) {
      struct spi_master *master;

      memcpy(&bi->board_info, info, sizeof(*info));
      mutex_lock(&board_lock);
      list_add_tail(&bi->list, &board_list);
      list_for_each_entry(master, &spi_master_list, list)
         spi_match_master_to_boardinfo(master, &bi->board_info);
      mutex_unlock(&board_lock);
   }

   return 0;
}
```

对于此处，n 为 1，在程序中首先创建相应的内存，在 for 循环中，将信息保存到内存中，然后插入 board_list 链表，接着遍历 spi_master_list 链表，注意此处，由于 device_initcall 的优先级高于 module_init，所以此时 spi_master_list 链表为空，那么还不能调用 spi_match_master_to_boardinfo 函数创建 spi 设备，具体的创建设备将在 spi 总线驱动的探测函数中，使用 spi_register_master()函数创建设备。

15.2.4 spi_driver 结构

根据 linux 的设备模型，有 device 就必定有 driver 与之对应，上一节介绍的 spi_device 结构中内嵌了 device 结构字段 dev，同样地，代表驱动程序的 spi_driver 结构也内嵌了 device_driver 结构：

```
struct spi_driver {
      const struct spi_device_id *id_table;
      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;
};
```

id_table 字段用于指定该驱动可以驱动的设备名称，总线的匹配函数会把 id_table 中指定的名字和 spi_device 结构中的 modalias 字段进行比较，两者相符即表示匹配成功，然后出发 spi_driver 的 probe 回调函数被调用，从而完成驱动程序的初始化工作。通用接口层提供

如下 API 来完成 spi_driver 的注册：

```
int spi_register_driver(struct spi_driver *sdrv)
{
	sdrv->driver.bus = &spi_bus_type;
	if (sdrv->probe)
		sdrv->driver.probe = spi_drv_probe;
	if (sdrv->remove)
		sdrv->driver.remove = spi_drv_remove;
	if (sdrv->shutdown)
		sdrv->driver.shutdown = spi_drv_shutdown;
	return driver_register(&sdrv->driver);
}
```

需要注意的是，这里的 spi_driver 结构代表的是具体的 SPI 协议驱动程序。

15.2.5 spi_message 和 spi_transfer 结构

要完成和 SPI 设备的数据传输工作，我们还需要另外两个数据结构：spi_message 和 spi_transfer。spi_message 包含了一个的 spi_transfer 结构序列，一旦控制器接收了一个 spi_message，其中的 spi_transfer 应该按顺序被发送，并且不能被其它 spi_message 打断，所以我们认为 spi_message 就是一次 SPI 数据交换的原子操作。下面我们看看这两个数据结构的定义：

```
struct spi_message {
	struct list_head	transfers;

	struct spi_device	*spi;

	unsigned		is_dma_mapped:1;

	/* completion is reported through a callback */
	void			(*complete)(void *context);
	void			*context;
	unsigned		frame_length;
	unsigned		actual_length;
	int			status;

	struct list_head	queue;
	void			*state;
};
```

链表字段 queue 用于把该结构挂在代表控制器的 spi_master 结构的 queue 字段上，控制器上可以同时被加入多个 spi_message 进行排队。另一个链表字段 transfers 则用于链接挂在本 message 下的 spi_tranfer 结构。complete 回调函数则会在该 message 下的所有 spi_transfer 都被传输完成时被调用，以便通知协议驱动处理接收到的数据以及准备下一批需要发送的数据。我们再来看看 spi_transfer 结构：

```
struct spi_transfer {
	const void	*tx_buf;
	void		*rx_buf;
	unsigned	len;

	dma_addr_t	tx_dma;
	dma_addr_t	rx_dma;

	unsigned	cs_change:1;
	u8		tx_nbits;
	u8		rx_nbits;
	u8		bits_per_word;
	u16		delay_usecs;
	u32		speed_hz;

	struct list_head transfer_list;
};
```

首先，transfer_list 链表字段用于把该 transfer 挂在一个 spi_message 结构中，tx_buf 和 rx_buf 提供了非 dma 模式下的数据缓冲区地址，len 则是需要传输数据的长度，tx_dma 和 rx_dma 则给出了 dma 模式下的缓冲区地址。原则来讲，spi_transfer 才是传输的最小单位，之所以又引进了 spi_message 进行打包，我觉得原因是：有时候希望往 spi 设备的多个不连续的地址（或寄存器）一次性写入，如果没有 spi_message 进行把这样的多个 spi_transfer 打包，因为通常真正的数据传送工作是在另一个内核线程（工作队列）中完成的，不打包的后果就是会造成更多的进程切换，效率降低，延迟增加，尤其对于多个不连续地址的小规模数据传送而言就更为明显。

通用接口层提供了一系列用于操作和维护 spi_message 和 spi_transfer 的 API 函数，这里也列一下。

用于初始化 spi_message 结构：

- void spi_message_init(struct spi_message *m);

把一个 spi_transfer 加入到一个 spi_message 中（注意，只是加入，未启动传输过程），和移除一个 spi_transfer：

- void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);
- void spi_transfer_del(struct spi_transfer *t);

以上两个 API 的组合，初始化一个 spi_message 并添加数个 spi_transfer 结构：

- void spi_message_init_with_transfers(struct spi_message *m, struct spi_transfer *xfers, unsigned int num_xfers);

分配一个自带数个 spi_transfer 机构的 spi_message：

- struct spi_message *spi_message_alloc(unsigned ntrans, gfp_t flags);

发起一个 spi_message 的传送操作：

异步版本　int spi_async(struct spi_device *spi, struct spi_message *message);

同步版本　int spi_sync(struct spi_device *spi, struct spi_message *message);

利用以上这些 API 函数，SPI 设备的协议驱动程序就可以完成与某个 SPI 设备的数据交换工作，同时也可以看到，因为有通用接口层的隔离，控制器驱动对于协议驱动程序来说是透明的，也就是说，协议驱动程序只关心具体需要处理和交换的数据，无需关心控制器是如何传送这些数据的。spi_master，spi_message，spi_transfer 这几个数据结构的关系可以用图 15.4 来描述：

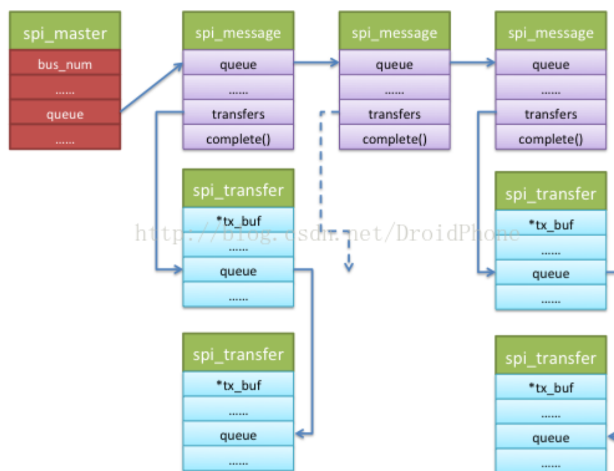


图 15.4 SPI 数据结构的关系

总结一下，协议驱动发送数据的流程大致是这样的：

1) 定义一个 spi_message 结构；
2) 用 spi_message_init 函数初始化 spi_message；
3) 定义一个或数个 spi_transfer 结构，初始化并为数据准备缓冲区并赋值给 spi_transfer 相应的字段（tx_buf，rx_buf 等）；
4) 通过 spi_message_init 函数把这些 spi_transfer 挂在 spi_message 结构下；
5) 如果使用同步方式，调用 spi_sync()，如果使用异步方式，调用 spi_async()；
   a) 另外，通用接口层也为一些简单的数据传输提供了独立的 API 来完成上述的组合过程：
6) int spi_write(struct spi_device *spi, const void *buf, size_t len); ---- 同步方式发送数据。
7) int spi_read(struct spi_device *spi, void *buf, size_t len); ---- 同步方式接收数据。
8) int spi_sync_transfer(struct spi_device *spi, struct spi_transfer *xfers, unsigned int num_xfers); ---- 同步方式，直接传送数个 spi_transfer，接收和发送。
9) int spi_write_then_read(struct spi_device *spi, const void *txbuf, unsigned n_tx, void *rxbuf, unsigned n_rx); ---- 先写后读。
10) ssize_t spi_w8r8(struct spi_device *spi, u8 cmd); ---- 写 8 位，然后读 8 位。
11) ssize_t spi_w8r16(struct spi_device *spi, u8 cmd); ---- 写 8 位，然后读 16 位。

15.3 SPI 控制器驱动

整个 SPI 驱动架构可以分为协议驱动、通用接口层和控制器驱动三大部分。其中，控制器驱动负责最底层的数据收发工作，为了完成数据的收发工作，控制器驱动需要完成以下这些功能：

1. 申请必要的硬件资源，例如中断，DMA 通道，DMA 内存缓冲区等等；
2. 配置 SPI 控制器的工作模式和参数，使之可以和相应的设备进行正确的数据交换工作；
3. 向通用接口层提供接口，使得上层的协议驱动可以通过通用接口层访问控制器驱动；
4. 配合通用接口层，完成数据消息队列的排队和处理，直到消息队列变空为止；

15.3.1 定义控制器设备

SPI 控制器遵循 linux 的设备模型框架，所以，一个 SPI 控制器在代码中对应一个 device 结构，对于嵌入式系统，我们通常把 SPI 控制器作为一个平台设备来对待，所以，只要在板级的代码中为 SPI 控制器定义一个 platform_device 结构即可。下面以 Samsung 的 SOC 芯片：S3C6410，做为例子，看看如何定义这个 platform_device。以下的代码来自：/arch/arm/plat-samsung/devs.c 中：

```
static struct resource s3c64xx_spi0_resource[] = {
    [0] = DEFINE_RES_MEM(S3C_PA_SPI0, SZ_256),
    [1] = DEFINE_RES_DMA(DMACH_SPI0_TX),
    [2] = DEFINE_RES_DMA(DMACH_SPI0_RX),
    [3] = DEFINE_RES_IRQ(IRQ_SPI0),
};

struct platform_device s3c64xx_device_spi0 = {
    .name       = "s3c6410-spi",
    .id     = 0,
    .num_resources  = ARRAY_SIZE(s3c64xx_spi0_resource),
```

```
    .resource   = s3c64xx_spi0_resource,
    .dev = {
        .dma_mask        = &samsung_device_dma_mask,
        .coherent_dma_mask  = DMA_BIT_MASK(32),
    },
};
```

由此可见，在这个 platform_device 中，定义了控制器所需的寄存器地址、DMA 通道资源和 IRQ 编号，设备的名字定义为：s3c64xx-spi，这个名字用于后续和相应的控制器驱动相匹配。在 machine 的初始化代码中，需要注册这个代表 SPI 控制器的平台设备，另外，也会通过 s3c64xx_spi0_set_platdata 函数设置平台相关的参数供后续的控制器驱动使用：

```
static struct platform_device *crag6410_devices[] __initdata = {
        ......
        &s3c64xx_device_spi0,
        ......
};

static void __init xxxx_machine_init(void)
{

        s3c64xx_spi0_set_platdata(NULL, 0, 2);
        //注册平台设备
        platform_add_devices(crag6410_devices, ARRAY_SIZE(crag6410_devices));
}
```

s3c64xx_spi0_set_platdata 函数的定义如下：

```
void __init s3c64xx_spi0_set_platdata(int (*cfg_gpio)(void), int src_clk_nr,
                int num_cs)
{
    struct s3c64xx_spi_info pd;
    ......
    pd.num_cs = num_cs;
    pd.src_clk_nr = src_clk_nr;
    pd.cfg_gpio = (cfg_gpio) ? cfg_gpio : s3c64xx_spi0_cfg_gpio;
        ......
    s3c_set_platdata(&pd, sizeof(pd), &s3c64xx_device_spi0);
}
```

上述函数主要是指定了控制器使用到的 gpio 配置、片选引脚个数和时钟配置等信息。这些信息在后面的控制器驱动中要使用到。

15.3.2 注册 SPI 控制器的 platform_driver

上一节中，把 SPI 控制器注册为一个 platform_device，相应地，对应的驱动就应该是一个平台驱动：platform_driver，它们通过 platform bus 进行相互匹配。以下的代码来自：/drivers/spi/spi-s3c64xx.c

```
static struct platform_driver s3c64xx_spi_driver = {
        .driver = {
                .name   = "s3c64xx-spi",
                .owner = THIS_MODULE,
                .pm = &s3c64xx_spi_pm,
                .of_match_table = of_match_ptr(s3c64xx_spi_dt_match),
        },
        .remove = s3c64xx_spi_remove,
        .id_table = s3c64xx_spi_driver_ids,
};
MODULE_ALIAS("platform:s3c64xx-spi");

static int __init s3c64xx_spi_init(void)
{
        return platform_driver_probe(&s3c64xx_spi_driver, s3c64xx_spi_probe);
}
subsys_initcall(s3c64xx_spi_init);
```

显然，系统初始化阶段（subsys_initcall 阶段），通过 s3c64xx_spi_init()，注册了一个平

台驱动，该驱动的名字正好也是：s3c64xx-spi，自然地，平台总线会把它和上一节定义的 platform_device 匹配上，并且触发 probe 回调被调用（就是 s3c64xx_spi_probe 函数）。当然，这里的匹配是通过 id_table 字段完成的：

```
static struct platform_device_id s3c64xx_spi_driver_ids[] = {
	{
		.name		= "s3c2443-spi",
		.driver_data	= (kernel_ulong_t)&s3c2443_spi_port_config,
	}, {
		.name		= "s3c6410-spi",
		.driver_data	= (kernel_ulong_t)&s3c6410_spi_port_config,
	},
	......
	{ },
};
```

15.3.3 注册 spi_master

在 linux 设备模型看来，代表 SPI 控制器的是第一节所定义的 platform_device 结构，但是对于 SPI 通用接口层来说，代表控制器的是 spi_master 结构，关于 spi_master 结构的描述，请参看第二节。我们知道，设备和驱动匹配上后，驱动的 probe 回调函数就会被调用，而 probe 回调函数正是对驱动程序和设备进行初始化的合适时机，本例中，对应的 probe 回调是：s3c64xx_spi_probe：

```
static int s3c64xx_spi_probe(struct platform_device *pdev)
{
	......

	/* 分配一个 spi_master 结构 */
	master = spi_alloc_master(&pdev->dev,
				sizeof(struct s3c64xx_spi_driver_data));
	......

	platform_set_drvdata(pdev, master);
	......
	master->dev.of_node = pdev->dev.of_node;
	master->bus_num = sdd->port_id;
	master->setup = s3c64xx_spi_setup;
	master->cleanup = s3c64xx_spi_cleanup;
	master->prepare_transfer_hardware = s3c64xx_spi_prepare_transfer;
	master->transfer_one_message = s3c64xx_spi_transfer_one_message;
	master->unprepare_transfer_hardware = s3c64xx_spi_unprepare_transfer;
	master->num_chipselect = sci->num_cs;
	master->dma_alignment = 8;
	master->bits_per_word_mask = SPI_BPW_MASK(32) | SPI_BPW_MASK(16) |
					SPI_BPW_MASK(8);
	/* the spi->mode bits understood by this driver: */
	master->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
	master->auto_runtime_pm = true;

	......
	/* 向通用接口层注册 spi_master 结构 */
	if (spi_register_master(master)) {
		dev_err(&pdev->dev, "cannot register SPI master\n");
		ret = -EBUSY;
		goto err3;
	}

	......
}
```

上述函数，除了完成必要的硬件资源初始化工作以外，最重要的工作就是通过 spi_alloc_master 函数分配了一个 spi_master 结构，初始化该结构，最终通过 spi_register_master 函数完成了对控制器的注册工作。从代码中我们也可以看出，spi_master

结构中的几个重要的回调函数已经被赋值，这几个回调函数由通用接口层在合适的时机被调用，以便完成控制器和设备之间的数据交换工作。

15.3.4 实现 spi_master 结构的回调函数

事实上，SPI 控制器驱动程序的主要工作，就是要实现 spi_master 结构中的几个回调函数，其它的工作逻辑，均由通用接口层帮我们完成，通用接口层会在适当的时机调用这几个回调函数，这里我只是介绍一下各个回调函数的作用，具体的实现例子，请各位自行阅读代码树中各个平台的例子（代码位于：/drivers/spi/）。

```
int (*setup)(struct spi_device *spi)
```

当协议驱动希望修改控制器的工作模式或参数时，会调用通用接口层提供的 API：spi_setup()，该 API 函数最后会调用 setup 回调函数来完成设置工作。

```
int (*transfer)(struct spi_device *spi, struct spi_message *mesg)
```

目前已经可以不用我们自己实现该回调函数，初始化时直接设为 NULL 即可，目前的通用接口层已经实现了消息队列化，注册 spi_master 时，通用接口层会提供实现好的通用函数。现在只有一些老的驱动还在使用该回调方式，新的驱动应该停止使用该回调函数，而是应该使用队列化的 transfer_one_message 回调。需要注意的是，我们只能选择其中一种方式，设置了 transfer_one_message 回调，就不能设置 transfer 回调，反之亦然。

```
void (*cleanup)(struct spi_device *spi)
```

当一个 SPI 从设备（spi_device 结构）被释放时，该回调函数会被调用，以便释放该从设备所占用的硬件资源。

```
int (*prepare_transfer_hardware)(struct spi_master *master)
int (*unprepare_transfer_hardware)(struct spi_master *master)
```

这两个回调函数用于在发起一个数据传送过程前和后，给控制器驱动一个机会，申请或释放某些必要的硬件资源，例如 DMA 资源和内存资源等等。

```
int (*prepare_message)(struct spi_master *master, struct spi_message *message)
int (*unprepare_message)(struct spi_master *master, struct spi_message *message)
```

这两个回调函数也是用于在发起一个数据传送过程前和后，给控制器驱动一个机会，对 message 进行必要的预处理或后处理，比如根据 message 需要交换数据的从设备，设定控制器的正确工作时钟、字长和工作模式等。

```
int (*transfer_one_message)(struct spi_master *master, struct spi_message *mesg)
```

当通用接口层发现 master 的队列中有消息需要传送时，会调用该回调函数，所以该函数是真正完成一个消息传送的工作函数，当传送工作完成时，应该调用 spi_finalize_current_message 函数，以便通知通用接口层，发起队列中的下一个消息的传送工作。

15.4 SPI 数据传输的队列化

SPI 数据传输可以有两种方式：同步方式和异步方式。所谓同步方式是指数据传输的发起者必须等待本次传输的结束，期间不能做其它事情，用代码来解释就是，调用传输的函数后，直到数据传输完成，函数才会返回。而异步方式则正好相反，数据传输的发起者无需等待传输的结束，数据传输期间还可以做其它事情，用代码来解释就是，调用传输的函数后，函数会立刻返回而不用等待数据传输完成，我们只需设置一个回调函数，传输完成后，该回调函数会被调用以通知发起者数据传送已经完成。同步方式简单易用，很适合处理那些少量数据的单次传输。但是对于数据量大、次数多的传输来说，异步方式就显得更加合适。

对于 SPI 控制器来说，要支持异步方式必须要考虑以下两种状况：

对于同一个数据传输的发起者，既然异步方式无需等待数据传输完成即可返回，返回后，

该发起者可以立刻又发起一个 message，而这时上一个 message 还没有处理完。

对于另外一个不同的发起者来说，也有可能同时发起一次 message 传输请求。

队列化正是为了为了解决以上的问题，所谓队列化，是指把等待传输的 message 放入一个等待队列中，发起一个传输操作，其实就是把对应的 message 按先后顺序放入一个等待队列中，系统会在不断检测队列中是否有等待传输的 message，如果有就不停地调度数据传输内核线程，逐个取出队列中的 message 进行处理，直到队列变空为止。SPI 通用接口层为我们实现了队列化的基本框架。

15.4.1 spi_transfer 的队列化

回顾一下通用接口层的介绍，对协议驱动来说，一个 spi_message 是一次数据交换的原子请求，而 spi_message 由多个 spi_transfer 结构组成，这些 spi_transfer 通过一个链表组织在一起，我们看看这两个数据结构关于 spi_transfer 链表的相关字段：

```
struct spi_transfer {
        ......
        const void      *tx_buf;
        void            *rx_buf;
        ......

        struct list_head transfer_list;
};

struct spi_message {
        struct list_head        transfers;

        struct spi_device       *spi;
        ......
        struct list_head        queue;
        ......
};
```

可见，一个 spi_message 结构有一个链表头字段：transfers，而每个 spi_transfer 结构都包含一个链表头字段：transfer_list，通过这两个链表头字段，所有属于这次 message 传输的 transfer 都会挂在 spi_message.transfers 字段下面。我们可以通过以下 API 向 spi_message 结构中添加一个 spi_transfer 结构：

```
static inline void
spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
{
        list_add_tail(&t->transfer_list, &m->transfers);
}
```

通用接口层会以一个 message 为单位，在工作线程中调用控制器驱动的 transfer_one_message 回调函数来完成 spi_transfer 链表的处理和传输工作，关于工作线程，我们留在后面讨论。

15.4.2 spi_message 的队列化

一个或者多个协议驱动程序可以同时向控制器驱动申请多个 spi_message 请求，这些 spi_message 也是以链表的形式被过在表示控制器的 spi_master 结构体的 queue 字段下面：

```
struct spi_master {
        struct device   dev;
        ......
        bool                            queued;
        struct kthread_worker           kworker;
        struct task_struct              *kworker_task;
        struct kthread_work             pump_messages;
        spinlock_t                      queue_lock;
        struct list_head                queue;
        struct spi_message              *cur_msg;
```

```
    ......
}
```

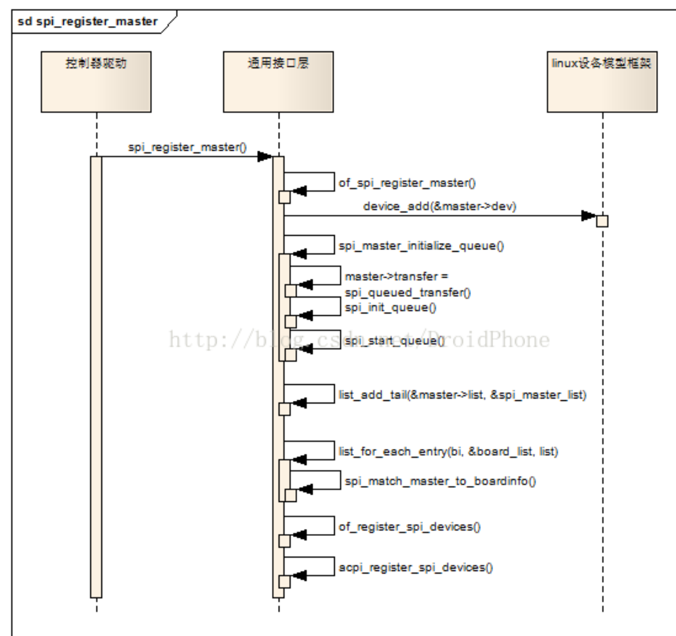
以下的 API 可以被协议驱动程序用于发起一个 message 传输操作：

```
extern int spi_async(struct spi_device *spi, struct spi_message *message)
```

spi_async 函数是发起一个异步传输的 API，它会把 spi_message 结构挂在 spi_master 的 queue 字段下，然后启动专门为 spi 传输准备的内核工作线程，由该工作线程来实际处理 message 的传输工作，因为是异步操作，所以该函数会立刻返回，不会等待传输的完成，这时，协议驱动程序（可能是另一个协议驱动程序）可以再次调用该 API，发起另一个 message 传输请求，结果就是，当工作线程被唤醒时，spi_master 下面可能已经挂了多个待处理的 spi_message 结构，工作线程会按先进先出的原则来逐个处理这些 message 请求，每个 message 传送完成后，对应 spi_message 结构的 complete 回调函数就会被调用，以通知协议驱动程序准备下一帧数据。这就是 spi_message 的队列化。工作线程唤醒时，spi_master、spi_message 和 spi_transfer 之间的关系可以用图 15.4 表示。

15.4.3 队列以及工作线程的初始化

通过第三节，SPI 控制器驱动在初始化时，会调用通用接口层提供的 API：spi_register_master，来完成控制器的注册和初始化工作，和队列化相关的字段和工作线程的初始化工作正是在该 API 中完成的。先把该 API 的调用序列图贴出来：



http://blog.csdn.net/DroidPhone

图 15.5 SPI 总线的 API 调用序列图

我们当然不希望自己实现一套队列化框架，所以，如果你在实现一个新的 SPI 控制器驱动，请记住，不要在控制器驱动中实现并赋值 spi_master 结构的 transfer 回调字段！进入 spi_master_initialize_queue 函数看看：

```
static int spi_master_initialize_queue(struct spi_master *master)
{
    ......
    master->queued = true;
    master->transfer = spi_queued_transfer;
    if (!master->transfer_one_message)
        master->transfer_one_message = spi_transfer_one_message;

    /* Initialize and start queue */
    ret = spi_init_queue(master);
    ......
```

```
        ret = spi_start_queue(master);
        ......
}
```

该函数把 master->transfer 回调字段设置为默认的实现函数：spi_queued_transfer，如果控制器驱动没有实现 transfer_one_message 回调，用默认的 spi_transfer_one_message 函数进行赋值。然后分别调用 spi_init_queue 和 spi_start_queue 函数初始化队列并启动工作线程。spi_init_queue 函数最主要的作用就是建立一个内核工作线程：

```
static int spi_init_queue(struct spi_master *master)
{
        ......

        INIT_LIST_HEAD(&master->queue);
        ......
        init_kthread_worker(&master->kworker);
        master->kworker_task = kthread_run(kthread_worker_fn,
                                   &master->kworker, "%s",
                                   dev_name(&master->dev));
        ......
        init_kthread_work(&master->pump_messages, spi_pump_messages);

        ......

        return 0;
}
```

内核工作线程的工作函数是：spi_pump_messages，该函数是整个队列化关键实现函数，我们将会在下一节中讨论该函数。spi_start_queue 就很简单了，只是唤醒该工作线程而已：

```
static int spi_start_queue(struct spi_master *master)
{
        ......

        master->running = true;
        master->cur_msg = NULL;
        ......
        queue_kthread_work(&master->kworker, &master->pump_messages);

        return 0;
}
```

自此，队列化的相关工作已经完成，系统等待 message 请求被发起，然后在工作线程中处理 message 的传送工作。

15.4.4 队列化的工作机制及过程

当协议驱动程序通过 spi_async 发起一个 message 请求时，队列化和工作线程被激活，触发一些列的操作，最终完成 message 的传输操作。先看看 spi_async 函数的调用序列图：

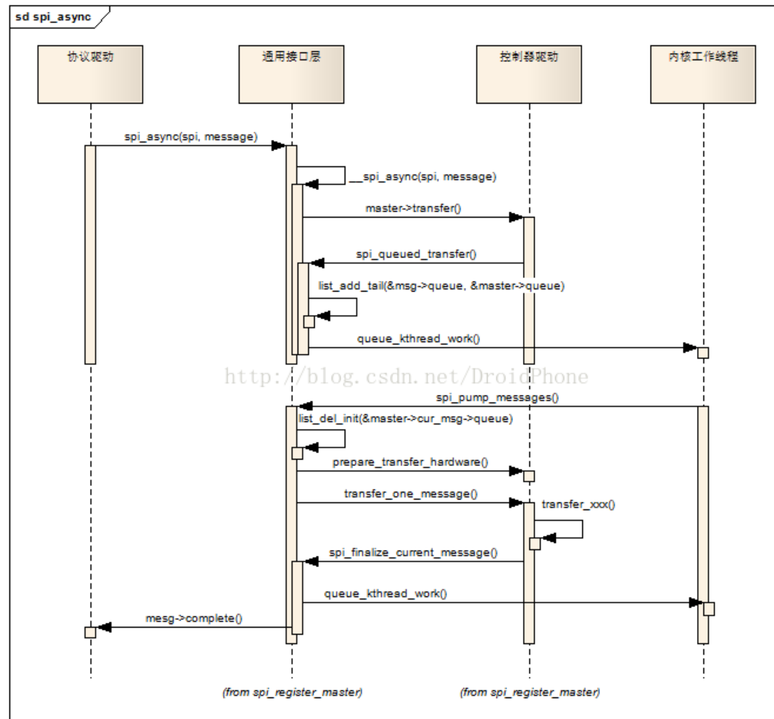


图 15.6 spi_async 函数的调用序列

spi_async 会调用控制器驱动的 transfer 回调，前面一节已经讨论过，transfer 回调已经被设置为默认的实现函数：spi_queued_transfer，该函数只是简单地把 spi_message 结构加入 spi_master 的 queue 链表中，然后唤醒工作线程。工作线程的工作函数是 spi_pump_messages，它首先把该 spi_message 从队列中移除，然后调用控制器驱动的 prepare_transfer_hardware 回调来让控制器驱动准备必要的硬件资源，然后调用控制器驱动的 transfer_one_message 回调函数完成该 message 的传输工作，控制器驱动的 transfer_one_message 回调函数在完成传输后，必须要调用 spi_finalize_current_message 函数，通知通用接口层继续处理队列中的下一个 message，另外，spi_finalize_current_message 函数也会调用该 message 的 complete 回调函数，以便通知协议驱动程序准备下一帧数据。

关于控制器驱动的 transfer_one_message 回调函数，我们的控制器驱动可以不用实现该函数，通用接口层已经为我们准备了一个标准的实现函数：spi_transfer_one_message，这样，我们的控制器驱动就只要实现 transfer_one 回调来完成实际的传输工作即可，而不用关心何时需压气哦调用 spi_finalize_current_message 等细节。这里顺便也贴出 transfer_one_message 的代码：

```
static int spi_transfer_one_message(struct spi_master *master,
                                    struct spi_message *msg)
{
        ......
        spi_set_cs(msg->spi, true);

        list_for_each_entry(xfer, &msg->transfers, transfer_list) {
                ......
                reinit_completion(&master->xfer_completion);

                ret = master->transfer_one(master, msg->spi, xfer);
                ......

                if (ret > 0)
                        wait_for_completion(&master->xfer_completion);

                ......
```

```
		if (xfer->cs_change) {
			if (list_is_last(&xfer->transfer_list,
					 &msg->transfers)) {
				keep_cs = true;
			} else {
				cur_cs = !cur_cs;
				spi_set_cs(msg->spi, cur_cs);
			}
		}

		msg->actual_length += xfer->len;
	}

out:
	if (ret != 0 || !keep_cs)
		spi_set_cs(msg->spi, false);

	......

	spi_finalize_current_message(master);

	return ret;
}
```

逻辑很清晰，这里就不再解释了。因为很多时候读者使用的内核版本和我写作时使用的版本不一样，经常会有人问有些函数或者结构不一样，所以这里顺便声明一下我使用的内核版本：3.13.0 -rc6。

15.5 实例分析-驱动编写之 SPI 设备静态注册 spidev.c

CS0 操作的是 W25X40 内容是 PMON，不能改动。

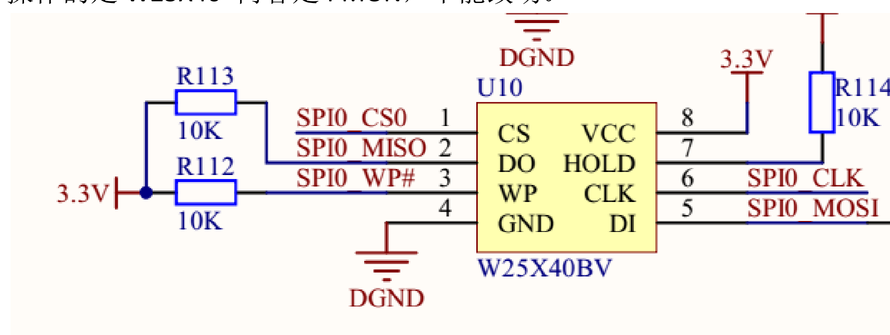


图 15.7 开发板 SPI 总线 CS0 控制的芯片

CS2 操作 SD 卡。

SPI0_CS3 89 SPI0_CS3
SPI0_MISO 90 SPI0_MISO
SPI0_MOSI 91 SPI0_MOSI
SPI0_CS0 92 SPI0_CS0
SPI0_CS1 93 SPI0_CS1
SPI0_CLK 94 SPI_CLK

图 15.8 开发板 SPI 总线 CS2 控制的 SD 卡引脚图

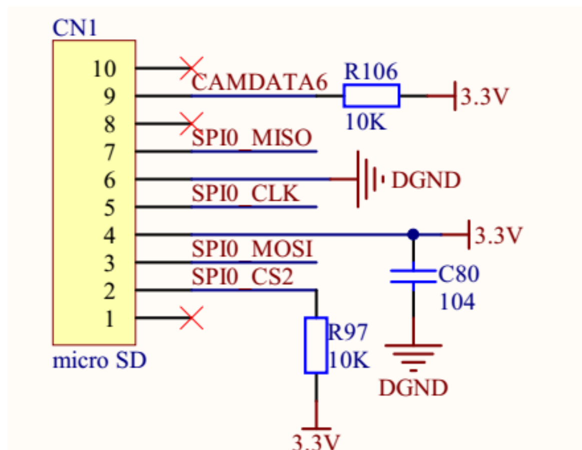


图 15.9 开发板 SPI 总线 CS2 控制的 SD 卡

CS1 和 CS3 是空余的。

SPI0_CS1	93	82	Sdio_Dat2
SPI0_CLK	95	78	Sdio_clk
SPI0_MISO	90	80	Sdio_Cmd
SPI0_MOSI	91	79	Sdio_Dat0
DGND			
SPI0_CS3	89	84	CAMCLKOUT

图 15.10 开发板 SPI 总线 CS1 复用图

如果不想为自己的 SPI 设备写驱动，那么可以用 Linux 自带的 spidev.c 提供的驱动程序。要使用 spidev.c 的驱动，只要在登记设备时，把设备名设置成 spidev 就可以。spidev.c 会在 device 目录下自动为每一个匹配的 SPI 设备创建设备节点，节点名”spi%d”。之后，用户程序可以通过字符型设备的通用接口控制 SPI 设备。

需要注意的是，spidev 创建的设备在设备模型中属于虚拟设备，它的 class 是 spidev_class。它的父设备是在 boardinfo 中定义的 spi 设备。

下面就采用这种方法

1. 先创建一个 spi_board_info 结构描述 spi 设备信息，调用 spi_register_board_info 将这个结构添加到 board_list 中。

以上一定要在注册 spi 控制器驱动即 spi master 前。

平台文件添加 spidev ,使用 CS3

```
{
        .modalias          = "spidev",
        .bus_num           = 0,
        .chip_select       = SPI0_CS3,
        .max_speed_hz      = 25000000,
        .mode = SPI_MODE_3,//  初始电平是高电平  第二个时钟采样数据
},
```

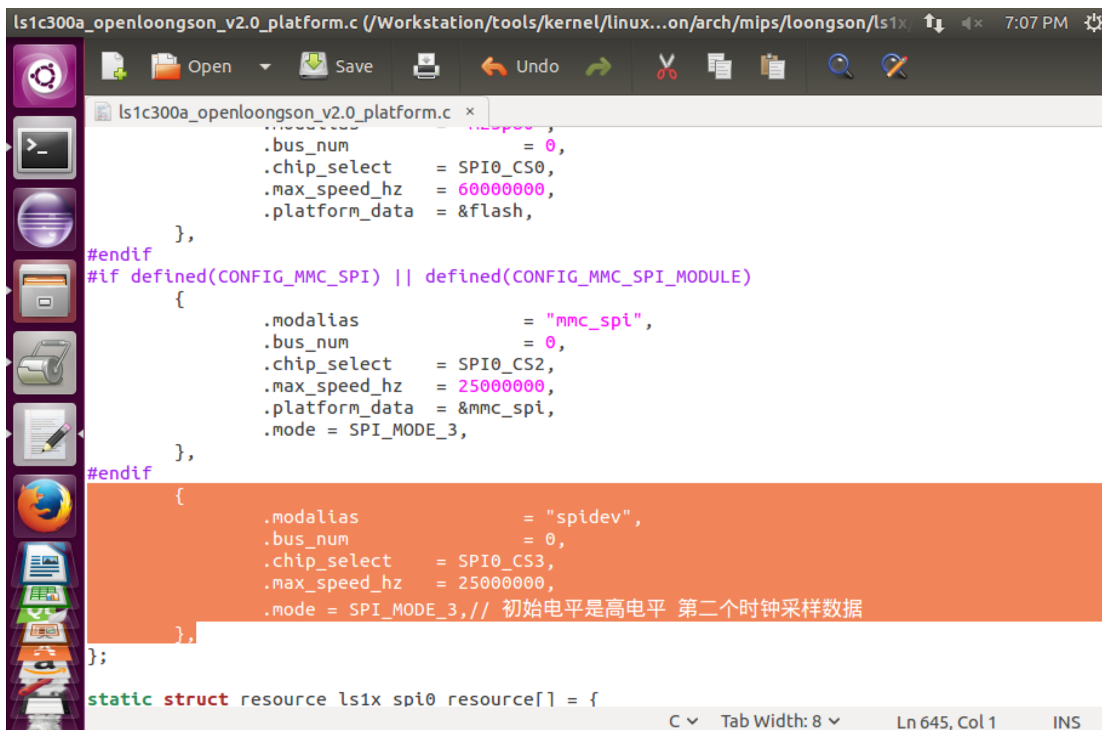


图 15.11 内核平台文件修改 1

调用下面函数，就把上面一个设备“spidev”登记到/sys/bus/spi 下了，

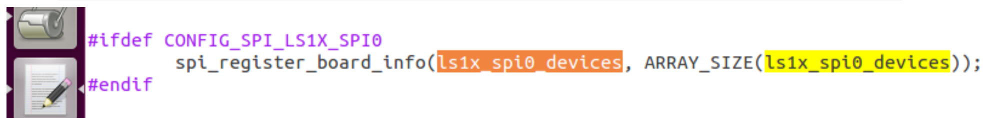


图 15.11 内核平台文件修改 2

spi_register_master 注册 SPI 控制器驱动，此时会调用 scan_boardinfo 扫描 board_list，根据 spi_board_info 调用 spi_new_device 生成 spi_device 结构，用 spi_add_device 添加设备。

重启后下载到开发板上，ls /sys/class 能看到新加的类：

```
[root@Loongson:/sys/class]#ls
backlight    graphics    input       mtd           sound        video4linux
bdi          hidraw      leds        net           spi_master   video_output
block        hwmon       mdio_bus    rtc           spidev       vtconsole
bsg          i2c-adapter mem         scsi_device   tty
firmware     i2c-dev     misc        scsi_disk     ubi
gpio         ieee80211   mmc_host    scsi_host     vc
[root@Loongson:/sys/class]#
```

登记的设备：

```
[root@Loongson:/]#ls /sys/bus/spi/devices
spi0.2    spi0.3
[root@Loongson:/]#
```

spi0.2 是 SD 卡，spi0.3 是 spidev

查询设备节点：

```
[root@Loongson:/]#ls /dev |grep spi
spidev0.3
[root@Loongson:/]#
```

如果没有设备节点，内核选择自动添加驱动：user mode spi device driver support。

```
Devices Drivers --->
[*]    SPI support     --->
<*>      user mode spi device driver support
```

15.6 实例分析-驱动编写之 SPI 设备动态注册 spike.c

参考网址：https://github.com/scottellis/spike。

动态主册 SPI 设备的步骤如下:

1. 得到管理总线的 spi_master 控制器指针(句柄);
2. 为总线分配 spi_device 结构;
3. 验证没有其他的设备已经在这条总线 bus.cs 上注册过了;
4. 使用设备特定的值(speed,datasize,etc)来填充 spi_device;
5. 将新的 spi_device 添加到总线;

源码如下：

```
/*spike.c*/
/*
   spike.c

   Copyright Scott Ellis, 2010

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

   spike v0.1 loads and registers a spi driver for a device at the bus/
   cable select specified by the constants SPI_BUS.SPI_BUS_CS1
*/

#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/mutex.h>
#include <linux/slab.h>
#include <linux/cdev.h>
#include <linux/spi/spi.h>
#include <linux/string.h>
#include <asm/uaccess.h>

#define USER_BUFF_SIZE 128

#define SPI_BUS 0
#define SPI_BUS_CS1 1
#define SPI_BUS_SPEED 1000000


const char this_driver_name[] = "spike";


struct spike_dev {
        struct semaphore spi_sem;
        struct semaphore fop_sem;
        dev_t devt;
        struct cdev cdev;
        struct class *class;
```

```
	struct spi_device *spi_device;
	char *user_buff;
};

static struct spike_dev spike_dev;


static ssize_t spike_read(struct file *filp, char __user *buff, size_t count,
			loff_t *offp)
{
	size_t len;
	ssize_t status = 0;

	if (!buff)
		return -EFAULT;

	if (*offp > 0)
		return 0;

	if (down_interruptible(&spike_dev.fop_sem))
		return -ERESTARTSYS;

	if (!spike_dev.spi_device)
		strcpy(spike_dev.user_buff, "spi_device is NULL\n");
	else if (!spike_dev.spi_device->master)
		strcpy(spike_dev.user_buff, "spi_device->master is NULL\n");
	else
		sprintf(spike_dev.user_buff, "%s ready on SPI%d.%d\n",
			this_driver_name,
			spike_dev.spi_device->master->bus_num,
			spike_dev.spi_device->chip_select);


	len = strlen(spike_dev.user_buff);

	if (len < count)
		count = len;

	if (copy_to_user(buff, spike_dev.user_buff, count))	{
		printk(KERN_ALERT "spike_read(): copy_to_user() failed\n");
		status = -EFAULT;
	} else {
		*offp += count;
		status = count;
	}

	up(&spike_dev.fop_sem);

	return status;
}

static int spike_open(struct inode *inode, struct file *filp)
{
	int status = 0;

	if (down_interruptible(&spike_dev.fop_sem))
		return -ERESTARTSYS;

	if (!spike_dev.user_buff) {
		spike_dev.user_buff = kmalloc(USER_BUFF_SIZE, GFP_KERNEL);
		if (!spike_dev.user_buff)
			status = -ENOMEM;
	}

	up(&spike_dev.fop_sem);

	return status;
```

```
}

static int spike_probe(struct spi_device *spi_device)
{
	if (down_interruptible(&spike_dev.spi_sem))
		return -EBUSY;

	spike_dev.spi_device = spi_device;

	up(&spike_dev.spi_sem);

	return 0;
}

static int spike_remove(struct spi_device *spi_device)
{
	if (down_interruptible(&spike_dev.spi_sem))
		return -EBUSY;

	spike_dev.spi_device = NULL;

	up(&spike_dev.spi_sem);

	return 0;
}

static int __init add_spike_device_to_bus(void)
{
	struct spi_master *spi_master;
	struct spi_device *spi_device;
	struct device *pdev;
	char buff[64];
	int status = 0;

	spi_master = spi_busnum_to_master(SPI_BUS);
	if (!spi_master) {
		printk(KERN_ALERT "spi_busnum_to_master(%d) returned NULL\n",
			SPI_BUS);
		printk(KERN_ALERT "Missing modprobe omap2_mcspi?\n");
		return -1;
	}

	spi_device = spi_alloc_device(spi_master);
	if (!spi_device) {
		put_device(&spi_master->dev);
		printk(KERN_ALERT "spi_alloc_device() failed\n");
		return -1;
	}

	spi_device->chip_select = SPI_BUS_CS1;

	/* Check whether this SPI bus.cs is already claimed */
	snprintf(buff, sizeof(buff), "%s.%u",
			dev_name(&spi_device->master->dev),
			spi_device->chip_select);

	pdev = bus_find_device_by_name(spi_device->dev.bus, NULL, buff);
	if (pdev) {
		/* We are not going to use this spi_device, so free it */
		spi_dev_put(spi_device);

		/*
		 * There is already a device configured for this bus.cs
		 * It is okay if it us, otherwise complain and fail.
		 */
		if (pdev->driver && pdev->driver->name &&
				strcmp(this_driver_name, pdev->driver->name)) {
```

```
printk(KERN_ALERT
		"Driver [%s] already registered for %s\n",
		pdev->driver->name, buff);
			status = -1;
		}
	} else {
		spi_device->max_speed_hz = SPI_BUS_SPEED;
		spi_device->mode = SPI_MODE_0;
		spi_device->bits_per_word = 8;
		spi_device->irq = -1;
		spi_device->controller_state = NULL;
		spi_device->controller_data = NULL;
		strlcpy(spi_device->modalias, this_driver_name, SPI_NAME_SIZE);

		status = spi_add_device(spi_device);
		if (status < 0) {
			spi_dev_put(spi_device);
			printk(KERN_ALERT "spi_add_device() failed: %d\n",
				status);
		}
	}

	put_device(&spi_master->dev);

	return status;
}

static struct spi_driver spike_driver = {
	.driver = {
		.name =	this_driver_name,
		.owner = THIS_MODULE,
	},
	.probe = spike_probe,
	.remove = __devexit_p(spike_remove),
};

static int __init spike_init_spi(void)
{
	int error;

	error = spi_register_driver(&spike_driver);
	if (error < 0) {
		printk(KERN_ALERT "spi_register_driver() failed %d\n", error);
		return error;
	}

	error = add_spike_device_to_bus();
	if (error < 0) {
		printk(KERN_ALERT "add_spike_to_bus() failed\n");
		spi_unregister_driver(&spike_driver);
		return error;
	}

	return 0;
}

static const struct file_operations spike_fops = {
	.owner =	THIS_MODULE,
	.read =	spike_read,
	.open =		spike_open,
};

static int __init spike_init_cdev(void)
{
	int error;

	spike_dev.devt = MKDEV(0, 0);
```

```
	error = alloc_chrdev_region(&spike_dev.devt, 0, 1, this_driver_name);
	if (error < 0) {
		printk(KERN_ALERT "alloc_chrdev_region() failed: %d \n",
			error);
		return -1;
	}

	cdev_init(&spike_dev.cdev, &spike_fops);
	spike_dev.cdev.owner = THIS_MODULE;

	error = cdev_add(&spike_dev.cdev, spike_dev.devt, 1);
	if (error) {
		printk(KERN_ALERT "cdev_add() failed: %d\n", error);
		unregister_chrdev_region(spike_dev.devt, 1);
		return -1;
	}

	return 0;
}

static int __init spike_init_class(void)
{
	spike_dev.class = class_create(THIS_MODULE, this_driver_name);

	if (!spike_dev.class) {
		printk(KERN_ALERT "class_create() failed\n");
		return -1;
	}

	if (!device_create(spike_dev.class, NULL, spike_dev.devt, NULL,
			this_driver_name)) {
		printk(KERN_ALERT "device_create(..., %s) failed\n",
			this_driver_name);
		class_destroy(spike_dev.class);
		return -1;
	}

	return 0;
}

static int __init spike_init(void)
{
	memset(&spike_dev, 0, sizeof(spike_dev));

	sema_init(&spike_dev.spi_sem, 1);
	sema_init(&spike_dev.fop_sem, 1);

	if (spike_init_cdev() < 0)
		goto fail_1;

	if (spike_init_class() < 0)
		goto fail_2;

	if (spike_init_spi() < 0)
		goto fail_3;

	return 0;

fail_3:
	device_destroy(spike_dev.class, spike_dev.devt);
	class_destroy(spike_dev.class);

fail_2:
	cdev_del(&spike_dev.cdev);
	unregister_chrdev_region(spike_dev.devt, 1);
```

```
fail_1:
        return -1;
}
module_init(spike_init);

static void __exit spike_exit(void)
{
        spi_unregister_device(spike_dev.spi_device);
        spi_unregister_driver(&spike_driver);

        device_destroy(spike_dev.class, spike_dev.devt);
        class_destroy(spike_dev.class);

        cdev_del(&spike_dev.cdev);
        unregister_chrdev_region(spike_dev.devt, 1);

        if (spike_dev.user_buff)
                kfree(spike_dev.user_buff);
}
module_exit(spike_exit);


MODULE_AUTHOR("Scott Ellis");
MODULE_DESCRIPTION("spike module - an example SPI driver");
MODULE_LICENSE("GPL");
MODULE_VERSION("0.1");
```

安装 spike 模块后，出现了 SPI0.1 新的 device 。0.2 是 SD 卡， 0.3 是上节建立的 spidev.

```
[root@Loongson:/]#insmod spike.ko
 [root@Loongson:/dev]#ls /sys/bus/spi/devices
spi0.1    spi0.2    spi0.3
```

查看设备，已经有了节点。

```
 [root@Loongson:/]#cd dev
[root@Loongson:/dev]#ls
console             mtd1                 spike       tty8
cpu_dma_latency     mtd1ro               tty         tty9
dsp                 mtd2                 tty0        ttyS0
fb0                 mtd2ro               tty1        ttyS1
full                mtdblock0            tty10       ttyS2
i2c-0               mtdblock1            tty11       ttyS3
i2c-1               mtdblock2            tty12       ubi_ctrl
i2c-2               network_latency      tty13       urandom
input               network_throughput   tty14       vcs
kmsg                null                 tty15       vcs1
ls1f-pwm            ptmx                 tty2        vcsa
mem                 pts                  tty3        vcsa1
mmcblk0             random               tty4        watchdog
mmcblk0p1           root                 tty5        zero
mtd0                rtc0                 tty6
mtd0ro              spidev0.3            tty7
```

15.7 编写测试程序

在内核目录\Documentation\spi 下，有 spidev_test.c 可以参考。以下程序作了修改，不停地发送字符 0x55。

```
/*spidev_test.c*/
/*
 * SPI testing utility (using spidev driver)
 *
 * Copyright (c) 2007  MontaVista Software, Inc.
 * Copyright (c) 2007  Anton Vorontsov <avorontsov@ru.mvista.com>
 *
 * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License.
 *
 * Cross-compile with cross-gcc -I/path/to/cross-kernel/include
 */

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

static void pabort(const char *s)
{
	perror(s);
	abort();
}

static const char *device = "/dev/spidev0.3";
static uint8_t mode;
static uint8_t bits = 8;
static uint32_t speed = 50000;
static uint16_t delay;
unsigned char buf_me[1] = {0x55};

static void transfer(int fd)
{
	int ret;
	uint8_t tx[] = {
		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, };
	struct spi_ioc_transfer tr = {
		.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);
	if (ret < 1)
		pabort("can't send spi message");

	for (ret = 0; ret < ARRAY_SIZE(tx); ret++) {
		if (!(ret % 6))
			puts("");
		printf("%.2X ", rx[ret]);
	}
	puts("");
}

static void print_usage(const char *prog)
{
```

```
	printf("Usage: %s [-DsbdlHOLC3]\n", prog);
	puts("  -D --device   device to use (default /dev/spidev1.1)\n"
	     "  -s --speed    max speed (Hz)\n"
	     "  -d --delay    delay (usec)\n"
	     "  -b --bpw      bits per word \n"
	     "  -l --loop     loopback\n"
	     "  -H --cpha     clock phase\n"
	     "  -O --cpol     clock polarity\n"
	     "  -L --lsb      least significant bit first\n"
	     "  -C --cs-high  chip select active high\n"
	     "  -3 --3wire    SI/SO signals shared\n");
	exit(1);
}

static void parse_opts(int argc, char *argv[])
{
	while (1) {
		static const struct option lopts[] = {
			{ "device",  1, 0, 'D' },
			{ "speed",   1, 0, 's' },
			{ "delay",   1, 0, 'd' },
			{ "bpw",     1, 0, 'b' },
			{ "loop",    0, 0, 'l' },
			{ "cpha",    0, 0, 'H' },
			{ "cpol",    0, 0, 'O' },
			{ "lsb",     0, 0, 'L' },
			{ "cs-high", 0, 0, 'C' },
			{ "3wire",   0, 0, '3' },
			{ "no-cs",   0, 0, 'N' },
			{ "ready",   0, 0, 'R' },
			{ NULL, 0, 0, 0 },
		};
		int c;

		c = getopt_long(argc, argv, "D:s:d:b:lHOLC3NR", lopts, NULL);

		if (c == -1)
			break;

		switch (c) {
		case 'D':
			device = optarg;
			break;
		case 's':
			speed = atoi(optarg);
			break;
		case 'd':
			delay = atoi(optarg);
			break;
		case 'b':
			bits = atoi(optarg);
			break;
		case 'l':
			mode |= SPI_LOOP;
			break;
		case 'H':
			mode |= SPI_CPHA;
			break;
		case 'O':
			mode |= SPI_CPOL;
			break;
		case 'L':
			mode |= SPI_LSB_FIRST;
			break;
		case 'C':
			mode |= SPI_CS_HIGH;
			break;
```

```
		case '3':
			mode |= SPI_3WIRE;
			break;
		case 'N':
			mode |= SPI_NO_CS;
			break;
		case 'R':
			mode |= SPI_READY;
			break;
		default:
			print_usage(argv[0]);
			break;
		}
	}
}

int main(int argc, char *argv[])
{
	int ret = 0;
	int fd;

	parse_opts(argc, argv);

	fd = open(device, O_RDWR);
	if (fd < 0)
		pabort("can't open device");



	/*
	 * spi mode
	 */
	ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
	if (ret == -1)
		pabort("can't set spi mode");

	ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
	if (ret == -1)
		pabort("can't get spi mode");

	/*
	 * bits per word
	 */
	ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't set bits per word");

	ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't get bits per word");

	/*
	 * max speed hz
	 */
	ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
	if (ret == -1)
		pabort("can't set max speed hz");

	ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
	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);
```

```
            while(1){
                write(fd,buf_me,1);
            }
        close(fd);

        return ret;
}
```

在虚拟机中用以下命令编译，出现错误：

```
root@ubuntu:/Workstation/examples/DrvProg/5.SPI# mipsel-linux-gcc -I /Workstation/tools/kernel/linux-3.0.82-openloongson/include/ spidev_test.c -o spidev_test
In file included from /opt/gcc-4.3-ls232/sysroot/usr/include/asm/types.h:15,
                 from /Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:4,
                 from spidev_test.c:21:
/Workstation/tools/kernel/linux-3.0.82-openloongson/include/asm-generic/int-ll64.h:11:29: error: asm/bitsperlong.h: No such file or directory
In file included from spidev_test.c:21:
/Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:13:2: warning: #warning "Attempt to use kernel headers from user space, see http://kernelnewbies.org/KernelHeaders"
root@ubuntu:/Workstation/examples/DrvProg/5.SPI# mipsel-linux-gcc -I /Workstation/tools/kernel/linux-3.0.82-openloongson/include/ spidev_test.c -o spidev_test
In file included from spidev_test.c:21:
/Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:13:2: warning: #warning "Attempt to use kernel headers from user space, see http://kernelnewbies.org/KernelHeaders"
```

说明在 int-ll64.h 中找不到 asm/bitsperlong.h。

下面修改内核目录/linux-3.0.82-openloongson/include/asm-generic/int-ll64.h 中的

```
#include <asm/bitsperlong.h>
```

改成：

```
#include <asm-generic/bitsperlong.h>
```

重新编译后成功，生成 spidev_test。拷备入开发板，运行正常。

16. CAN 总线驱动开发

16.1 智龙开发板硬件 CAN 接口

开发板的底板上电路图使用 GPIO57，56—CAN1　GPIO55，54—CAN0。

注意：扩展板丝印层上的 CAN0_H 和 CAN0_L 错误，要对调。

EJTAG_SEL	95	55	GPIO00	CAMCLKOUT	I2C_SDA0	CAN0_RX	UART3_RX	SDRAM_CS1	GPIO
EJTAG_TCK	96	56	GPIO01	CAMPCLKIN	I2C_SCL0	CAN0_TX	UART3_TX		GPIO
EJTAG_TMS	97	57	GPIO04	CAMDATA1	I2C_SDA2	PWM0	UART2_RX		GPIO
EJTAG_TDO	98	58	GPIO03	CAMHSYNC	I2C_SCL1	CAN1_TX	UART1_TX		GPIO
EJTAG_TDI	99	59	GPIO02	CAMVSYNC	I2C_SDA1	CAN1_RX	UART1_RX		GPIO
EJTAG_RST	100	60	GPIO05	CAMDATA0	I2C_SCL2	PWM1	UART2_TX		GPIO
CORE_VDD	101	61							
IO_VDD	102	62							
CAMDATA7	103	–	GPIO57		LCD_R2	CAN1_TX	I2C_SCL2	UART7_RX	
CAMDATA6	104	–	GPIO56		LCD_R1	CAN1_RX	I2C_SDA2	UART7_TX	
CAMDATA5	105	–	GPIO55		LCD_R0	CAN0_TX	I2C_SCL1	UART8_RX	
CAMDATA4	106	–	GPIO54		LCD_G1	CAN0_RX	I2C_SDA1	UART8_TX	

图 16.1 开发板上 CAN 接口复用

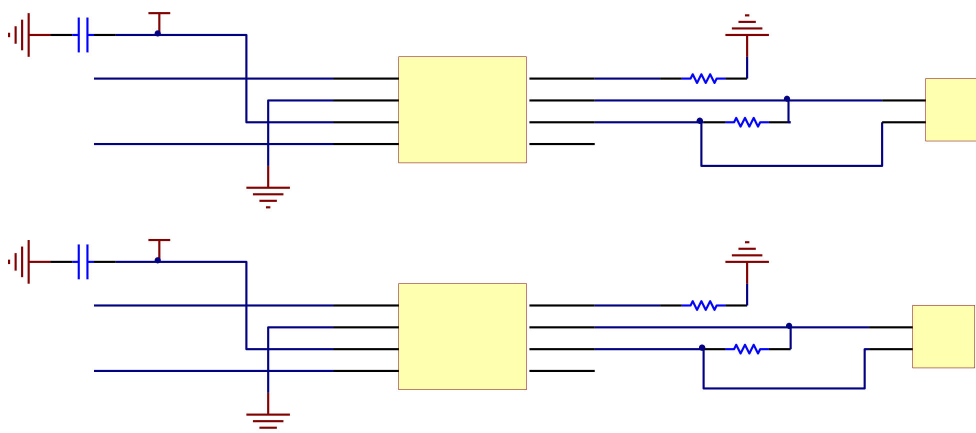


图 16.2 开发板 CAN 接口电路原理图

采用 GPIO55、54，第三复用。复用成功后，LED5 灯（orange）灭掉，不再动作。

14	CAMDATA2	led3 108	52		LCD_B2	SPI1_CS3	PWM2	UART9_TX/UART8_RTS	56 57第三功能
15	CAMDATA5	105	55		LCD_R0	CAN0_TX	I2C_SCL1	UART8_RX	CAN1_RX CAN1_TX 依赖于：
16	CAMDATA4	led5 106	54		LCD_G1	CAN0_RX	I2C_SDA1	UART8_TX	CONFIG_CAN_SJA1000_PLATFORM 54 55第三功能
17	CAMDATA7	103	57		LCD_R2	CAN1_TX	I2C_SCL2	UART7_RX	CAN0_RX CAN0_TX
18	CAMDATA6	104	56		LCD_R1	CAN1_RX	I2C_SDA2	UART7_TX	依赖于： CONFIG_CAN_SJA1000_PLATFORM

图 16.3 开板 CAN 接口与 LED 重复的部分

GPIO56 用作 SD 卡探测，也不能用作 CAN1。

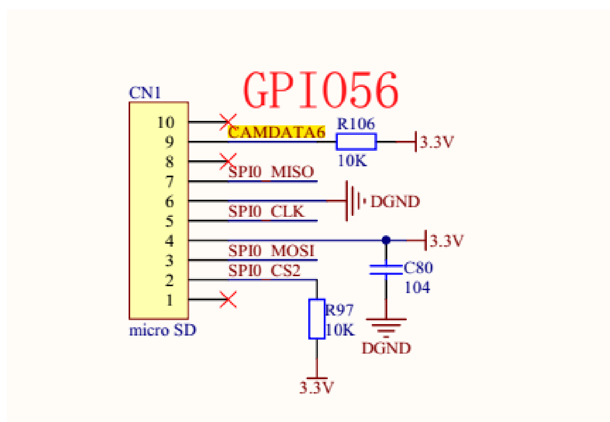


图 16.4 开发板 CAN 接口与 SD 卡探测重复部分

内核配置中，启用 CAN 控制器。

```
[*] Networking support    --->
  [*]     CAN bus subsystem support    --->
          --- CAN bus subsystem support
          [*]     Raw CAN Protocol (raw access with CAN-ID filtering)
          [*]     Broadcast Manager CAN Protocol (with content filtering)
              CAN Device Drivers    --->
                <*> Philips/NXP SJA1000 devices --->
                --- Philips/NXP SJA1000 devices --->
                <*>     Generic Platform Bus based SJA1000 driver
                [*]         Loong1CAN Controller 0
                [*]         Loong1CAN Controller 1
```

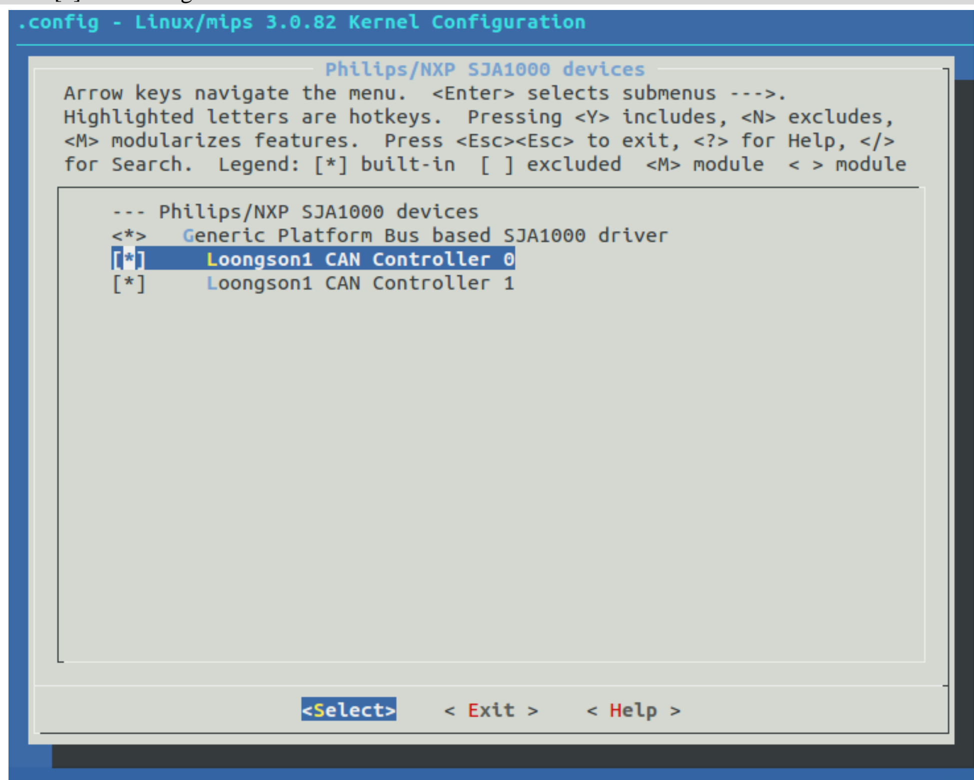


图 16.5 内核配置 CAN 控制器

在平台文件 ls1c300a_openloongson_v2.0_platform.c 中删除：

```
/*54  用作 CAN0 RX
    {
        .name                   = "led_orange",
        .gpio           = 54,
        .active_low     = 1,
```

```
                .default_trigger    = "timer",
                .default_trigger    = NULL,
                .default_state      = LEDS_GPIO_DEFSTATE_ON,
        },*/
```

在平台文件 static void ls1x_can_setup(void) 函数中添加：

```
#ifdef CONFIG_LS1X_CAN0
        /* 设置复用关系 can0 gpio54/55 */
        __raw_writel(__raw_readl(LS1X_CBUS_FIRST1) & (~0x00c00000), LS1X_CBUS_FIRST1);
        __raw_writel(__raw_readl(LS1X_CBUS_SECOND1) & (~0x00c00000), LS1X_CBUS_SECOND1);
        __raw_writel(__raw_readl(LS1X_CBUS_THIRD1) | 0x00c00000, LS1X_CBUS_THIRD1);
        __raw_writel(__raw_readl(LS1X_CBUS_FOURTHT1) & (~0x00c00000), LS1X_CBUS_FOURTHT1);

        /* 使能 can0 控制器 */
        __raw_writel(__raw_readl(LS1X_MUX_CTRL0) & (~CAN0_SHUT), LS1X_MUX_CTRL0);
#endif        //#ifdef CONFIG_LS1X_CAN0

#ifdef CONFIG_LS1X_CAN1
        /* 设置复用关系 can1 gpio56/57 */
        __raw_writel(readl(LS1X_CBUS_FIRST1) &(~( 1 << (57 -    32))) ,LS1X_CBUS_FIRST1);
        __raw_writel(readl(LS1X_CBUS_FIRST1) &( ~(1 << (56 -    32))) ,LS1X_CBUS_FIRST1);
        __raw_writel(readl(LS1X_CBUS_SECOND1) &(~( 1 << (57 -    32))) ,LS1X_CBUS_SECOND1);
        __raw_writel(readl(LS1X_CBUS_SECOND1) &( ~(1 << (56 -    32))) ,LS1X_CBUS_SECOND1);
        __raw_writel(readl(LS1X_CBUS_FOURTHT1) &(~( 1 << (57 -    32))) ,LS1X_CBUS_FOURTHT1);
        __raw_writel(readl(LS1X_CBUS_FOURTHT1) &( ~(1 << (56 -    32))) ,LS1X_CBUS_FOURTHT1);
        __raw_writel(readl(LS1X_CBUS_THIRD1) |( 1 << (57 -    32)) ,LS1X_CBUS_THIRD1);
        __raw_writel(readl(LS1X_CBUS_THIRD1) |( 1 << (56 -    32)) ,LS1X_CBUS_THIRD1); */
        /* 使能 can1 控制器 */
        __raw_writel(__raw_readl(LS1X_MUX_CTRL0) & (~CAN1_SHUT), LS1X_MUX_CTRL0);
#endif        //#ifdef CONFIG_LS1X_CAN1
```

16.2 SocketCAN

原文地址：这篇文章主要针对 can 协议簇（aka socket can)。

http://lxr.linux.no/linux+v2.6.34/Documentation/networking/can.txt

但是这篇文档没有涉及广播管理协议套接字 (SOCK_DGRAM) 的内容。

另外一篇比较好的 Socket CAN 的英文文档是（详细介绍了广播管理协议套接字）：

Low Level CAN Framework Application Programmers Interface

http://www.brownhat.org/docs/socketcan/llcf-api.html#SECTION00051000000000000000

16.2.1 概述--什么是 Socket CAN？

socketcan 子系统是在 Linux 下 CAN 协议(Controller Area Network)实现的一种实现方法。CAN 是一种在世界范围内广泛用于自动控制、嵌入式设备和汽车领域的网络技术。Linux 下最早使用 CAN 的方法是基于字符设备来实现的，与之不同的是 Socket CAN 使用伯克利的 socket 接口和 linux 网络协议栈，这种方法使得 can 设备驱动可以通过网络接口来调用。Socket CAN 的接口被设计的尽量接近 TCP/IP 的协议，让那些熟悉网络编程的程序员能够比较容易的学习和使用。

16.2.2 动机--为什么使用 socket API 接口？

在 Socket CAN 之前 Linux 中已经有了一些 CAN 的实现方法，那为什么还要启动 Socket CAN 这个项目呢？大多数已经存在的实现方法仅仅作为某个具体硬件的设备驱动，它们往往基于字符设备并且提供的功能很少。那些方案通常是由一个针对具体硬件的设备驱动提供的字符设备接口来实现原始 can 帧的发送和接收，并且直接和控制器硬件打交道。帧队列和 ISO-TP 这样的高层协议必须在用户空间来实现。就像串口设备接口一样，大多数基于字符设备的实现在同一时刻仅仅支持一个进程的访问。如果更换了 CAN 控制器，那么同时也要更

换另一个设备驱动，并且需要大多数应用程序重新调整以适应新驱动的 API。

Socket CAN 被设计用来克服以上种种不足。这种新的协议族实现了用户空间的 socket 接口，它构建于 Linux 网络层之上，因此可以直接使用已有的队列功能。CAN 控制器的设备驱动将自己作为一个网络设备注册进 Linux 的网络层，CAN 控制器收到的 CAN 帧可以传输给高层的网络协议和 CAN 协议族，反之，发送的帧也会通过高层给 CAN 控制器。传输协议模块可以使用协议族提供的接口注册自己，所以可以动态的加载和卸载多个传输协议。事实上，CAN 核心模块不提供任何协议，也不能在没有加载其它协议的情况下单独使用。同一时间可以在相同或者不同的协议上打开多个套接字，可以在相同或者不同的 CAN ID 上同时监听和发送(listen/send)。几个同时监听具有相同 ID 帧的套接字可以在匹配的帧到来后接收到相同的内容。如果一个应用程序希望使用一个特殊的协议（比如 ISO-TP）进行通信，只要在打开套接字的时候选择那个协议就可以了，接下来就可以读取和写入应用数据流了，根本无需关心 CAN-ID 和帧的结构等信息。

使用字符设备也可以让用户空间获得类似的便利，但是这中解决方案在技术上不够优雅，原因如下：

*复杂的操作。使用 Socket CAN，只需要向 socket(2)函数传递协议参数并使用 bind(2)选择 CAN 接口和 CAN ID，基于字符设备实现这样的功能却要使用 ioctl(2)来完成所有的操作。

*无法代码复用。字符设备无法使用 Linux 的网络队列代码，所以必须为 CAN 网络重写这部分功能。

*缺乏抽象性。在大多数已经实现的字符设备方案中，硬件相关的 CAN 控制器设备驱动直接提供应用程序需要的字符设备。在 Unix 系统中，无论对于字符设备还是块设备，这都是不常见的。比如，你不会为某个串口、电脑上的某个声卡、访问磁带和硬盘的 SCSI/IDE 控制器直接创建一个字符设备。相反，你会将向应用程序提供统一字符设备/块设备接口的功能交给一个抽象层来做，你自己仅仅提供硬件相关的设备驱动接口。这种抽象是通过子系统来完成的，比如 tty 层子系统、声卡子系统和 SCSI/IDE 子系统。

实现 CAN 设备驱动最简单的办法就是直接提供一个字符设备而不使用（或不完全使用）抽象层，大多数已经存在的驱动也是这么做的。但是正确的方法却要增加抽象层以支持各种功能，如注册一个特定的 CAN-ID，支持多次打开的操作和这些操作之间的 CAN 帧复用，支持 CAN 帧复杂的队列功能，还要提供注册设备驱动的 API。然而使用 Linux 内核提供的网络框架将会大大简化，这就是 Socket CAN 要做的。

在 Linux 中实现 CAN 功能最自然和合适的方式就是使用内核的网络框架。

16.2.3 Socket CAN 详解

就像第二章所讲的那样，使用 Socket CAN 的主要目的就是为用户空间的应用程序提供基于 Linux 网络层的套接字接口。与广为人知的 TCP/IP 协议以及以太网不同，CAN 总线没有类似以太网的 MAC 层地址，只能用于广播。CAN ID 仅仅用来进行总线的仲裁。因此 CAN ID 在总线上必须是唯一的。当设计一个 CAN-ECU(Electronic Control Unit 电子控制单元）网络的时候，CAN-ID 可以映射到具体的 ECU。因此 CAN-ID 可以当作发送源的地址来使用。

16.2.3.1 接收队列

允许多个应用程序同时访问网络导致了新的问题出现，那就是不同的应用程序可能会在同一个 CAN 网络接口上对具有相同 CAN-ID 的帧感兴趣。Socket CAN 的核心部分实现了 Socket

CAN 的协议族，通过高效的接收队列解决了这个问题。比如一个用户空间的程序打开了一个原始 CAN 套接字，原始协议模块将向 CAN 套接字的核心模块申请用户空间需要的一系列 CAN-ID。Socket CAN 的核心向 CAN 协议模块提供预约和解约 CAN-ID 的接口--can_rx_(un)register()，无论这个 CAN-ID 是针对一个具体的 CAN 接口还是所有已知的 CAN 接口（参考第 5 章）。

为了优化 CPU 的运行效率，每个设备都对应一个接收队列，这样比较容易实现各种报文过滤规则。

16.2.3.2 发送帧的本地回环

在其它种类的网络中，在相同或者不同网络节点上的应用程序都可以相互交换数据。

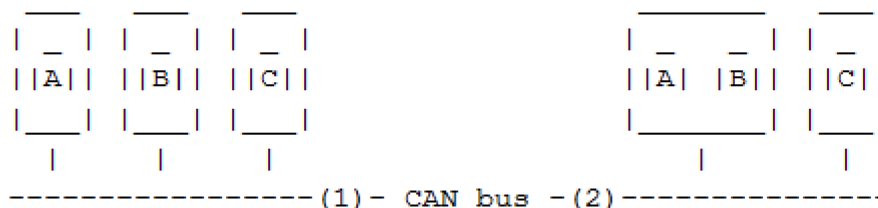


图 16.6 CAN 节点通讯示例

请看图 16.6 的两个例子。为了保证应用程序 A 在两个例子中能够接收到同样的例子（例 2 中 A 和 B 在同一个 CAN 设备上），发送出去的 CAN 帧需要能够在本地回环。

Linux 下的网络设备仅仅处理物理媒介上帧的发送和接受。总线仲裁机制下，高优先级的帧会将低优先级的帧延后。为了正确反映实际的通信活动，回环必须在正确传输成功之后进行。如果 CAN 网络的硬件不支持回环功能，一种低效的方案是使用 Socket CAN 核心部分来实现软件回环。具体的情况请参考 6.2 小节。

CAN 网络的回环功能是默认开启的。由于 RT-SocketCAN 的特殊需求，每个套接字的回环功能可以被独立关闭。CAN 原始套接字的控制选项请参考 4.1 小节。

*当你在同一个节点上运行 CAN 分析命令“candump”或者“cansniffer”的时候就会发现回环功能真的很有用。

16.2.3.3 网络安全相关

CAN 网络是一种现场总线，仅仅支持没有路由和安全控制的广播传输。大部分应用程序都需要要直接处理原始 CAN 帧，所以和其它类型的网络一样，CAN 网络对所有的用户（而不仅仅是 root 用户）访问没有任何限制。由于当前 CAN_RAW 和 CAN_BCM 的实现仅仅支持对 CAN 接口的读写，所以允许所有的用户访问 CAN 并不影响其它类型网络的安全性。为了使能非 root 用户对 CAN_RAW 和 CAN_BCM 协议套接字的访问，必须在编译内核的时候选上 Kconfig 的 CAN_RAW_USER/CAN_BCM_USER 选项。

16.2.3.4 网络故障监测

使用 CAN 总线的时候，可能会遇到物理和 mac（media access control）层的问题。为了方便用户分析物理收发器的硬件错误、总线仲裁错误和不同的 ECU(Electrical Conversion Unit, 电气转换装置)引起的错误，对于底层（物理和 mac 层）的监测和记录是至关重要的。拥有精确时间戳的错误监测对于诊断错误是非常重要的。基于以上原因，CAN 接口的驱动可以可以选择性的产生所谓的错误帧，这些帧以和其它的 CAN 帧一样的方式传递给用户程序。当一个物理层或者 MAC 层的错误被(CAN 控制器)检测到之后，驱动创建一个相应的错误帧。错误帧可以被应用程序通过 CAN 的过滤机制请求得到。过滤机制允许选择需要的错误帧的类型。默认情况下，接收错误帧的功能是禁止的。

CAN 错误帧的详细格式定义在 linux 头文件中：include/linux/can/error.h。

16.2.4 如何使用 Socket CAN

就像 TCP/IP 协议一样，在使用 CAN 网络之前你首先需要打开一个套接字。CAN 的套接字使用到了一个新的协议族，所以在调用 socket(2)这个系统函数的时候需要将 PF_CAN 作为第一个参数。当前有两个 CAN 的协议可以选择，一个是原始套接字协议（ raw socket protocol），另一个是广播管理协议 BCM（broadcast manager）。你可以这样来打开一个套接字：

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

或者

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

在成功创建一个套接字之后，你通常需要使用 bind(2)函数将套接字绑定在某个 CAN 接口上（这和 TCP/IP 使用不同的 IP 地址不同，参见第 3 章）。在绑定 (CAN_RAW)或连接(CAN_BCM)套接字之后，你可以在套接字上使用 read(2)/write(2)，也可以使用 send(2)/sendto(2)/sendmsg(2)和对应的 recv*操作。当然也会有 CAN 特有的套接字选项，下面将会说明。

基本的 CAN 帧结构体和套接字地址结构体定义在 include/linux/can.h：

```
/*
* 扩展格式识别符由 29 位组成。其格式包含两个部分：11 位基本 ID、18 位扩展 ID。
* Controller Area Network Identifier structure
*
* bit 0-28        : CAN 识别符 (11/29 bit)
* bit 29         :  错误帧标志 (0 = data frame, 1 = error frame)
* bit 30         :  远程发送请求标志 (1 = rtr frame)
* bit 31         :帧格式标志 (0 = standard 11 bit, 1 = extended 29 bit)
*/
typedef __u32 canid_t;

struct can_frame {
        canid_t can_id;    /* 32 bit CAN_ID + EFF/RTR/ERR flags */
          __u8        can_dlc; /*  数据长度: 0 .. 8 */
          __u8        data[8] __attribute__((aligned(8)));
  };
```

结构体的有效数据在 data[]数组中，它的字节对齐是 64bit 的，所以用户可以比较方便

的在 data[]中传输自己定义的结构体和共用体。CAN 总线中没有默认的字节序。在 CAN_RAW 套接字上调用 read(2)，返回给用户空间的数据是一个 struct can_frame 结构体。

就像 PF_PACKET 套接字一样，sockaddr_can 结构体也有接口的索引，这个索引绑定了特定接口：

```
struct sockaddr_can {
        sa_family_t can_family;
        int             can_ifindex;
        union {
                /* transport protocol class address info (e.g. ISOTP) */
                struct { canid_t rx_id, tx_id; } tp;
                /* reserved for future CAN protocols address information */
        } can_addr;
};
```

指定接口索引需要调用 ioctl()（比如对于没有错误检查 CAN_RAW 套接字）:

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));

(..)
```

为了将套接字和所有的 CAN 接口绑定，接口索引必须是 0。这样套接字便可以从所有使能的 CAN 接口接收 CAN 帧。recvfrom(2)可以指定从哪个接口接收。在一个已经和所有 CAN 接口绑定的套接字上，sendto(2)可以指定从哪个接口发送。

从一个 CAN_RAW 套接字上读取 CAN 帧也就是读取 struct can_frame 结构体:

```
struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));

if (nbytes < 0) {
        perror("can raw socket read");
        return 1;
}

/* paranoid check ... */
if (nbytes < sizeof(struct can_frame)) {
        fprintf(stderr, "read: incomplete CAN frame\n");
        return 1;
}

/* do something with the received CAN frame */
```

写 CAN 帧也是类似的，需要用到 write (2)函数：

```
nbytes = write(s, &frame, sizeof(struct can_frame));
```

如果套接字跟所有的 CAN 接口都绑定了（addr.can_index = 0），推荐使用 recvfrom(2) 获取数据源接口的信息：

```
struct sockaddr_can addr;
struct ifreq ifr;
socklen_t len = sizeof(addr);
struct can_frame frame;

nbytes = recvfrom(s, &frame, sizeof(struct can_frame),
                  0, (struct sockaddr*)&addr, &len);

/* get interface name of the received CAN frame */
ifr.ifr_ifindex = addr.can_ifindex;
ioctl(s, SIOCGIFNAME, &ifr);
printf("Received a CAN frame from interface %s", ifr.ifr_name);
```

对于绑定了所有接口的套接字，向某个端口发送数据必须指定接口的详细信息：

```
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_family  = AF_CAN;

nbytes = sendto(s, &frame, sizeof(struct can_frame),
                0, (struct sockaddr*)&addr, sizeof(addr));
```

16.2.4.1 使用 can_filter 的原始套接字 (RAW socket)

CAN_RAW 套接字的用法和 CAN 字符设备的用法是类似的。为了使用 CAN 套接字的新特性，在绑定原始套接字的时候将会默认开启以下特性：

- filter 将会接收所有的数据
- 套接字仅仅接收有效的数据帧（=> no error frames）
- 发送帧的回环功能被开启（参见 3.2 节）
- （回环模式下）套接字不接收它自己发送的帧

这些特性的设置可以在绑定之前和之后修改。为了使用 CAN_RAW 套接字相关的选项，必须包含<linux/can/raw.h>。

16.2.4.1.1 原始套接字选项 CAN_RAW_FILTER

CAN_RAW 套接字的接收可以使用 CAN_RAW_FILTER 套接字选项指定的多个过滤规则（过滤器）来过滤。

过滤规则（过滤器）的定义在 include/linux/can.h 中:

```
struct can_filter {
```

```
        canid_t can_id;
        canid_t can_mask;
    };
```

过滤规则的匹配：

```
    <received_can_id> & mask == can_id & mask

/*
#define CAN_INV_FILTER 0x20000000U /* to be set in can_filter.can_id */
#define CAN_ERR_FLAG     0x20000000U /* error frame */
*/
```

这和大家所熟知的 CAN 控制器硬件过滤非常相似。可以使用 CAN_INV_FILTER 这个宏将 can_filter 结构体的成员 can_id 中的比特位反转。和 CAN 控制器的硬件过滤形成鲜明对比的是，用户可以为每一个打开的套接字设置多个独立的过滤规则（过滤器）:

```
/*
 /* valid bits in CAN ID for frame formats */
#define CAN_SFF_MASK 0x000007FFU /*  标准帧格式 (SFF) */
#define CAN_EFF_MASK 0x1FFFFFFFU /*  扩展帧格式 (EFF) */
#define CAN_ERR_MASK 0x1FFFFFFFU /*  忽略 EFF, RTR, ERR 标志 */
 */

    struct can_filter rfilter[2];

    rfilter[0].can_id     = 0x123;
    rfilter[0].can_mask = CAN_SFF_MASK;
    rfilter[1].can_id     = 0x200;
    rfilter[1].can_mask = 0x700;

    setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

为了在指定的 CAN_RAW 套接字上禁用接收过滤规则，可以这样：

```
  setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

在一些极端情况下不需要读取数据，可以把过滤规则清零（所有成员设为 0），这样原始套接字就会忽略接收到的 CAN 帧。在这种仅仅发送数据（不读取）的应用中可以在内核中省略接收队列，以此减少 CPU 的负载（虽然只能减少一点点）。

16.2.4.1.2 原始套接字选项 CAN_RAW_ERR_FILTER

正如 3.4 节所说，CAN 接口驱动可以选择性的产生错误帧，错误帧和正常帧以相同的方式传给应用程序。可能产生的错误被分为不同的种类，使用适当的错误掩码可以过滤它们。为了注册所有可能的错误情况，CAN_ERR_MASK（0x1FFFFFFFU）这个宏可以用来作为错误掩码。这个错误掩码定义在 linux/can/error.h。

```
    can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );

    setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER,
               &err_mask, sizeof(err_mask));
```

16.2.4.1.3 原始套接字选项 CAN_RAW_LOOPBACK

为了满足众多应用程序的需要，本地回环功能默认是开启的（详细情况参考 3.2 节）。但是在一些嵌入式应用场景中（比如只有一个用户在使用 CAN 总线），回环功能可以被关闭（各个套接字之间是独立的）：

```
int loopback = 0; /* 0 = disabled, 1 = enabled (default) */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

16.2.4.1.4 原始套接字选项 CAN_RAW_RECV_OWN_MSGS

在本地回环功能开启的情况下，所有的发送帧都会被回环到在相应 CAN 接口上注册了同样 CAN-ID（和发送帧的相同）的套接字上。发送 CAN 帧的套接字被假设不想接收自己发送的 CAN 帧，因此在发送套接字上的回环功能默认是关闭的。可以在需要的时候改变这一默认行为：

```
int recv_own_msgs = 1; /* 0 = disabled (default), 1 = enabled */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS,
              &recv_own_msgs, sizeof(recv_own_msgs));
```

16.2.4.2 广播管理协议套接字 (SOCK_DGRAM)

（没有内容）

16.2.4.3 面向连接的传输协议 (SOCK_SEQPACKET)

（没有内容）

16.2.4.4 无连接的传输协议 (SOCK_DGRAM)

（没有内容）

16.2.5 Socket CAN 核心模块

CAN 套接字的核心模块实现了 PF_CAN 协议族。CAN 协议模块在核心模块运行后加载。CAN 核心模块为协议模块提供了申请需要的 ID 的接口（参考 3.1 小节）。

16.2.5.1 can.ko 模块的参数

- stats_timer: 为了计算 CAN 套接字的统计信息（比如最近一秒的帧数和每秒最大的帧数），can.ko 调用一个定时间隔为 1 秒的定时器，默认情况下这个定时器是开启的。这个定时器也可以在模块参数中传入 stattimer=0 来禁止。

```
- debug :    (removed since SocketCAN SVN r546)
```

16.2.5.2 procfs 接口

就像 3.1 节描述的那样，CAN 套接字核心借助于一些带有过滤规则的队列向 CAN 协议模块传递接收到的 CAN 帧。可以在 procfs 中查看这些接收队列的的过滤规则和匹配规则的次数。所有的条目都包了设备名和协议模块标识：

```
foo@bar:~$ cat /proc/net/can/rcvlist_all

receive list 'rx_all': (vcan3: no entry)
  (vcan2: no entry)
  (vcan1: no entry)
  device   can_id   can_mask   function   userdata   matches   ident
  vcan0     000     00000000   f88e6370   f6c6f400         0     raw
 (any: no entry)
（yll 补充：function 是一个函数指针，userdata 是一个 void *指针，所以这两个值打印出来没有太大意义）
```

在这个例子中，这个应用程序接收所有 vcan0 上传输的数据.

rcvlist_all - 没有过滤规则的队列
rcvlist_eff - 扩展帧（EFF）的队列
rcvlist_err - 错误帧队列
rcvlist_fil - 通过过滤规则的队列
rcvlist_inv - 未通过过滤规则的队列
rcvlist_sff - 标准帧的队列

在/proc/net/can 中还有另外一些文件：

stats - CAN 套接字核心的统计信息（接收/发送的帧数，匹配率等）
reset_stats - 复位统计信息
version - 打印 CAN 套接字核心的版本和 ABI 的版本

16.2.5.3 写一个自己的 CAN 协议模块

要在 PF_CAN 中增加一个新的协议，必须在 include/linux/can.h 中为新的协议增加相应的定义。包含 include/linux/can.h 这个文件便可以使用增加的原型和定义。内核除了提供了注册 CAN 协议和 CAN 设备的通知列表的功能，也提供了在一个特定 CAN 接口上注册感兴趣

的 CAN 帧或者发送 CAN 帧的功能。

```
    can_rx_register     - 在一个特定接口上注册希望接收到的 CAN 帧的信息 （yll:这个函数的定义在内核的 net/can/af_can.c 中）
    can_rx_unregister - 注销上面的申请
    can_send            - 发送 CAN 帧（可以选择是否开启本地回环）
```

详细的信息请参考内核中的源码 net/can/af_can.c、net/can/raw.c、net/can/bcm.c。

16.2.6 CAN 网络驱动

编写一个 CAN 网络设备驱动要比写一个 CAN 字符设备驱动要容易的多。和编写其它网络设备驱动类似，你只要处理以下事宜：

- TX :将套接字缓冲区的 CAN 帧发送到 CAN 控制器
- RX :从 CAN 控制器的 CAN 帧读取到套接字缓冲区

Documentation/networking/netdevices.txt 中是网络设备驱动的例子。下面将会介绍 CAN 网络设备驱动的一些独有特性。

16.2.6.1 常见设置

```
    dev->type    = ARPHRD_CAN; /* the netdevice hardware type */
    dev->flags = IFF_NOARP;    /* CAN has no arp */
    dev->mtu      = sizeof(struct can_frame);
```

结构体 can_frame 是 PF_CAN 协议族套接字缓冲区的数组载体。

16.2.6.2 发送帧的本地回环

如 3.2 小节所述，CAN 网络设备驱动应该支持类似 TTY 设备回显功能的本地回环功能。如果驱动支持这个功能，则要设备 IFF_ECHO 标志来防止 PF_CAN 核心回显发送帧（又称回环）。

```
    dev->flags = (IFF_NOARP | IFF_ECHO);
```

16.2.6.3 CAN 控制器的硬件过滤

为了减小一些嵌入式系统的中断负载，一些 CAN 控制器支持多个 CAN-ID 或者多个 CAN-ID 区间的过滤功能。硬件过滤功能在不同的控制器之间差异很大，并且不能同时满足多个用户的不同过滤需求。在单用户应用中使用控制器的硬件过滤或许还有意义，但是在一个多用户系统中驱动层的过滤将会影响所有用户。PF_CAN 核心内置的过滤规则集合允许对每个套接字独立的设置多个过滤规则。因此使用硬件过滤属于嵌入式系统中“手动调整”的范畴。从 2002 年开始笔者一直使用拥有四路 SJA1000 CAN 控制器的 MPC603e @133MHz，

总线负载虽然很高，但是到目前为止还没有什么问题。

16.2.6.4 虚拟的 CAN 驱动 (vcan)

和网络回环设备一样，vcan 提供一个虚拟的本地 CAN 接口。CAN 中一个有效的地址包括：

- 一个有效的 CAN 标识符（CAN ID）
- 这个 CAN 标识符将要发往的总线（比如 can0）.

所以在一般的应用场景中往往需要多个 vcan 接口。

vcan 接口允许在没有控制器硬件的情况下进行发送和接收。vcan 网络设备的命名一般采用‘vcanX’,比如 can1 vcan2 等。当编译为单独的模块的时候，vcan 驱动的模块名为 vcan.ko。

vcan 驱动从 linux2.6.24 开始支持 netlink 接口，使得创建 vcan 网络设备变的可能。可以使用 ip(8)命令工具管理 vcan 网络设备的创建和移除：

- 创建一个 vcan 网络接口：

```
$ ip link add type vcan
```

- 使用给定的名字 'vcan42'创建 一个 vcan 网络接口：

```
$ ip link add dev vcan42 type vcan
```

- 移除 vcan 网络接口'vcan42':

```
$ ip link del vcan42
```

16.2.6.5 CAN 网络设备驱动接口

CAN 网络设备驱动提供了进行安装、配置和监控 CAN 网络设备的接口。可以使用 IPROUTE2 工具集中的“ip”命令通过 netlink 接口配置 CAN 设备，比如设置波特率。本章剩余的部分将会简介如何使用这一工具。另外这些接口使用一些通用的数据结构并且提供了一系列常用的功能，这些功能都是 CAN 网络设备驱动需要用到的。请参考 SJA1000 或者 MSCAN 的驱动去了解如何使用它们。模块的名字是 can-dev.ko。

16.2.6.5.1 Netlink 接口--设置/获取设备属性

CAN 设备必须使用 netlink 来配置。在"include/linux/can/netlink.h"有对 netlink 消息类型的定义和简短描述。IPROUTE2 工具集中的“ip”命令可以使用 CAN 的 netlink 支持，下面是使用示例：

- 设置 CAN 设备属性：

```
$ ip link set can0 type can help
Usage: ip link set DEVICE type can [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |
[ tq TQ prop-seg PROP_SEG phase-seg1 PHASE-SEG1 phase-seg2 PHASE-SEG2 [ sjw SJW ] ]
[ loopback { on | off } ]
[ listen-only { on | off } ]
[ triple-sampling { on | off } ]
[ restart-ms TIME-MS ]
[ restart ]
Where: BITRATE       := { 1..1000000 } SAMPLE-POINT   := { 0.000..0.999 }
       TQ            := { NUMBER }
       PROP-SEG      := { 1..8 }
       PHASE-SEG1    := { 1..8 }
       PHASE-SEG2    := { 1..8 }
       SJW           := { 1..4 }
       RESTART-MS    := { 0 | NUMBER }
```

- 显示 CAN 设备的详情和统计信息：

```
$ ip -details -statistics link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP qlen 10
    link/can
     can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100
     bitrate 125000 sample_point 0.875
     tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
     sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
     clock 8000000
     re-started bus-errors arbit-lost error-warn error-pass bus-off
     41         17457      0          41         42         41
     RX: bytes  packets  errors  dropped overrun mcast
     140859     17608    17457   0       0       0
     TX: bytes  packets  errors  dropped carrier collsns
     861        112      0       41      0       0
```

下面是上面一些名词的解释：

```
"<TRIPLE-SAMPLING>"
```

表示选中的 CAN 控制器的模式：LOOPBACK, LISTEN-ONLY, or TRIPLE-SAMPLING。

```
"state ERROR-ACTIVE"
```

CAN 控制器的当前状态："ERROR-ACTIVE", "ERROR-WARNING", "ERROR-PASSIVE", "BUS-OFF" or "STOPPED"

```
"restart-ms 100"
```

自动重启的延时时间。如果设为非零值， 在总线关闭的情况下，在设定的数量毫秒后 CAN 控制器被自动触发。这个功能默认是关闭的。

```
"bitrate 125000 sample_point 0.875"
```

使用 bits/sec 作为单位显示位时间并显示 0.000～0.999 的采样点位置。如果内核中使能了统计位时间的功能(CONFIG_CAN_CALC_BITTIMING=y)，位时间可以使用"bitrate"参数来设置。可选的"sample-point"也是可以配置的。默认使用 0.000 这个推荐值。

```
"tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1"
```

以 ns 为单位显示时间份额（tq-time quanta）、传播段（prop-seg ： propagation segment）、相位缓冲段 1 和 2（phase-seg：phase buffer），以 tq 为单位显示同步跳转宽度（sjw：synchronisation jump width）。这些变量允许定义与硬件无关的位时序，这也是 Bosch CAN 2.0 spec 所推荐的（参考第八章 http://www.semiconductors.bosch.de/pdf/can2spec.pdf）。

```
"sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1 clock 8000000"
```

显示 CAN 控制器的比特时序常量，这里的例子以 sja1000 为例。时间段（tseg -time segment）1 和 2 的最小和最大值，以 tq 为单位的同步跳转宽度，比特速率的预分频器（brp--pre-scaler）和 CAN 系统时钟（以 HZ 为单位）。这些常量可以被用户空间的比特时序统计算法所使用。

```
"re-started bus-errors arbit-lost error-warn error-pass bus-off"
```

显示重启的次数、总线和仲裁丢失错误，错误主动（error-warning）、错误被动（error-passive）、和总线关闭的状态变化。接收的过载错误在 统计信息的"overrun"域下面列出。

16.2.6.5.2 设置 CAN 的比特_时序

CAN 比特时序参数可以使用硬件无关的定义方法。这些参数是： "tq", "prop_seg", "phase_seg1", "phase_seg2" 和 "sjw":

```
$ ip link set canX type can tq 125 prop-seg 6 \
                    phase-seg1 7 phase-seg2 2 sjw 1
```

在内核选项 CONFIG_CAN_CALC_BITTIMING 被使能的情况下，如果比特率（波特率）参数 "bitrate"被设置了，CAN 的这些参数将会生效：

```
$ ip link set canX type can bitrate 125000
```

请注意，这条命令在大部分使用标准波特率的 CAN 控制器上工作良好，但是使用一些少见的波特率值（如 115000）和时钟频率值将会失败。禁用内核的 CONFIG_CAN_CALC_BITTIMING 选项可以节省一些内存空间并且允许用户空间的命令工具完全的控制比特时序参数。使用 CAN 控制器的比特时序常量就可以达到这个目的（用户空间控制比特时序）。下面的命令将会列出这些变量：

```
$ ip -details link show can0
...
    sja1000: clock 8000000 tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
```

16.2.6.5.3 启动和停止 CAN 网络设备

一个 CAN 网络设备可以使用"ifconfig canX up/down" 或者 "ip link set canX up/down"来开启和关闭。为了避免错误的默认值，必须在启动 CAN 设备之前设置它的比特时序参数：

```
$ ip link set canX up type can bitrate 125000
```

如果总线上出现太多的错误设备可能进入总线关闭状态（也就是从总线脱离）。进入总线关闭状态之后设备不会再发送和接收信息。给"restart-ms"设置一个非零值可以开启总线关闭自动恢复的功能（也就是进入总线关闭状态后重新开启），下面是一个示例：

```
$ ip link set canX type can restart-ms 100
```

应用程序可以通过监测 CAN 的错误帧意识到已经进入总线关闭状态，并且可以使用以下命令重启：

```
$ ip link set canX type can restart
```

注意，重启也会生成一个 CAN 错误帧（参见 3.4 节）。

16.2.6.6 支持 Socket CAN 的硬件

"drivers/net/can"中的“Kconfig”文件中可以查看到所有支持的硬件列表。在 CAN 套接字项目网站上（参见第 7 章）有更多驱动何以获得，当然也有对早些时期版本内核的支持 。

16.2.7 学习 Socket CAN 的相关资源

你可在 BerliOS OSS 项目网站的 CAN 套接字页面中发现更多资源，比如用户空间工具、对旧版内核的支持、更多的驱动、邮件列表等：

http://developer.berlios.de/projects/socketcan

如果你有任何问题或者发现了 BUG，不要迟疑，立马发送邮件到 CAN 套接字的用户邮件列表。但是在发送之前请首先搜索邮件记录中是否已经有了相同的问题。

16.2.8. 贡献者名单

=========

Oliver Hartkopp (PF_CAN core, filters, drivers, bcm, SJA1000 driver) Urs Thuermann (PF_CAN core, kernel integration, socket interfaces, raw, vcan) Jan Kizka (RT-SocketCAN core, Socket-API reconciliation)

Wolfgang Grandegger (RT-SocketCAN core & drivers, Raw Socket-API reviews, CAN device driver interface, MSCAN driver) Robert Schwebel (design reviews, PTXdist integration)

Marc Kleine-Budde (design reviews, Kernel 2.6 cleanups, drivers) Benedikt Spranger (reviews)

Thomas Gleixner (LKML reviews, coding style, posting hints)

Andrey Volkov (kernel subtree structure, ioctls, MSCAN driver) Matthias Brukner (first SJA1000 CAN netdevice implementation Q2/2003) Klaus Hitschler (PEAK driver integration)

Uwe Koppe (CAN netdevices with PF_PACKET approach)

Michael Schulze (driver layer loopback requirement, RT CAN drivers review) Pavel Pisa (Bit-timing calculation)

Sascha Hauer (SJA1000 platform driver)

Sebastian Haas (SJA1000 EMS PCI driver)

Markus Plessing (SJA1000 EMS PCI driver)

Per Dalen (SJA1000 Kvaser PCI driver)

Sam Ravnborg (reviews, coding style, kbuild help)

==
作者：yuanlulu
http://blog.csdn.net/yuanlulu

版权没有，但是转载请保留此段声明

16.3 测试工具

测试 can 需要 ip,can-utils 和 libsocketcan 库。通过 ip 工具配置 can,如速率,启用和禁用 can 等。

16.4 Socket CAN 在智龙上的测试-使用工具 iproute2

ifocnfig -a 能看到 can 节点

```
[root@Loongson:/ can_tools]#ifconfig -a
can0        Link encap:UNSPEC    HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
            NOARP    MTU:16    Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:10
            RX bytes:0 (0.0 B)    TX bytes:0 (0.0 B)
            Interrupt:6
```

16.4.1 下载编译

1）下载源码

地址： http://pkgs.fedoraproject.org/repo/pkgs/iproute/

这里下载的是 iproute2-3.1.0.tar.bz2。

或者 在虚拟机上运行以下命令

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git
```

2）Makefile 中修改交叉编译工具链

```
CC = mipsel-linux-gcc
HOSTCC = mipsel-linux-gcc
SUBDIRS=lib ip
```

make 编译后有错如下图：

```
root@ubuntu: /Workstation/tools/iproute/iproute2-3.1.0
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESO
LVE_HOSTNAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_can.o iplink_ca
n.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESO
LVE_HOSTNAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvlan.o iplin
k_macvlan.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESO
LVE_HOSTNAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvtap.o iplin
k_macvtap.c
mipsel-linux-gcc -Wl,-export-dynamic  ip.o ipaddress.o ipaddrlabel.o iproute.o i
prule.o ipnetns.o rtm_map.o iptunnel.o ip6tunnel.o tunnel.o ipneigh.o ipntable.o
 iplink.o ipmaddr.o ipmonitor.o ipmroute.o ipprefix.o iptuntap.o ipxfrm.o xfrm_s
tate.o xfrm_policy.o xfrm_monitor.o iplink_vlan.o link_veth.o link_gre.o iplink_
can.o iplink_macvlan.o iplink_macvtap.o ../lib/libnetlink.a ../lib/libutil.a  -l
resolv ../lib/libnetlink.a ../lib/libutil.a -ldl -o ip
ipnetns.o: In function `netns_exec':
ipnetns.c:(.text+0x538): undefined reference to `setns'
collect2: ld returned 1 exit status
<builtin>: recipe for target 'ip' failed
make[1]: *** [ip] Error 1
make[1]: Leaving directory '/Workstation/tools/iproute/iproute2-3.1.0/ip'
Makefile:43: recipe for target 'all' failed
make: *** [all] Error 2
root@ubuntu:/Workstation/tools/iproute/iproute2-3.1.0#
```

图 16.7 虚拟机中编译 CAN 工具

根据需要修改内核文件 ipnetns.c 文件，屏蔽调用 setns 的函数。

```
Open  Save  Undo
sysinfo.h ×  Makefile ×  ipnetns.c ×
#define MNT_DETACH      0x00000002      /* Just detach
#endif /* MNT_DETACH */

#ifdef HAVE_SETNS   原来是#ifndef
static int setns(int fd, int nstype)
{
#ifdef __NR_setns
        return syscall(__NR_setns, fd, nstype);
#else
```

图 16.8 修改内核文件 ipnetns.c

3）Make

再次编译后成功。生成 IP 目录。

```
root@ubuntu: /Workstation/tools/iproute/iproute2-3.1.0
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_vlan.o iplink_vlan.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o link_veth.o link_veth.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o link_gre.o link_gre.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_can.o iplink_can.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvlan.o iplink_macvlan.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvtap.o iplink_macvtap.c
mipsel-linux-gcc -Wl,-export-dynamic  ip.o ipaddress.o ipaddrlabel.o iproute.o iprule.o
ipnetns.o rtm_map.o iptunnel.o ip6tunnel.o tunnel.o ipneigh.o ipntable.o iplink.o ipmadd
r.o ipmonitor.o ipmroute.o ipprefix.o iptuntap.o ipxfrm.o xfrm_state.o xfrm_policy.o xfr
m_monitor.o iplink_vlan.o link_veth.o link_gre.o iplink_can.o iplink_macvlan.o iplink_ma
cvtap.o ../lib/libnetlink.a ../lib/libutil.a  -lresolv ../lib/libnetlink.a ../lib/libuti
l.a -ldl -o ip
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o rtmon.o rtmon.c
mipsel-linux-gcc -Wl,-export-dynamic  rtmon.o ../lib/libnetlink.a ../lib/libutil.a  -lre
solv ../lib/libnetlink.a ../lib/libutil.a -ldl -o rtmon
make[1]: Leaving directory '/Workstation/tools/iproute/iproute2-3.1.0/ip'
root@ubuntu:/Workstation/tools/iproute/iproute2-3.1.0#
```

图 16.9 虚拟机中编译 CAN 工具成功

```
root@ubuntu: /Workstation/tools/iproute/iproute2-3.1.0
ar rcs libnetlink.a ll_map.o libnetlink.o
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o utils.o utils.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o rt_names.o rt_names.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o ll_types.o ll_types.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o ll_proto.o ll_proto.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o ll_addr.o ll_addr.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o inet_proto.o inet_proto.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o dnet_ntop.o dnet_ntop.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o dnet_pton.o dnet_pton.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o ipx_ntop.o ipx_ntop.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -fPIC   -c -o ipx_pton.o ipx_pton.c
ar rcs libutil.a utils.o rt_names.o ll_types.o ll_proto.o ll_addr.o inet_proto.o  dnet_n
top.o dnet_pton.o ipx_ntop.o ipx_pton.o
make[1]: Leaving directory '/Workstation/tools/iproute/iproute2-3.1.0/lib'
```

图 16.10 虚拟机中编译生成目标文件

```
root@ubuntu: /Workstation/tools/iproute/iproute2-3.1.0
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_vlan.o iplink_vlan.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o link_veth.o link_veth.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o link_gre.o link_gre.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_can.o iplink_can.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvlan.o iplink_macvlan.c
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o iplink_macvtap.o iplink_macvtap.c
mipsel-linux-gcc -Wl,-export-dynamic  ip.o ipaddress.o ipaddrlabel.o iproute.o iprule.o
ipnetns.o rtm_map.o iptunnel.o ip6tunnel.o tunnel.o ipneigh.o ipntable.o iplink.o ipmadd
r.o ipmonitor.o ipmroute.o ipprefix.o iptuntap.o ipxfrm.o xfrm_state.o xfrm_policy.o xfr
m_monitor.o iplink_vlan.o link_veth.o link_gre.o iplink_can.o iplink_macvlan.o iplink_ma
cvtap.o ../lib/libnetlink.a ../lib/libutil.a  -lresolv ../lib/libnetlink.a ../lib/libuti
l.a -ldl -o ip
mipsel-linux-gcc -D_GNU_SOURCE -O2 -Wstrict-prototypes -Wall -I../include -DRESOLVE_HOST
NAMES -DLIBDIR=\"/usr/lib/\" -DHAVE_SETNS   -c -o rtmon.o rtmon.c
mipsel-linux-gcc -Wl,-export-dynamic  rtmon.o ../lib/libnetlink.a ../lib/libutil.a  -lre
solv ../lib/libnetlink.a ../lib/libutil.a -ldl -o rtmon
make[1]: Leaving directory '/Workstation/tools/iproute/iproute2-3.1.0/ip'
root@ubuntu:/Workstation/tools/iproute/iproute2-3.1.0#
```

图 16.11 虚拟机中编译生成 ip 文件

4)拷备 ip 文件到开发板

将 ip 工具 copy 到开发板的文件系统下，这里拷备到/can_tools/下。

```
[root@Loongson:/can_tools]#tftp -r ip -g 193.169.2.215
ip                    100% |*******************************|   290k    0:00:00 ETA
```

5)在开发板上应用 IP link 的命令查看设备

```
 [root@Loongson:/can_tools]#./ip link
-/bin/sh: ./can_tools: Permission denied
[root@Loongson:/can_tools]#chmod u+x ip
[root@Loongson:/can_tools]#./ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN qlen 10
    link/can
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether aa:37:80:e3:d9:f9 brd ff:ff:ff:ff:ff:ff
4: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 00:0b:81:96:76:24 brd ff:ff:ff:ff:ff:ff
```

16.4.2 配置运行命令

配置, can 接口需要在 down 状态下才能配置

```
[root@Loongson:/can_tools]#./ip link set can0 down
[root@Loongson:/can_tools]#./ip link set can0 up type can bitrate 250000
sja1000_platform sja1000_platform.0: setting BTR0=0x23 BTR1=0x04
[root@Loongson:/can_tools]#./ip -details link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UNKNOWN qlen 10
    link/can
    can state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
    bitrate 250000 sample-point 0.857
    tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1
    sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
    clock 63000000
```

down 下的状态

```
[root@Loongson:/can_tools]#./ip link set can0 down
[root@Loongson:/can_tools]#./ip -details link show can0
2: can0: <NOARP,ECHO> mtu 16 qdisc pfifo_fast state DOWN qlen 10
    link/can
    can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
    bitrate 250000 sample-point 0.857
    tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1
    sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
    clock 63000000
```

启动

```
 [root@Loongson:/can_tools]# ifconfig can0 up
[root@Loongson:/can_tools]#ifconfig
can0          Link encap:UNSPEC    HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
              UP RUNNING NOARP    MTU:16    Metric:1
              RX packets:0 errors:0 dropped:0 overruns:0 frame:0
              TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:10
              RX bytes:0 (0.0 B)    TX bytes:0 (0.0 B)
              Interrupt:6
```

loopback 状态 loopback 状态，使用 loopback 模式可测试本机的 can 驱动是否正常工作.

```
[root@Loongson:/can_tools]#./ip link set can0 down
[root@Loongson:/can_tools]#./ip link set can0 type can bitrate 25000 loopback on
RTNETLINK answers: Operation not supported
[root@Loongson:/can_tools]# ./ip -details link show can0
2: can0: <NOARP,ECHO> mtu 16 qdisc pfifo_fast state DOWN qlen 10
    link/can
    can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
    bitrate 250000 sample-point 0.857
    tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1
    sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
    clock 63000000
```

回环模式不支持。

16.5 Socket CAN 在智龙上的测试-使用工具 canutils

can-utils 和 libsocketcan 可以使用 buildroot 编译。

```
Target packages > Libraries > Networking
[*] libsocketcan
  Target packages > Networking applications
[*] can-utils
```

由本系统采用的 busybox，这里重新下载 canutils-4.0.6.tar.bz2 编译。

16.5.1 安装 libsocketcan

1）下载 libsocketcan

canutils 编译需要 libsocketcan 库的支持，需要下载 libsocketcan。

下载地址：http://www.pengutronix.de/software/libsocketcan/download/ 笔者下载的是 libsocketcan 0.0.10

2）配置编译工具链

--host 是指定交叉工具链

```
./configure --host=mipsel-linux
```

3)编译 libsocketcan - make

```
root@ubuntu:/Workstation/tools/SocketCAN/libsocketcan-0.0.10# make
Making all in include
make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'
make    all-am
make[2]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'
make[2]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'
make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'
Making all in config
make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/config'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/config'
Making all in src
make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src'
  CC          libsocketcan.lo
  CCLD        libsocketcan.la
make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src'
make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10'
make[1]: Nothing to be done for 'all-am'.
make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10'
```

在'/Workstation/tools/SocketCAN/libsocketcan-0.0.10/下生成了文件 libsocketcan.lo、CCLD　　libsocketcan.la 。

16.5.2 安装 canutils

1）下载并配置 canutils-4.0.6。

下载地址：http://public.pengutronix.de/software/socket-can/canutils/ 。

这里下载 canutils-4.0.6.tar.bz2。

解压 canutils-4.0.6.tar.bz2，执行 configure 命令。(其中--host 是指定交叉工具链，--prefix 是指定库的生成位置(不用)，libsocketcan_LIBS 是指定 canconfig 需要链接的库，LDFLAGS 是指定外部库的路径，CPPFLAGS 是指定外部头文件的路径)

配置 canutils

```
./configure --host=mipsel-linux libsocketcan_LIBS=-lsocketcan
LDFLAGS="L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs "
libsocketcan_CFLAGS="-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include"
CFLAGS="-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include"
```

2）编译 make

```
# make
Making all in include
make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'
make    all-am
make[2]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'
make[2]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'
make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'
Making all in config
make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/config'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/config'
Making all in src
make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/src'
mipsel-linux-gcc -DHAVE_CONFIG_H -I. -I../include -I../include -I../include -DPF_CAN=29
-DAF_CAN=PF_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT
candump.o -MD -MP -MF .deps/candump.Tpo -c -o candump.o candump.c
```

```
mv -f .deps/candump.Tpo .deps/candump.Po
/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc
-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o candump candump.o
libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o
candump candump.o
mipsel-linux-gcc -DHAVE_CONFIG_H -I. -I../include -I../include -I../include -DPF_CAN=29
-DAF_CAN=PF_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT
cansend.o -MD -MP -MF .deps/cansend.Tpo -c -o cansend.o cansend.c
mv -f .deps/cansend.Tpo .deps/cansend.Po
/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc
-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansend cansend.o
libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o
cansend cansend.o
mipsel-linux-gcc -DHAVE_CONFIG_H -I. -I../include -I../include -I../include -DPF_CAN=29
-DAF_CAN=PF_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT
canecho.o -MD -MP -MF .deps/canecho.Tpo -c -o canecho.o canecho.c
mv -f .deps/canecho.Tpo .deps/canecho.Po
/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc
-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o canecho canecho.o
libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o
canecho canecho.o
mipsel-linux-gcc -DHAVE_CONFIG_H -I. -I../include -I../include -I../include -DPF_CAN=29
-DAF_CAN=PF_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT
cansequence.o -MD -MP -MF .deps/cansequence.Tpo -c -o cansequence.o cansequence.c
mv -f .deps/cansequence.Tpo .deps/cansequence.Po
/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc
-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansequence cansequence.o
libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o
cansequence cansequence.o
mipsel-linux-gcc -DHAVE_CONFIG_H -I. -I../include -I../include -I../include -DPF_CAN=29
-DAF_CAN=PF_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT
canconfig.o -MD -MP -MF .deps/canconfig.Tpo -c -o canconfig.o canconfig.c
mv -f .deps/canconfig.Tpo .deps/canconfig.Po
/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc
-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o canconfig canconfig.o
-L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs -lsocketcan
libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2
-o .libs/canconfig canconfig.o -L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs
/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs/libsocketcan.so -Wl,-rpath
-Wl,/usr/local/libsocketcan/lib
make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/src'
Making all in man
make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/man'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/man'
make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6'
make[1]: Nothing to be done for 'all-am'.
make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6'
```

在 SRC 目录下生成 candump cansend canecho cansequence 4 个文件。

./lib 下 canconfig 1 个文件。

3)将编译好的 cansend 等 5 个文件复制到对应开发板/can_tools 下。

运行 canconfig 时出错：

```
[root@Loongson:/can_tools]#./canconfig can0 up
./canconfig: error while loading shared libraries: libsocketcan.so.2: cannot open shared object file: No such file or
directory
```

需将 libsocketcan.so.2.2.1 拷贝到开发板的/lib 文件夹下，由于 libsocketcan.so.2 是链接文件，需要建立 libsocketcan.so.2.2.1 的快捷方式 libsocketcan.so.2。

```
[root@Loongson:/lib]#tftp -r libsocketcan.so.2.2.1 -g 193.169.2.215
[root@Loongson:/lib]#ln -s libsocketcan.so.2.2.1 libsocketcan.so.2
```

附录：

虽然程序编译时是优先使用指定的库 link,但是运行时，动态库还是从系统的位置找的.

因为想避免安装这个库，所需要将工程中动态库的地址手动加入到系统中。

主要有 3 种方法：

1. 用 ln 将需要的 so 文件链接到/usr/lib 或者/lib 这两个默认的目录下边

```
ln -s /where/you/install/lib/*.so /usr/lib
sudo ldconfig
```

2.修改 LD_LIBRARY_PATH

```
export LD_LIBRARY_PATH=/where/you/install/lib:$LD_LIBRARY_PATH
sudo ldconfig
```

3.修改/etc/ld.so.conf，然后刷新

```
vim /etc/ld.so.conf
add /where/you/install/lib

sudo ldconfig
```

16.5.3 使用 canutils

1) 配置 can0

```
[root@Loongson:/can_tools]#./canconfig can0 bitrate 250000
sja1000_platform sja1000_platform.0: setting BTR0=0x23 BTR1=0x04
can0 bitrate: 250000, sample-point: 0.857
```

2） 开启 / 重启 / 关闭 CAN 总线

```
[root@Loongson:/can_tools]#./canconfig can0 start
can0 state: ERROR-ACTIVE
[root@Loongson:/can_tools]#./canconfig can0 restart
Device is not in BUS_OFF, no use to restart
can0: failed to restart
[root@Loongson:/can_tools]#./canconfig can0 stop
can0 state: STOPPED
[root@Loongson:/can_tools]#
```

3) 查看 CAN 总线状态

```
[root@Loongson:/can_tools]#./canecho can0
interface-in = can0, interface-out = can0, family = 29, type = 3, proto = 1
read: Network is down
```

4）发送信息

cansend canX --identifier=ID + 数据

```
[root@Loongson:/can_tools]#./cansend can0 --identifier=0x123 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
interface = can0, family = 29, type = 3, proto = 1
write: Network is down
```

5)接收数据

candump canX

```
  [root@Loongson:/can_tools]#./candump can0
interface = can0, family = 29, type = 3, proto = 1
<0x001> [8] 00 01 02 03 04 05 06 07
<0x002> [8] 00 01 02 03 04 05 06 076)
```

使用滤波器接收 ID 匹配的数据

candump canX –-filter=ID:mask

```
[root@Loongson:/can_tools]#./candump can0 --filter=0x123:0x7ff
id: 0x00000123 mask: 0x000007ff
interface = can0, family = 29, type = 3, proto = 1
```

7）总结

至此，使用 Socket 方式的 CAN 总线驱动设计的就介绍完了，用户可以使用 Socket 套接字的方式，参照 canutils 的源码设计自己的应用程序。

16.6 编写 CAN 的 socket 收发测试程序 canapp

16.6.1 程序设计说明

下面具体介绍使用 SocketCAN 实现通信时使用的应用程序开发接口。

(1). 初始化

SocketCAN 中大部分的数据结构和函数在头文件 linux/can.h 中进行了定义。 CAN 总线套接字的创建采用标准的网络套接字操作来完成。网络套接字在头文件 sys/socket.h 中定义。 套接字的初始化方法如下：

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);//创建 SocketCAN 套接字
strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);//指定 can0 设备
addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr *)&addr, sizeof(addr)); //将套接字与 can0 绑定
```

(2). 数据发送

在数据收发的内容方面， CAN 总线与标准套接字通信稍有不同，每一次通信都采用 can_ frame 结构体将数据封装成帧。 结构体定义如下：

```
struct can_frame {
canid_t can_id;//CAN 标识符
__u8 can_dlc;//数据场的长度
__u8 data[8];//数据
};
```

can_id 为帧的标识符， 如果发出的是标准帧， 就使用 can_id 的低 11 位； 如果为扩展帧， 就使用 0～ 28 位。 can_id 的第 29、 30、 31 位是帧的标志位，用来定义帧的类型，定义如下：

```
#define CAN_EFF_FLAG 0x80000000U //扩展帧的标识
#define CAN_RTR_FLAG 0x40000000U //远程帧的标识
#define CAN_ERR_FLAG 0x20000000U //错误帧的标识，用于错误检查
```

数据发送使用 write 函数来实现。 如果发送的数据帧(标识符为 0x123)包含单个字节(0xAB)的数据，可采用如下方法进行发送：

```
struct can_frame frame;
frame.can_id = 0x123;//如果为扩展帧，那么 frame.can_id = CAN_EFF_FLAG | 0x123;
frame.can_dlc = 1; //数据长度为 1
frame.data[0] = 0xAB; //数据内容为 0xAB

int nbytes = write(s, &frame, sizeof(frame)); //发送数据
if(nbytes != sizeof(frame)) //如果 nbytes 不等于帧长度，就说明发送失败
	printf("Error\n!");
```

如果要发送远程帧(标识符为 0x123)，可采用如下方法进行发送：

```
struct can_frame frame;
frame.can_id = CAN_RTR_FLAG | 0x123;
write(s, &frame, sizeof(frame));
```

(3). 数据接收

数据接收使用 read 函数来完成，实现如下：

```
struct can_frame frame;
int nbytes = read(s, &frame, sizeof(frame));
```

当然， 套接字数据收发时常用的 send、 sendto、 sendmsg 以及对应的 recv 函数也都可以用于 CAN 总线数据的收发。

(4). 错误处理

当帧接收后，可以通过判断 can_id 中的 CAN_ERR_FLAG 位来判断接收的帧是否为错误帧。 如果为错误帧，可以通过 can_id 的其他符号位来判断错误的具体原因。

错误帧的符号位在头文件 linux/can/error.h 中定义。

(5). 过滤规则设置

在数据接收时，系统可以根据预先设置的过滤规则，实现对报文的过滤。过滤规则使用 can_filter 结构体来实现，定义如下:

```
struct can_filter {
        canid_t can_id;
        canid_t can_mask;
};
```

过滤的规则为：

接收到的数据帧的 can_id & mask == can_id & mask

通过这条规则可以在系统中过滤掉所有不符合规则的报文，使得应用程序不需要对无关的报文进行处理。在 can_filter 结构的 can_id 中，符号位 CAN_INV_FILTER 在置位时可以实现 can_id 在执行过滤前的位反转。

用户可以为每个打开的套接字设置多条独立的过滤规则，使用方法如下：

```
struct can_filter rfilter[2];
rfilter[0].can_id = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK; //#define CAN_SFF_MASK 0x000007FFU
rfilter[1].can_id = 0x200;
rfilter[1].can_mask = 0x700;
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));//设置规则
```

在极端情况下，如果应用程序不需要接收报文，可以禁用过滤规则。这样的话，原始套接字就会忽略所有接收到的报文。在这种仅仅发送数据的应用中，可以在内核中省略接收队列，以此减少 CPU 资源的消耗。禁用方法如下：

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0); //禁用过滤规则
```

通过错误掩码可以实现对错误帧的过滤， 例如：

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );
setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, err_mask, sizeof(err_mask));
```

(6). 回环功能设置

在默认情况下， 本地回环功能是开启的，可以使用下面的方法关闭回环/开启功能：

```
int loopback = 0; // 0 表示关闭, 1 表示开启( 默认)
setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

在本地回环功能开启的情况下，所有的发送帧都会被回环到与 CAN 总线接口对应的套接字上。 默认情况下，发送 CAN 报文的套接字不想接收自己发送的报文，因此发送套接字上的回环功能是关闭的。可以在需要的时候改变这一默认行为：

```
int ro = 1; // 0 表示关闭( 默认), 1 表示开启
setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS, &ro, sizeof(ro));
```

16.6.2 程序发送示例

http://blog.csdn.net/reille/article/details/49949651

```
/* 1. 报文发送程序 canappsend.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>

#ifndef PF_CAN
#define PF_CAN 29
```

```
#endif

#ifndef AF_CAN
#define AF_CAN PF_CAN
#endif

int main()
{
        int s, nbytes;
        struct sockaddr_can addr;
        struct ifreq ifr;
        struct can_frame frame[2] = {{0}};

        s = socket(PF_CAN, SOCK_RAW, CAN_RAW);//创建套接字
        strcpy(ifr.ifr_name, "can0" );
        ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备
        addr.can_family = AF_CAN;
        addr.can_ifindex = ifr.ifr_ifindex;
        bind(s, (struct sockaddr *)&addr, sizeof(addr));//将套接字与 can0 绑定
        //禁用过滤规则，本进程不接收报文，只负责发送
        setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
        //生成两个报文
        frame[0].can_id = 0x11;
        frame[0]. can_dlc = 1;
        frame[0].data[0] = 'Y';
        frame[1].can_id = 0x22;
        frame[1]. can_dlc = 1;
        frame[1].data[0] = 'N';
        //循环发送两个报文
        while(1)
        {
                nbytes = write(s, &frame[0], sizeof(frame[0])); //发送 frame[0]
                if(nbytes != sizeof(frame[0]))
                {
                        printf("Send Error frame[0]\n!");
                        break; //发送错误，退出
                }
                sleep(1);
                nbytes = write(s, &frame[1], sizeof(frame[1])); //发送 frame[1]
                if(nbytes != sizeof(frame[1]))
                {
                        printf("Send Error frame[1]\n!");
                        break;
                }
                sleep(1);
        }
        close(s);
        return 0;
}
```

16.6.3 程序接收示例

```
/* 2. 报文过滤接收程序 canappreceive.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>

#ifndef PF_CAN
```

```
#define PF_CAN 29
#endif

#ifndef AF_CAN
#define AF_CAN PF_CAN
#endif

int main()
{
        int s, nbytes;
        struct sockaddr_can addr;
        struct ifreq ifr;
        struct can_frame frame;
        struct can_filter rfilter[1];

        s = socket(PF_CAN, SOCK_RAW, CAN_RAW); //创建套接字
        strcpy(ifr.ifr_name, "can0" );
        ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备
        addr.can_family = AF_CAN;
        addr.can_ifindex = ifr.ifr_ifindex;
        bind(s, (struct sockaddr *)&addr, sizeof(addr)); //将套接字与 can0 绑定
        //定义接收规则，只接收标识符等于 0x11 的报文
        rfilter[0].can_id = 0x11;
        rfilter[0].can_mask = CAN_SFF_MASK;
        //设置过滤规则
        setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
        while(1)
        {
                nbytes = read(s, &frame, sizeof(frame)); //接收报文
                //显示报文
                if(nbytes > 0)
                {
                        printf("ID=0x%X DLC=%d data[0]=0x%X\n", frame.can_id,
                                frame.can_dlc, frame.data[0]);
                }
        }
        close(s);
        return 0;
}
```

16.6.4 发送接收测试

1）发送

以下用 iproute2 在开发板上操作打开 CAN 设备。

配置并启动设备

```
[root@Loongson:/can_tools]#./ip link set can0 up type can bitrate 250000
sja1000_platform sja1000_platform.0: setting BTR0=0x23 BTR1=0x04
```

或者用 canutils 打开 CAN 设备。

```
[root@Loongson:/can_tools]#./canconfig can0 bitrate 250000
sja1000_platform sja1000_platform.0: setting BTR0=0x23 BTR1=0x04
can0 bitrate: 250000, sample-point: 0.857
[root@Loongson:/can_tools]#./canconfig can0 start
can0 state: ERROR-ACTIVE
```

将 2 个开发板的 CANH、 CANL 联接在一起，分别下载 canappsend、canappreceive 并运行：

```
[root@Loongson:/app]#chmod u+x canappsend
[root@Loongson:/app]#./canappsend
```

接收程序过滤接收 ID=0x11 的报文：

```
[root@Loongson:/app]#chmod u+x canappreceive
[root@Loongson:/app]#./canappreceive
ID=0x11 DLC=1 data[0]=0x59
```

所以只收到以下报文：’Y’=0x59

```
frame[0].can_id = 0x11;
frame[0]. can_dlc = 1;
frame[0].data[0] = 'Y';
```

以下报文收不到：

```
frame[1].can_id = 0x22;
frame[1]. can_dlc = 1;
frame[1].data[0] = 'N';
```

17. LCD 应用开发

参考网址：

http://www.openloongson.org/forum.php?mod=viewthread&tid=141&extra=page%3D1。

17.1 硬件接口

开发板的扩展板上电路如图 17.1 示。

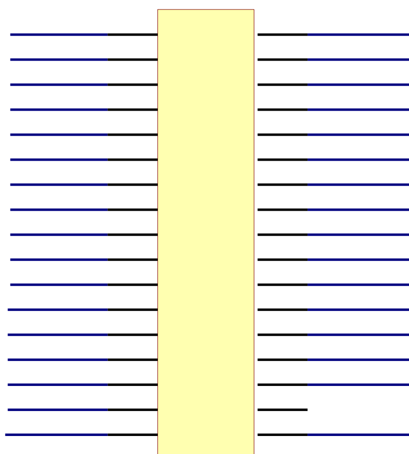


图 17.1 开发板扩展板 LCD 接口

17.2 GPIO 口操作函数

LCD 模块操作实际上是 GPIO 口的操作。以前在单片机中，可以简单置位、复位，在 linux 系统中，操作 GPIO 的方法有很多，可采用中级篇中的《GPIO 硬件编程方法》节，但最直接的方法是寄存器操作。

操作寄存器主要是对 GPIO 的相关寄存器操作，如图所示。在应用编程中，需要使用 mmap 系统调用。使用 mmap 映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,不必再调用 read,write 等系统调用。相关操作参考《9. 内核访问外设 I/O 资源》。

寄存器名称	地址	读/写(R/W)	功能描述	复位值
GPIO_CFG0	0xbfd0_10c0	R/W	GPIO0 配置寄存器 1 表示配置为 GPIO，0 表示无效	0xfc00_3f3f
GPIO_EN0	0xbfd0_10d0	R/W	GPIO0 方向寄存器 0 表示配置为输出，1 表示配置为输入	0xfc00_3f3f
GPIO_IN0	0xbfd0_10e0	R	GPIO0 输入寄存器	0x0
GPIO_OUT0	0xbfd0_10f0	R/W	GPIO0 输出寄存器 为 1 输出高 为 0 输出低	0x0

图 17.2 GPIO 相关寄存器

```
/*gpio_fun.c*/
#include<stdio.h>
#include<stdlib.h>
#include<stdint.h>
#include<string.h>
#include<unistd.h>
#include<sys/mman.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>



#define GPIO_CFG0   0xC0
#define GPIO_EN0    0xD0
#define GPIO_IN0    0xE0
#define GPIO_OUT0   0xF0

#define CBUS_FIRST0       0x1C0

unsigned char * map_base;
int    fd;

int gpio_open(void)
{
        fd = open("/dev/mem", O_RDWR|O_SYNC);
        if (fd == -1)    {
                return (-1);
        }

        /* °Ñ 0x1fd01000 ¿ªÊŒ0x1000×ÖœÚ£¬Ó³Éäµœmap_base */
        map_base = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x1fd01000);

        if (map_base == 0)    {
                printf("NULL pointer!\n");
        }
        return 0;

}

void gpio_close(void)
{
        close(fd);
        munmap(map_base, 0x1000);
}

void gpio_cfg_init(int gpio_number, int function)
{
        uint16_t addr_offset;
        uint16_t bit_offset;
        int index;
        uint32_t data_temp;

        index = gpio_number / 32;
        bit_offset = gpio_number % 32;
        addr_offset = GPIO_CFG0 + index*4;

        data_temp = *(volatile unsigned long *)(map_base+addr_offset);
        if(function == 0)
        {
                data_temp = data_temp & (~(1<<bit_offset));//clr
        }
        else
        {
                data_temp = data_temp | ((1<<bit_offset));//set
```

```
	}

	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
void gpio_en_init(int gpio_number, int function)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index;
	uint32_t data_temp;

	index = gpio_number / 32;
	bit_offset = gpio_number % 32;
	addr_offset = GPIO_EN0 + index*4;

	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	if(function == 0)
	{
		data_temp = data_temp & (~(1<<bit_offset));//clr
	}
	else
	{
		data_temp = data_temp | ((1<<bit_offset));//set
	}

	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
void gpio_func_init(int gpio_number, int function)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index,i;
	uint32_t data_temp;

	index = gpio_number / 32;
	bit_offset = gpio_number % 32;
	addr_offset = CBUS_FIRST0 + index*4;

	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	for(i=1;i<6;i++)
	{
		data_temp = *(volatile unsigned long *)(map_base+addr_offset+(i-1)*0x10);
		if(i == function)//set
		{
			data_temp = data_temp | ((1<<bit_offset));//set
		}
		else//clr
		{
			data_temp = data_temp & (~(1<<bit_offset));//clr
		}
		*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
	}
}

void gpio_set(int gpio_number)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index;
	uint32_t data_temp;

	index = gpio_number / 32;
	bit_offset = gpio_number % 32;
	addr_offset = GPIO_OUT0 + index*4;

	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
```

```
	{
		data_temp = data_temp | ((1<<bit_offset));//set
	}

	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
uint32_t gpio_get(int gpio_number)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index;
	uint32_t data_temp;

	index = gpio_number / 32;
	bit_offset = gpio_number % 32;
	addr_offset = GPIO_IN0 + index*4;

	data_temp = *(volatile unsigned long *)(map_base+addr_offset);

	return	(data_temp >> bit_offset)&0x01;
}
void gpio_clr(int gpio_number)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index;
	uint32_t data_temp;

	index = gpio_number / 32;
	bit_offset = gpio_number % 32;
	addr_offset = GPIO_OUT0 + index*4;

	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	{
		data_temp = data_temp & (~(1<<bit_offset));//clr
	}

	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
uint32_t data_get(int gpio_start_number, int len)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index,i;
	uint32_t data_temp;

	index = gpio_start_number / 32;
	bit_offset = gpio_start_number % 32;
	addr_offset = GPIO_IN0 + index*4;
	if((bit_offset+len)>32)
	{
		len = 32 - bit_offset;
	}
	data_temp = *(volatile unsigned long *)(map_base+addr_offset);

	for( i=bit_offset+len;i<32;i++)
	{
		data_temp = data_temp & (~(1<<i));//clr
	}

	return data_temp>>bit_offset;
}
void data_write(int gpio_start_number, int len ,uint32_t data)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index,i;
```

```
	uint32_t data_temp;

	index = gpio_start_number / 32;
	bit_offset = gpio_start_number % 32;
	addr_offset = GPIO_OUT0 + index*4;
	if((bit_offset+len)>32)
	{
		len = 32 - bit_offset;
	}
	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	{
		for( i=0;i<len;i++)
		{
			if(0x00 == (0x01&(data>>i))){
				data_temp = data_temp & (~(1<<(bit_offset+i)));//clr
			}
			else{
				data_temp = data_temp | (1<<(bit_offset+i)) ;//set
			}
		}
	}
	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
void gpio_cfg_data(int gpio_start_number, int len ,uint32_t data)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index,i;
	uint32_t data_temp;

	index = gpio_start_number / 32;
	bit_offset = gpio_start_number % 32;
	addr_offset = GPIO_CFG0 + index*4;
	if((bit_offset+len)>32)
	{
		len = 32 - bit_offset;
	}
	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	{
		for( i=0;i<len;i++)
		{
			if(0x00 == data){
				data_temp = data_temp & (~(1<<(bit_offset+i)));//clr
			}
			else if(0x01 == data){
				data_temp = data_temp | (1<<(bit_offset+i)) ;//set
			}
		}
	}
	*(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
void gpio_en_data(int gpio_start_number, int len ,uint32_t data)
{
	uint16_t addr_offset;
	uint16_t bit_offset;
	int index,i;
	uint32_t data_temp;

	index = gpio_start_number / 32;
	bit_offset = gpio_start_number % 32;
	addr_offset = GPIO_EN0 + index*4;
	if((bit_offset+len)>32)
	{
		len = 32 - bit_offset;
	}
	data_temp = *(volatile unsigned long *)(map_base+addr_offset);
	{
```

```
                for( i=0;i<len;i++)
                {
                        if(0x00 == data){
                                data_temp = data_temp & (~(1<<(bit_offset+i)));//clr
                        }
                        else if(0x01 == data){
                                data_temp = data_temp | (1<<(bit_offset+i)) ;//set
                        }
                }
        }
        *(volatile unsigned long *)(map_base+addr_offset)=data_temp;
}
/* Usage:
  * ./gpio_fun cfg    [gpio_num] [0] //  用法
  * ./gpio_fun en    [gpio_num] [0]
  * ./gpio_fun out    [gpio_num] [0]
  * ./gpio_fun in    [gpio_num] [0]
  * ./gpio_fun dataout    [gpio_num] [len] data//len bit
  * ./gpio_fun datain    [gpio_num] [len] data// len bit
  * ./gpio_fun cfgdata    [gpio_num] [len] 0/1// 8 bit
  * ./gpio_fun endata    [gpio_num] [len] 0/1// 8 bit
  * ./gpio_fun function    [gpio_num] functionnum
  */
int main(int argc, char *argv[])
{
        unsigned int buf[3];
        if(argc<3) {
                printf("gpio_fun cfg/en/out/in 50 0\r\n");
                printf("gpio_fun dataout 58 6 85\r\n");
                printf("gpio_fun datain 58 6\r\n");
                printf("gpio_fun cfgdata 58 6 0/1 \r\n");
                printf("gpio_fun endata 58 6 0/1 \r\n");
                printf("gpio_fun function 4 85 ;set gpio85 to function 4 \r\n");
                return (-2);
        }

        if(gpio_open()<0)
                return (-1);


        /* gpio_num */
       buf[0] = strtoul(argv[2], NULL, 0);

        /* function    */
if(argc>=4)
{
        buf[1] = strtoul(argv[3], NULL, 0);
}
        /* data    */
if(argc==5)
{
        buf[2] = strtoul(argv[4], NULL, 0);
}
       if (strcmp(argv[1], "cfg") == 0)
       {
                gpio_cfg_init(buf[0], buf[1]);
       }
       else if (strcmp(argv[1], "en") == 0)
       {
                gpio_en_init(buf[0], buf[1]);
        }
       else if (strcmp(argv[1], "out") == 0)
       {
                if(1 == buf[1]){
                        gpio_set(buf[0]);
                }
                else if(0 == buf[1]){
```

```
                gpio_clr(buf[0]);
            }
        }
        else if (strcmp(argv[1], "in") == 0)
        {
            printf("gpio%d = %d\r\n",buf[0], gpio_get(buf[0]) );
        }
        else if (strcmp(argv[1], "dataout") == 0)
        {
            if(argc>=5)
            {
                data_write(buf[0],buf[1],buf[2]);
            }
            else
            {
                printf("gpio_fun argc<5 !\r\n" );
            }
        }
        else if (strcmp(argv[1], "datain") == 0)
        {
            printf("gpio%d read %d bit= 0x%02x\r\n",buf[0], buf[1], data_get(buf[0],buf[1]));
        }
        else if (strcmp(argv[1], "cfgdata") == 0)
        {
            if(argc>=5)
            {
                gpio_cfg_data(buf[0],buf[1],buf[2]);
            }
            else
            {
                printf("gpio_fun argc<5 !\r\n" );
            }
        }
        else if (strcmp(argv[1], "endata") == 0)
        {
            if(argc>=5)
            {
                gpio_en_data(buf[0],buf[1],buf[2]);
            }
            else
            {
                printf("gpio_fun argc<5 !\r\n" );
            }
        }
        else if (strcmp(argv[1], "function") == 0)
        {
            if(argc>=4)
            {
                gpio_func_init(buf[0],buf[1]);
            }
            else
            {
                printf("gpio_fun argc<4 !\r\n" );
            }
        }
        gpio_close();
        return (0);
}
```

代码中，main 函数测试用，测试成功后，将 main 改成其它名字。

17.3 LCD 操作

这里使用 LCD 是来自于

https://item.taobao.com/item.htm?spm=a1z10.5-c.w4002-1891666121.32.hqEYu4&id=251

42720480。

使用控制器 NT35310 ，屏幕大小为 480*320。

直接在驱动例程上进行修改，将 GPIO 口的操作换成 7.3 节的 GPIO 口操作的函数，还延时函数的替换。代码太长，这里不再列出。这里只说明初始化过程。

首先从 0x00 处读取 ID，如果读不到，再从 0xD3 处读判断是否 9341，如果不是再从 0xBF 读判断是否 6804，如果不是再从 0xD4 判断是否 35310，如果不是再从 0XDA00 读判断是否 NT35510，如果不是再从 0XA1 读判断是否 SSD1963。

```
	lcddev.id = LCD_ReadReg(0x0000);
	if(lcddev.id<0XFF||lcddev.id==0XFFFF||lcddev.id==0X9300)//读到 ID 不正确,新增 lcddev.id==0X9300 判断，因为 9341 在未被复位的情况下会被读成 9300
	{
		//尝试 9341 ID 的读取
		LCD_WR_REG(0XD3);
		LCD_RD_DATA();			//dummy read
		LCD_RD_DATA();		//读到 0X00
		lcddev.id=LCD_RD_DATA();	//读取 93
		lcddev.id<<=8;
		lcddev.id|=LCD_RD_DATA();	//读取 41
		if(lcddev.id!=0X9341)	//非 9341,尝试是不是 6804
		{
			LCD_WR_REG(0XBF);
			LCD_RD_DATA();		//dummy read
			LCD_RD_DATA();		//读回 0X01
			LCD_RD_DATA();		//读回 0XD0
			lcddev.id=LCD_RD_DATA();//这里读回 0X68
			lcddev.id<<=8;
			lcddev.id|=LCD_RD_DATA();//这里读回 0X04
			if(lcddev.id!=0X6804) //也不是 6804,尝试看看是不是 NT35310
			{
				LCD_WR_REG(0XD4);
				LCD_RD_DATA();			//dummy read
				LCD_RD_DATA();			//读回 0X01
				lcddev.id=LCD_RD_DATA();	//读回 0X53
				lcddev.id<<=8;
				lcddev.id|=LCD_RD_DATA();	//这里读回 0X10
				if(lcddev.id!=0X5310)		//也不是 NT35310,尝试看看是不是 NT35510
				{
					LCD_WR_REG(0XDA00);
					LCD_RD_DATA();		//读回 0X00
					LCD_WR_REG(0XDB00);
					lcddev.id=LCD_RD_DATA();//读回 0X80
					lcddev.id<<=8;
					LCD_WR_REG(0XDC00);
					lcddev.id|=LCD_RD_DATA();//读回 0X00
					if(lcddev.id==0x8000)lcddev.id=0x5510;//NT35510 读回的 ID 是 8000H,为方便区分,我们强制设置为 5510
					if(lcddev.id!=0X5510)		//也不是 NT5510,尝试看看是不是 SSD1963
					{
						LCD_WR_REG(0XA1);
						lcddev.id=LCD_RD_DATA();
						lcddev.id=LCD_RD_DATA();	//读回 0X57
						lcddev.id<<=8;
						lcddev.id|=LCD_RD_DATA();	//读回 0X61
						if(lcddev.id==0X5761)lcddev.id=0X1963;//SSD1963 读回的 ID 是 5761H,为方便区分,我们强制设置为 1963
					}
				}
			}
		}
	}
```

```
    printf(" LCD ID:%x\r\n",lcddev.id); //打印 LCD ID
```

最终运行结果为：LCD 控制器为 NT35310。

```
[root@Loongson:/app]#./testlcd
Init LCD
 LCD ID:5310
DRIVER GET LCD ID:0x5310
ili9341.c-Init_ili9341():ILI9341 init Successful!
```

17.4 编写 Makefile

本应用程序中使用的文件结构如下图示。

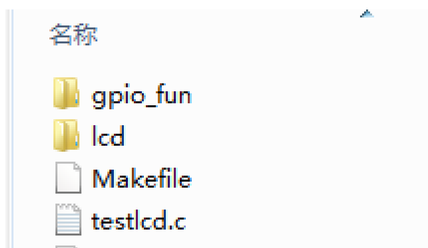


图 17.3 LCD 使用相关文件目录

其中 gpio_fun 中为 GPIO 口的操作函数。Lcd 中为 LCD 的驱动。

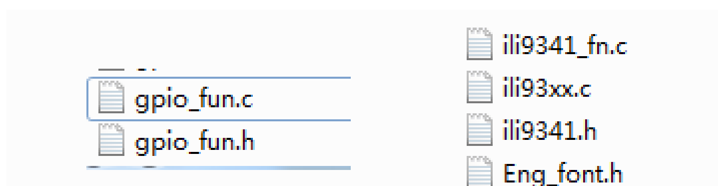


图 17.4 LCD 操作文件

下面编写通用的 Makefile，指明了头文件的位置，目标文件，并指定输出文件 testlcd。

```
# 指令编译器和选项
CC=mipsel-linux-gcc
CFLAGS=-Wall -std=gnu99 -Wstrict-prototypes


# 目标文件
TARGET=testlcd
SRCS = testlcd.c \
   ./gpio_fun/gpio_fun.c \
   ./lcd/ili9341_fn.c\
   ./lcd/ili93xx.c

INC = -I./gpio_fun -I./lcd

OBJS = $(SRCS:.c=.o)

$(TARGET):$(OBJS)
	@echo TARGET:$@
	@echo OBJECTS:$^
	$(CC) -o $@ $^

clean:
	rm -rf $(TARGET) $(OBJS)

%.o:%.c
	$(CC) $(CFLAGS) $(INC) -o $@ -c $<
```

17.5 代码及运行结果

17.5.1 显示字符

```
LCD_Clear(BLUE); //设置背景色
LCD_ShowString(30,40,400,24,24,"OPEN LOONGSON Smart Loong V2.0^_^");
LCD_ShowString(30,70,200,16,16,"TFTLCD TEST ");
LCD_ShowString(30,90,200,16,16,"sundm75");
LCD_ShowString(30,110,200,16,16,lcd_id);        //显示 LCD ID
LCD_ShowString(30,130,200,12,12,"2017/5/24");
```

17.5.2 显示图片

用 Image2Lcd 转换图片。

（1）设置输出数据格式 C 语言数组

（2）设置输出灰度 16 位真彩色

（3）设置最大宽度高度不得大于 LCD 的宽、高（320、480）。

（4）最后数据头文件 picture.h。

图 17.5 图片到数据转换操作界面

代码如下：

```
/*    画 picture */
/************************************************************************
* 名      称：void lcd_draw_picture(u16 StartX,u16 StartY,u16 EndX,u16 EndY,u16 *picture)
* 功      能：在指定座标范围显示一副图片
* 入口参数：StartX       行起始座标
*               StartY       列起始座标
*               EndX          行结束座标
*               EndY          列结束座标
                picture          图片头指针
* 出口参数：无
* 说      明：图片取模格式为水平扫描，16 位颜色模式（RGB565)
* 调用方法：lcd_draw_picture(0,0,100,100,(u16*)demo);
************************************************************************/
```

```
void lcd_draw_picture(u16 start_x, u16 start_y, u16 end_x, u16 end_y, u16 *picture)
{
	u32 length;
	u32 i;
	u16 x, y;

	x = start_x;
	y = start_y;

	length = (end_x - start_x + 1) * (end_y - start_y + 1) ;
	for (i = 0; i < length; i++)
	{
		LCD_Fast_DrawPoint(x, y, *picture++);

		y++;
		if(y > end_y)
		{
			x++;
			y = start_y;
		}
	}
}

/*-------------测试 LCD 函数，在当前 LCD 屏幕中心画出 PictureAddr 图片-------------*/
void DrawPicture_Center(u16 *PictureAddr)
{

	u16 PictureWidth;
	u16 PictureHeight;
	u16 * picuturepoint;

	picuturepoint=PictureAddr;

	PictureHeight = picuturepoint[1];
	PictureWidth  = picuturepoint[2];

	lcd_draw_picture((480-PictureWidth+1)/2, (320-PictureHeight+1)/2, ((480+PictureWidth+1)/2)-1,
				((320+PictureHeight+1)/2)-1, (u16 *)(PictureAddr + 3));
}
```

17.5.3 显示汉字

用 字模 III 增强版 ,生成字模的汉字库。

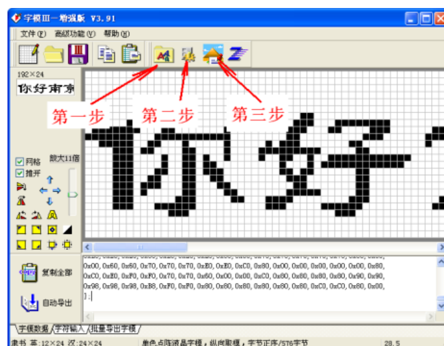


图 17.6 生成字模汉字库 1

第一步：

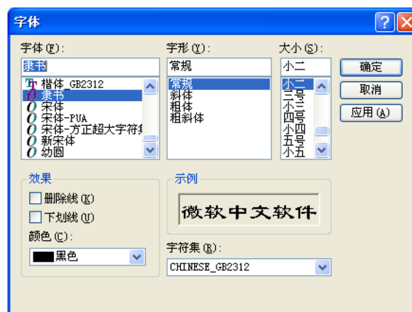


图 17.7 生成字模汉字库 2

第二步：

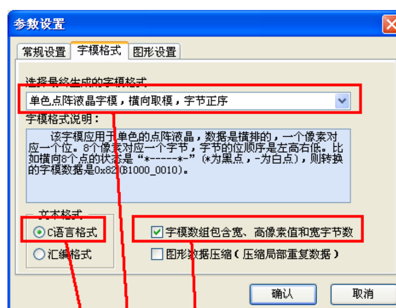


图 17.8 生成字模汉字库 3

汉字显示软件操作：

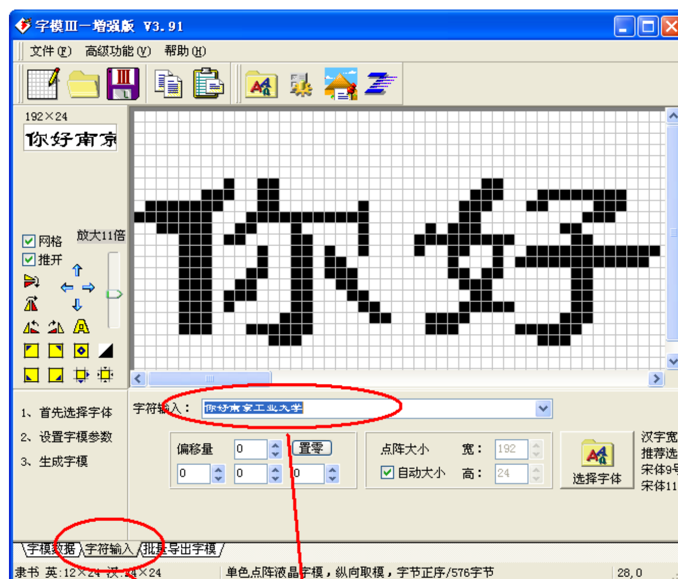


图 17.9 生成字模汉字库 4

第三步，生成字模数据

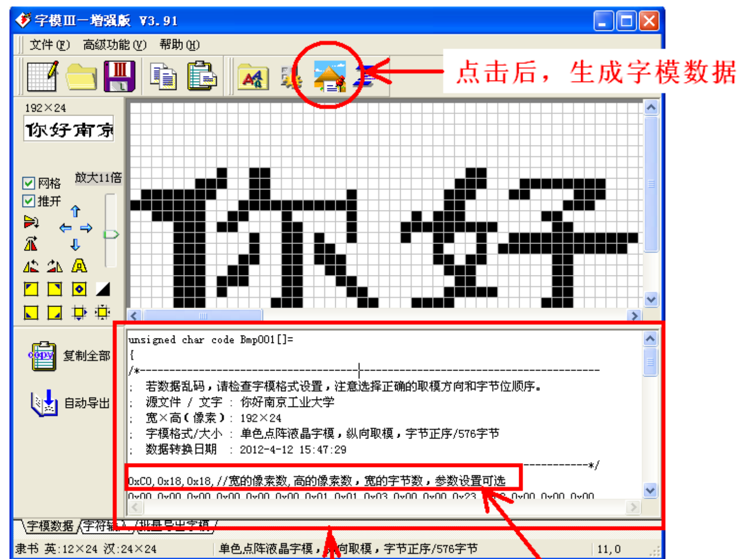


图 17.10 生成字模汉字库 5

代码如下：

```
/*****************************************************************************
* 名      称：void lcd_draw_hanzi(u16 x,u16 y,u8 *hanziaddr,u16 charColor,u16 bkColor)
* 功      能：在指定座标显示一个由字模软件生成的汉字字符
* 入口参数：x              行座标
*           y              列座标
            hanziaddr    汉字头指针
*           charColor    字符的颜色
*           bkColor      字符背景颜色
* 出口参数：无
* 说      明：显示范围限定为汉字头文件中，用横向取模的单色点阵液晶字模，横向取模，字节正序
* 调用方法：lcd_draw_hanzi(10,10,(u8*)Str_table1,White, HYALINE);
*****************************************************************************/
void lcd_draw_hanzi(u16 x,u16 y,u8 *hanziaddr,u16 charColor,u16 bkColor)    //
{
  u16 i=0;
  u16 j=0;
  u16 m=0;
  u8 width,high, wbyte;

  u8 *hanzipoint = hanziaddr;
  u8 tmp_char = 0;

  width = *hanzipoint++;
  high = *hanzipoint++;
  wbyte = *hanzipoint++;
  if(HYALINE == bkColor)
  {
    for (i=0;i<(high);i++)
    {
      for (j=0;j<wbyte;j++)
      {
        tmp_char=*hanzipoint++;
        for(m=0;m<8;m++)
        {
          if ( (tmp_char >>7-m) & 0x01 == 0x01)
          {
            POINT_COLOR = charColor;//  字符颜色
```

```
                    LCD_DrawPoint(x+j*8+m,y+i);
                }
                else
                {
                  // do nothing //  透明背景
                }
            }
          }
      }
   }
   else
   {
      for (i=0;i<(high);i++)
      {
         for (j=0;j<wbyte;j++)
         {
            tmp_char=*hanzipoint++;
            for(m=0;m<8;m++)
            {
               if ( (tmp_char >>7-m) & 0x01 == 0x01)
                {
                  POINT_COLOR = charColor;//  字符颜色
                    LCD_DrawPoint(x+j*8+m,y+i);
                }
                else
                {
                  POINT_COLOR = bkColor;//  背景颜色
                    LCD_DrawPoint(x+i,y+i-8);
                }
            }
          }
      }
   }
}
```

17.5.3 画线、圆圈

```
LCD_DrawLine(20,20, 20,300); //画线
LCD_DrawLine(410,30, 410,290);//画线
LCD_DrawRectangle(3,15,450,310); //画矩形
LCD_ShowString(30,40,400,24,24,"Picture 214 X 300");//写字符
```

主程序 testlcd.c：

```
int main(int argc,char * argv[])
{
     u8 x=0;
     u8 lcd_id[64];                  //存放 LCD ID 字符串
    gpio_open();
    Init_ili9341();
    POINT_COLOR=RED;
    sprintf((char*)lcd_id,"LCD ID:%04X",lcddev.id);//将 LCD ID 打印到 lcd_id 数组。

    while(1)
    {
         switch(x)
         {
                case 0:LCD_Clear(WHITE);
                LCD_Display_Dir(1);                  //横屏
                POINT_COLOR = BLACK;
                BACK_COLOR = WHITE;
         LCD_ShowString(30,40,400,24,24,"OPEN LOONGSON Smart Loong V2.0^_^");
         LCD_ShowString(30,70,200,16,16,"TFTLCD TEST ");
         LCD_ShowString(30,90,200,16,16,"sundm75");
```

```
		LCD_ShowString(30,110,200,16,16,lcd_id);		//显示 LCD ID

		LCD_ShowString(30,130,200,12,12,"2017/5/24");

			break;
			case 1://LCD_Clear(BLACK);
			POINT_COLOR = WHITE;
			BACK_COLOR = BLACK;
		LCD_ShowString(30,40,400,24,24,"OPEN LOONGSON Smart Loong V2.0^_^");
		LCD_ShowString(30,70,200,16,16,"TFTLCD TEST ");
		LCD_ShowString(30,90,200,16,16,"sundm75");
		LCD_ShowString(30,110,200,16,16,lcd_id);		//显示 LCD ID

		LCD_ShowString(30,130,200,12,12,"2017/5/24");

			break;
			case 2://LCD_Clear(BLUE);
			POINT_COLOR = RED;
			BACK_COLOR = BLUE;
		LCD_ShowString(30,40,400,24,24,"OPEN LOONGSON Smart Loong V2.0^_^");
		LCD_ShowString(30,70,200,16,16,"TFTLCD TEST ");
		LCD_ShowString(30,90,200,16,16,"sundm75");
		LCD_ShowString(30,110,200,16,16,lcd_id);		//显示 LCD ID

		LCD_ShowString(30,130,200,12,12,"2017/5/24");

			break;
			case 3://LCD_Clear(RED);
			POINT_COLOR = BLUE;
			BACK_COLOR = RED;
		LCD_ShowString(30,40,400,24,24,"OPEN LOONGSON Smart Loong V2.0^_^");
		LCD_ShowString(30,70,200,16,16,"TFTLCD TEST ");
		LCD_ShowString(30,90,200,16,16,"sundm75");
		LCD_ShowString(30,110,200,16,16,lcd_id);		//显示 LCD ID

		LCD_ShowString(30,130,200,12,12,"2017/5/24");

			break;
			case 4:
				LCD_Display_Dir(0);			//竖屏
				LCD_Clear(BLUE);
				lcd_draw_hanzi(5,420,(u8*)Hanzi, WHITE, HYALINE);
				LCD_Display_Dir(1);			//横屏
				DrawPicture_Center(gImage_picture);
				msleep(500);
			break;
			case 5:
				LCD_Display_Dir(1);			//横屏
				POINT_COLOR = RED;
				BACK_COLOR = BLUE;
				LCD_DrawLine(20,20, 20,300);
				POINT_COLOR = GREEN;
				LCD_DrawLine(405,30, 405,290);
				LCD_DrawRectangle(3,15,450,310);
				LCD_Display_Dir(0);			//竖屏
				LCD_ShowString(30,45,400,29,24,"Smart Loong V2.0");
				LCD_ShowString(100,20,400,4,24,"Loongson");
				POINT_COLOR = RED;
				LCD_Draw_Circle(30,15,10);
				LCD_Draw_Circle(50,15,10);
				LCD_Draw_Circle(70,15,10);
				LCD_Draw_Circle(90,15,10);
				LCD_Draw_Circle(110,15,10);
				LCD_Draw_Circle(130,15,10);
				LCD_Draw_Circle(150,15,10);
				LCD_Draw_Circle(180,15,10);
```

```
                        LCD_Draw_Circle(200,15,10);
                        LCD_Draw_Circle(220,15,10);
                        LCD_Draw_Circle(240,15,10);
                        LCD_Draw_Circle(260,15,10);
                        LCD_Draw_Circle(280,15,10);
                        LCD_Draw_Circle(300,15,10);
                        msleep(800);
                    break;

            }
            x++;
            if(x==6)x=0;
            msleep(300);
    }
        gpio_close();



    return 0;
}
```

程序运行结果：

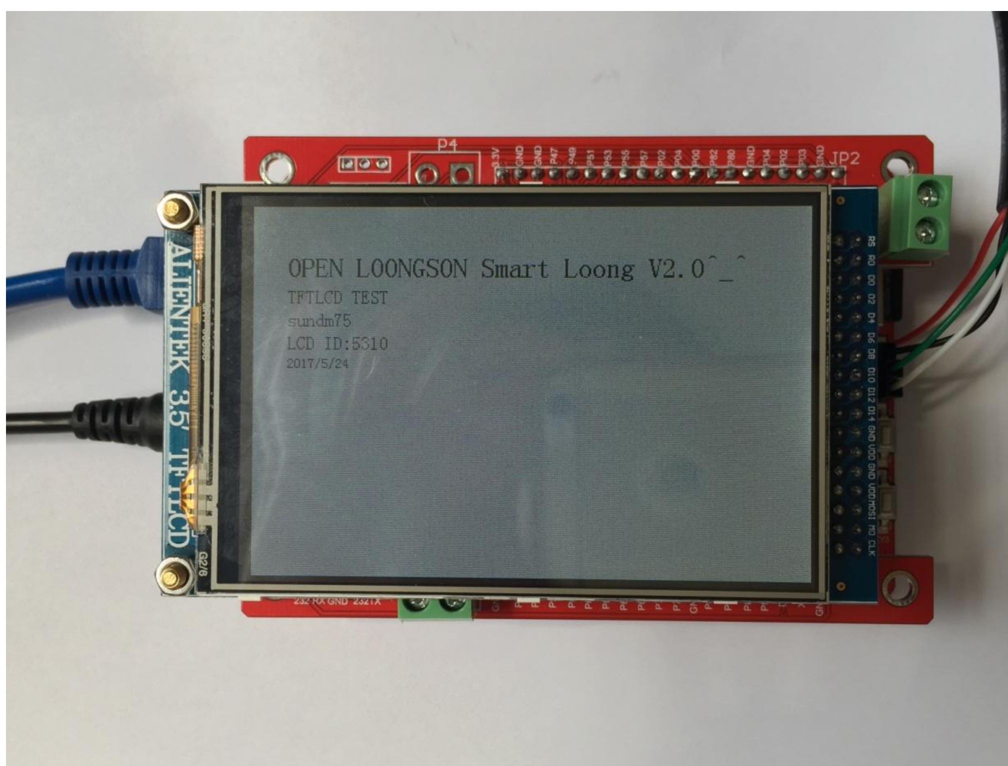


图 17.11 LCD 显示不同大小字符

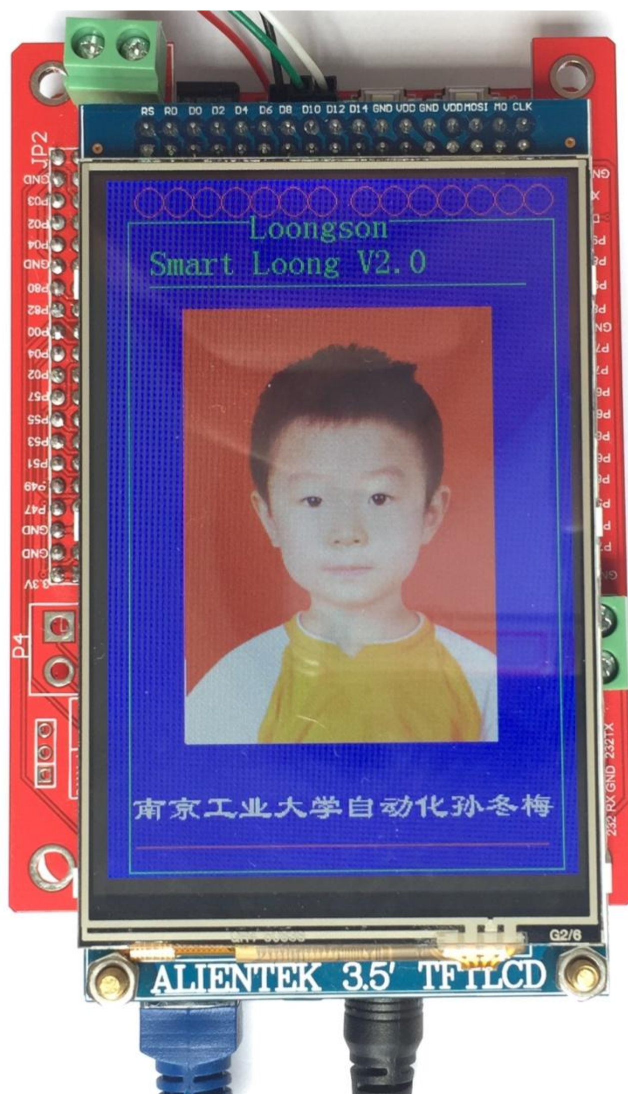


图 17.12 LCD 显示图片

18. ADC 应用开发

18.1 配置 ADC 驱动

```
Devices Drivers    --->
  <*>     Hardware Monitoring support     --->
      <*>     loongson1 built-in ADC
```

配置如下选项。

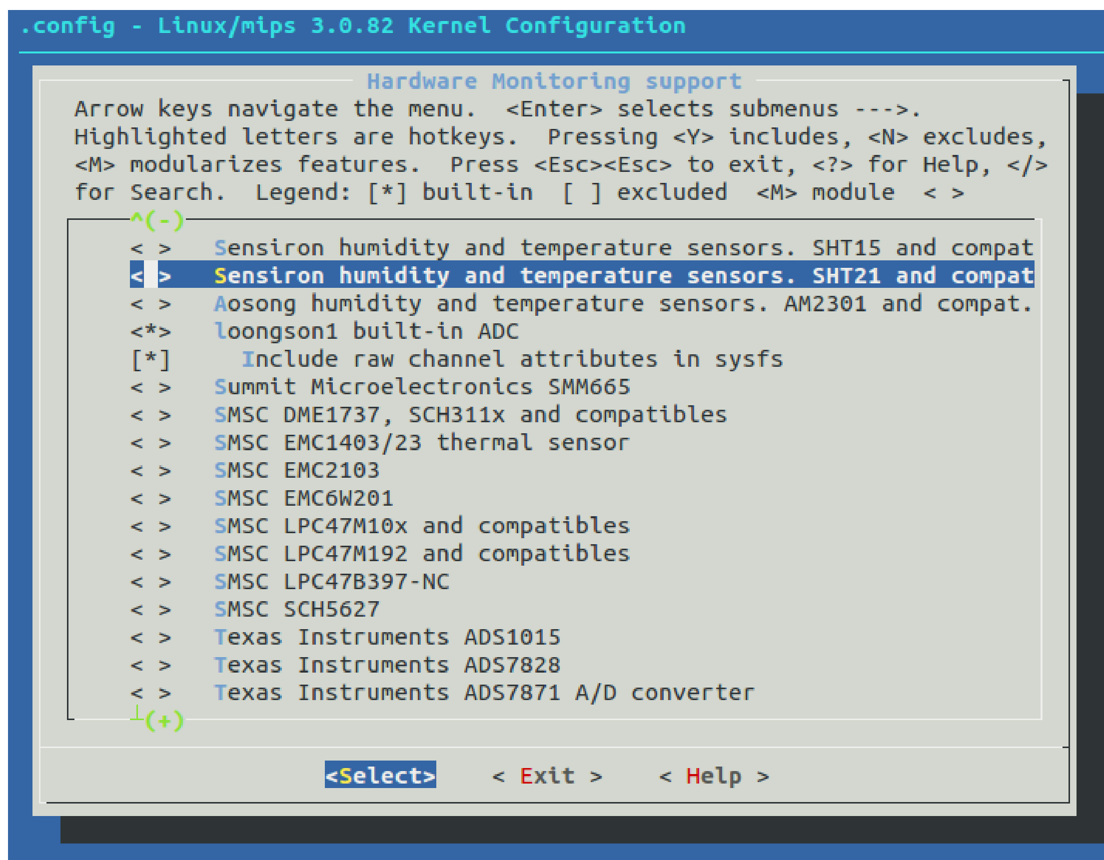


图 18.1 内核配置 ADC

18.2 硬件管脚分配

使用管脚 GPIO85，86，87，88

78	ADC_D0	85		
79	ADC_D1	86		
80	ADC_XP	87		
81	ADC_YP	88		

图 18.2 开发板 ADC 引脚复用

18.3 应用测试

重启系统后，进入 ADC 目录。一共有 4 个 adcN_raw（N=0~3），分别对应 ADC_D0,ADC_D1, ADC_XP,ADC_YP。

```
[root@Loongson:/]#cd /sys/class/hwmon/hwmon0/device/ [root@Loongson:/sys/devices/platform/ls1x-hwmon]#ls
adc0_raw    adc2_raw    driver      in0_input   modalias    subsystem
adc1_raw    adc3_raw    hwmon       in0_label   power       uevent
```

读取 ADC_D0(GPIO85)输入的值，D0 悬空。

```
[root@Loongson:/sys/devices/platform/ls1x-hwmon]#cat adc0_raw
977
```

将开发板上 D0 脚接入 GND，再执行命令

```
[root@Loongson:/sys/devices/platform/ls1x-hwmon]#cat adc0_raw    //D0 管脚接 GND
0
```

将开发板上 D0 脚接入 3V3，再执行命令

```
[root@Loongson:/sys/devices/platform/ls1x-hwmon]#cat adc0_raw    //D0 管脚接 GND3V3
1023
```

其它 ADCN 可同样操作。

18.4 应用层编程

应用层使用系统调用即可以控制。

```
/**************************************************************************
 Loongson1s1c hwmon-adc driver test
  device node is "/sys/class/hwmon/hwmon0/device/adcN_raw"
 **************************************************************************/
/*testadc.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/fcntl.h>

int main(int argc, char *argv[])
{
    int fd = -1, ret;
    int adc_ch, value;
    char buffer[5];
    char dev_name[50];

    if ( argc < 2 )
    {
        printf("usage adc_test chanel(0~7)\n");
        exit(1);
    }
    else
    {
        adc_ch = atoi(argv[1]);
    }

    if ( (adc_ch >= 0) && (adc_ch <= 7) )
    {
        sprintf(dev_name, "/sys/class/hwmon/hwmon0/device/adc%d_raw\0", adc_ch);
        printf("dev name %s\n", dev_name);

        if ((fd = open(dev_name, O_RDONLY)) < 0)
        {
            perror("open error");
            exit(1);
        }
    }
    else
```

```
    {
        printf("adc_ch is (0~7)\n");
        exit(1);
    }

    ret = read(fd, buffer, 4);
    if (ret < 0)
    {
        perror("read error");
        exit(1);
    }

    value = atoi(buffer);
    printf("ADC %d current value is %d\n", adc_ch, value);

    close(fd);
    return 0;
}
```

编译后，下载到开发板，运行命令：./testadc 0， 后面 0 表示使用 ADC_D0。

```
[root@Loongson:/]#./testadc 0
dev name /sys/class/hwmon/hwmon0/device/adc0_raw
ADC 0 current value is 0    //D0 管脚接 GND
[root@Loongson:/]#./testadc 0
dev name /sys/class/hwmon/hwmon0/device/adc0_raw
ADC 0 current value is 1023       //D0 管脚接 3V3
```

19. 内核访问外设 I/O 资源

19.1 MIPS 的内存映射

在 32 位 MIPS 体系结构下，最多可寻址 4GB 地址空间。这 4GB 空间的分配是怎样的呢？看下面这张图：

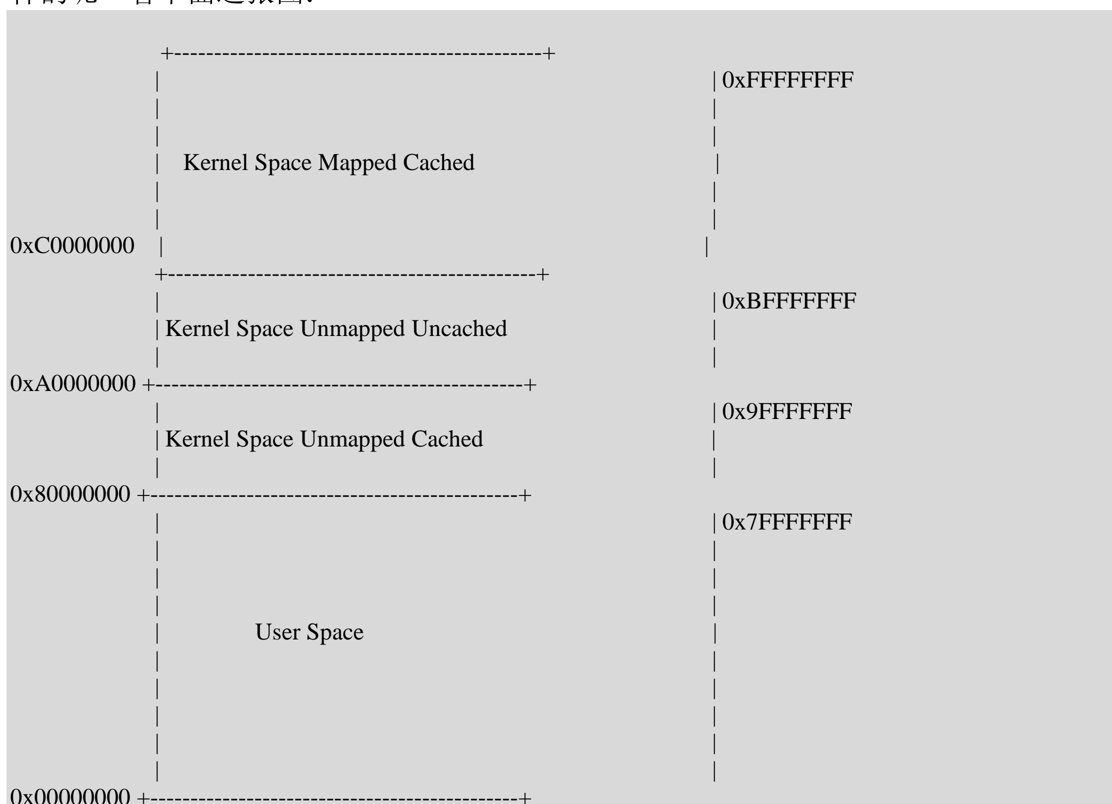


图 19.1 MIPS 体系结构逻辑寻址空间分布

上图是 MIPS 处理器的逻辑寻址空间分布图。我们看到，2GB 以下的地址空间，也就是从 0x00000000 到 0x7FFFFFFF 的这一段空间，为 User Space，可以在 User Mode 下访问，当然，在 Kernel Mode 下也是可以访问的。程序在访问 User Space 的内存时，会通过 MMU 的 TLB，映射到实际的物理地址上。也就是说，这一段逻辑地址空间和物理地址空间的对应关系，是由 MMU 中的 TLB 表项决定的。

从 0x80000000 到 0xFFFFFFFF 的一段为 Kernel Space，仅限于 Kernel Mode 访问。如果在 User Mode 下试图访问这一段内存，将会引发系统的一个 Exception。MIPS 的 Kernel Space 又可以划分为三部分。首先是通过 MMU 映射到物理地址的 1GB 空间，地址范围从 0xC0000000 到 0xFFFFFFFF。这 1GB 空间可以用来访问实际的 DRAM 内存，可以为操作系统的内核所用。

MIPS 的 Kernel Space 中，还有两段特殊的地址空间，分别是从 0x80000000 到 0x9FFFFFFF 的 Kernel Space Unmapped Cached 和 0xA0000000 到 0xBFFFFFFF 的 Kernel Space Unmapped Uncached。之所以说它们特殊，是因为这两段逻辑地址到物理地址的映射关系是硬件直接确定的，不通过 MMU，而且两段实际上是重叠的，均对应 0x00000000 到 0x20000000 的物理地址。那么，为什么一段同样的物理地址有两个逻辑地址对应呢？它们的区别又在哪里呢？

原来，这是 MIPS 的设计特色之一。软件在访问 Kernel Space Unmapped Uncached 这段

地址空间的时候，不经过 MIPS 的 Cache。这样，虽然速度会比较慢，但是，对于硬件 I/O 寄存器来说，就不存在所谓的 Cache 一致性问题。Cache 一致性问题，是指硬件将某个地址的内容跳过软件而改变了，Cache 中的内容尚未同步。这样，如果软件读取该地址，有可能从 Cache 中获取到错误的内容。将硬件 I/O 寄存器设定在这段地址空间，就可以避免 Cache 一致性带来的问题。MIPS 的程序上电启动地址 0xBFC00000，也落在这段地址空间内。——上电时，MMU 和 Cache 均未初始化，因此，只有这段地址空间可以正常读取并处理。

另一段特殊的地址 Kernel Space Unapped Cached，与前者类似，直接映射到 0x00000000 到 0x20000000，与 Kernel Space Unmapped Uncached 重叠。因为通过 Cache，这段地址空间的访问速度比前者为快。一般地，这段内存空间用于内核代码段，或者内核中的堆栈。

显然地，当工程师们换算 Kernel Space 中的这两段的物理地址和逻辑地址时，只需要改变地址的高 3bit 就可以了。

那么，什么时候需要使用物理地址，什么时候需要使用逻辑地址呢？我们知道，逻辑地址是程序中访问的内存地址，譬如，下面的这条指令：

lw a0, 128(t2)

这条指令的内容是从 t2 寄存器内的地址 + 偏移 128 字节处，读取一个 word (4Byte)到寄存器 a0 内。如果 t2 的值为 0x88200100，则最终访问的物理地址为 0x88200180。

而物理地址，从工程上可以理解为，将逻辑分析仪连接到内存总线(Memory Bus)上，逻辑分析仪指示的地址，就是物理地址。假如，在上一个例子中，我们把逻辑分析仪接到处理器的前端内存总线，我们就可以看到，执行该指令时，系统访问的物理地址为 0x08200180。物理地址和逻辑地址的换算，不仅限于电子工程师在设计硬件线路时需要。在内核工程师编写支持 DMA 的外部设备驱动时，需要将向操作系统申请到的数据缓冲区地址（当然，这是一个逻辑地址）转换为物理地址，并“告诉”相关外设。这样，外设就可以在收到数据后，使用 DMA 模式储存在系统的主存中，并向系统发起一个 IRQ。操作系统在 IRQ 的 handler 中，从外设的相应 IO 寄存器读取到这段内存的地址（当然，是物理地址）并转换为逻辑地址并处理之。这个过程中，如果没有正确使用和分辨物理地址和逻辑地址，驱动程序便会导致内核的一个 panic 错误。

物理地址到逻辑地址的映射关系是由什么决定的呢？除了上面提到的两段 Unmapped 的地址空间，其余都是由 TLB 确定的，由 MMU 来执行。

默认外设 I/O 资源是不在 Linux 内核空间中的（如 sram 或硬件接口寄存器等），若需要访问该外设 I/O 资源，必须先将其地址映射到内核空间中来，然后才能在内核空间中访问它。

Linux 内核访问外设 I/O 内存资源的方式有两种：动态映射(ioremap)和静态映射(map_desc)。

19.2 动态映射(ioremap)方式

动态映射方式是大家使用了比较多的，也比较简单。即直接通过内核提供的 ioremap 函数动态创建一段外设 I/O 内存资源到内核虚拟地址的映射表，从而可以在内核空间中访问这段 I/O 资源。

ioremap 宏定义在 asm/io.h 内：

```
#define ioremap(cookie,size)    __ioremap(cookie,size,0)
```

__ioremap 函数原型为(arm/mm/ioremap.c)：

```
void __iomem * __ioremap(unsigned long phys_addr, size_t size, unsigned long flags);
```

phys_addr：要映射的起始的 IO 地址

size：要映射的空间的大小

flags：要映射的 IO 空间和权限有关的标志

该函数返回映射后的内核虚拟地址(3G-4G). 接着便可以通过读写该返回的内核虚拟地址去访问之这段 I/O 内存资源。

示例程序 regeditor_drv.c。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/irq.h>
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <linux/device.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/poll.h>
#include <linux/spinlock.h>

#define KER_RW_R8       100
#define KER_RW_R16      101   /* 切记！cmd 命令值不能乱定义 */
#define KER_RW_R32      102

#define KER_RW_W8       103
#define KER_RW_W16      104
#define KER_RW_W32      105

static int major;
static struct class *class;

static long ker_rw_ioctl( struct file *file, unsigned int cmd, unsigned long arg)
{
    volatile unsigned char  *p8;
    volatile unsigned short *p16;
    volatile unsigned int   *p32;
    unsigned int val;
    unsigned int addr;

    unsigned int buf[2];

    copy_from_user(buf, (const void __user *)arg,8);

    addr = buf[0];
    val  = buf[1];

    p8  = (volatile unsigned char *)ioremap(addr, 4);
    p16 = (volatile unsigned short *) p8;
    p32 = (volatile unsigned int *)p8;

    switch (cmd)
    {
        case KER_RW_R8:
        {
            val = *p8;
            copy_to_user((void __user *)(arg+4), &val, 4);
            break;
        }

        case KER_RW_R16:
        {
            val = *p16;
            copy_to_user((void __user *)(arg+4), &val, 4);
            break;
```

```
            }

            case KER_RW_R32:
            {
                  val = *p32;
                  copy_to_user((void __user *)(arg+4), &val, 4);
                  break;
            }

            case KER_RW_W8:
            {
                  *p8 = val;
                  break;
            }

            case KER_RW_W16:
            {
                  *p16 = val;
                  break;
            }

            case KER_RW_W32:
            {
                  *p32 = val;
                  break;
            }
      }

      iounmap(p8);
      return 0;
}


static struct file_operations ker_rw_ops = {
      .owner      = THIS_MODULE,
      .unlocked_ioctl          = ker_rw_ioctl,

       };

       static int ker_rw_init(void)
       {
             major = register_chrdev(0, "ker_rw", &ker_rw_ops);

             class = class_create(THIS_MODULE, "ker_rw");

             /*  为了让 mdev 根据这些信息来创建设备节点  */
             device_create(class, NULL, MKDEV(major, 0), NULL, "ker_rw"); /* /dev/ker_rw */

             return 0;
       }

       static void ker_rw_exit(void)
       {
             device_destroy(class,MKDEV(major, 0));
             class_destroy(class);
             unregister_chrdev(major, "ker_rw");
       }

       module_init(ker_rw_init);
       module_exit(ker_rw_exit);


       MODULE_LICENSE("GPL");
```

编写 makefile。

```
obj-m := regeditor_drv.o
#定义目录变量
```

```
KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson
PWD := $(shell pwd)
all:
# make 文件
	make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS_COMPILE=mipsel-linux-
clean:
	rm -rf *.o   *.mod.c *.ko
```

驱动编译后，在开发板加载 regeditor_drv.ko,后运行以下测试程序 regeditor

```
/*regeditor.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

#define KER_RW_R8        100
#define KER_RW_R16       101
#define KER_RW_R32       102

#define KER_RW_W8        103
#define KER_RW_W16       104
#define KER_RW_W32       105


/* Usage:
 * ./regeditor r8   addr [num]   //  用法
 * ./regeditor r16 addr [num]
 * ./regeditor r32 addr [num]
 *
 * ./regeditor w8   addr val
 * ./regeditor w16 addr val
 * ./regeditor w32 addr val
 */

void print_usage(char *file)
{
	printf("Usage:\n");
	printf("%s <r8 | r16 | r32> <phy addr> [num]\n", file);
	printf("%s <w8 | w16 |w32> <phy addr> <val>\n", file);
}

int main(int argc, char **argv)
{
	int fd;
	unsigned int buf[2];
	unsigned int i;
	unsigned int num;


	if ((argc != 3) && (argc != 4))
	{
		print_usage(argv[0]);
		return -1;
	}

	fd = open("/dev/ker_rw", O_RDWR);
	if (fd < 0)
	{
		printf("can't open /dev/ker_rw\n");
```

```
		return -2;
	}

	/* addr */
	buf[0] = strtoul(argv[2], NULL, 0);

	if (argc == 4)
	{
		buf[1] = strtoul(argv[3], NULL, 0);
		num	= buf[1];
	}
	else
	{
		num = 1;
	}

	if (strcmp(argv[1], "r8") == 0)
	{
		for ( i = 0; i < num; i++)
		{
			ioctl(fd, KER_RW_R8, buf);	/* val = buf[1] */
			printf("%02d. [%08x] = %02x\n", i, buf[0], (unsigned char)buf[1]);
			buf[0] += 1;
		}
	}
	else if (strcmp(argv[1], "r16") == 0)
	{
		for ( i = 0; i < num; i++)
		{
			ioctl(fd, KER_RW_R16, buf);	/* val = buf[1] */
			printf("%02d. [%08x] = %04x\n", i, buf[0], (unsigned short)buf[1]);
			buf[0] += 2;
		}
	}
	else if (strcmp(argv[1], "r32") == 0)
	{
		for ( i = 0; i < num; i++)
		{
			ioctl(fd, KER_RW_R32, buf);	/* val = buf[1] */
			printf("%02d. [%08x] = %08x\n", i, buf[0], (unsigned int)buf[1]);
			buf[0] += 4;
		}
	}
	 else if (strcmp(argv[1], "w8") == 0)
	 {
		 ioctl(fd, KER_RW_W8, buf);	/* val = buf[1] */
	 }
	 else if (strcmp(argv[1], "w16") == 0)
	 {
		 ioctl(fd, KER_RW_W16, buf);	/* val = buf[1] */
	 }
	 else if (strcmp(argv[1], "w32") == 0)
	 {
		 ioctl(fd, KER_RW_W32, buf);	/* val = buf[1] */
	 }
	 else
	 {
		 printf(argv[0]);
		 return -1;
	 }

	 return 0;

	}
```

运行结果：

```
[root@Loongson:/loongson_tools]#./regeditor r32 0x1fd011e4
```

```
00. [1fd011e4] = 00c00000
```

19.3 静态映射(map_desc)方式

下面介绍静态映射方式即通过 map_desc 结构体静态创建 I/O 资源映射表。

内核提供了在系统启动时通过 map_desc 结构体静态创建 I/O 资源到内核地址空间的线性映射表(即 page table)的方式，这种映射表是一种一一映射的关系。程序员可以自己定义该 I/O 内存资源映射后的虚拟地址。创建好了静态映射表，在内核或驱动中访问该 I/O 资源时则无需再进行 ioreamp 动态映射，可以直接通过映射后的 I/O 虚拟地址去访问它。

下面详细分析这种机制的原理并举例说明如何通过这种静态映射的方式访问外设 I/O 内存资源。

内核提供了一个重要的结构体 struct machine_desc ,这个结构体在内核移植中起到相当重要的作用,内核通过 machine_desc 结构体来控制系统体系**架构**相关部分的初始化。

machine_desc 结构体的成员包含了体系架构相关部分的几个最重要的初始化函数，包括 map_io, init_irq, init_machine 以及 phys_io , timer 成员等。

machine_desc 结构体定义如下：

```
struct map_desc {//用到的结构体
/*
* Note! The first four elements are used
* by assembler code in head-armv.S
*/
 unsigned long virtual;   //虚拟地址
 unsigned long pfn;   //__phys_to_pfn(物理地址)，就是物理页框号
 unsigned long length;   //大小
 unsigned int type;   //类型
};
```

常见问题：

1、用户空间（进程）是否有高端内存概念？

用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问 3G 物理内存，而内核进程可以访问所有物理内存。

2、64 位内核中有高端内存吗？

目前现实中，64 位 Linux 内核不存在高端内存，因为 64 位内核可以支持超过 512GB 内存。若机器安装的物理内存超过内核地址空间范围，就会存在高端内存。

3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32 位系统用户进程最大可以访问 3GB，内核代码可以访问所有物理内存。

64 位系统用户进程最大可以访问超过 512GB，内核代码可以访问所有物理内存。

4、高端内存和物理地址、逻辑地址、线性地址的关系？

高端内存只和逻辑地址有关系，和逻辑地址、物理地址没有直接关系。

5、为什么不把所有的地址空间都分配给内核？

若把所有地址空间都给内存，那么用户进程怎么使用内存？怎么保证内核使用内存和用户进程不起冲突？

（1）让我们忽略 Linux 对段式内存映射的支持。 在保护模式下，我们知道无论 CPU 运行于用户态还是核心态，CPU 执行程序所访问的地址都是虚拟地址，MMU 必须通过读取控制寄存器 CR3 中的值作为当前页面目录的指针，进而根据分页内存映射机制（参看相关文

档）将该虚拟地址转换为真正的物理地址才能让 CPU 真 正的访问到物理地址。

（2）对于 32 位的 Linux，其每一个进程都有 4G 的寻址空间，但当一个进程访问其虚拟内存空间中的某个地址时又是怎样实现不与其它进程的虚拟空间混淆 的呢？每个进程都有其自身的页面目录 PGD，Linux 将该目录的指针存放在与进程对应的内存结构 task_struct.(struct mm_struct)mm->pgd 中。每当一个进程被调度（schedule()）即将进入运行态时，Linux 内核都要用该进程的 PGD 指针设 置 CR3（switch_mm()）。

（3）当创建一个新的进程时，都要为新进程创建一个新的页面目录 PGD，并从内核的页面目录 swapper_pg_dir 中复制内核区间页面目录项至新建进程页面目录 PGD 的相应位置，具体过程如下：

do_fork() --> copy_mm() --> mm_init() --> pgd_alloc() --> set_pgd_fast() --> get_pgd_slow() --> memcpy(&PGD + USER_PTRS_PER_PGD, swapper_pg_dir + USER_PTRS_PER_PGD, (PTRS_PER_PGD - USER_PTRS_PER_PGD) * sizeof(pgd_t))

这样一来，每个进程的页面目录就分成了两部分，第一部分为“用户空间”，用来映射其整个进程空间（0x0000 0000－0xBFFF FFFF）即 3G 字节的虚拟地址；第二部分为“系统空间”，用来映射（0xC000 0000－0xFFFF FFFF）1G 字节的虚拟地址。可以看出 Linux 系统中每个进程的页面目录的第二部分是相同的，所以从进程的角度来看，每个进程有 4G 字节的虚拟空间， 较低的 3G 字节是自己的用户空间，最高的 1G 字节则为与所有进程以及内核共享的系统空间。

（4）现在假设我们有如下一个情景：

在进程 A 中通过系统调用 sethostname(const char *name,seze_t len)设置计算机在网络中的“主机名”。

在该情景中我们势必涉及到从用户空间向内核空间传递数据的问题，name 是用户空间中的地址，它要通过系统调用设置到内核中的某个地址中。让我们看看这个 过程中的一些细节问题：系统调用的具体实现是将系统调用的参数依次存入寄存器 ebx,ecx,edx,esi,edi（最多 5 个参数，该情景有两个 name 和 len），接着将系统调用号存入寄存器 eax，然后通过中断指令“int 80”使进程 A 进入系统空间。由于进程的 CPU 运行级别小于等于为系统调用设置的陷阱门的准入级别 3，所以可以畅通无阻的进入系统空间去执行为 int 80 设置的函数指针 system_call()。由于 system_call()属于内核空间，其运行级别 DPL 为 0，CPU 要将堆栈切换到内核堆栈，即 进程 A 的系统空间堆栈。我们知道内核为新建进程创建 task_struct 结构时，共分配了两个连续的页面，即 8K 的大小，并将底部约 1k 的大小用于 task_struct（如 #define alloc_task_struct() ((struct task_struct *) __get_free_pages(GFP_KERNEL,1))）,而其余部分内存用于系统空间的堆栈空间，即当从用户空间转入系统空间时，堆栈指针 esp 变成了（alloc_task_struct()+8192），这也是为什么系统空间通常用宏定义 current（参看其实现）获取当前进程的 task_struct 地址的原因。每次在进程从用户空间进入系统空间之初，系统堆栈就已经被依次压入用户堆栈 SS、用户堆栈指针 ESP、EFLAGS、 用户空间 CS、EIP，接着 system_call()将 eax 压入，再接着调用 SAVE_ALL 依次压入 ES、DS、EAX、EBP、EDI、ESI、 EDX、ECX、EBX，然后调用 sys_call_table+4*%EAX，本情景为 sys_sethostname()。

（5）在 sys_sethostname()中，经过一些保护考虑后，调用 copy_from_user(to,from,n)，其中 to 指向内核空间 system_utsname.nodename，譬如 0xE625A000，from 指向用户空间譬如 0x8010FE00。现在进程 A 进入了内核，在 系统空间中运行，MMU 根据其 PGD 将虚拟地址完成到物理地址的映射，最终完成从用户空间到系统空间数据的复制。准备复制之前内核先要确定用户空间地址和 长度的合法性，至于从该用户空间地址开始的某个长度的整个区间是否已经映射并不去检查，如果区间内某个地址未映射或读写权限等问题出现时，则视为坏地址， 就产生一个页面异常，让页面异常服务程序处理。过程如 下：

copy_from_user()->generic_copy_from_user()->access_ok()+__copy_user_zeroing().

（6）小结：
*进程寻址空间 0~4G
*进程在用户态只能访问 0~3G，只有进入内核态才能访问 3G~4G
*进程通过系统调用进入内核态
*每个进程虚拟空间的 3G~4G 部分是相同的
*进程从用户态进入内核态不会引起 CR3 的改变但会引起堆栈的改变

19.4 mmap 系统调用

19.4.1mmap 系统调用

mmap 将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。munmap 执行相反的操作，删除特定地址区域的对象映射。

当使用 mmap 映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,不必再调用 read,write 等系统调用.但需注意,直接对该段内存写时不会写入超过当前文件大小的内容.

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

基于文件的映射，在 mmap 和 munmap 执行过程的任何时刻，被映射文件的 st_atime 可能被更新。如果 st_atime 字段在前述的情况下没有得到更新，首次对映射区的第一个页索引时会更新该字段的值。用 PROT_WRITE 和 MAP_SHARED 标志建立起来的文件映射，其 st_ctime 和 st_mtime 在对映射区写入之后，但在 msync()通过 MS_SYNC 和 MS_ASYNC 两个标志调用之前会被更新。

用法：

```
#include <sys/mman.h>
#include <unistd.h>
void *mmap(void *start, size_t length, int prot, int flags,
int fd, off_t offset);
int munmap(void *start, size_t length);
```

返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回 0。失败时，mmap()返回 MAP_FAILED[其值为(void *)-1]，munmap 返回-1。errno 被设为以下的某个值

EACCES：访问出错

EAGAIN：文件已被锁定，或者太多的内存已被锁定

EBADF：fd 不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE：已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM：内存不足，或者进程已超出最大内存映射数量

EPERM：权能不足，操作不允许

ETXTBSY：已写的方式打开文件，同时指定 MAP_DENYWRITE 标志

SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

参数：

start：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过 or 运算合理地组合在一起

PROT_EXEC //页内容可以被执行

PROT_READ //页内容可以被读取

PROT_WRITE //页可以被写入

PROT_NONE //页不可访问

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

MAP_FIXED //使用指定的映射起始地址，如果由 start 和 len 参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。

MAP_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到 msync()或者 munmap()被调用，文件实际上不会被更新。

MAP_PRIVATE //建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。

MAP_DENYWRITE //这个标志被忽略。

MAP_EXECUTABLE //同上

MAP_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。

MAP_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。

MAP_GROWSDOWN //用于堆栈，告诉内核 VM 系统，映射区可以向下扩展。

MAP_ANONYMOUS //匿名映射，映射区不与任何文件关联。

MAP_ANON //MAP_ANONYMOUS 的别称，不再被使用。

MAP_FILE //兼容标志，被忽略。

MAP_32BIT //将映射区放在进程地址空间的低 2GB，MAP_FIXED 指定时会被忽略。当前这个标志只在 x86-64 平台上得到支持。

MAP_POPULATE //为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。

MAP_NONBLOCK //仅和 MAP_POPULATE 一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

fd：有效的文件描述词。如果 MAP_ANONYMOUS 被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。

系统调用 munmap()

```
#include    <unistd.h>
#include    <sys/mman.h>
```

```
int munmap( void * addr, size_t len )
```

该调用在进程地址空间中解除一个映射关系，addr 是调用 mmap()时返回的地址，len 是映射区的大小。当映射关系解除后，对原来映射地址的访问将导致段错误发生。

系统调用 msync()

```
#include <sys/mman.h>
#include <unistd.h>
int msync ( void * addr , size_t len, int flags)
```

一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用 munmap（）后才执行该操作。可以通过调用 msync()实现磁盘上文件内容与共享内存区的内容一致。

19.4.2 系统调用 mmap()用于共享内存的两种方式

（1）使用普通文件提供的内存映射：适用于任何进程之间；此时，需要打开或创建一个文件，然后再调用 mmap()；典型调用代码如下：

```
fd=open(name, flag, mode);
if(fd<0)
    ...
ptr=mmap(NULL, len , PROT_READ|PROT_WRITE, MAP_SHARED , fd , 0);
```

通过 mmap()实现共享内存的通信方式有许多特点和要注意的地方

（2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用 mmap()，然后调用 fork()。那么在调用 fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承 mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而 mmap()返回的地址，却由父子进程共同维护。

对于具有亲缘关系的进程实现共享内存最好的方式应该是采用匿名内存映射的方式。此时，不必指定具体的文件，只要设置相应的标志即可.

19.4.3 mmap 进行内存映射的原理

mmap 系统调用的最终目的是将,设备或文件映射到用户进程的虚拟地址空间,实现用户进程对文件的直接读写,这个任务可以分为以下三步:

（1）在用户虚拟地址空间中寻找空闲的满足要求的一段连续的虚拟地址空间,为映射做准备(由内核 mmap 系统调用完成)

每个进程拥有 3G 字节的用户虚存空间。但是，这并不意味着用户进程在这 3G 的范围内可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用。

那么，内核怎样管理每个进程 3G 的虚存空间呢？概括地说，用户进程经过编译、链接后形成的映象文件有一个代码段和数据段（包括 data 段和 bss 段），其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，即全局变量和所有申明为 static 的局部变量，这些空间是进程所必需的基本要求，这些空间是在建立一个进程的运行映像时就分配好的。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的，如图 19.2 所示：

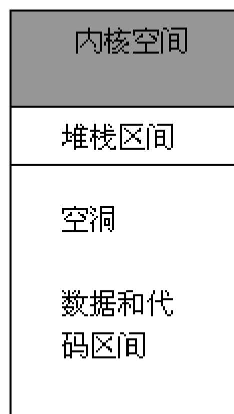


图 19.2 进程虚拟空间的划分

在内核中,这样每个区域用一个结构 struct vm_area_struct 来表示.它描述的是一段连续的、具有相同访问属性的虚存空间，该虚存空间的大小为物理内存页面的整数倍。可以使用 cat /proc//maps 来查看一个进程的内存使用情况,pid 是进程号.其中显示的每一行对应进程的一个 vm_area_struct 结构.

下面是 struct vm_area_struct 结构体的定义：

```
#include <linux/mm_types.h>

/* This struct defines a memory VMM memory area. */

struct vm_area_struct {
struct mm_struct * vm_mm; /* VM area parameters */
unsigned long vm_start;
unsigned long vm_end;

/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next;
pgprot_t vm_page_prot;
unsigned long vm_flags;

/* AVL tree of VM areas per task, sorted by address */
short vm_avl_height;
struct vm_area_struct * vm_avl_left;
struct vm_area_struct * vm_avl_right;

/* For areas with an address space and backing store,
vm_area_struct *vm_next_share;
struct vm_area_struct **vm_pprev_share;
struct vm_operations_struct * vm_ops;
unsigned long vm_pgoff; /* offset in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
struct file * vm_file;
unsigned long vm_raend;
void * vm_private_data; /* was vm_pte (shared mem) */
};
```

通常，进程所使用到的虚存空间不连续，且各部分虚存空间的访问属性也可能不同。所以一个进程的虚存空间需要多个 vm_area_struct 结构来描述。在 vm_area_struct 结构的数目较少的时候，各个 vm_area_struct 按照升序排序，以单链表的形式组织数据（通过 vm_next 指针指向下一个 vm_area_struct 结构）。但是当 vm_area_struct 结构的数据较多的时候，仍然采用链表组织的化，势必会影响到它的搜索速度。针对这个问题，vm_area_struct 还添加了 vm_avl_hight（树高）、vm_avl_left（左子节点）、vm_avl_right（右子节点）三个成员来实现 AVL 树，以提高 vm_area_struct 的搜索速度。

假如该 vm_area_struct 描述的是一个文件映射的虚存空间，成员 vm_file 便指向被映射的文件的 file 结构，vm_pgoff 是该虚存空间起始地址在 vm_file 文件里面的文件偏移，单位

为物理页面。

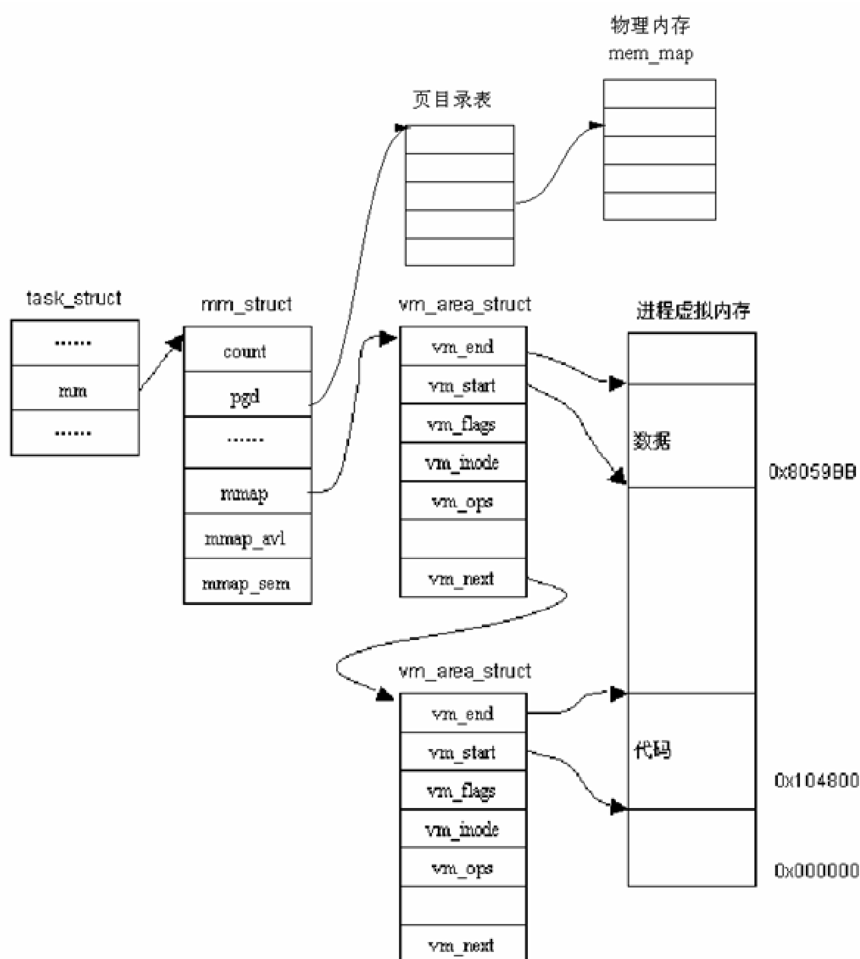


图 19.3 进程虚拟地址示意图

因此,mmap 系统调用所完成的工作就是准备这样一段虚存空间,并建立 vm_area_struct 结构体,将其传给具体的设备驱动程序.

（2）建立虚拟地址空间和文件或设备的物理地址之间的映射(设备驱动完成)

建立文件映射的第二步就是建立虚拟地址和具体的物理地址之间的映射,这是通过修改进程页表来实现的.mmap 方法是 file_opeartions 结构的成员:

```
int (*mmap)(struct file *,struct vm_area_struct *);
```

linux 有 2 个方法建立页表:

(1) 使用 remap_pfn_range 一次建立所有页表.

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn, unsigned long size, pgprot_t prot);
```

返回值:

成功返回 0, 失败返回一个负的错误值

参数说明:

vma 用户进程创建一个 vma 区域

virt_addr 重新映射应当开始的用户虚拟地址. 这个函数建立页表为这个虚拟地址范围从 virt_addr 到 virt_addr_size.

pfn 页帧号, 对应虚拟地址应当被映射的物理地址. 这个页帧号简单地是物理地址右移 PAGE_SHIFT 位. 对大部分使用, VMA 结构的 vm_paoff 成员正好包含你需要的值. 这个函数影响物理地址从 (pfn<

size 正在被重新映射的区的大小, 以字节.

prot 给新 VMA 要求的"protection". 驱动可(并且应当)使用在 vma->vm_page_prot 中找到的值.

(2) 使用 nopage VMA 方法每次建立一个页表项.

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);
```

返回值:

成功则返回一个有效映射页,失败返回 NULL.

参数说明:

address 代表从用户空间传过来的用户空间虚拟地址.

返回一个有效映射页.

(3) 使用方面的限制：

remap_pfn_range 不能映射常规内存，只存取保留页和在物理内存顶之上的物理地址。因为保留页和在物理内存顶之上的物理地址内存管理系统的各个子模块管理不到。640 KB 和 1MB 是保留页可能映射，设备 I/O 内存也可以映射。如果想把 kmalloc()申请的内存映射到用户空间，则可以通过 mem_map_reserve()把相应的内存设置为保留后就可以。

（3）当实际访问新映射的页面时的操作(由缺页中断完成)

(1) page cache 及 swap cache 中页面的区分：一个被访问文件的物理页面都驻留在 page cache 或 swap cache 中，一个页面的所有信息由 struct page 来描述。struct page 中有一个域为指针 mapping ，它指向一个 struct address_space 类型结构。page cache 或 swap cache 中的所有页面就是根据 address_space 结构以及一个偏移量来区分的。

(2) 文件与 address_space 结构的对应：一个具体的文件在打开后，内核会在内存中为之建立一个 struct inode 结构，其中的 i_mapping 域指向一个 address_space 结构。这样，一个文件就对应一个 address_space 结构，一个 address_space 与一个偏移量能够确定一个 page cache 或 swap cache 中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。

(3) 进程调用 mmap()时，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。

(4) 对于共享内存映射情况，缺页异常处理程序首先在 swap cache 中寻找目标页（符合 address_space 以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区 (swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到 page cache 中。进程最终将更新进程页表。

注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在 page cache 中根据 address_space 以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新.

(5) 所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。

19.4.4 内存映射的步骤:

用 open 系统调用打开文件，并返回描述符 fd.
用 mmap 建立内存映射，并返回映射首地址指针 start.
对映射(文件)进行各种操作，显示(printf)，修改(sprintf).
用 munmap(void *start, size_t lenght)关闭内存映射.
用 close 系统调用关闭文件 fd.

19.5 mmap 编程示例

修改自刘世伟的 https://github.com/lshw/loongson_tools.
读取 GPIO 的复用寄存器：cbus_dump.c

```
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include "ls1c_regs.h"

int main()
{
	unsigned char * map_base;
	FILE *f;
	int n, fd;

	fd = open("/dev/mem", O_RDWR|O_SYNC);
	if (fd == -1)	{
		return (-1);
	}
/* 把 bfd01000 开始 0x1000 字节，映射到 map_base */
	map_base = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x1fd01000);

	if (map_base == 0)	{
		printf("NULL pointer!\n");
	}

	unsigned long addr;
	unsigned char content;
	int i = 0,i1 = 0;
printf("[gpio_number]:	0	8	16	24	32	40	48	56	64	72	80	88	96 104 112 120");
	for(i1 = 0 ; i1 < 5 ; i1 ++) { // 每个 gpio5 个功能的定义分别是 bfd0110c0,bfd0110d0,bfd0110e0,bfd0110f0,bfd011200 开始的 16 个字节,
		printf("\nfun%d[1fd01%03x]:",i1 + 1,LS1X_CBUS_FIRST0 + i1 * 0x10);
		for(i = 0 ; i < 0x10 ; i ++) {
			printf("	%02X",*(volatile unsigned long *)(map_base+ LS1X_CBUS_FIRST0 + i1 * 0x10 + i)&0xFF);
		}
	}
	close(fd);
	printf("\n");
	munmap(map_base, 0x1000);
	return (0);
}
```

读取 GPIO 的复用配置：gpio_func.c
清除寄存器中的某一位：regs_bit_clr.c
读取寄存器中的某一位：regs_bit_get.c
设置寄存器中的某一位：regs_bit_set.c
读取某一个寄存器：regs_read.c
写值某一个寄存器：regs_write.c

附录 1　一天一个 linux 命令

1) 解压

tar xzvf xx.tar.gz -C tmp
-C 作用：将 xx.tar.gz 这个压缩包解压到当前目录下的 tmp 目录下，而不是当前目录下。
*.tar.gz　格式解压为　tar　zxvf　xx.tar.gz
*.tar.bz2　格式解压为　tar　jxvf　xx.tar.bz2
01-.tar 格式
解包：　tar xvf FileName.tar
打包：　tar cvf FileName.tar DirName（注：tar 是打包，不是压缩！）

02-.gz 格式
解压：　gunzip FileName.gz
解压：　gzip -d FileName.gz
压缩：　gzip FileName

03-.tar.gz 格式
解压：　tar zxvf FileName.tar.gz
压缩：　tar zcvf FileName.tar.gz DirName

04-.bz2 格式
解压 1：　bzip2 -d FileName.bz2
解压 2：　bunzip2 FileName.bz2
压　缩：　bzip2 -z FileName

05-.tar.bz2 格式
解压：　tar jxvf FileName.tar.bz2
压缩：　tar jcvf FileName.tar.bz2 DirName

06-.bz 格式
解压 1：　bzip2 -d FileName.bz
解压 2：　bunzip2 FileName.bz

07-.tar.bz 格式
解压：　tar jxvf FileName.tar.bz

08-.Z 格式
解压：　uncompress FileName.Z
压缩：　compress FileName

09-.tar.Z 格式
解压：　tar Zxvf FileName.tar.Z

压缩： tar Zcvf FileName.tar.Z DirName

10-.tgz 格式
解压： tar zxvf FileName.tgz

11-.tar.tgz 格式
解压： tar zxvf FileName.tar.tgz
压缩： tar zcvf FileName.tar.tgz FileName

12-.zip 格式
解压： unzip FileName.zip
压缩： zip FileName.zip DirName

13-.lha 格式
解压： lha -e FileName.lha
压缩： lha -a FileName.lha FileName

14-.rar 格式
解压： rar a FileName.rar
压缩： rar e FileName.rar
rar 请到：http://www.rarsoft.com/download.htm 下载！
解压后请将 rar_static 拷贝到/usr/bin 目录（其他由$PATH 环境变量
指定的目录也行）： cp rar_static /usr/bin/rar

2) 用 cat 命令将字符串追加到文件中

echo "1245350832" | cat >> file

3) echo

覆盖型写法 (文件里原来的内容被覆盖)
echo "aaa" > a.txt
echo aaa > a.txt

添加型写法 (新内容添加在原来内容的
echo "aaa" >> a.txt
echo aaa >> a.txt

echo 命令的功能是在显示器上显示一段文字，一般起到一个提示的作用。
该命令的一般格式为： echo [-n] 字符串
其中选项 n 表示输出文字后不换行；字符串能加引号，也能不加引号。用 echo 命令输出加引号的字符串时，将字符串原样输出；用 echo 命令输出不加引号的字符串时，将字符串中的各个单词作为字符串输出，各字符串之间用一个空格分割。
功能说明：显示文字。
语 法：echo [-ne][字符串]或 echo [--help][--version]

补充说明：echo 会将输入的字符串送往标准输出。输出的字符串间以空白字符隔开，并在最后加上换行号。

参　数：-n 不要在最后自动换行

-e 若字符串中出现以下字符，则特别加以处理，而不会将它当成一般

文字输出：

\a 发出警告声；

\b 删除前一个字符；

\c 最后不加上换行符号；

\f 换行但光标仍旧停留在原来的位置；

\n 换行且光标移至行首；

\r 光标移至行首，但不换行；

\t 插入 tab；

\v 与\f 相同；

\\ 插入\字符；

\nnn 插入 nnn（八进制）所代表的 ASCII 字符；

–help 显示帮助

–version 显示版本信息

4) 常用 cat 命令

命令简介

把档案串连接后传到基本输出（萤幕或加 > fileName 到另一个档案）。

命令语法

cat [-AbeEnstTuv] [--help] [--version] fileName

参数

-n	由 1 开始对所有输出的行数编号
b	由 1 开始对所有输出的行数编号
s	当遇到有连续两行以上的空白行，就代换为一行的空白行

范例

cat filename.txt 或 cat < filename.txt

该命令将文件内容在屏幕上显示出来。

cat filename.txt |more

可以将内容在屏幕上分页显示。

cat > filename.txt

该命令会把之后的输入的文本重定向到 filename.txt 文件中，ctrl+c 或 ctrl+d 退出。

cat -n textfile1 > textfile2

把 textfile1 的档案内容加上行号后输入 textfile2 这个档案里，textfile2 中原有内容会被覆盖。

cat -n textfile1 >> textfile2

把 textfile1 的档案内容加上行号后附加到 textfile2 这个档案里，textfile2 中原有内容不会被覆盖。

cat -b textfile1 textfile2 >> textfile3

把 textfile1 和 textfile2 的档案内容加上行号（空白行不加）之后将内容附加到 textfile3 里。

cat -n textfile1 textfile2>> textfile3

把 textfile1 和 textfile2 的档案内容加上行号（空白行也加）之后将内容附加到 textfile3 里。

cat '' > filename.txt

清空文件内容。

一定得注意'>会覆盖原有内容，'>>'则不会覆盖原有内容。

cat 主要有三大功能：

(1)一次显示整个文件。

cat filename

(2)从键盘创建一个文件。

cat > filename

只能创建新文件,不能编辑已有文件.

(3)将几个文件合并为一个文件。

cat file1 file2 > file

cat 具体命令格式为 : cat [-AbeEnstTuv] [--help] [--version] filename

说明：把档案串连接后传到基本输出(屏幕或加 > fileName 到另一个档案)

参数：

-n 或 –number 由 1 开始对所有输出的行数编号

-b 或 –number-nonblank 和 -n 相似，只不过对于空白行不编号

-s 或 –squeeze-blank 当遇到有连续两行以上的空白行，就代换为一行的空白行

-v 或 –show-nonprinting

范例：

把 linuxfile1 的档案内容加上行号后输入 linuxfile2 这个档案里

cat -n linuxfile1 > linuxfile2

把 linuxfile1 和 linuxfile2 的档案内容加上行号(空白行不加)之后将内容附加到 linuxfile3 里。

cat -b linuxfile1 linuxfile2 >> linuxfile3

范例：

把 linuxfile1 的档案内容加上行号后输入 linuxfile2 这个档案里

cat -n linuxfile1 > linuxfile2

把 linuxfile1 和 linuxfile2 的档案内容加上行号(空白行不加)后将内容附加到 linuxfile3 里。

cat -b linuxfile1 linuxfile2 >> linuxfile3

清空/etc/test.txt 档案内容

cat /dev/null > /etc/test.txt

5) tree 显示目录结构

6) 防火墙

卸载防火墙

```
apt-get remove iptables
```

关闭防火墙

```
service iptables stop
```

7) 内核启动参数

Linux 内核在启动的时候，能接收某些命令行选项或启动时参数。当内核不能识别某些硬件进而不能设置硬件参数或者为了避免内核更改某些参数的值，可以通过这种方式手动将这些参数传递给内核。

如果不使用启动管理器，比如直接从 BIOS 或者把内核文件用“cp zImage /dev/fd0”等方法直接从设备启动，就不能给内核传递参数或选项－－这也许是我们使用引导管理器比如 LILO 的好处之一吧。

Linux 的内核参数是以空格分开的一个字符串列表，通常具有如下形式：

name[=value_1][,value_2]...[,value_10]

“name”是关键字，内核用它来识别应该把“关键字”后面的值传递给谁，也就是如何处理这个值，是传递给处理例程还是作为环境变量或者抛给“init”。值的个数限制为 10，你可以通过再次使用该关键字使用超过 10 个的参数。

首 先，内核检查关键字是不是 `root='',`nfsroot='', `nfsaddrs='', `ro'', `rw'', `debug''或`init''，然后内核在 bootsetups 数组里搜索于该关键字相关联的已注册的处理函数，如果找到相关的已注册的处理函数，则调用这些函数并把关 键字后面的值作为参数传递给这些函数。比如你在启动时设置参数 name＝a,b,c,d，内核搜索 bootsetups 数组，如果发现“name”已注 册，则调用“name”的设置函数如 name_setup()，并把 a,b,c,d 传递给 name_setup()执行。

所有型如“name＝value”参数，如果没有被上面所述的设置函数接收，将被解释为系统启动后的环境变量，比如“TERM=vt100”就会被作为一个启动时参数。

所有没有被内核设置函数接收也没又被设置成环境变量的参数都将留给 init 进程处理，比如“single”。

常用的设备无关启动时参数。

1、init＝...

设置内核执行的初始化进程名，如果该项没有设置，内核会按顺序尝试/etc/init，/bin/init，/sbin/init， /bin/sh，如果所有的都没找到，内核会抛出 kernel panic：的错误。

2、nfsaddrs=...

设置从网络启动时 NFS 的启动地址，已字符串的形式给出。

3、nfsroot=...

设置网络启动时的 NFS 根名字，如果该字符串不是以 "/"、","、"."开始，默认指向“/tftp-boot”。

以上 2、3 在无盘站中很有用处。

4、no387

该选项仅当定义了 CONFIG_BUGi386 时才能用，某些 i387 协处理器芯片使用 32 位的保护模式时会有 BUG，比如一些浮点运算，使用这个参数可以让内核忽略 387 协处理器。

5、no-hlt

该选项仅当定义了 CONFIG_BUGi386 时才能用，一些早期的 i486DX-100 芯片在处理“hlt”指令时会有问题，执行该指令后不能可靠的返回操作系统，使用该选项，可以让 Linux 系统在 CPU 空闲的时候不要挂起 CPU。

6、root=...

该参数告诉内核启动时使用哪个设备作为根文件系统。比如可以指定根文件为 hda8：root=/dev/hda8。

7、ro 和 rw

ro 参数告诉内核以只读方式加载根文件系统，以便进行文件系统完整性检查，比如运行 fsck；rw 参数告诉内核以读写方式加载根文件系统，这是默认值。

8、reserve=...

保留端口号。格 式：reserve=iobase,extent[,iobase,extent]...，用来保护一定区域的 I/O 端口不被设备驱动程序自动探测。在某 些机器上，自动探测会失败，或者设备探测错误或者不想让内核初始化设备时会用到该参数；比如： reserve=0x300,32 device=0x300，除 device=0x300 外所有设备驱动不探测 0x300-0x31f 范围的 I/O 端口。

9、mem=...

限制内核使用的内存数量。早期 BIOS 设计为只能识别 64M 以下的内存，如果你的内存数量大于 64M，你可以指明，如果你指明的数量超过了实际安装的内存数量，系统崩溃是迟早的事情。 如：mem=0x1000000 意味着有 16M 内存，如果是 mem=0x6000000，就是 96M 内存了。

注意：很多机型把部分内存作为 BIOS 的映射，所以你在指定内存大小的时候一定要预留空间。你也可以在 pentium 或者更新的 CPU 上使用 mem=nopentium 关闭 4M 的页表，这要在内核配置时申明。

10、panic=N

默认情况，内核崩溃－－kernel panic 后会宕机而不会重启，你可以设置宕机多少秒之后重启机器；也可以在/proc/sys/kernel/panic 文件里设置。

11、reboot=[warm|cold][,[bios|hard]]

该选项仅当定义了 CONFIG_BUGi386 时才能用。2.0.22 的内核重启默认为 cool reboot，warm reboot 更快，使用"reboot=bios"可以继承 bios 的设置。

12、nosmp 和 maxcpus=N

仅当定义了 __SMP__，该选项才可用。可以用来禁用多 CPU 或者指明最多支持的 CPU 个数。

内核开发和调试的启动时参数

这些参数主要用在内核的开发和调试上，如果你不进行类似的工作，你可以简单的跳过本小节。

1、debug

Linux 的日志级别比较多(详细信息可以参看 Linux/kernel.h)，一般地，日志的守护进程 klogd 只把比 DEBUG 级别高的日志写进磁盘；如果使用该选项，klogd 也把内核的 DEBUG 信息写进日志。

2、profile=N

在做内核开发的时候，如果想清楚的 知道内核在什么地方耗用了多少 CPU 的时钟周期，可以使用核心的分析函数设置变量 prof_shift 为非 0 值，有两种方式可以实现：一种是在编译时指 定，另一种就是通过“profile=”来指定； 他给出了一个相当于最小单位－－即时钟周期；系统在执行内核代码的时候， profile[address >;>; prof_shift]的值就会累加，你也可以从/proc/profile 得到关于它的一些信息。

3、swap=N1,N2,N3,N4,N5,N6,N7,N8

设置内核交换算法的八个参数：max_page_age, page_advance, page_decline,page_initial_age, age_cluster_fract, age_cluster_min,

pageout_weight,bufferout_weight。

4、buff=N1,N2,N3,N4,N5,N6

设置内核缓冲内存管理的六个参数：max_buff_age, buff_advance, buff_decline,buff_initial_age, bufferout_weight, buffermem_grace。

使用 RAMDISK 的参数

(仅当内核配置并编译了 CONFIG_BLK_DEV_RAM)。一般的来说，使用 ramdisk 并不是一件好事，系统自己会更加有效的使用可用的内存；但是，在启动或者制作启 动盘时，使用 ramdisk 可以很方便的装载软盘等设备上的映象(尤其是安装程序、启动过程中)，因为在正真使用物理磁盘之前，必须要加载一些必要的模 块，比如文件系统模块，scsi 驱动等(可以参见我的 initrd-x.x.x.img 文件分析－制作安装程序不支持的根文件系统)。

早期的 ramdisk(比如 1.3.48 的核心)是静态分配的，必须以 ramdisk=N 来指定 ramdisk 的大小；现在 ramdisk 可以动态增加。一共有四个参数，两个布尔型，两个整形。

1、load_ramdisk=N

如果 N＝1，就加载 ramdisk；如果 N＝0，就不加载 ramdisk；默认值为 0。

2、prompt_ramdisk=N

N＝1，提示插入软盘；N＝0，不提示插入软盘；默认为 1。

3、ramdisk_size=N 或者 ramdisk=N

设定 ramdisk 的最大值为 N KB,默认为 4096KB。

4、ramdisk_start=N

设置 ramdisk 的开始块号为 N，当 ramdisk 有内核的映象文件是需要这个参数。

5、noinitrd

(仅当内核配置了选项 CONFIG_BLK_DEV_RAM 和 CONFIG_BLK_DEV_INITRD)现在的内核都可以支持 initrd 了，引导进程首先装载内核和一个 初始化的 ramdisk，然后内核将 initrd 转换成普通的 ramdisk，也就是读写模式的根文件系统设备。然后 Linuxrc 执行，然后装载真正的 根文件系统，之后 ramdisk 被卸载，最后执行启动序列，比如/sbin/init。

选项 noinitrd 告诉内核不执行上面的步骤，即使内核编译了 initrd，而是把 initrd 的数据写到 /dev/initrd，只是这是一个一次性的设备。

8) printk 内核打印

printk

编辑

本词条缺少**名片图**，补充相关内容使词条更完整，还能快速升级，赶紧来编辑吧！

printk 相当于 printf 的孪生姐妹，她们一个运行在用户态，另一个则在内核态被人们所熟知。但是根据不同的操作系统也会有不一样的效果，例如编写一个 hello word 内核模块，使用这个函数不一定会将内容显示到终端上，但是一定在内核缓冲区里，可以使用 dmesg.查看效果。

中文名

printk

属　　于

内核中运行

级　　别

可以指定输出的优先级

1 printk 概述

2 原型

3 输出格式

1)printk 概述 编辑

对于做嵌入式或者熟悉 linux 内核的人来说，对 printk 这个函数一定不会感到陌生。

printk 是在内核中运行的向控制台输出显示的函数，Linux 内核首先在内核空间分配一个静态缓冲区， 作为显示用的空间，然后调用 sprintf，格式化显示字符串，最后调用 tty_write 向终端进行信息的显示。但是根据不同的操作系统也会有不一样的 效果，例如编写一个 hello word 内核模块，使用这个函数不一定会将内容显示到终端上，但是一定在内核缓冲区里，可以使用 dmesg.查看效果。

printk 与 printf 的差异，是什么导致一个运行在内核态而另一个运行用户态？其实这两个函数几乎是相同的， 出现这种差异是因为 tty_write 函数需要使用 fs 指向的被显示的字符串，而 fs 是专门用于存放用户态段选择符的，因此，在内核态时，为了配合 tty_write 函数，printk 会把 fs 修改为内核态数据段选择符 ds 中的值，这样才能正确指向内核的数据缓冲区，当然这个操作会先对 fs 进行压栈保存，调用 tty_write 完毕后再出栈恢复。总结说来，printk 与 printf 的差异是由 fs 造成的，所以差异也是围绕对 fs 的处理。

2)原型 编辑

const char * fmt(...);

【示例】

与大多数展示 printf 的功能一样，我们也用一个 helloworld 的程序来演示 printk 的输出：

编写一个内核模块：

```
#include<linux/kernel.h>
#include<linux/module.h>
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include<linux/modversions.h>
#endif
MODULE_LICENSE("GPL");
int init_module()
{
printk("hello.word-this is the kernel speaking\n");
return 0;
}
void cleanup_module()
{
printk("Short is the life of a kernel module\n");
}
```

保存为文件 hello.c

编写一个 Makefile:

```
CC=gcc
MODCFLAGS:=-O6 -Wall -DMODULE -D__KERNEL__ -DLINUX
hello.o:hello.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c hello.c
echo insmod hello.o to turn it on
```

保存为文件 Makefile

执行 make

我们可以看到生成了一个 hello.o 的内核模块，我们想通过这个模块在插入内核的时候输出

"hello.word-this is the kernel speaking"

这样一条信息。

然后我们开始：

[root@localhost root]# insmod hello.o

[root@localhost root]#

并没有输出任何消息。why?

这也是 printf 和 printk 的一个不同的地方

用 printk，内核会根据日志级别，可能把消息打印到当前控制台上，这个控制台通常是一个字符模式的终端、一个串口打印机或是一个并口打印机。这些消息正常输出的前提是——日志输出级别小于 console_loglevel（在内核中数字越小优先级越高）。

没有指定日志级别的 printk 语句默认采用的级别是 DEFAULT_ MESSAGE_LOGLEVEL（这个默认级别一般为<4>,即与 KERN_WARNING 在一个级别上），其定义在 linux26/kernel/printk.c 中可以找到

日志级别一共有 8 个级别，printk 的日志级别定义如下（在 include/linux/kernel.h 中）：

#define KERN_EMERG 0/*紧急事件消息，系统崩溃之前提示，表示系统不可用*/

#define KERN_ALERT 1/*报告消息，表示必须立即采取措施*/

#define KERN_CRIT 2/*临界条件，通常涉及严重的硬件或软件操作失败*/

#define KERN_ERR 3/*错误条件，驱动程序常用 KERN_ERR 来报告硬件的错误*/

#define KERN_WARNING 4/*警告条件，对可能出现问题的情况进行警告*/

#define KERN_NOTICE 5/*正常但又重要的条件，用于提醒*/

#define KERN_INFO 6/*提示信息，如驱动程序启动时，打印硬件信息*/

#define KERN_DEBUG 7/*调试级别的消息*/

现在我们来修改 hello.c 程序，使 printk 的输出级别为最高：

printk("<0>""hello.word-this is the kernel speaking\n");

然后重新编译 hello.o,并插入内核：

[root@localhost root]# insmod hello.o

[root@localhost root]#

Message from syslogd@localhost at Sat Aug 15 05:32:22 2009 ...

localhost kernel: hello.word-this is the kernel speaking

hello,world 信息出现了。

其实 printk 始终是能输出信息的，只不过不一定是到了终端上。我们可以去 /var/log/messages 这个文件里面去查看。

如果 klogd 没有运行，消息不会传递到用户空间，只能查看/proc/kmsg

通过读写/proc/sys/kernel/printk 文件可读取和修改控制台的日志级别。查看这个文件的方法如下：

#cat /proc/sys/kernel/printk 6 4 1 7

上面显示的 4 个数据分别对应控制台日志级别、默认的消息日志级别、最低的控制台日志级别和默认的控制台日志级别。

可用下面的命令设置当前日志级别：

echo 8 > /proc/sys/kernel/printk

这样所有级别<8,(0-7)的消息都可以显示在控制台上.

3)输出格式 编辑

printk 函数：

KERN_EMERG"<0>"/*紧急事件消息，系统崩溃之前提示，表示系统不可用*/

KERN_ALERT"<1>"/*报告消息，表示必须立即采取措施*/

KERN_CRIT"<2>"/*临界条件，通常涉及严重的硬件或软件操作失败*/

KERN_ERR"<3>"/*错误条件，驱动程序常用 KERN_ERR 来报告硬件的错误*/

KERN_WARNING"<4>"/*警告条件，对可能出现问题的情况进行警告*/

KERN_NOTICE"<5>"/*正常但又重要的条件，用于提醒。常用于与安全相关的消息*/

KERN_INFO"<6>"/*提示信息，如驱动程序启动时，打印硬件信息*/

KERN_DEBUG"<7>"/*调试级别的消息*/

如果变量类型是 ，使用 prink 的格式说明符 ：

int %d 或者 %x(注： %d 是十进制， %x 是十六进制)

unsigned int %u 或者 %x

long %ld 或者 %lx

unsigned long %lu 或者 %lx

long long %lld 或者 %llx

unsigned long long %llu 或者 %llx

size_t %zu 或者 %zx

ssize_t %zd 或者 %zx

原始指针值必须用 %p 输出。

u64，即(unsigned long long)，必须用 %llu 或者 %llx 输出，如：

printk("%llu", (unsigned long long)u64_var);

s64，即(long long)，必须用 %lld 或者 %llx 输出，如 ：

printk("%lld", (long long)s64_var);

如果 (变量类型)<type> 的长度依赖一个配置选项 (例如： sector_t, blkcnt_t, phys_addr_t, resource_size_t) 或者 依赖相关的体系结构（例如： tcflag_t ），使用一个可能最大类型的格式说明符，并且显示转换它。如：

printk("test: sector number/total blocks: %llu/%llu\n",(unsigned long long)sector, (unsigned long long)blockcount);

9) linux 下显示隐藏文件

1，在图形命令下，用文件浏览器打开文件夹，按下 ctrl+h 组合键可显示隐藏文件合文件夹，再按一次取消显示。

2，也可以使用命令行显示。打开终端，输入 ls -a 即可显示所有的文件合文件夹，包括隐藏的文件和文件夹。

10) linux cp 说明

拷贝文件和目录是每一个操作系统的基本指令。备份行为基本上是创建文件和目录的副本。在 Linux 系统下，我们可以用 **cp** 命令来实现。

copy 命令是什么

正如我们在上文提到的，cp 是一个用来创建文件和目录副本的命令。在这里我们提供了一些在日常操作中可能用到的 **cp** 命令的实例。

1. 不带任何参数下，运行 cp

这是 cp 命令最基础的使用。 拷贝名为 myfile.txt 从一个位置到另一个位置，我们可以像这样子输入：

```
$ cp myfile.txt /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls /home/pungki/office/
movie  movie.list  music
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp myfile.txt /home/pungki/office/
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls /home/pungki/office/
movie  movie.list  music  myfile.txt
pungki@dev-machine:~/Documents$
```

如果我们没有输入绝对路径，这意味着我们正在当前目录下拷贝一个文件。在上面的实例中，myfile.txt 位于/home/pungki/Documents 目录下。如果我们当前目录正是/home/pungki/Documets,那么没有必要输入/home/pungki/Documents/myfile.txt 来拷贝文件。当/home/pungki/office 是一个目录，则文件会拷贝到里面。

2. 同时拷贝多个文件

要在同时拷贝多个文件，我们只需要将多个文件用空格隔开。如下示例：

```
$ cp file_1.txt file_2.txt file_3.txt /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls
file_1.txt  file_2.txt  file_3.txt  myfile.txt  temp
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp file_1.txt file_2.txt file_3.txt /home/pungki
/office/
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls /home/pungki/office/
file_1.txt  file_2.txt  file_3.txt  movie  movie.list  music  myfile.txt
pungki@dev-machine:~/Documents$
```

3. 拷贝一个目录

要拷贝一个目录的话会有点棘手。你需要添加 **-r** 或者 **-R** 选项来实现。**-r** 或**-R** 选项表明递归操作。无论该目录是否为空目录，**这个选项都是必要的**。如下示例：

```
$ cp -r directory_1 /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls
directory_1  file_1.txt  file_2.txt  file_3.txt  myfile.txt  temp
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp -r directory_1 /home/pungki/office/
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls /home/pungki/office/
directory_1  file_2.txt  movie       music
file_1.txt   file_3.txt  movie.list  myfile.txt
pungki@dev-machine:~/Documents$
```

需要注意的一件事，你需要移除在目录名尾部的斜杠。否则你会收到类似的错误信息 cp: omitting directory ‘directory_1/’

```
pungki@dev-machine:~/Documents$ ls
directory_1  file_1.txt  file_2.txt  file_3.txt  myfile.txt  temp
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp directory_1/ /home/pungki/office/
cp: omitting directory ‘directory_1/’
pungki@dev-machine:~/Documents$
```

如果你收到错误信息，则目录不会被拷贝到目标文件夹。

4. 创建文件的硬链接，而不是拷贝它们

拷贝文件意味着你必须使用一些存储空间来储存拷贝的文件。有时候出于某种原因，你可能想要创建“快捷方式”或者链接到文件，而不是拷贝它们。要做到这一点，我们可以使用**-l** 选项。

```
$ cp -l file_4.txt /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls -lvi *.txt
835337 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_1.txt
835372 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_2.txt
835373 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_3.txt
835386 -rw-rw-r-- 2 pungki pungki 54 Jan 11 07:36 file_4.txt
835545 -rw-rw-r-- 1 pungki pungki  0 Jan 10 03:52 myfile.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp -l file_4.txt /home/pungki/office/
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls -lvi /home/pungki/office/*.txt
279041 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 /home/pungki/office/file_1.txt
279042 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 /home/pungki/office/file_2.txt
279229 -rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 /home/pungki/office/file_3.txt
835386 -rw-rw-r-- 3 pungki pungki 54 Jan 11 07:36 /home/pungki/office/file_4.txt
279034 -rw-rw-r-- 1 pungki pungki  0 Jan 10 03:55 /home/pungki/office/myfile.txt
pungki@dev-machine:~/Documents$
```

从上图看出，我们看到 **file_4.txt** 的硬链接已经拷贝到**/home/pungki/office/file_4.txt**。标记有同样的 inode, **835386**。但是请注意，硬链接不能用来创建目录。下面让我们看一个例子。

原目录 directory_1 的 inode 值是 278230

```
pungki@dev-machine:~/Documents$ ls -lvi |grep directory_1
279230 drwxrwxr-x 2 pungki pungki 4096 Jan 10 05:30 directory_1
pungki@dev-machine:~/Documents$
```

原文件 file_5.txt 的 inode 值是 279231

```
pungki@dev-machine:~/Documents$ ls -lvi directory_1/
total 0
279231 -rw-rw-r-- 1 pungki pungki 0 Jan 10 05:30 file_5.txt
pungki@dev-machine:~/Documents$
```

对 directory_1 执行 cp 命令

```
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp -rl directory_1/ /home/pungki/office/
pungki@dev-machine:~/Documents$
```

拷贝的 directory_1 副本的 inode 值是 274800

```
pungki@dev-machine:~/Documents$ ls -lvi /home/pungki/office/ |grep directory_1
274800 drwxrwxr-x 2 pungki pungki 4096 Jan 11 09:35 directory_1
pungki@dev-machine:~/Documents$
```

拷贝的 file_5.txt 副本的 inode 值是 279231。跟它的原文件一样

```
pungki@dev-machine:~/Documents$ ls -lvi /home/pungki/office/directory_1/
total 0
279231 -rw-rw-r-- 2 pungki pungki 0 Jan 10 05:30 file_5.txt
pungki@dev-machine:~/Documents$
```

5. 创建文件的符号链接

也有一种链接叫做 **软链接** 或 **符号链接**。我们用 **-s** 选项来实现。下面是命令的示例。

```
$ cp -s /home/pungki/Documents/file_6.txt file_6.txt
```

创建符号链接只能在当前目录下进行。在上面的截图中，我们想要创建符号链接 **/home/pungki/office/file6.txt** 指向原文件 **/home/pungki/Documents/file6.txt**。但是为了创建符号链接，我必须在将**/home/pungki/office** 作为目标目录。一旦我设法进入目录，我就可以向上面一样运行 **cp -s** 命令。

现在你列出文件详情，你会看到**/home/pungki/office/file_6.txt** 指向了原文件。在其文件名后标记了箭头符号。

```
pungki@dev-machine:~/office$ cp -s /home/pungki/Documents/file_6.txt file_6.txt
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$ ls -l *.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_1.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_2.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_3.txt
-rw-rw-r-- 3 pungki pungki 54 Jan 11 07:36 file_4.txt
lrwxrwxrwx 1 pungki pungki 33 Jan 11 13:04 file_6.txt -> /home/pungki/Documents/
file_6.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 03:55 myfile.txt
pungki@dev-machine:~/office$
```

6. 不随符号链接拷贝原文件

我们可以用 **-P** 选项来实现。当对符号链接使用 cp 命令，它会照原样拷贝它自身。来看看下面的示例。

```
$ cp -P file_6.txt ./movie
```

```
pungki@dev-machine:~/office$ ls -l file_6.txt
lrwxrwxrwx 1 pungki pungki 33 Jan 11 13:04 file_6.txt -> /home/pungki/Documents/
file_6.txt
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$ cp -P file_6.txt ./movie
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$ ls -l ./movie/file_6.txt
lrwxrwxrwx 1 pungki pungki 33 Jan 11 17:47 ./movie/file_6.txt -> /home/pungki/Do
cuments/file_6.txt
pungki@dev-machine:~/office$
```

如你所见，cp 命令照原样拷贝 **file_6.txt** 自身。文件类型仍然是一个符号链接。

7. 随符号链接拷贝原文件

现在我们可以试一下**-L** 选项。基本上，这个刚好与上面的 -P 选项 相反。下面是个示例：

```
$ cp -L file_6.txt ./movie
```

```
pungki@dev-machine:~/office$ ls -l file_6.txt
lrwxrwxrwx 1 pungki pungki 33 Jan 11 13:04 file_6.txt -> /home/pungki/Documents/
file_6.txt
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$ cp -L file_6.txt ./movie
pungki@dev-machine:~/office$
pungki@dev-machine:~/office$ ls -l ./movie/file_6.txt
-rw-rw-r-- 1 pungki pungki 50 Jan 11 17:46 ./movie/file_6.txt
pungki@dev-machine:~/office$
```

使用这个选项，拷贝的文件将会和**-L**file_6.txt 原文件一样。我们可以从文件大小看出来。拷贝的文件有**-L**50 字节而当**-L**file_6.txt 作为符号链接时文件大小只有**-L**33 字节。

8. 文件归档

当我们去拷贝一个目录时，我们会用**-L**-r 或者**-L**-R 选项。但是我们也可以用**-a** 选项来归档文件。这样会创建文件和目录的**-L** 准确套录，如果有的话也可以包括符号链接。下面是示例：[译注：-a 会保留原文件或目录的属性]

```
$ cp -a directory_1/ /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls -l directory_1/
total 6572
-rw-rw-r-- 1 pungki pungki       0 Jan 10 05:30 file_5.txt
lrwxrwxrwx 1 pungki pungki      33 Jan 11 18:10 file_6.txt -> /home/pungki/Docum
ents/file_6.txt
-rw-rw-r-- 1 pungki pungki 6727680 Jan 11 18:04 testdisk-6.14.linux26.tar
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp -a directory_1/ /home/pungki/office/
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls -l /home/pungki/office/directory_1/
total 6572
-rw-rw-r-- 1 pungki pungki       0 Jan 10 05:30 file_5.txt
lrwxrwxrwx 1 pungki pungki      33 Jan 11 18:10 file_6.txt -> /home/pungki/Docum
ents/file_6.txt
-rw-rw-r-- 1 pungki pungki 6727680 Jan 11 18:04 testdisk-6.14.linux26.tar
pungki@dev-machine:~/Documents$
```

上列的命令会拷贝一个名为 directory1 的目录到**-L**/home/pungki/office 目录下。如你所见，**file6.txt**依然作为符号链接被复制。

9. 显示正在做什么

默认情况下，当拷贝作业成功时，我们仅仅会再次看到命令提示符。如果你想了解在拷贝文件时都发生了什么，我们可以用 -v 选项。

```
$ cp -v *.txt /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ cp -v *.txt /home/pungki/office/
'file_1.txt' -> '/home/pungki/office/file_1.txt'
'file_2.txt' -> '/home/pungki/office/file_2.txt'
'file_3.txt' -> '/home/pungki/office/file_3.txt'
'file_4.txt' -> '/home/pungki/office/file_4.txt'
'file_6.txt' -> '/home/pungki/office/file_6.txt'
'myfile.txt' -> '/home/pungki/office/myfile.txt'
pungki@dev-machine:~/Documents$
```

当我们从当前目录下拷贝所有的 txt 文件到 **/home/pungki/office** 目录，**-v** 选项会显示正在操作的过程。这些额外的信息会帮助我们了解更多拷贝过程。

10. 当原文件较目标文件新时拷贝

我们用 **-u** 选项来实现。下面是具体示例:

```
$ cp -vu *.txt /home/pungki/office
```

```
pungki@dev-machine:~/Documents$ ls -l *.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 11 23:14 file_1.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_2.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_3.txt
-rw-rw-r-- 2 pungki pungki 54 Jan 11 07:36 file_4.txt
-rw-rw-r-- 1 pungki pungki 50 Jan 11 13:01 file_6.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 03:52 myfile.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ vi file_1.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls -l *.txt
-rw-rw-r-- 1 pungki pungki 36 Jan 11 23:14 file_1.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_2.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_3.txt
-rw-rw-r-- 2 pungki pungki 54 Jan 11 07:36 file_4.txt
-rw-rw-r-- 1 pungki pungki 50 Jan 11 13:01 file_6.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 03:52 myfile.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp -vu *.txt /home/pungki/office/
'file_1.txt' -> '/home/pungki/office/file_1.txt'
pungki@dev-machine:~/Documents$
```

起初我们看到 **file_1.txt** 是 **0** 字节大小。然后我们用 vi 编辑，加入一些内容并保存。接

下来,我们发现文件大小已经变为了 36 个字节。与此同时在**/home/pungki/office** 目录中,我们**已经包含了所有 txt 文件**。当我们用 -u 选项,结合 -v 选项来查看具体操作,cp 命令会只拷贝比目标目录下新的文件。因此,我们看到只有 **file_1.txt 拷贝到 /home/pungki/office** 目录下。

11. 使用交互模式

交互模式下会询问是否覆盖目标目录下的文件。使用 **-i** 选项,启用交互模式。

```
$ cp -ir directory_1/ /home/pungki/office/
```

```
pungki@dev-machine:~/Documents$ cp -ir directory_1/ /home/pungki/office/
pungki@dev-machine:~/Documents$ cp -ir directory_1/ /home/pungki/office/
cp: overwrite '/home/pungki/office/directory_1/testdisk-6.14.linux26.tar'? y
cp: overwrite '/home/pungki/office/directory_1/file_5.txt'? y
cp: overwrite '/home/pungki/office/directory_1/file_6.txt'? y
pungki@dev-machine:~/Documents$
```

12. 创建备份文件

当目标目录已经含有同名文件,默认情况下 cp 命令会覆盖目标目录下的同名文件。使用 **--backup** 选项,cp 命令会为每一个现有的目标文件做一个备份。../office 相对于 /home/pungki/office 。下面是示例:

```
$ cp --backup=simple -v *.txt ../office
```

```
pungki@dev-machine:~/Documents$ ls -l /home/pungki/office/*.txt
-rw-rw-r-- 1 pungki pungki 36 Jan 11 19:49 /home/pungki/office/file_1.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 11 19:49 /home/pungki/office/file_2.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 11 19:49 /home/pungki/office/file_3.txt
-rw-rw-r-- 1 pungki pungki 54 Jan 11 19:49 /home/pungki/office/file_4.txt
-rw-rw-r-- 1 pungki pungki 50 Jan 11 19:49 /home/pungki/office/file_6.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 11 19:49 /home/pungki/office/myfile.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp --backup=simple -v *.txt ../office/
'file_1.txt' -> '../office/file_1.txt' (backup: '../office/file_1.txt~')
'file_2.txt' -> '../office/file_2.txt' (backup: '../office/file_2.txt~')
'file_3.txt' -> '../office/file_3.txt' (backup: '../office/file_3.txt~')
'file_4.txt' -> '../office/file_4.txt' (backup: '../office/file_4.txt~')
'file_6.txt' -> '../office/file_6.txt' (backup: '../office/file_6.txt~')
'myfile.txt' -> '../office/myfile.txt' (backup: '../office/myfile.txt~')
pungki@dev-machine:~/Documents$
```

正如我们看到的,**--backup=simple** 选项会创建一个在文件名末尾用波浪符标记(~)的备份文件。**--backup** 选项也有一些其他控制:

none, off:从不备份(即使给出 --backup)

numbered, t:用编号备份

existing, nil :如果编号备份存在则使用编号备份,否者用简易备份[译注:也就是用波浪号]

simple, never :总是使用简易备份

13. 只拷贝文件属性

cp 命令也提供给我们 **--attributes-only** 选项。顾名思义,这个选项只会拷贝文件名及其属性,不会拷贝任何数据。下面是示例:

```
$ cp --attributes-only file_6.txt -v ../office
```

```
pungki@dev-machine:~/Documents$ ls -l *.txt
-rw-rw-r-- 1 pungki pungki 36 Jan 11 19:24 file_1.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_2.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 04:22 file_3.txt
-rw-rw-r-- 2 pungki pungki 54 Jan 11 07:36 file_4.txt
-rw-rw-r-- 1 pungki pungki 50 Jan 11 13:01 file_6.txt
-rw-rw-r-- 1 pungki pungki  0 Jan 10 03:52 myfile.txt
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ cp --attributes-only file_6.txt -v ../office
'file_6.txt' -> '../office/file_6.txt'
pungki@dev-machine:~/Documents$
pungki@dev-machine:~/Documents$ ls -l /home/pungki/office/file_6.txt
-rw-rw-r-- 1 pungki pungki 0 Jan 11 20:01 /home/pungki/office/file_6.txt
pungki@dev-machine:~/Documents$
```

从上图看出，原文件 file_6.txt 有 50 字节大小。用了 **--attributes-only** 选项，拷贝的文件只有 0 字节大小。这是因为文件内容并没有拷贝。

14. 强制拷贝

用了 **-f** 选项会强制进行拷贝操作。如果目标文件不能打开，可以用 -f 尝试一下。

```
$ cp -f *.txt -v ../office
```

```
pungki@dev-machine:~/Documents$ cp -f *.txt -v ../office/
'file_1.txt' -> '../office/file_1.txt'
'file_2.txt' -> '../office/file_2.txt'
'file_3.txt' -> '../office/file_3.txt'
'file_4.txt' -> '../office/file_4.txt'
'file_6.txt' -> '../office/file_6.txt'
'myfile.txt' -> '../office/myfile.txt'
pungki@dev-machine:~/Documents$
```

15. 在拷贝之前先删除目标

我们可以用，**--remove-destination 选项** 实现。这个选项与上面的**-f 选项**形成对照。如果 cp 命令在目标目录下发现同名文件，cp 命令会先删除目标文件，然后再拷贝一份新的。下面是示例：

```
$ cp --remove-destination *.txt -v ../office
```

```
pungki@dev-machine:~/Documents$ cp --remove-destination *.txt -v ../office/
removed '../office/file_1.txt'
'file_1.txt' -> '../office/file_1.txt'
removed '../office/file_2.txt'
'file_2.txt' -> '../office/file_2.txt'
removed '../office/file_3.txt'
'file_3.txt' -> '../office/file_3.txt'
removed '../office/file_4.txt'
'file_4.txt' -> '../office/file_4.txt'
removed '../office/file_6.txt'
'file_6.txt' -> '../office/file_6.txt'
removed '../office/myfile.txt'
'myfile.txt' -> '../office/myfile.txt'
pungki@dev-machine:~/Documents$
```

总结

cp 命令是 Linux 下最基础的命令之一。对于那些想要学习 Linux 的人，必须得把这个命令掌握。当然你也可以在你的终端下键入 **man cp** 或者 **cp --help** 来显示更多帮助信息。

11) subsys_initcall 说明

http://my.oschina.net/u/572632/blog/305492

摘要 linux 子系统的初始化_subsys_initcall()

kernel

(1)概述

内核选项的解析完成之后，各个子系统的初始化即进入第二部分—入口函数的调用。通常 USB、PCI 这样的子系统都会有一个名为 subsys_initcall 的入口，如果你选择它们作为研究内核的切入点，那么就请首先找到它。

(2)section 的声明

C 语言中 attribute 属性的 section 是在目标文件链接时可以用于主动定制代码的位置，具体可以 WIKI，下面看 linux kernel 中是如何定义的。

以下代码来自 linux 内核源码中 include/linux/init.h 文件。下面使用相同语法规则的变量名存放了各个初始化函数的地址。

更重要的是其 section 属性也是按照一定规则构成的。

于 section 见 http://lihuize123123.blog.163.com/blog/static/878290522010420111428109/

```
1  /* initcalls are now grouped by functionality into separate
2    * subsections. Ordering inside the subsections is determined
3    * by link order.
4    * For backwards compatibility, initcall() puts the call in
5    * the device init subsection.
6    *
7    * The `id' arg to __define_initcall() is needed so that multiple initcalls
8    * can point at the same handler without causing duplicate-symbol build errors.
9    */
10
11#define __define_initcall(level,fn,id) \
12        static initcall_t __initcall_##fn##id __used \
13        __attribute__((__section__(".initcall" level ".init"))) = fn
14
15/*
16  * Early initcalls run before initializing SMP.
17  *
18  * Only for built-in code, not modules.
19  */
20#define early_initcall(fn)          __define_initcall("early",fn,early)
21
22/*
23  * A "pure" initcall has no dependencies on anything else, and purely
24  * initializes variables that couldn't be statically initialized.
25  *
26  * This only exists for built-in code, not for modules.
27  */
28#define pure_initcall(fn)           __define_initcall("0",fn,0)
29
30#define core_initcall(fn)           __define_initcall("1",fn,1)
31#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
32#define postcore_initcall(fn)           __define_initcall("2",fn,2)
33#define postcore_initcall_sync(fn) __define_initcall("2s",fn,2s)
34#define arch_initcall(fn)           __define_initcall("3",fn,3)
35#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
36#define subsys_initcall(fn)             __define_initcall("4",fn,4)
37#define subsys_initcall_sync(fn)        __define_initcall("4s",fn,4s)
38#define fs_initcall(fn)                 __define_initcall("5",fn,5)
39#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
40#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
41#define device_initcall(fn)             __define_initcall("6",fn,6)
42#define device_initcall_sync(fn)        __define_initcall("6s",fn,6s)
43#define late_initcall(fn)           __define_initcall("7",fn,7)
44#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
45
46#define __initcall(fn) device_initcall(fn)
47
48#define __exitcall(fn) \
49        static exitcall_t __exitcall_##fn __exit_call = fn
```

```
50
51#define console_initcall(fn) \
52        static initcall_t __initcall_##fn \
53        __used __section(.con_initcall.init) = fn
54
55#define security_initcall(fn) \
56        static initcall_t __initcall_##fn \
57        __used __section(.security_initcall.init) = fn
```

(3)注册

这些入口有个共同的特征，它们都是使用 __define_initcall 宏定义的。它们的调用也不是随便的，而是按照一定顺序的，这个顺序就取决于__define_initcall 宏。 __define_initcall 宏用来将指定的函数指针放到.initcall.init 节里。

.initcall.init 节

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多 节，如文本、数据、init 数据、bass 等等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节 映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。vmlinux.lds 是存在于 arch/<target>/目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量 处。在 vmlinux.lds 文件里查找 initcall.init 就可以看到下面的内容

```
1  #define INITCALLS                                        \
2          *(.initcallearly.init)                           \
3         VMLINUX_SYMBOL(__early_initcall_end) = .;                 \
4            *(.initcall0.init)                             \
5            *(.initcall0s.init)                            \
6            *(.initcall1.init)                             \
7            *(.initcall1s.init)                            \
8            *(.initcall2.init)                             \
9            *(.initcall2s.init)                            \
10           *(.initcall3.init)                             \
11           *(.initcall3s.init)                            \
12           *(.initcall4.init)                             \
13           *(.initcall4s.init)                            \
14           *(.initcall5.init)                             \
15           *(.initcall5s.init)                            \
16        *(.initcallrootfs.init)                           \
17           *(.initcall6.init)                             \
18           *(.initcall6s.init)                            \
19           *(.initcall7.init)                             \
20           *(.initcall7s.init)
```

这就告诉我们.initcall.init 节又分成了 7 个子节，而 xxx_initcall 入口函数指针具体放在哪一个子节里边儿是由 xxx_initcall 的定义中，__define_initcall 宏的参数决定 的，比如 core_initcall 将函数指针放在.initcall1.init 子节，device_initcall 将函数指针放在 了.initcall6.init 子节等等。各个子节的顺序是确定的，即先调用.initcall1.init 中的函数指针再调用.initcall2.init 中的函数指针，等等。不同的入口函数被放在不同的子节中，因此也就决定了它们的调用顺序。

注意：设备驱动程序中常见的 module_init(x)函数，查看 init.h 文件发现

```
1  /**
2    * module_init() - driver initialization entry point
3    * @x: function to be run at kernel boot time or module insertion
4    *
5    * module_init() will either be called during do_initcalls() (if
6    * builtin) or at module insertion time (if a module).    There can only
7    * be one per module.
8    */
9  #define module_init(x) __initcall(x);
10
```

```
11#define __initcall(fn) device_initcall(fn)
12
13/* Don't use these in modules, but some people do... */
14#define early_initcall(fn)         module_init(fn)
15#define core_initcall(fn)          module_init(fn)
16#define postcore_initcall(fn)          module_init(fn)
17#define arch_initcall(fn)          module_init(fn)
18#define subsys_initcall(fn)             module_init(fn)
19#define fs_initcall(fn)                 module_init(fn)
20#define device_initcall(fn)             module_init(fn)
21#define late_initcall(fn)          module_init(fn)
```

```
1  #define __define_initcall(level,fn) \
2         static initcall_t __initcall_##fn __used \
3         __attribute__((__section__(".initcall" level ".init"))) = fn
4
5  /* Userspace initcalls shouldn't depend on anything in the kernel, so we'll
6    * make them run first.
7    */
8  #define __initcall(fn) __define_initcall("1", fn)
9
10#define __exitcall(fn) static exitcall_t __exitcall_##fn __exit_call = fn
11
12#define __init_call        __used __section(.initcall.init)
```

这样推断 module_init 调用优先级为 6 低于 subsys_initcall 调用优先级 4.

在定义 MODULE 的情况下对 subsys_initcall 的定义，等价于使用 module_init。

(4)调用

那些入口函数的调用由 do_initcalls 函数来完成。

do_initcall 函数通过 for 循环，由__initcall_start 开始，直到__initcall_end 结束，依次调用识别到的初始化函数。而位于 __initcall_start 和__initcall_end 之间的区域组成了.initcall.init 节，其中保存了由 xxx_initcall 形式的宏标记的函数地址，do_initcall 函数可以很轻松的取得函数地址并执行其指向的函数。

.initcall.init 节所保存的函数地址有一定的优先级，越前面的函数优先级越高，也会比位于后面的函数先被调用。

由 do_initcalls 函数调用的函数不应该改变其优先级状态和禁止中断。因此，每个函数执行后，do_initcalls 会检查该函数是否做了任何变化，如果有必要，它会校正优先级和中断状态。

另外，这些被执行的函数有可以完成一些需要异步执行的任务，flush_scheduled_work 函数则用于确保 do_initcalls 函数在返回前等待这些异步任务结束。

```
1  static void __init do_initcalls(void)
2  {
3          initcall_t *fn;
4
5          for (fn = __early_initcall_end; fn < __initcall_end; fn++)
6                  do_one_initcall(*fn);
7
8          /* Make sure there is no pending stuff from the initcall sequence */
9          flush_scheduled_work();
10 }
11
12 int __init_or_module do_one_initcall(initcall_t fn)
13 {
14         int count = preempt_count();
15         int ret;
16
17         if (initcall_debug)
18                 ret = do_one_initcall_debug(fn);
19         else
20                 ret = fn();
```

```
21
22      msgbuf[0] = 0;
23
24      if (ret && ret != -ENODEV && initcall_debug)
25              sprintf(msgbuf, "error code %d ", ret);
26
27      if (preempt_count() != count) {
28              strlcat(msgbuf, "preemption imbalance ", sizeof(msgbuf));
29              preempt_count() = count;
30      }
31      if (irqs_disabled()) {
32              strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
33              local_irq_enable();
34      }
35      if (msgbuf[0]) {
36              printk("initcall %pF returned with %s\n", fn, msgbuf);
37      }
38
39      return ret;
40 }
```

12) rm 删除目录

rm a.txt　删除普通文件 a.txt

rm -r a/　删除目录 a

rm -rf　a/　强制删除目录 a

-f　表示强制

13) tar 打包

tar 命令支持 exclude 选项，将特定的文件和目录不打包，这样就能避免 log/.o/.a/.svn 等文件被打包进去的烦恼了

具体用法，假设要将 test 目录打包，test 目录下有 log src bin lib 等目录，其中 log 里有很多日志文件，lib 里有很多.o/.a 文件，这些是我们不想打包的

tar -czv --exclude=*.log --exclude=*.o --exclude-vcs -f test.tgz test

1.用 tar 打包文件时，带点的文件是无法打包到，在 linux 中，其中也就是以 "." 开头命名的文件，

①例如：.htaccess 文件

在打包时，加一个-rf 参数就行了！

tar -rf　htaccess.tar .htaccess　这样就把.htaccess 文件打包成功了！

②直接将隐藏文件所在的目录打包，例如：.htaccess 文件在 public_html 文件

只需要 tar zcf public_html.tar.gz public_html

③也可以用 find+tar 命令实现

```
tar cf file.tar $(find /path（打包文件的路径） -type f)
```

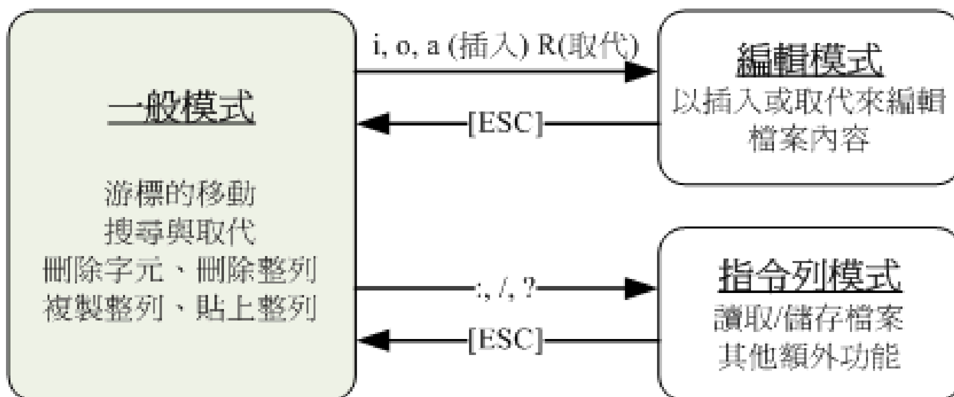
2.打包 public_html 这个文件，而这个 public_html 这个文件里的 cache 文件是你不想打包的。

tar cvf public_html --exclude cache

另：tar --exclude home/update/redhat/* --exclude lib --exclude usr/share （注：根目录的符号不加）

附录 2 VIM 图例及常用操作

1） vim 三种模式



2）vim 功能键

排序:(例: 1–5行排序:光标放在1行上,然后!5G,然后sort)
字体设定:(set guifont=Raize\ Bold\ 13)
键绑定:(如: map h <Insert>;map <F2> h{ } 按下F2后进入Insert模式并打印{ })

动作 移动光标或欲定义操作的范围
指令 直接执行的指令
红色指令进入编辑模式
操作 后接用以表示操作范围的指令
extra 特殊功能需额外输入

主要ex指令：
:w (保存)
:q (退出)
:e (打开文件) <Tab>
:%s/x/y/g (以y全文替换x)
:h (帮助)

其它重要指令：
Ctrl-R: 重做
Ctrl-F/-B: 向前(下)翻页/向后(上)翻页
Ctrl-E/-Y: 向前(下)一列/向后(上)一列
Ctrl-V: 切换visual模式

(1) 在复制/粘贴/删除 指令前使用 "x (x=a...z) 使用指令的寄存器(如: "ay$ 复制该行目前位置到行尾的内容到寄存器'a')
(2) 命令前添加数字 重复指定次数的操作 (如: 2p, d2w, 5i, d4j)
(3) 重复光标所在字符处指定的查找 (dd = 删除本行, >>= 行首缩进)
(4) ZZ 保存并退出
(5) zt: 移動游標所在行至畫面頂端, zb: 底端, zz: 中央
(6) gg: 文件首;ddp/P: 交换上下两行 daw: 删除单词;全选: ggVG

w,e,b指令
b(小写): quux(foo, bar, baz);
B(大写): quux(foo, bar, baz);

(数字)n + >> :行共同缩进(如:2>>表示这行到下一行共同缩进)

3）自己的~/.vimrc 的配置

```
"==
"Author :sundm75
"Version:1.0
"==

"按编程语言的语法，对代码进行彩色标示，术语叫做语法高亮
syntax on

"显示行数标示
set number

"打开状态栏的坐标信息
set ruler
```

```
"取消底部状态栏显示。1 为关闭，2 为开启。
set laststatus=1

"将输入的命令显示出来，便于查看当前输入的信息
set showcmd

"设置魔术匹配控制，可以通过:h magic 查看更详细的帮助信息
set magic

"设置 vim 存储的历史命令记录的条数
set history=100

"下划线高亮显示光标所在行
set cursorline

"插入右括号时会短暂地跳转到匹配的左括号
set showmatch

"搜索时忽略大小写
set ignorecase

"不对匹配的括号进行高亮显示
let loaded_matchparen=1

"在执行宏命令时，不进行显示重绘；在宏命令执行完成后，一次性重绘，以便提高性能。
set lazyredraw

"设置一个 tab 对应 4 个空格
set tabstop=4

"在按退格键时，如果前面有 4 个空格，则会统一清除
set softtabstop=4

"cindent 对 c 语法的缩进更加智能灵活，
"而 shiftwidth 则是在使用&lt;和&gt;进行缩进调整时用来控制缩进量。
"换行自动缩进，是按照 shiftwidth 值来缩进的
set cindent shiftwidth=4

"最基本的自动缩进
set autoindent shiftwidth=4

"比 autoindent 稍智能的自动缩进
set smartindent shiftwidth=4

"将新增的 tab 转换为空格。不会对已有的 tab 进行转换
set expandtab

"高亮显示搜索匹配到的字符串
set hlsearch

"在搜索模式下，随着搜索字符的逐个输入，实时进行字符串匹配，并对首个匹配到的字符串高亮显示
set incsearch

"设置自定义快捷键的前导键
let mapleader=","

"利用前导键加 b，则可以在一个单子两边加上大括号
map b wbi{<Esc>ea}<Esc>

"使用前导键加 w 来实现加速文件保存，来代替:w!加回车
nmap w :w!<CR>

"匹配那些末尾有空格或 TAB 的行。（es：Endspace Show）
map es :/.*\s\+$<CR>
```

```
"删除行末尾的空格或 TAB（ed：Endspace Delete）
map ed :s#\s\+$##<CR>

"如果所选行的行首没有#，则给所选行行首加上注释符#（#a：# add）
map #a :s/^\([^#]\s*\)/#\1/<CR>

"如果所选行行首有#，则将所选行行首所有的#都去掉（#d：# delete）
map #d :s/^#\+\(\s*\)/\1/<CR>

"如果所选行的行首没有//，则给所选行行首加上注释符//（/a：/ add）
map /a :s/^\([^\/\/]\s*\)/\/\/\1/<CR>

"如果所选行行首有//，则将所选行行首的//都去掉（/d：/ delete）
map /d :s/^\/\/\(\s*\)/\1/<CR>

"用 F6 隐藏 ^M
nmap <F6> :e    ++ff=dos<CR>

set fileencodings=utf-8,gbk,gb2312,gb18030
set termencoding=utf-8
set fileformats=unix,dos
set encoding=prc
```

附录 3 busybox 下载及配置

插入 U 盘、SD 等设备，即可在根目录的/media 目录下建立相应的文件夹，挂载对用的磁盘文件。

1.busybox 定义及版本

制作微型 Linux，要借助一个软件，这里讲的是 busybox，首先要知道 busybox 是什么？它 是一个含有很多个最常用 linux 命令和工具的软件，例如：ls，cp，echo, grep,mount 等。

在当前需要做一个程序，编译出来之后，这个程序要想运行它还需要依赖很多库文件，要向移植这个命令过去，就要把它所依赖的库文件也一并复制过去，这样它才能正常运行。前面制作的小 linux 的时候要移植 bash，先用 ldd 去查看它所依赖的库，这是因为使用动态连接的方式去编译的 程序。事实上，也完全可以实现将它所依赖的库直接编译进这个程序，这样可能会使程序的体积变大，但是把它移动到哪里都能直接用，因为所依赖的库都 直接做进里面了。那编译 busybox 的时候，为了让它移植的过程尽可能简化，直接编译 busybox 的时候，把它编译成静态的方式，把它所依赖 的库直接做进 busybox。在 http://www.busybox.net/

可以查看 busybox 的版本，开发版最新的是 BusyBox 1.24.1 (unstable)。这里使用的都是稳定版。

首先要在原有的虚拟机上装上一个 IDE 格式的硬盘，并且分两个 区/dev/hda1 和/dev/hda2。并且还要创建目录/mnt/boot 和/mnt/sysroot，然后把/dev/hda1、/dev /hda2 分别挂载到/mnt/boot、/mnt/sysroot 下面。(这个过程在前面制作简单 linux 的时候有详细版，相同的地方到/dev /hda1 和/dev/hda2 在/etc/fstab 文件中挂起，用 mount 查看一下是否挂起）这里就直接到的服务器上下载了，如果朋友们想 尝试一下，到网上下载稳定版的 BusyBox 1.22.1 (stable)。下面就开始演示这个过程。

二、装载微 Linux

2、开始下载 busybox

直接到服务器上下载了，如果朋友们想尝试一下，到网上下载稳定版的 BusyBox 1.20.2 (stable)。下载的命令是 get busybox-1.20.2.tar.bz2

因为最新版本的 busybox 要依赖更新版的内核头文件，我们使用的是 2.6.38 的内核，而它当中不具备新版的 busybox 所具备的某些功能，所以编译过程中很可能会出错

```
[root@localhost ~]# lftp 172.16.0.1  进入服务器
lftp 172.16.0.1:~> cd /pub/Sources/Busybox  busybox软件包放的路径
cd ok, cwd=/pub/Sources/Busybox
lftp 172.16.0.1:/pub/Sources/Busybox> ls
-rw-r--r--    1 500      500       1055142 Jul 23  2010 OpenGUI-5.5.14.tar.gz
-rw-r--r--    1 500      500        213992 Jul 23  2010 SDL-1.2.14-1.i586.rpm
-rw-r--r--    1 500      500       4014154 Jul 23  2010 SDL-1.2.14.tar.gz
-rw-r--r--    1 500      500       3945256 Jul 22  2010 buildroot-2010.05.tar.gz
-rw-r--r--    1 500      500       2200830 Jun 04  2010 busybox-1.14.4.tar.bz2
-rw-r--r--    1 500      500       1987727 Jun 04  2010 busybox-1.15.3.tar.bz2
-rw-r--r--    1 500      500       2007183 Jun 04  2010 busybox-1.16.0.tar.bz2
-rw-r--r--    1 500      500       2008548 Mar 28  2010 busybox-1.16.1.tar.bz2
-rw-r--r--    1 500      500       2167188 Dec 05  2011 busybox-1.19.3.tar.bz2
-rw-r--r--    1 500      500       2186738 Oct 12 06:01 busybox-1.20.2.tar.bz2  准备下载的包
```

下载完成后，解压 busybox，要先进入 busybox 然后编译，它的编译方式跟内核一样，要先执行 make menuconfig【提示：在执行 make menuconfig 的时候不要把屏幕缩的太小，会报错的，这里不演示了，有兴趣的可以尝试一下】，事先也说过它可以模仿很多命名，所以要进行 个别的选择，一个一个选太麻烦所以它给一个目录。

```
[root@localhost ~]# ls
anaconda-ks.cfg  busybox-1.20.2.tar.bz2  Desktop  install.log  install.log.syslog
[root@localhost ~]# tar xf busybox-1.20.2.tar.bz2
[root@localhost ~]# cd busybox-1.20.2                和编辑内核的步骤一样
[root@localhost busybox-1.20.2]# make menuconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/basic/split-include
  HOSTCC  scripts/basic/docproc
```

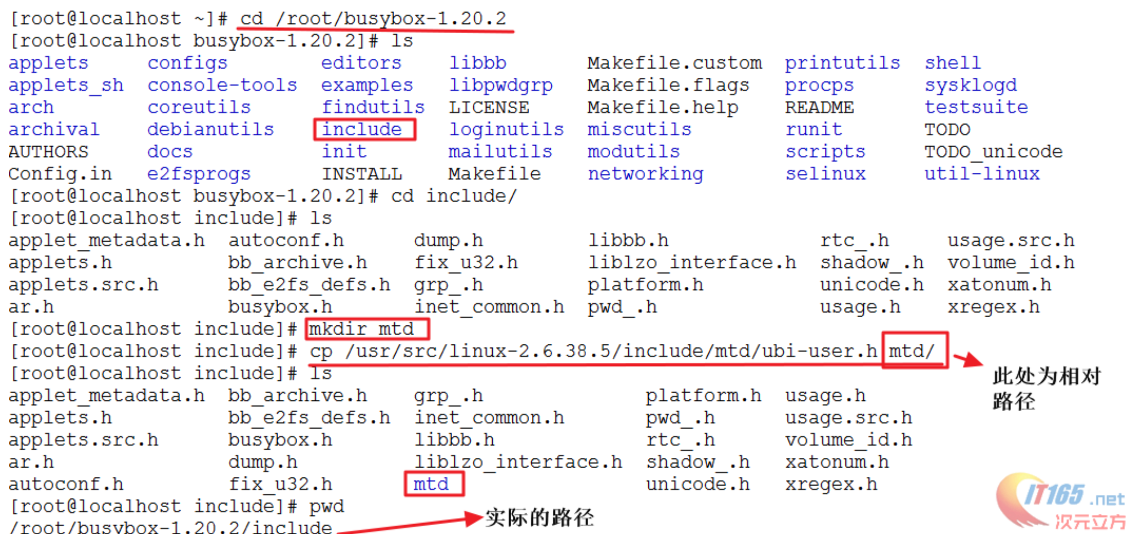
3、编辑 busybox

4、安装 busybox

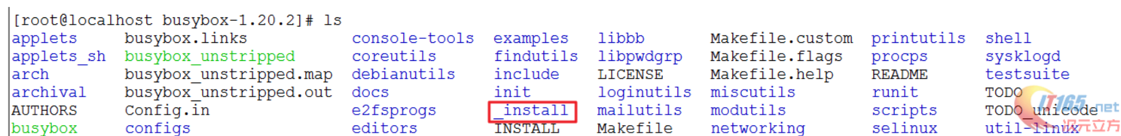
这个时候直接去编译安装 busybox，直接执行 make install 这时候可能会报错的，因为最新版本的 busybox 要依赖更新版的内核头文件，我们使用的是 2.6.18 的内核，而它当中不具备新版的 busybox 所具备的某些功能，所以编译过程中很可能会出错，我们要使用 2.6.28 以上的版本，我们在这里选用 2.6.38 的版本，主要的依赖就是/usr/src/linux-2.6.38.5/include/mtd/ubi-user.h 这个文件，把它复制过去。

```
[root@localhost ~]# ls
anaconda-ks.cfg  busybox-1.20.2.tar.bz2  install.log         linux-2.6.38.5.tar.bz2
busybox-1.20.2   Desktop                 install.log.syslog
[root@localhost ~]# tar xf linux-2.6.38.5.tar.bz2 -C /usr/src
[root@localhost ~]# cd /usr/src
[root@localhost src]# ls
debug  kernels  linux-2.6.38.5  redhat
[root@localhost src]# cd linux-2.6.38.5/
[root@localhost linux-2.6.38.5]# ls
arch     crypto         fs       Kbuild   MAINTAINERS  README          security  virt
block    Documentation  include  Kconfig  Makefile     REPORTING-BUGS  sound
COPYING  drivers        init     kernel   mm           samples         tools
CREDITS  firmware       ipc      lib      net          scripts         usr
[root@localhost linux-2.6.38.5]# cd include
[root@localhost include]# ls
acpi         crypto  Kbuild  linux      media  net     rdma   scsi   target  video
asm-generic  drm     keys    math-emu   mtd    pcmcia  rxrpc  sound  trace   xen
[root@localhost include]# cd mtd                        这就是我们所需要的文件，
[root@localhost mtd]# ls                                有了这个文件后执行make
inftl-user.h  Kbuild  mtd-abi.h  mtd-user.h  nftl-user.h  ubi-user.h  install 就不会报错了
```

这时候我们进入到/root/busybox-1.20.2 下,在 busybox-1.20.2 下也有一个 include,我们在里面创建一个 mtd 目录，然后把把 inubi-user.h 复制到里面。



然后我们就开始执行 make instll 了。（因为程序很小，这个速度是很快的）【提示：这里一定要注意在哪个目录下面，我们要在 busybox-1.20.2 下面执行 make install】这时候在当前目录下创建了一个_install，所有创建的文件都在这里面，我们就以_install 为蓝本去创建一些我们所需要的文 件。



5、用_instll 先创建 initrd

我们用_instll 先创建 initrd，但是我们在创建 initrd 的时候我们先给它一个临时目录 /tmp/busybox。

```
[root@localhost busybox-1.20.2]# cp _install /tmp/busybox -a
[root@localhost busybox-1.20.2]# cd /tmp/busybox
[root@localhost busybox]# ls
bin  linuxrc  sbin  usr
[root@localhost busybox]# rm -rf linuxrc
[root@localhost busybox]# mkdir proc sys etc dev lib/modules tmp -pv
```

这里我们打算使用红帽提供的内核，但这个内核不支持 ext3【提示：内核不支持 ext3，就意味着它没法访问真正的根文件系统】,这里的 initrd 主要存在目的就是为内核提供所需要依赖访问根文件系统所需要的模块了。我们要借助 initrd 来装载 ext3 这个模块了。我们在 /busybox/lib/modules 装载两个模块。

```
[root@localhost busybox]# modinfo ext3
filename:       /lib/modules/2.6.18-308.el5/kernel/fs/ext3/ext3.ko
license:        GPL
description:    Second Extended Filesystem with journaling extensions
author:         Remy Card, Stephen Tweedie, Andrew Morton, Andreas Dilger, Theodore Ts'o and o
thers
srcversion:     26DC008FC415305C5F65313
depends:        jbd
vermagic:       2.6.18-308.el5 SMP mod_unload 686 REGPARM 4KSTACKS gcc-4.1
module_sig:     883f3504f2326dddc995cc0b59af121112b66609f56f4d643dcbbde3384fc7aee2342f5b331f63
1b09d15ace08361a0e2dfa6ec1ec46718b122bde923
[root@localhost busybox]# modinfo jbd
filename:       /lib/modules/2.6.18-308.el5/kernel/fs/jbd/jbd.ko
license:        GPL
```

```
[root@localhost busybox]# cp  /lib/modules/2.6.18-308.el5/kernel/fs/jbd/jbd.ko lib/module
[root@localhost busybox]# cp  /lib/modules/2.6.18-308.el5/kernel/fs/ext3/ext3.ko lib/modu
```

下来我们就要提供 init 脚本，因为 busybox 提供的没有 bash，只有 sh。进行编辑 vim init。编辑完成退出后还需要加入权限（chmod +x init）

```
.#!/bin/sh        ####【注意：这里一定要写/bin/sh】
.#
.mount -t proc proc /proc       ####挂载 proc 文件系统
.mount -t sysfs sysfs /sys      ####挂载 sys 文件系统
.insmod /lib/modules/jbd.ko     ####加载模块
.insmod /lib/modules/ext3.ko
.mdev –s          ####探测额外硬件【下面有详细解释】
.mount -t ext3 /dev/hda2 /mnt/sysroot    ####挂载根文件系统
.exec switch_root /mnt/sysroot    /sbin/init      ####切换虚根到真正的文件系统
```

有时候我们可能需要系统来探测额外的硬件，并触发这些设备的，红帽给我们提供的是 udev，而 busybox 给我们提供的是 mdev，而这个 mdev 在/tmp/busybox/mnt/sysroot/sbin 下。

```
[root@localhost busybox]# ls sbin
acpid       fbsplash     hdparm      klogd      mkdosfs     nameif       setconsole         syslogd
adjtimex    fdisk        hwclock     loadkmap   mke2fs      pivot_root   slattach           tunctl
arp         findfs       ifconfig    logread    mkfs.ext2   poweroff     start-stop-daemon  udhcpc
blkid       freeramdisk  ifdown      losetup    mkfs.minix  raidautorun  sulogin            vconfig
blockdev    fsck         ifenslave   lsmod      mkfs.vfat   reboot       swapoff            watchdog
bootchartd  fsck.minix   ifup        makedevs   mkswap      rmmod        swapon             zcip
depmod      getty        init        man        modinfo     route        switch_root
devmem      halt         insmod      mdev       modprobe    runlevel     sysctl
[root@localhost busybox]# sbin/mdev
BusyBox v1.20.2 (2013-03-30 14:37:25 CST) multi-call binary.

Usage: mdev [-s]

mdev -s is to be run during boot to scan /sys and populate /dev.
```

mdev -s
就能够扫描**sys**目录，并和**dev**目录去获取所有的硬件设备

在/busybox/mnt/创建 sysroot。

```
[root@localhost busybox]# mkdir mnt/sysroot -pv
mkdir: created directory `mnt'
mkdir: created directory `mnt/sysroot'
[root@localhost busybox]# ls
bin  dev  etc  init  lib  mnt  proc  sbin  sys  tmp  usr
```

我们还需要手动创建设备文件，分别是 dev/console 和 dev/null【建议：使用手动创建】

```
[root@localhost busybox]# mknod dev/console c 5 1
[root@localhost busybox]# mknod dev/null c 1 3
[root@localhost busybox]# tree dev
dev
|-- console
`-- null

0 directories, 2 files
```

这时候我们就可以将此目录做成 initrd 了。大小是不是很小呀！

```
[root@localhost busybox]# find . | cpio -H newc --quiet -o | gzip -9 > /mnt/boot/initrd.gz
[root@localhost busybox]# ls -lh /mnt/boot/initrd.gz
-rw-r--r-- 1 root root 951K Mar 31 11:12 /mnt/boot/initrd.gz   是不是很小
```

6、准备内核

这个时候我们就要复制内核了，并且重命名为 vmlinuz。

cp /boot/vmlinuz-2.6.18-308.el5 /mnt/boot/vmlinuz

7、安装 grub【提示：安装完成以后我们最好还是查看一下/mnt/boot,看看是否和我们期望的一样】

```
[root@localhost busybox]# grub-install --root-directory=/mnt /dev/hda
Probing devices to guess BIOS drives. This may take a long time.
Installation finished. No error reported.
This is the contents of the device map /mnt/boot/grub/device.map.
Check if this is correct or not. If any of the lines is incorrect,
fix it and re-run the script `grub-install'.

(fd0)   /dev/fd0
(hd0)   /dev/hda
(hd1)   /dev/sda
```

8、提供配置文件

[root@localhost busybox]# vim /mnt/boot/grub/grub.conf

```
default=0

timeout=1

title shuaige Linux (2.6.18)

root(hd0,0)

kernel /vmlinuz ro root=/dev/hda2

initrd /initrd.gz
```

9、开始准备 sysroot 了，【提示：这个目录就用不着了，我们要回到原目录下】，我们把/ busybox-1.20.2 下面的_install 复制粘贴在/mnt/sysroot 里面。

```
[root@localhost busybox-1.20.2]# cd
```

[root@localhost ~]# cd busybox-1.20.2

[root@localhost busybox-1.20.2]# cp _install/* /mnt/sysroot –a

10、创建目录

我们在/mnt/sysroot 下创建一些目录。

命令：mkdir proc sys dev tmp var/{log,lock.run} lib/moduies etc/rc.d/init.d root boot mnt media –pv

```
[root@localhost sysroot]# ls
bin  linuxrc  lost+found  sbin  usr
[root@localhost sysroot]# rm -rf linuxrc
[root@localhost sysroot]# mkdir proc sys dev tmp var/{log,lock.run} lib/moduies etc/rc.d/init.d root boo
mnt media -pv
```

11、创建配置文件

配置/mnt/sysroot /etc/inittab【提示：busybox 所支持的 inittab 和我们的 inittab 是不太一样的】里面不需要设定级别

::sysinit:/etc/rc.d/rc.sysinit 初始化脚本

console::respawn:-/bin/sh

::ctrlaltdel:sbin/reboot 按下 Ctrl +Alt+Del 执行 reboot www.it165.net

::shutdown:/bin/umount -a –r 执行 shutdown 命令的时候，要卸载文件系统

12、为系统准备一个“文件系统表”配置文件

内容需要挂载到/mnt/sysroot/etc/fstab，内容如下：

```
sysfs       /sys      sysfs    defaults   0 0
proc        /proc     proc     defaults   0 0
/dev/hda1   /boot     ext3     defaults   0 0
/dev/hda2   /         ext3     defaults   1 1
```

13、创建设备文件

这里的真正的根文件系统也是需要创建文件设备的。

```
[root@localhost sysroot]# mknod dev/console c 5 1
[root@localhost sysroot]# mknod dev/null c 1 3
```

14、建立系统初始化脚本

[root@localhost sysroot] # vim etc/rc.d/rc.sysinit 内容如下：

```
#!/bin/sh

echo -e "\tWelcome to \033[31mShuaiGe\033[0m   Linux"

echo -e "Remounting the root filesystem ..........[   \033[32mOK \033[0m   ]"

mount -t proc proc /proc

mount -t sysfs sysfs /sys

mount -o remount,rw /

echo -e "Creating the files of devic ...................[   \033[32mOK\033[0m   ]"

mdev -s

echo -e "Mounting the filesystem......................[   \033[32mOK\033[0m   ]"

mount -a

swapon -a
```

15、然后给它执行权限。到这里我们的微虚拟机的制作过程已告一段落，只要注意到我

上面所提的注意事项，完成这个是不成问题的。

```
[root@localhost sysroot]#chmod +x etc/rc.d/rc.sysinit
```

附录 3 PMON 常用命令

龙芯 CPU LS1C300A 的 BIOS 是 pmon，负责引导系统内核。龙芯指令集是 MIPS32。MIPS 的名字为“Microcomputer without interlocked pipeline stages”的缩写。另外一个通常的非正式的说法是“Millions of instructions per second”。编译工具为 mips- elf-gcc。

系统上电启动按空格键后即可进入 PMON 设置界面。在 PMON 的命令行上可以输入命令设置启动参数，参数被烧到 Flash 里面，重新启动后生效。

PMON 中内置了很多命令，下面列举主要的：

类型	命令	说明	例子	例子含义
帮助	h	查看帮助信息	h	列出所有可以使用命令
			h ping	查看 ping 命令的用法
调试	d1	读某个地址的值(读一个 byte)	d1 0x80300000	查看地址 0x80300000 处的值
	d2	读某个地址的值(读一个 halt word)	d1 0x80300000	查看地址 0x80300000 处的值
	d4	读某个地址的值(读一个 word)	d4 0x80300000	查看地址 0x80300000 处的值
	m1	在某个地址处写入一个值(写入一个 byte 大小)	m1 0x80300000 0x12	在地址 0x80300000 处写入 0x12
	m2	在某个地址处写入一个值(写入一个 halt word 大小的值)	m2 0x80300000 0x1234	在地址 0x80300000 处写入 0x1234
	m4	在某个地址处写入一个值(写入一个 word 大小的值)	m4 0x80300000 0x12345678	在地址 0x80300000 处写入 0x12345678
				读出此地址值 PMON>d4 0x80300000；可以看到读出来的值为 0x12345678
内存	mt	内存测试命令	mt	测试板的内存是否正常
	load	下载 linux 内核到内存	load tftp://192.168.3.18/vmlinux	通过网络从 IP 为 192.168.3.18 的主机上下载内核 vmlinx 到内存
测试外设	ac97_read	测试 ac97（录音）	ac97_read	录音，有 5s 钟的时间录音；与 ac97_test 配合着测试 ac97 设备是否正常
	ac97_test	测试 ac97,放出刚才录进去的声音	ac97_test	播放刚才录进去的声音；和 ac97_read 配合着用

网络	ifaddr	设置板的 ip 地址(只当次有效，断电后会丢失)	ifaddr syn0 192.168.3.25	设置板的 ip 地址为 192.168.3.25
	ping	测试网络	ping 192.168.3.1	测试与 192.168.3.1 网口是否连通
环境管理	set	设置环境变量；设置的参数会保存到norflash 高位地址，在 pmon 一开始运行时就会自动去调用	set	列出所有已经设置好的环境变量
			set ifconfig syn0:192.168.3.88	设置开发板的 IP 地址，重启开发板后 IP 地址固定存在
			set al /dev/mtd0	自动从 nandflash 的 mtd0 分区 load 内存，设置板在一上电时自动执行 load 内核到内存操作
			set append 'console=ttyS2'	设置板的运行的启动参数
	env	查看板上已经设好的环境变量	env	列出所以环境变量
FLASH	devcp	PMON 上的拷贝下载，通常拷贝从网络下载的文件到 Nor FlashNand Flash 中	devcp tftp://192.168.3.18/vmlinux /dev/mtd0	从网络下载 vmlinux 到内存中并拷贝到 nandflash 中
	mtd_erase	擦除 Nand Flash 某分区的数据	mtd_erase /dev/mtd1	擦除 Nand Flash 分区 1 的数据
系统管理	reboot	重启 PMON	reboot	重启 PMON
其他	devls	查看设备列表	devls -n	查看网络设备

设置显示分辨率：相应内核启动参数加上 video= ls1bfb : 480x272-16@60 （以 480x272 分辨率为例）；如果使用 vga 接口的显示器，则启动参数为 video= ls1bfb :vga80 0x600-16@60 （以 800x600 分辨率为例），由于开发板没有 vga 接口，所以不使用该参数。

配置网卡:

```
ifconfig syn0 10.0.0.2
```

可以用 ping 命令测试网卡

```
ping 10.0.0.1
```

命令行设置从网卡启动:

```
ifconfig syn0 10.0.0.2
load tftp://10.0.0.1/vmlinux
g console=ttyS2,1152 00 rdinit=/sbin/init initcall_debug=1
```

命令行烧 nandflash （用于更新 PMON ）:

```
ifconfig syn0 10.0.0.2
devcp tftp://10.0.0.1/gzom.bin /dev/mtd0
```

命令行从 nand 启动:

```
load /dev/mtd0
```

```
g console=ttyS2,115200 rdinit=/sbin/init initcall_debug=1
```

设置自动启动:

环境变量 ifconfig 用来每次启动的时候自动设置网卡地址

```
set ifconfig syn0:10.0.0.2:255.255.255.0
```

设置从不同介质启动内核（假设内核名称为 vmlinux）：

```
set al /dev/fs/yaffs2@mtd1/boot/vmlinux    从 yaffs2 分区里面的 boot 目录中的 vmlinux 来
```

引导

```
set al /dev/mtd0    从 nandflash 的第一个分区引导
set al /dev/fs/ext2@usb0/boot/vmlinux    如果从 usb 光盘引导
set al tftp://10.0.0.3/vml inux    从 tftp 服务器引导
Set al http://10.0.0.3/vmlinux    从 http 引导
Set al nfs://10.0.0.3/vmlinux    从 nfs 引导
set al /dev/ram@0xbe000000,0x1000000    从地址 0xbe000000 引导
```

设置内核启动参数：

```
set append 'root=/dev/mtdblock2
console=tty'
```

从 nand 的第二个分区作为根文件系统

```
set append ‘ root=/dev/nfs
nfsroot=192.168.1.1:/mnt/hdb1/nfs
ip=192.168.1.89:::::eth0 console=tty ’
```

nfs 服务器 192.168.1.1 的/mnt/hdb1/nfs 作为根

文件系统，网卡 eth0,ip 192.168.1.89

```
set append ‘rdinit=/sbin/init console=tty ’    内核里面自带的 ramdisk 作为系统
```

设置 PMON 系统时间：

```
set TZ +8    设置时区+8 区
set date 200805011200.01    设置日期 2008- 5-01 12:00:01
```

设置网卡：

```
Set ethaddr 00:01:02:03:04:05    设置网卡 MAC 地址是 00:01:02:03:04:05
set ifconfig syn0:10.0.0.89    设置 pmon 启动后网卡（syn0 代表 1b gmac 网
```

络控制器,注：可以用 devls 列出 PMON 设备）ip 为 10.0.0.89

以上是一些通用的设置，具体到 1C 参考板的缺省参数设置是：

```
set al /de v/mtd0    从 nand 加载内核
set append ‘root=/dev/mtdblock2
console=tty ’
```

文件系统位于 nand 第二个分区

```
set bootdelay 3    启动延迟 3 秒
```

附录 4 module_init 和 module_exit 分析

http://blog.sina.com.cn/s/blog_8e9c63c701019zsj.html

module_init()

linux 内核中通过 module_init 函数指定初始化函数，分两种：

1、 静态编译进内核

对于静态编译进内核的，什么时候执行呢？arch/arm/kernel/head.S

start_kernel()->rest_init()->kernel_init()->do_basic_setup()->do_initcalls()

在执行 do_initcalls 之前调用了：

driver_init(); //建立设备模型子系统

在没有定义#ifndef MODULE 的情况下，基本上表明模块是要编译进内核的(obj-y)

```
#define module_init(x) __initcall(x);
#define __initcall(fn) device_initcall(fn)
#define device_initcall(fn) __define_initcall("6",fn,6)
#define __define_initcall(level,fn,id) \
static initcall_t __initcall_##fn##id __used \
__attribute__((__section__(".initcall" level ".init"))) = fn
```

所以，module_init(x)最终展开为：

```
static initcall_t __initcall_##fn##id __used \
__attribute__((__section__(".initcall" level ".init"))) = fn
```

更直白点，假设 driver 所对应的模块的初始化函数为 int gpio_init(void)，那么 module_init(gpio_init)实际上等于:

```
static initcall_t __initcall_gpio_init_6 __used __attribute__((__section__(".initcall6.init"))) = gpio_init;
```

就是声明一类型为 initcall_t（typedef int (*initcall_t)(void)）函数指针类型的变量 __initcall_gpio_init_6 并将 gpio_init 赋值与它。

这里的函数指针变量声明比较特殊的地方在于，将这个变量放在了一名为".initcall6.init"节中。接下来结合 vmlinux.lds 中的

```
.initcall.init : AT(ADDR(.initcall.init) - (0xc0000000 -0x00000000)) {
__initcall_start = .;
*(.initcallearly.init) __early_initcall_end = .; *(.initcall0.init) *(.initcall0s.init) *(.initcall1.init) *(.initcall1s.init)
*(.initcall2.init) *(.initcall2s.init) *(.initcall3.init) *(.initcall3s.init) *(.initcall4.init) *(.initcall4s.init)
*(.initcall5.init) *(.initcall5s.init) *(.initcallrootfs.init) *(.initcall6.init) *(.initcall6s.init) *(.initcall7.init)
*(.initcall7s.init)
__initcall_end = .;
}
```

以及 do_initcalls：

```
static void __init do_initcalls(void)
{
initcall_t *call;
for (call = __initcall_start; call < __initcall_end; call++)
do_one_initcall(*call);

flush_scheduled_work();
}
```

那么就不难理解模块中的 module_init 中的初始化函数何时被调用了：在系统启动过程中 start_kernel()->rest_init()->kernel_init()->do_basic_setup()->do_initcalls()。

在静态加载驱动的时候，我们那个 gpio_init_cleanup 和 module_exit 函数永远不会被执行，所以去掉是完全可以的，不过为了程序看起来结构清晰，也为了与动态加载的程序兼容，还是建议保留着。

静态编译进内核，在系统启动的时候，会按顺序初始化相应的函数，当系统动态检测到设备存在时，会自动执行相应的函数，生成设备名。

2、按照模块动态加载

运行 make menuconfig，在内核配置中进入 Loadable module support，选择 Enable loadable module support 和 Kernel module loader(NEW)两个选项。在应用程序配置中进入 busybox,选择 insmod, rmmod, lsmod 三个选项。

执行 insmod 时，会执行 module_init()对应的函数。

然后 mknod 设备节点。

附录 5.创建一个与你的驱动程序对应的设备节点

设备节点：被创建在/dev 下，是连接内核与用户层的枢纽，就是设备是接到对应哪种接口的哪个 ID 上。作用及使用：类 unix 系统对设备的访问都是基于文件形式的。在类 unix 系统中，你要访问一个硬件设备。一般和访问一个普通文件差不多。因此，/dev 下的设备节点就被作为这样的一类特殊文件来存在。在驱动程序中同样需要实现各种文件的操作调用，如 open,release,read,write,ioctl 等。应用程序通过 open("/dev/xxx",O_RDWR)这样的代码来打开设备。驱动程序通过这样的节点向应用程序提供各种服务：如 read,write,ioctl 等，即应用程序通过 read write 等函数读写设备文件，内核会调用驱动程序里面的相应函数，在那些函数里面会实现读写设备的功能。

问：如果有个设备永远只被一个应用程序使用，那我直接调用驱动里面的函数不要节点是不是也可以了？

答：不可以。

1、驱动程序里面的函数在一般情况下，应用程序是调用不到的。它被隐藏在 VFS（虚拟文件系统）的后面。2、由于存在了 VFS。所以，linux 下的文件（包括各种设备）都是可以被多个应用程序打开的，从而也可以被多个应用程序使用。3、对于存在临界资源的设备，一般在驱动程序中需要对临界资源进行保护。从而使得多个应用程序或进程能安全的操作设备。4、正是引入了 VFS 这个架构，使得在 linux 下对设备的访问方法基本相同。例如：向屏幕画图可以用 write()系统调用。而向串口写入数据也可以用 write()系统调用。

inux 下生成驱动设备节点文件的方法

http://zhidao.baidu.com/link?url=fXaOk5xopUA47H5AFm8dczMBOk9Qb6_-FK39RTKiFyO4BO_Ffh1_F_wNGp61_S_ja--Z_dBDAFK5c-XI0N2_XzIhFU3ufttI081A5N1BO-W

Linux 下生成驱动设备节点文件的方法有 3 个：1、手动 mknod；2、利用 devfs；3、利用 udev。

在刚开始写 Linux 设备驱动程序的时候，很多时候都是利用 mknod 命令手动创建设备节点，实际上 Linux 内核为我们提供了一组函数，可以用来在模块加载的时候自动在/dev 目录下创建相应设备节点，并在卸载模块时删除该节点。

在 2.6.17 以前，在/dev 目录下生成设备文件很容易，

```
devfs_mk_bdev
devfs_mk_cdev
devfs_mk_symlink
devfs_mk_dir
devfs_remove
```

这几个是纯 devfs 的 api，2.6.17 以前可用，但后来 devfs 被 sysfs+udev 的形式取代，同时期 sysfs 文件系统可以用的 api：class_device_create_file，在 2.6.26 以后也不行了，现在，使用的是 device_create ，从 2.6.18 开始可用

```
struct device *device_create(struct class *class, struct device *parent,
 dev_t devt, const char *fmt, ...)
```

从 2.6.26 起又多了一个参数 drvdata： the data to be added to the device for callbacks 不会用可以给此参数赋 NULL

```
struct device *device_create(struct class *class, struct device *parent,
dev_t devt, void *drvdata, const char *fmt, ...)
```

下面着重讲解第三种方法 udev

在驱动用加入对 udev 的支持主要做的就是：在驱动初始化的代码里调用 class_create(...)为该设备创建一个 class，再为每个设备调用 device_create(...)(在 2.6 较早的内核中用 class_device_create)创建对应的设备。

内核中定义的 struct class 结构体，顾名思义，一个 struct class 结构体类型变量对应一个类，内核同时提供了 class_create(...)函数，可以用它来创建一个类，这个类存放于 sysfs 下面，一旦创建好了这个类，再调用 device_create(...)函数来在/dev 目录下创建相应的设备节点。这样，加载模块的时候，用户空间中的 udev 会自动响应 device_create(...)函数，去/sysfs 下寻找对应的类从而创建设备节点。这个过程可参考文中《6.6 led 子系统剖析》。

struct class 和 class_create(...) 以及 device_create(...)都包含在在/include/linux/device.h 中，使用的时候一定要包含这个头文件，否则编译器会报错。

struct class 定义在头文件 include/linux/device.h 中

class_create(...)在/drivers/base/class.c 中实现

device_create(...)函数在/drivers/base/core.c 中实现

class_destroy(...),device_destroy(...)也在/drivers/base/core.c 中实现调用过程类似如下：

```
static struct class *spidev_class;

/*-------------------------------------------------------------------------*/

static int __devinit spidev_probe(struct spi_device *spi)
{
....

        dev =device_create(spidev_class, &spi->dev, spidev->devt,
        spidev, "spidev%d.%d",
        spi->master->bus_num, spi->chip_select);
...
}

static int __devexit spidev_remove(struct spi_device *spi)
{
......
        device_destroy(spidev_class, spidev->devt);
.....

return 0;
}

static struct spi_driver spidev_spi = {
        .driver =
        {
                .name = "spidev",
                .owner = THIS_MODULE,
        },
        .probe = spidev_probe,
        .remove = __devexit_p(spidev_remove),

};

/*-------------------------------------------------------------------------*/
static int __init spidev_init(void)
{
....

        spidev_class =class_create(THIS_MODULE, "spidev");
        if (IS_ERR(spidev_class)) {
        unregister_chrdev(SPIDEV_MAJOR, spidev_spi.driver.name);
        return PTR_ERR(spidev_class);
        }
....
}
```

```
module_init(spidev_init);

static void __exit spidev_exit(void)
{
......
	class_destroy(spidev_class);
......
}
module_exit(spidev_exit);

MODULE_DESCRIPTION("User mode SPI device interface");
MODULE_LICENSE("GPL");
```

下面以一个简单字符设备驱动来展示如何使用这几个函数，在《6.3 最简单的 Linux 驱动》上添加修改而成 hello_drivernod.c。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>

int HELLO_MAJOR = 0;
int HELLO_MINOR = 0;
int NUMBER_OF_DEVICES = 2;

struct class *my_class;
struct cdev cdev;
dev_t devno;
struct hello_dev
{
	struct device *dev;
	dev_t chrdev;
	struct cdev cdev;
};
static struct hello_dev *my_hello_dev = NULL;

struct file_operations hello_fops =
{
	.owner = THIS_MODULE
};

static int __init hello_init (void)
{
	int err = 0;
	struct device *dev;

	my_hello_dev = kzalloc(sizeof(struct hello_dev), GFP_KERNEL);
	if (NULL == my_hello_dev)
	{
		printk("%s kzalloc failed!\n",__func__);
		return -ENOMEM;
	}
	devno = MKDEV(HELLO_MAJOR, HELLO_MINOR);
	if (HELLO_MAJOR)
		err= register_chrdev_region(my_hello_dev->chrdev, 2, "memdev");
	else
	{
		err = alloc_chrdev_region(&my_hello_dev->chrdev, 0, 2, "memdev");
		HELLO_MAJOR = MAJOR(devno);
	}
	if (err)
	{
		printk("%s alloc_chrdev_region failed!\n",__func__);
		goto alloc_chrdev_err;
```

```
	}
	printk("MAJOR IS %d\n",HELLO_MAJOR);

	cdev_init(&(my_hello_dev->cdev), &hello_fops);
	my_hello_dev->cdev.owner = THIS_MODULE;

	err = cdev_add(&(my_hello_dev->cdev), my_hello_dev->chrdev, 1);
	if (err)
	{
		printk("%s cdev_add failed!\n",__func__);
		goto cdev_add_err;
	}
	printk (KERN_INFO "Character driver Registered\n");

	my_class =class_create(THIS_MODULE,"hello_char_class"); //类名为 hello_char_class
	if(IS_ERR(my_class))
	{
		err = PTR_ERR(my_class);
		printk("%s class_create failed!\n",__func__);
		goto class_err;
	}

	dev = device_create(my_class,NULL,my_hello_dev->chrdev,NULL,"memdev%d",0); //设备名为 memdev
	if (IS_ERR(dev))
	{
		err = PTR_ERR(dev);
		printk ("%s device_create failed!\n",__func__);
		goto device_err;
	}
	printk("hello module initialization\n");
	return 0;

	device_err:
	device_destroy(my_class, my_hello_dev->chrdev);
	class_err:
	cdev_del(my_hello_dev->chrdev);
	cdev_add_err:
	unregister_chrdev_region(my_hello_dev->chrdev, 1);
	alloc_chrdev_err:
	kfree(my_hello_dev);
	return err;
}

static void __exit hello_exit (void)
{
	cdev_del (&(my_hello_dev->cdev));
	unregister_chrdev_region (my_hello_dev->chrdev,1);
	device_destroy(my_class, devno); //delete device node under /dev  必须先删除设备，再删除 class 类
	class_destroy(my_class); //delete class created by us
	printk (KERN_INFO "char driver cleaned up\n");
}

module_init (hello_init);
module_exit (hello_exit);
MODULE_LICENSE ("GPL");\
MODULE_AUTHOR("sundm75");
MODULE_DESCRIPTION("Board First module device driver test");
```

编写 Makefile 文件：

```
#定义生成的目标
obj-m := hello_drivernod.o
#定义目录变量
KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson
PWD := $(shell pwd)
all:
```

```
# make 文件
	make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS_COMPILE=mipsel-linux-
clean:
	rm -rf *.o   *.mod.c *.ko
```

```
[root@Loongson:/]#insmod hello_drivernod.ko
MAJOR IS 0
Character driver Registered
hello module initialization
[root@Loongson:/]#cd dev
```

class 下添加了 hello_char_class 的类

```
COM16 - PuTTY
mem                 network_throughput  tty15               vcs
memdev0             null                tty2                vcs1
mtd0                ptmx                tty3                vcsa
mtd0ro              pts                 tty4                vcsa1
mtd1                random              tty5                watchdog
mtd1ro              root                tty6                zero
mtd2                rtc0                tty7
[root@Loongson:/dev]#cd ..
[root@Loongson:/]#ls
bin                 lib                 proc                usr
dev                 linuxrc             root                var
etc                 lost+found          sbin
hello_drivernod.ko  mnt                 sys
home                opt                 tmp
[root@Loongson:/]#cd class
-/bin/sh: cd: can't cd to class
[root@Loongson:/]#cd sys
[root@Loongson:/sys]#ls
block      class      devices    fs         module
bus        dev        firmware   kernel     power
[root@Loongson:/sys]#cd class
[root@Loongson:/sys/class]#ls
bdi                 i2c-dev             mtd                 spidev
block               input               net                 tty
bsg                 leds                rtc                 ubi
firmware            mdio_bus            scsi_device         vc
gpio                mem                 scsi_disk           vtconsole
hello_char_class    misc                scsi_host
i2c-adapter         mmc_host            spi_master
[root@Loongson:/sys/class]#
```

这样，模块加载后，就能在/dev 目录下找到 memdev0 这个设备节点了。如果想在开机时自动加载，可采用 6.5 方法进行内核配置驱动 。

```
COM16 - PuTTY
i2c-2              mtdblock2          tty12              ttyS3
input              mtdblock3          tty13              ubi_ctrl
kmsg               network_latency    tty14              urandom
mem                network_throughput tty15              vcs
memdev0            null               tty2               vcs1
mtd0               ptmx               tty3               vcsa
mtd0ro             pts                tty4               vcsa1
mtd1               random             tty5               watchdog
mtd1ro             root               tty6               zero
mtd2               rtc0               tty7
[root@Loongson:/dev]#mdev -s
[root@Loongson:/dev]#ls
console            mtd2ro             tty                tty8
cpu_dma_latency    mtd3               tty0               tty9
full               mtd3ro             tty1               ttyS0
i2c-0              mtdblock0          tty10              ttyS1
i2c-1              mtdblock1          tty11              ttyS2
i2c-2              mtdblock2          tty12              ttyS3
input              mtdblock3          tty13              ubi_ctrl
kmsg               network_latency    tty14              urandom
mem                network_throughput tty15              vcs
memdev0            null               tty2               vcs1
mtd0               ptmx               tty3               vcsa
mtd0ro             pts                tty4               vcsa1
mtd1               random             tty5               watchdog
mtd1ro             root               tty6               zero
mtd2               rtc0               tty7
[root@Loongson:/dev]#cd ..
[root@Loongson:/]#ls
bin                lib                proc               usr
```

另一个例子:内核中的 drivers/i2c/i2c-dev.c

在 i2cdev_attach_adapter 中调用

```
device_create(i2c_dev_class, &adap->dev,MKDEV(I2C_MAJOR, adap->nr), NULL,"i2c-%d", adap->nr);
```

这样在 dev 目录就产生 i2c-0 或 i2c-1 节点

udev: udev 是硬件平台无关的，属于 user space 的进程，它脱离驱动层的关联而建立在操作系统之上，基于这种设计实现，我们可以随时修改及删除/dev 下的设备文件名称和指向，随心所欲地按照我们的愿望安排和管理设备文件系统，而完成如此灵活的功能只需要简单地修改 udev 的配置文件即可，无需重新启动操作系统。udev 已经使得我们对设备的管理如探囊取物般轻松自如。

接下来就是 udev 应用，udev 是应用层的东西，udev 需要内核 sysfs 和 tmpfs 的支持，sysfs 为 udev 提供设备入口和 uevent 通道，tmpfs 为 udev 设备文件提供存放空间。

首先，由于在 kernel 启动未完成以前我们的设备文件不可用，如果使用 mtd 设备作为 rootfs 的挂载点，这个时候/dev/mtdblock 是不存在的，无法让 kernel 找到 rootfs，kernel 只好停在那里惊慌。这个问题可以通过给 kernel 传递设备号的方式来解决，在 linux 系统中，mtdblock 的主设备号是 31，part 号 从 0 开始，那么以前的/dev/mtdblock/3 就等同于 31:03，以次类推，所以我们只需要修改 bootloader 传给 kernel 的 cmd line 参数，使 root=31:03，就可以让 kernel 在 udevd 未起来之前成功的找到 rootfs。

其次，需要做的工作就是重新生成 rootfs，把 udevd 和 udevstart 复制到/sbin 目录。然后 在/etc/下为 udev 建立设备规则，这可以说是 udev 最为复杂的一步。这篇文章提供了最完整的指导：Writing udev rules (http://reactivated.net/writing_udev_rules.html)。

udev 的源码可以在去相关网站下载，然后就是对其在运行环境下的移植，指定交叉编译环境，修改 Makefile 下的 CROSS_COMPILE，如为 mipsel-linux-，DESTDIR=xxx，或直接 make CROSS_COMPILE=mipsel-linux-,DESTDIR=xxx 并 install。

把主要生成的 udevd、udevstart 拷贝 rootfs 下的/sbin/目录内，udev 的配置文件 udev.conf 和 rules.d 下的 rules 文件拷贝到 rootfs 下的/etc/目录内

并在 rootfs/etc/init.d/rc.sysinit 中添加以下几行：

```
echo "Starting udevd..."
/sbin/udevd --daemon
/sbin/udevstart
```

（原 rcS 内容如下：

```
# mount filesystems
/bin/mount -t proc /proc /proc
/bin/mount -t sysfs sysfs /sys
/bin/mount -t tmpfs tmpfs /dev
# create necessary devices
/bin/mknod /dev/null c 1 3
/bin/mkdir /dev/pts
/bin/mount -t devpts devpts /dev/pts
/bin/mknod /dev/audio c 14 4
/bin/mknod /dev/ts c 10 16
)
```

这样当系统启动后，udevd 和 udevstart 就会解析配置文件，并自动在/dev 下创建设备节点文件.

mdev 的使用方法和原理：http://blog.csdn.net/hugerat/archive/2008/12/03/3437099.aspx

mdev 是 busybox 自带的一个简化版的 udev，适合于嵌入式的应用场合。其具有使用简单的特点。它的作用，就是在系统启动和热插拔或动态加载驱动程序时，自动产生驱动程序所需的节点文件。在以 busybox 为基础构建嵌入式 linux 的根文件系统时，使用它是最优的选择。

（1）在编译时加上对 mdev 的支持：

```
Linux System Utilities    --->
[*]mdev
[*]    Support /etc/mdev.conf
[*]        Support subdirs/symlinks
[*]            Support regular expressions substitutions
[*]        Support command execution at device addition/removal
[*]    Support loading of firmwares
```

（2）在启动时加上使用 mdev 的命令：

fstab 列出了 linux 开机时自动挂载的文件系统的列表，内容：

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

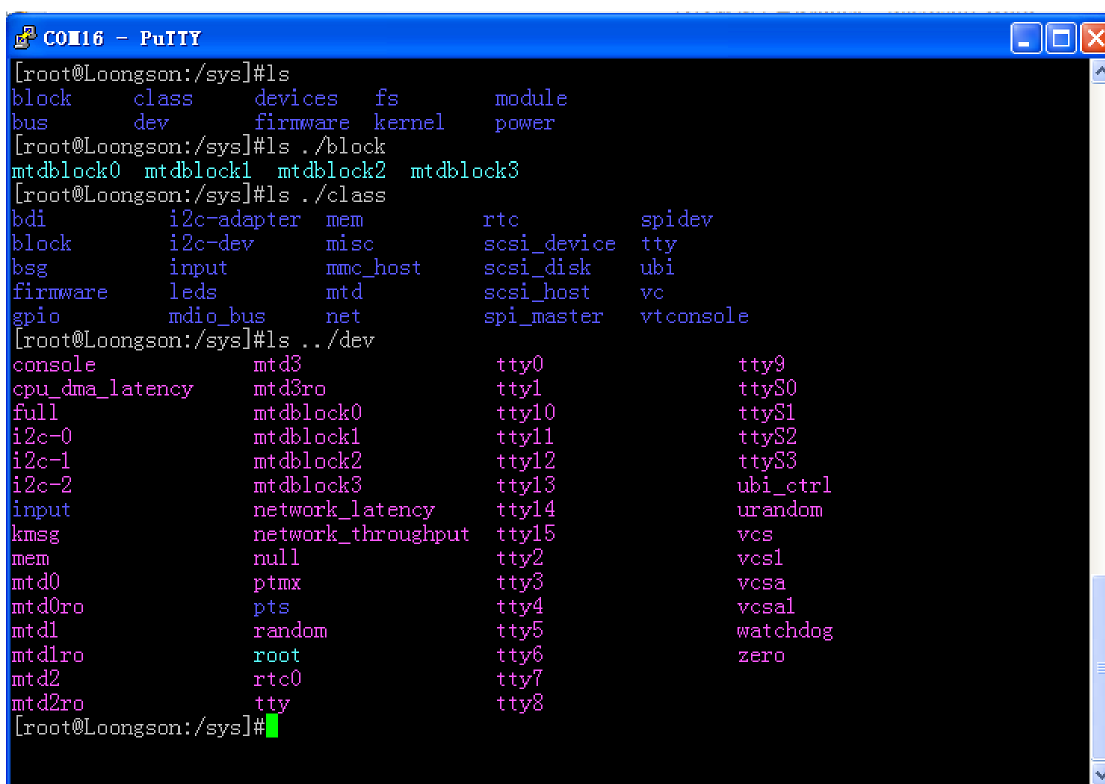
fstab 中存放了与分区有关的重要信息，其中每一行为一个分区记录，每一行又可分为六个部份

1. 第一项是 mount 的储存装置的实体位置，如 proc
2. 第二项就是您想要将其加入至哪个目录位置，如/proc,就是在安装时提示的挂入点。
3. 第三项就是所谓的 local filesystem，其包含了以下格式：如 ext、ext2、msdos、iso9660、nfs、swap 等，或如上例的 ext2，可以参见/prco/filesystems 说明。
4. 第四项就是 mount 时，所要设定的状态，如 ro（只读）或如上例的 defaults（包括了其它参数如 rw、suid、exec、auto、nouser、async），可以参见「mount nfs」。
5. 第五项是提供 DUMP 功能，在系统 DUMP 时是否需要 BACKUP 的标志位，其内定值是 0。
6. 第六项是设定此 filesystem 是否要在开机时做 check 的动作，除了 root 的 filesystem 其必要的 check 为 1 之外，其它皆可视需要设定，内定值是 0。

在创建的根文件系统（rootfs）中的 etc/ init.d/rc.sysinit 文件中添加了如下指令：

```
#mount all filesystem defined in "/etc/fstab"
echo "#mount all....."
/bin/mount -a

echo "#Starting mdev....."
echo /sbin/mdev > /proc/sys/kernel/hotplug #  设置热插拔事件处理程序为 mdev
/sbin/mdev -s #设备节点维护程序 mdev 初始化
```

执行 mdev –s 的流程：以‘-s’为参数调用位于/sbin 目录写的 mdev（其实是个链接，作用是传递参数给/bin 目录下的 busybox 程序并调用它），mdev 扫描 /sys/class 和/sys/block 中所有的类设备目录，如果在目录中含有“dev”的属性文件，且文件中包含的是设备号，则 mdev 就利用这些信息为这个设备在 /dev 下创建设备节点文件。一般只在启动时才执行一次“mdev -s”。



注意：

mdev 有两个主要的应用：初始化对象和动态更新。两个应用都需要内核 sysfs 的支持。为了实现动态更新，你必须在内核配置时增加热挺拔支持。

以下是系统初始化脚本中一个典型的使用 mdev 的代码片段：

[1] mount -t sysfs sysfs /sys

[2] echo /bin/mdev > /proc/sys/kernel/hotplug

[3] mdev -s

简单说明一下上面的代码：

[1]你必须在执行 mdev 前挂载 /sys 。

[2] 命令内核在增删设备时执行 /bin/mdev ，使设备节点文件会被创建和删除。

[3] 设置 mdev，让它在系统启动时创建所有的设备节点。

当然，一个对 mdev 更完整的安装还必须在以上代码片段前执行下面的命令：

[4] mount -t tmpfs mdev /dev

[5] mkdir /dev/pts

[6] mount -t devpts devpts /dev/pts

[4]确保 /dev 是 tmpfs 文件系统(假设文件系统在 flash 外运行)。

[5] 创建/dev/pts 挂载点

[6] 在 /dev/pts 挂载 devpts 文件系统

linux 中的热插拔和 mdev 机制:

http://blog.sina.com.cn/s/blog_af9acfc601018sax.html

附录 6 linux kernel 从入口到 start_kernel 的代码分析

http://www.baidu.com/link?url=oLv-1UIQfRpc_74lkUy3DFA32RT1sdzjh2wnIKJY2jZdCHn7Zt0VAaXTr829q4Ft7nGC32Dlo2uZl4J4Fx1O5wME0Y-uyiEyBkv3DVg3UCu&wd=&eqid=d184df580011c8df00000005567b4586

linux kernel 从入口到 start_kernel 的代码分析

本文的很多内容是参考了网上某位大侠的文章写的<<>>，有些东西是直接从他那 copy 过来的。

最近分析了一下 u-boot 的源码，并写了分文档， 为了能够衔接那篇文章，这次又把 arm linux 的启动代码大致分析了一下，特此写下了这篇文档。一来是大家可以看看 u-boot 到底是如何具体跳转到 linux 下跑的，二来也为自己更深入的学习 linux kernel 打下基础。

本文以 arm 版的 linux 为例，从 kernel 的第一条指令开始分析,一直分析到进入 start_kernel()函数，也就是 kernel 启动的汇编部分，我们把它称之为第一部分， 以后有时间在把启动的第二部分在分析一下。我们当前以 linux-2.6.18 内核版本作为范例来分析,本文中所有的代码前面都会加上行号以便于讲解。

由于启动部分有一些代码是平台相关的,虽然大部分的平台所实现的功能都比较类似,但是为了更好的对 code 进行说明,对于平台相关的代码,我们选择 smdk2410 平台，CPU 是 s3c2410(arm 核是 arm920T)进行分析。

另外,本文是以未压缩的 kernel 来分析的.对于内核解压缩部分的 code,在 arch/arm/boot/compressed 中,本文不做讨论。

一. 启动条件

通常从系统上电执行的 boot loader 的代码， 而要从 boot loader 跳转到 linux kernel 的第一条指令处执行需要一些特定的条件。关于对 boot loader 的分析请看我的另一篇文档 u-boot 源码分析。

这里讨论下进入到 linux kernel 时必须具备的一些条件,这一般是 boot loader 在跳转到 kernel 之前要完成的:

1. CPU 必须处于 SVC(supervisor)模式,并且 IRQ 和 FIQ 中断都是禁止的;
2. MMU(内存管理单元)必须是关闭的，此时虚拟地址就是物理地址;
3. 数据 cache(Data cache)必须是关闭的
4. 指令 cache(Instruction cache)可以是打开的,也可以是关闭的,这个没有强制要求;
5. CPU 通用寄存器 0 (r0)必须是 0;
6. CPU 通用寄存器 1 (r1)必须是 ARM Linux machine type (关于 machine type，我们后面会有讲解)
7. CPU 通用寄存器 2 (r2) 必须是 kernel parameter list 的物理地址(parameter list 是由 boot loader 传递给 kernel,用来描述设备信息属性的列表)。

更详细的关于启动 arm linux 之前要做哪些准备工作可以参考，“Booting ARM Linux" 文档

二. starting kernel

首先，我们先对几个重要的宏进行说明(我们针对有 MMU 的情况)：

宏	位置	默认值	说明
KERNEL_RAM_ADDR	arch/arm/kernel/head.S +26	0xc0008000	kernel 在 RAM 中的虚拟地址
PAGE_OFFSET	include/asm-arm/memeory.h +50	0xc0000000	内核空间的起始虚拟地址
TEXT_OFFSET	arch/arm/Makefile +131	0x00008000	内核在 RAM 中起始位置相对于 RAM 起始地址的偏移
TEXTADDR	arch/arm/kernel/head.S +49	0xc0008000	kernel 的起始虚拟地址
PHYS_OFFSET	include/asm-arm/arch- *** /memory.h	平台相关	RAM 的起始物理地址，对于 s3c2410 来说在 include/asm-arm/arch-s3c2410/memory.h 下定义，值为 0x30000000(ram 接在片选 6 上)

内核的入口是 stext,这是在 arch/arm/kernel/vmlinux.lds.S 中定义的:

```
00011: ENTRY(stext)
```

对于 vmlinux.lds.S,这是 ld script 文件,此文件的格式和汇编及 C 程序都不同,本文不对 ld script 作过多的介绍,只对内核中用到的内容进行讲解,关于 ld 的详细内容可以参考 ld.info

这里的 ENTRY(stext) 表示程序的入口是在符号 stext.

而符号 stext 是在 arch/arm/kernel/head.S 中定义的:

下面我们将 arm linux boot 的主要代码列出来进行一个概括的介绍,然后,我们会逐个的进行详细的讲解.

在 arch/arm/kernel/head.S 中 72 - 94 行,是 arm linux boot 的主代码:

```
00072: ENTRY(stext)
00073:    msr    cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ ensure svc mode
00074:                                       @ and irqs disabled
00075:    mrc    p15, 0, r9, c0, c0            @ get processor id
00076:    bl     __lookup_processor_type        @ r5=procinfo r9=cpuid
00077:    movs   r10, r5                        @ invalid processor (r5=0)?
00078:    beq    __error_p                      @ yes, error 'p'
00079:    bl     __lookup_machine_type          @ r5=machinfo
00080:    movs   r8, r5                         @ invalid machine (r5=0)?
00081:    beq    __error_a                      @ yes, error 'a'
00082:    bl     __create_page_tables
```

在进入 linux kernel 前要确保在管理模式下，并且 IRQ,FIQ 都是关闭的，因此在 00073 行就是要确保这几个条件成立。

1. 确定 processor type

arch/arm/kernel/head.S 中:

```
00075:      mrc      p15, 0, r9, c0, c0          @ get processor id
00076:      bl       __lookup_processor_type     @ r5=procinfo r9=cpuid
00077:      movs     r10, r5                     @ invalid processor (r5=0)?
00078:      beq      __error_p                   @ yes, error 'p'
```

75 行: 通过 cp15 协处理器的 c0 寄存器来获得 processor id 的指令. 关于 cp15 的详细内容可参考相关的 arm 手册，也可直接参考 s3c2410 的 data sheet。

76 行: 跳转到__lookup_processor_type.在__lookup_processor_type 中,会把找到匹配的 processor type 对象存储在 r5 中。

77,78 行: 判断 r5 中的 processor type 是否是 0,如果是 0,说明系统中没找到匹配当前 processor type 的对象，则跳转到__error_p(出错)。系统中会预先定义本系统支持的 processor type 对象集。

__lookup_processor_type 函数主要是根据从 cpu 中获得的 processor id 和系统中预先定义的本系统能支持的 proc_info 集进行匹配,看系统能否支持当前的 processor, 并将匹配到的 proc_info 的基地址存到 r5 中, 0 表示没有找到对应的 processor type.

下面我们分析__lookup_processor_type 函数。

arch/arm/kernel/head-common.S 中:

```
00145:      .type     __lookup_processor_type, %function
00146: __lookup_processor_type:
00147:      adr       r3, 3f
00148:      ldmda     r3, {r5 - r7}
00149:      sub       r3, r3, r7                 @ get offset between virt&phys
00150:      add       r5, r5, r3                 @ convert virt addresses to
00151:      add       r6, r6, r3                 @ physical address space
00152: 1:   ldmia     r5, {r3, r4}               @ value, mask
00153:      and       r4, r4, r9                 @ mask wanted bits
00154:      teq       r3, r4
00155:      beq       2f
00156:      add       r5, r5, #PROC_INFO_SZ      @ sizeof(proc_info_list)
00157:      cmp       r5, r6
00158:      blo       1b
00159:      mov       r5, #0                     @ unknown processor
00160: 2:   mov       pc, lr
00161:
00162: /*
00163:  * This provides a C-API version of the above function.
00164:  */
```

```
00165: ENTRY(lookup_processor_type)
00166:        stmfd       sp!, {r4 - r7, r9, lr}
00167:        mov        r9, r0
00168:        bl         __lookup_processor_type
00169:        mov        r0, r5
00170:        ldmfd       sp!, {r4 - r7, r9, pc}
00171:
00172: /*
00173:   * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
00174:   * more information about the __proc_info and __arch_info structures.
00175:   */
00176:        .long       __proc_info_begin
00177:        .long       __proc_info_end
00178:   3:   .long       .
00179:        .long       __arch_info_begin
00180:        .long       __arch_info_end
```

145, 146 行是函数定义

147 行: 取地址指令,这里的 3f 是向前 symbol 名称是 3 的位置,即第 178 行,将该地址存入 r3. 这里需要注意的是,adr 指令取址,获得的是基于 pc 的一个地址,要格外注意,这个地址是 3f 处的"运行时地址",由于此时 MMU 还没有打开,也可以理解成物理地址(实地址).(详细内容可参考 arm 指令手册)

148 行: 因为 r3 中的地址是 178 行的位置的地址,因而执行完后:

r5 存的是 176 行符号 __proc_info_begin 的地址;
r6 存的是 177 行符号 __proc_info_end 的地址;
r7 存的是 3f 处的地址.

这里需要注意链接地址和运行时地址的区别. r3 存储的是运行时地址(物理地址),而 r7 中存储的是链接地址(虚拟地址).

__proc_info_begin 和__proc_info_end 是在 arch/arm/kernel/vmlinux.lds.S 中:

```
00031:              __proc_info_begin = .;
00032:                     *(.proc.info.init)
00033:              __proc_info_end = .;
```

这里是声明了两个变量:__proc_info_begin 和 __proc_info_end,其中等号后面的"."是 location counter(详细内容请参考 ld.info)

这三行的意思是: __proc_info_begin 的位置上,放置所有文件中的 ".proc.info.init" 段的内容,然后紧接着是 __proc_info_end 的位置.

kernel 使用 struct proc_info_list 来描述 processor type.

在 include/asm-arm/procinfo.h 中:

```
00029: struct proc_info_list {
00030:        unsigned int           cpu_val;
00031:        unsigned int           cpu_mask;
00032:        unsigned long          __cpu_mm_mmu_flags;      /*
used by head.S */
```

```
00033:          unsigned  long                    __cpu_io_mmu_flags;          /*  used
by head.S */
00034:          unsigned  long                    __cpu_flush;                 /*  used
by head.S */
00035:          const char                    *arch_name;
00036:          const char                    *elf_name;
00037:          unsigned int                  elf_hwcap;
00038:          const char                    *cpu_name;
00039:          struct processor              *proc;
00040:          struct cpu_tlb_fns            *tlb;
00041:          struct cpu_user_fns           *user;
00042:          struct cpu_cache_fns          *cache;
00043:
};
```

我们当前以 s3c2410 为例,其 processor 是 920t 的.

在 arch/arm/mm/proc-arm920.S 中:

```
00448:        .section ".proc.info.init", #alloc, #execinstr
00449:
00450: .type     __arm920_proc_info,#object
00451:        __arm920_proc_info:
00452:        .long     0x41009200
004523:       .long     0xff00fff0
00454:        .long     PMD_TYPE_SECT | /
00455:              PMD_SECT_BUFFERABLE | /
00456:              PMD_SECT_CACHEABLE | /
00457:              PMD_BIT4 | /
00458:              PMD_SECT_AP_WRITE | /
00459:              PMD_SECT_AP_READ
00460:        .long     PMD_TYPE_SECT | /
00461:               PMD_BIT4 | /
00462:               PMD_SECT_AP_WRITE | /
00463:               PMD_SECT_AP_READ
00464:        b     __arm920_setup
00465:        .long     cpu_arch_name
00466:        .long     cpu_elf_name
00467:        .long     HWCAP_SWP | HWCAP_HALF | HWCAP_THUMB
00468:        .long     cpu_arm920_name
00469:        .long     arm920_processor_functions
00470:        .long     v4wbi_tlb_fns
00471:        .long     v4wb_user_fns
00472:   #ifndef CONFIG_CPU_DCACHE_WRITETHROUGH
00473:        .long    arm920_cache_fns
00474:   #else
00475:        .long     v4wt_cache_fns
```

```
00476:    #endif
00477:          .size      __arm920_proc_info, . - __arm920_proc_info
```

从 448 行,我们可以看到 __arm920_proc_info 被放到了".proc.info.init"段中.对照 struct proc_info_list,我们可以看到 __cpu_flush 的定义是在 464 行,即__arm920_setup.(我们将在"4. 调用平台特定的__cpu_flush 函数"一节中详细分析这部分的内容.)

我们继续分析__lookup_processor_type

149 行: 从上面的分析我们可以知道 r3 中存储的是 3f 处的物理地址,而 r7 存储的是 3f 处的虚拟地址,这一行是计算当前程序运行的物理地址和虚拟地址的差值,将其保存到 r3 中.

150 行: 将 r5 存储的虚拟地址(__proc_info_begin)转换成物理地址

151 行: 将 r6 存储的虚拟地址(__proc_info_end)转换成物理地址

152 行: 对照 struct proc_info_list,可以得知,这句是将当前 proc_info 的 cpu_val 和 cpu_mask 分别存

到 r3, r4 中

153 行: r9 中存储了 processor id(arch/arm/kernel/head.S 中的 75 行),与 r4 的 cpu_mask 进行逻辑与得到我们需要的值

154 行: 将 153 行中得到的值与 r3 中的 cpu_val 进行比较

155 行: 如果相等,说明我们找到了对应的 processor type,跳到 160 行,返回

156 行: 如果不相等, 将 r5 指向下一个 proc_info,

157 行: 和 r6 比较,检查是否到了__proc_info_end.

158 行: 如果没有到__proc_info_end,表明还有 proc_info 配置,返回 152 行继续查找

159 行: 执行到这里,说明所有的 proc_info 都匹配过了,但是没有找到匹配的,将 r5 设置成 0(unknown processor)

160 行: 返回

2. 确定 machine type

继续分析 head.S,确定了 processor type 之后，就要确定 machine type 了

arch/arm/kernel/head.S 中:

```
00079:          bl          __lookup_machine_type              @ r5=machinfo
00080:          movs        r8,  r5                                  @ invalid
machine (r5=0)?
00081:          beq         __error_a                          @ yes, error 'a'
```

79 行: 跳转到__lookup_machine_type 函数, 和 proc_info 一样，在系统中也预先定义好了本系统能支持的 machine type 集， 在__lookup_machine_type 中,就是要查找系统中是否有对当前 machine type 的支持， 如果查找到则会把 struct machine_desc 的基地址(machine type)存储在 r5 中。

80,81 行: 将 r5 中的 machine_desc 的基地址存储到 r8 中,并判断 r5 是否是 0,如果是 0,说明是无效的 machine type,跳转到__error_a(出错)

__lookup_machine_type 函数

下面我们分析__lookup_machine_type 函数:

arch/arm/kernel/head-common.S 中:

```
00176:          .long          __proc_info_begin
```

```
00177:          .long           __proc_info_end
00178: 3:       .long           .
00179:          .long           __arch_info_begin
00180:          .long           __arch_info_end
00181:
00182: /*
00183:   * Lookup machine architecture in the linker-build list of architectures.
00184:   * Note that we can't use the absolute addresses for the __arch_info
00185:   * lists since we aren't running with the MMU on (and therefore, we are
00186:   * not in the correct address space).   We have to calculate the offset.
00187:   *
00188:   *   r1 = machine architecture number
00189:   * Returns:
00190:   *   r3, r4, r6 corrupted
00191:   *   r5 = mach_info pointer in physical address space
00192:   */
00193:   .type    __lookup_machine_type, %function
00194: __lookup_machine_type:
00195:       adr r3, 3b
00196:       ldmia    r3, {r4, r5, r6}
00197:       sub r3, r3, r4              @ get offset between virt&phys
00198:       add r5, r5, r3              @ convert virt addresses to
00199:       add r6, r6, r3              @ physical address space
00200:   1:  ldr r3, [r5, #MACHINFO_TYPE]     @ get machine type
00201:       teq r3, r1                  @ matches loader number?
00202:       beq 2f                      @ found
00203:       add r5, r5, #SIZEOF_MACHINE_DESC    @ next machine_desc
00204:       cmp r5, r6
00205:       blo 1b
00206:       mov r5, #0                  @ unknown machine
00207: 2:    mov pc, lr
```

实际上上面这段代码的原理和确定 processor type 的原理是一样的。

内核中,一般使用宏 MACHINE_START 来定义 machine type。
对于 smdk2410 来说，在 arch/arm/mach-s3c2410/Mach-smdk2410.c 中:

```
MACHINE_START(SMDK2410, "SMDK2410") /* @TODO: request a new identifier and switch
                                     * to SMDK2410 */
    /* Maintainer: Jonas Dietsche */
    .phys_io        = S3C2410_PA_UART,
    .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C2410_SDRAM_PA + 0x100,
    .map_io         = smdk2410_map_io,
    .init_irq       = s3c24xx_init_irq,
    .init_machine   = smdk_machine_init,
```

```
        .timer          = &s3c24xx_timer,
MACHINE_END
```

195 行：把 3b 处的地址存入 r3 中，3b 处的地址就是 178 行处的地址。

196 行: 把 3b 处开始的连续地址即 3b 处的地址，__arch_info_begin，__arch_info_end 依次存入 r4,r5,r6.

197 行: r3 中存储的是 3b 处的物理地址,而 r4 中存储的是 3b 处的虚拟地址,这里计算处物理地址和虚拟地址的差值,保存到 r3 中

198 行: 将 r5 存储的虚拟地址(__arch_info_begin)转换成物理地址

199 行: 将 r6 存储的虚拟地址(__arch_info_end)转换成物理地址

200 行: MACHINFO_TYPE 在 arch/arm/kernel/asm-offset.c 101 行定义, 这里是取 struct machine_desc 中的 nr(architecture number) 到 r3 中

201 行: 将 r3 中取到的 machine type 和 r1 中的 machine type(见前面的"启动条件")进行比较

202 行: 如果相同,说明找到了对应的 machine type,跳转到 207 行的 2f 处,此时 r5 中存储了对应的 struct machine_desc 的基地址

203 行: 如果不匹配, 则取下一个 machine_desc 的地址

204 行: 和 r6 进行比较,检查是否到了__arch_info_end.

205 行: 如果没到尾,说明还有 machine_desc,返回 200 行继续查找.

206 行: 执行到这里,说明所有的 machind_desc 都查找完了,并且没有找到匹配的, 将 r5 设置成 0(unknown machine).

207 行: 返回

3. 创建页表

继续分析 head.S,确定了 processor type 和 machine type 之后，就是创建页表。

通过前面的两步,我们已经确定了 processor type 和 machine type.

此时,一些特定寄存器的值如下所示:

r8 = machine info (struct machine_desc 的基地址)

r9 = cpu id (通过 cp15 协处理器获得的 cpu id)

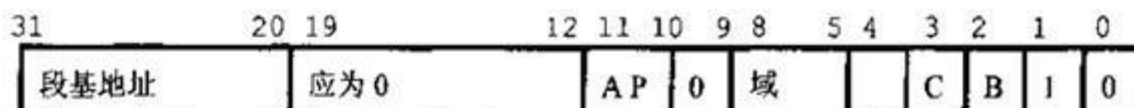
r10 = procinfo (struct proc_info_list 的基地址)

创建页表是通过函数 __create_page_tables 来实现的.

这里,我们使用的是 arm 的 L1 主页表,L1 主页表也称为段页表(section page table)，L1 主页表将 4 GB 的地址空间分成若干个 1 MB 的段(section),因此 L1 页表包含 4096 个页表项(section entry). 每个页表项是 32 bits(4 bytes)

因而 L1 主页表占用 4096 *4 = 16k 的内存空间.

对于 ARM920,其 L1 section entry 的格式为可参考 arm920t TRM):



它的地址翻译过程如下：

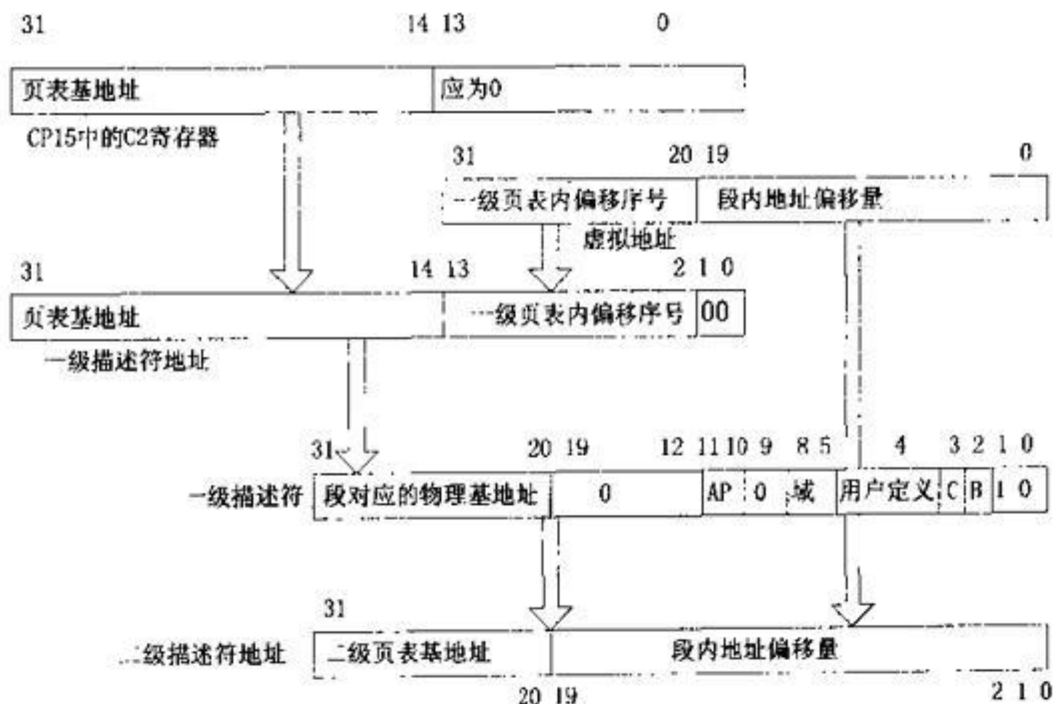


图 5.4 基于段的地址变换的过程

下面我们来分析 __create_page_tables 函数:

在 arch/arm/kernel/head.S 中:

```
00206:          .type       __create_page_tables, %function
00207: __create_page_tables:
00208:          pgtbl       r4                                      @ page table
address
00209:
00210:         /*
00211:          * Clear the 16K level 1 swapper page table
00212:          */
00213:         mov         r0, r4
00214:         mov         r3, #0
00215:         add         r6, r0, #0x4000
00216: 1:     str         r3, [r0], #4
00217:         str         r3, [r0], #4
00218:         str         r3, [r0], #4
00219:         str         r3, [r0], #4
00220:         teq         r0, r6
00221:         bne         1b
00222:
00223:              ldr         r7,  [r10,  #PROCINFO_MM_MMUFLAGS]        @
mm_mmuflags
00224:
00225:         /*
```

```
00226:          * Create identity mapping for first MB of kernel to
00227:          * cater for the MMU enable.    This identity mapping
00228:          * will be removed by paging_init().    We use our current program
00229:          * counter to determine corresponding section base address.
00230:          */
00231:         mov         r6, pc, lsr #20                 @ start of kernel
section
00232:         orr         r3, r7, r6, lsl #20             @ flags + kernel base
00233:         str         r3, [r4, r6, lsl #2]            @ identity mapping
00234:
00235:         /*
00236:          * Now setup the pagetables for our kernel direct
00237:          * mapped region. We round TEXTADDR down to the
00238:          * nearest megabyte boundary.    It is assumed that
00239:          * the kernel fits within 4 contigous 1MB sections.
00240:          */
00241:         add     r0, r4,   #(TEXTADDR & 0xff000000) >> 18        @ start of
kernel
00242:         str     r3, [r0, #(TEXTADDR & 0x00f00000) >> 18]!
00243:         add     r3, r3, #1 << 20
00244:         str     r3, [r0, #4]!           @ KERNEL + 1MB
00245:         add     r3, r3, #1 << 20
00246:         str     r3, [r0, #4]!           @ KERNEL + 2MB
00247:         add     r3, r3, #1 << 20
00248:         str     r3, [r0, #4]            @ KERNEL + 3MB
00249:
00250:         /*
00251:          * Then map first 1MB of ram in case it contains our boot params.
00252:          */
00253:         add         r0, r4, #PAGE_OFFSET >> 18
00254:         orr         r6, r7, #PHYS_OFFSET
00255:         str         r6, [r0]

        ...

00314:         mov         pc, lr
00315:         .ltorg
```

206, 207 行：函数声明

208 行：通过宏 pgtbl 将 r4 设置成页表的基地址(物理地址)

宏 pgtbl 在 arch/arm/kernel/head.S 中：

```
00042:         .macro          pgtbl, rd
00043:         ldr             /rd, =(__virt_to_phys(KERNEL_RAM_ADDR - 0x4000))
00044:         .endm
```

可以看到,页表是位于 KERNEL_RAM_ADDR 下面 16k 的位置

宏 __virt_to_phys 是在 incude/asm-arm/memory.h 中:

```
00125: #ifndef __virt_to_phys
00126: #define __virt_to_phys(x)          ((x) - PAGE_OFFSET + PHYS_OFFSET)
00127: #define __phys_to_virt(x)          ((x) - PHYS_OFFSET + PAGE_OFFSET)
00128: #endif
```

下面从 213 行 - 221 行, 是将这 16k 的页表清 0.

213 行: r0 = r4, 将页表基地址存在 r0 中

214 行: 将 r3 置成 0

215 行: r6 = 页表基地址 + 16k, 可以看到这是页表的尾地址

216 - 221 行: 循环,从 r0 到 r6 将这 16k 页表用 0 填充.

223 行: 获得 proc_info_list 的__cpu_mm_mmu_flags 的值,并存储到 r7 中. (宏 PROCINFO_MM_MMUFLAGS 是在 arch/arm/kernel/asm-offset.c 中定义)

231 行: 通过 pc 值的高 12 位(右移 20 位),得到 kernel 的 section 基址(从上面的图可以看出),并存储到 r6 中.因为当前是通过运行时地址得到的 kernel 的 section 地址,因而是物理地址.

232 行: r3 = r7 | (r6 << 20); flags + kernel base,得到页表中需要设置的值.

233 行: 设置页表: mem[r4 + r6 * 4] = r3,这里,因为页表的每一项是 32 bits(4 bytes),所以要乘以 4(<<2).

上面这三行,设置了 kernel 当前运行的 section(物理地址所在的 page entry)的页表项

241--248 行: TEXTADDR 是内核的起始虚拟地址(0xc0008000), 这几行是设置 kernel 起始 4M 虚拟地址的页表项(个人觉得 242 行设置的页表项和上面 233 行设置的页表项是同一个，因为 r3 没有变，就是 kernel 头 1M 的页表项)。

/* TODO: 这两行的 code 很奇怪,为什么要先取 TEXTADDR 的高 8 位(Bit[31:24])0xff000000,然后再取后面的 8 位(Bit[23:20])0x00f00000*/

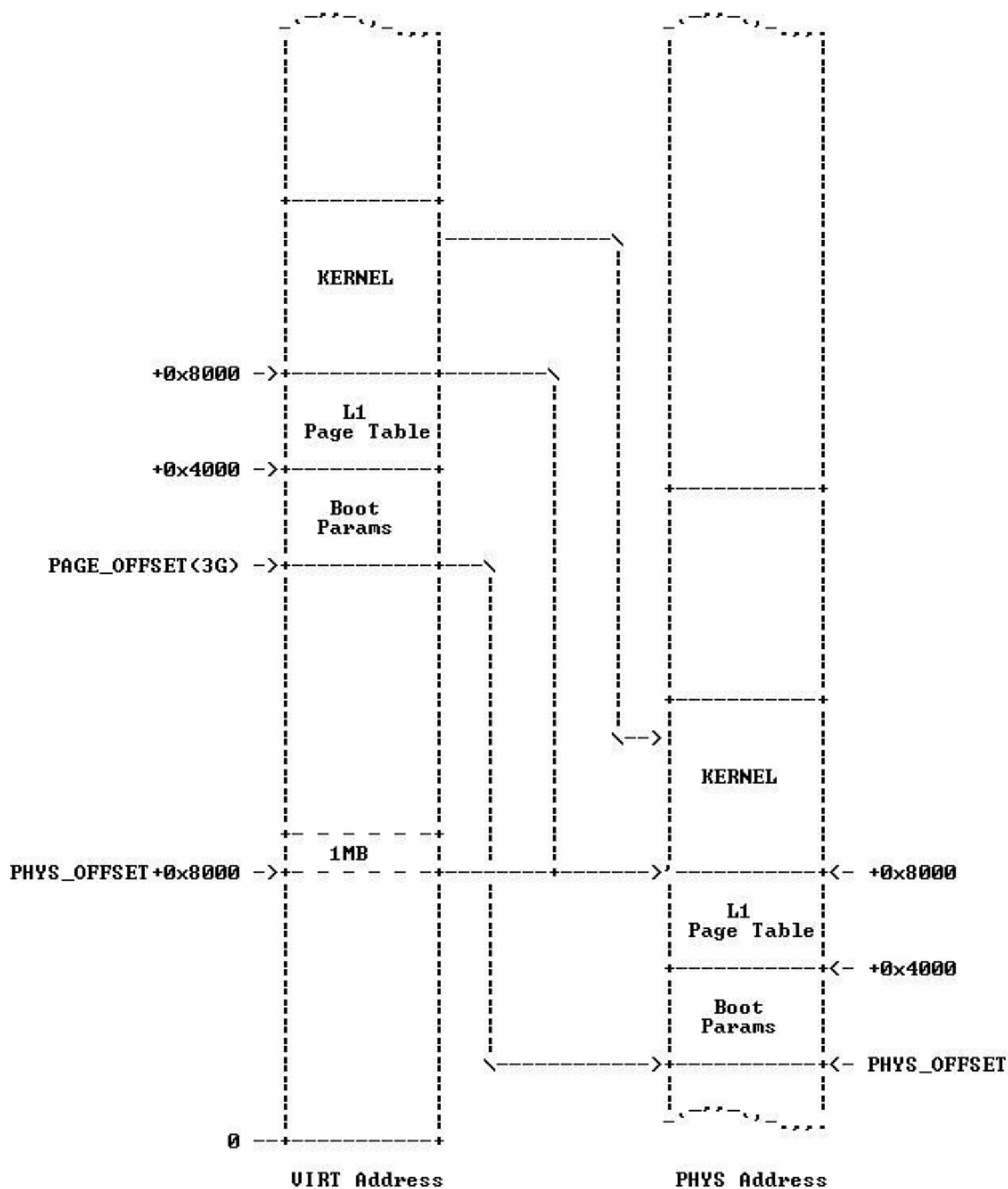
253 行: 将 r0 设置为 RAM 第一兆虚拟地址的页表项地址(page entry)

254 行: r7 中存储的是 mmu flags, 逻辑或上 RAM 的起始物理地址,得到 RAM 第一个 MB 页表项的值.

255 行: 设置 RAM 的第一个 MB 虚拟地址的页表.

上面这三行是用来设置 RAM 中第一兆虚拟地址的页表. 之所以要设置这个页表项的原因是 RAM 的第一兆内存中可能存储着 boot params.

这样,kernel 所需要的基本的页表我们都设置完了, 如下图所示:



4. 调用平台特定的 __cpu_flush 函数

当 __create_page_tables 返回之后

此时,一些特定寄存器的值如下所示:

r4 = pgtbl (page table 的物理基地址)

r8 = machine info (struct machine_desc 的基地址)

r9 = cpu id (通过 cp15 协处理器获得的 cpu id)
r10 = procinfo (struct proc_info_list 的基地址)

在我们需要开启 mmu 之前,做一些必须的工作:清除 ICache, 清除 DCache, 清除 Writebuffer, 清除 TLB 等.这些一般是通过 cp15 协处理器来实现的,并且是平台相关的. 这就是__cpu_flush 需要做的工作

在 arch/arm/kernel/head.S 中

```
00091:      ldr      r13, __switch_data          @ address to jump to after
00092:                                           @ mmu has been enabled
00093:      adr      lr, __enable_mmu            @ return (PIC) address
00094:      add      pc, r10, #PROCINFO_INITFUNC
```

第 91 行: 将 r13 设置为 __switch_data 的地址
第 92 行: 将 lr 设置为 __enable_mmu 的地址
第 93 行: r10 存储的是 procinfo 的基地址, PROCINFO_INITFUNC 是在 arch/arm/kernel/asm-offset.c 中 107 行定义. 该行将 pc 设为 proc_info_list 的 __cpu_flush 函数的地址, 即下面跳转到该函数.在分析 __lookup_processor_type 的时候,我们已经知道,对于 ARM920t 来说,其__cpu_flush 指向的是函数 __arm920_setup

下面我们来分析函数 __arm920_setup
在 arch/arm/mm/proc-arm920.S 中:

```
00385: .type    __arm920_setup, #function
00386: __arm920_setup:
00387:      mov r0, #0
00388:      mcr p15, 0, r0, c7, c7      @ invalidate I,D caches on v4
00389:      mcr p15, 0, r0, c7, c10, 4 @ drain write buffer on v4
00390: #ifdef CONFIG_MMU
00391:      mcr p15, 0, r0, c8, c7      @ invalidate I,D TLBs on v4
00392: #endif
00393:      adr r5, arm920_crval
00394:      ldmia    r5, {r5, r6}
00395:      mrc p15, 0, r0, c1, c0      @ get control register v4
00396:      bic r0, r0, r5
00397:      orr r0, r0, r6
00398:      mov pc, lr
00399:      .size    __arm920_setup, . - __arm920_setup
```

385,386 行: 定义__arm920_setup 函数。
387 行： 设置 r0 为 0。
388 行： 使数据 cahche, 指令 cache 无效。
389 行： 使 write buffer 无效。
391 行： 使数据 TLB,指令 TLB 无效。

393 行： 获取 arm920_crval 的地址，并存入 r5。
394 行： 获取 arm920_crval 地址处的连续 8 字节分别存入 r5,r6。
arm920_crval 在 arch/arm/mm/proc-arm920t.c:

```
    .type    arm920_crval, #object
arm920_crval:
    crval    clear=0x00003f3f, mmuset=0x00003135, ucset=0x00001130
```

由此可知，r5 = 0x00003f3f, r6 = 0x00003135

395 行： 获取 CP15 下控制寄存器的值，并存入 r0。
396 行： 通过查看 arm920_crval 的值可知该行是清除 r0 中相关位，为以后对这些位的赋值做准备。
397 行： 设置 r0 中的相关位，即为 mmu 做相应设置。
398 行： 函数返回。

5. 开启 mmu

开启 mmu 是由函数 __enable_mmu 实现的.

在进入 __enable_mmu 的时候, r0 中已经存放了控制寄存器 c1 的一些配置(在上一步中进行的设置), 但是并没有真正的打开 mmu, 在 __enable_mmu 中,我们将打开 mmu.

此时,一些特定寄存器的值如下所示:

```
r0 = c1 parameters       (用来配置控制寄存器的参数)
r4 = pgtbl               (page table 的物理基地址)
r8 = machine info        (struct machine_desc 的基地址)
r9 = cpu id              (通过 cp15 协处理器获得的 cpu id)
r10 = procinfo           (struct proc_info_list 的基地址)
```

在 arch/arm/kernel/head.S 中:

```
00146:          .type       __enable_mmu, %function
00147: __enable_mmu:
00148: #ifdef CONFIG_ALIGNMENT_TRAP
00149:          orr         r0, r0, #CR_A
00150: #else
00151:          bic         r0, r0, #CR_A
00152: #endif
00153: #ifdef CONFIG_CPU_DCACHE_DISABLE
00154:          bic         r0, r0, #CR_C
00155: #endif
00156: #ifdef CONFIG_CPU_BPREDICT_DISABLE
00157:          bic         r0, r0, #CR_Z
00158: #endif
00159: #ifdef CONFIG_CPU_ICACHE_DISABLE
00160:          bic         r0, r0, #CR_I
00161: #endif
00162:          mov           r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | /
00163:                           domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | /
```

```
00164:                              domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | /
00165:                              domain_val(DOMAIN_IO, DOMAIN_CLIENT))
00166:          mcr          p15, 0, r5, c3, c0, 0                @ load domain access register
00167:          mcr          p15, 0, r4, c2, c0, 0                @ load page table pointer
00168:          b            __turn_mmu_on
00169:
00170: /*
00171:  * Enable the MMU.   This completely changes the structure of the visible
00172:  * memory space.   You will not be able to trace execution through this.
00173:  * If you have an enquiry about this, *please* check the linux-arm-kernel
00174:  * mailing list archives BEFORE sending another post to the list.
00175:  *
00176:  *   r0   = cp#15 control register
00177:  *   r13 = *virtual* address to jump to upon completion
00178:  *
00179:  * other registers depend on the function called upon completion
00180:  */
00181:  .align   5
00182:  .type    __turn_mmu_on, %function
00183:__turn_mmu_on:
00184:       mov r0, r0
00185:       mcr p15, 0, r0, c1, c0, 0        @ write control reg
00186:       mrc p15, 0, r3, c0, c0, 0        @ read id reg
00187:       mov r3, r3
00188:       mov r3, r3
00189:       mov pc, r13
```

146，147 行: 定义__enable_mmu 函数。

148--152 行：根据配置使能或禁止地址对齐错误检测。

153--155 行：根据配置使能或禁止数据 cache。

156--158 行：reserved。

159--161 行：根据配置使能或禁止指令 cache。

162--165 行：配置相应的访问权限并存入 r5。

166 行：把访问权限写入 CP15 协处理器。

167 行：把页表地址写入 CP15 协处理器。

168 行：跳转到__turn_mmu_on 来打开 MMU。

接下来就是打开 MMU 了，我们看它的代码：

第 185 行: 写 cp15 的控制寄存器 c1, 这里是打开 mmu 的动作,同时会打开 cache 等(根据 r0 相应的配置)

第 186 行: 读取 id 寄存器.

第 187 - 188 行：两个 nop.

第 189 行：取 r13 到 pc 中,我们前面已经看到了，r13 中存储的是 __switch_data (在 arch/arm/kernel/head.S 91 行),下面会跳到 __switch_data.

第 187,188 行的两个 nop 是非常重要的,因为在 185 行打开 mmu 动作之后,要等到 3 个 cycle 之后才会生效,这和 arm 的流水线有关系.

因而,在打开 mmu 动作之后又加了两个 nop 动作.

6. 切换数据

下面我们就来看__switch_data:

在 arch/arm/kernel/head-common.S 中:

```
00014:          .type       __switch_data, %object
00015: __switch_data:
00016:          .long       __mmap_switched
00017:          .long       __data_loc                      @ r4
00018:          .long       __data_start                    @ r5
00019:          .long       __bss_start                     @ r6
00020:          .long       _end                            @ r7
00021:          .long       processor_id                    @ r4
00022:          .long       __machine_arch_type             @ r5
00023:          .long       cr_alignment                    @ r6
00024:          .long       init_thread_union + THREAD_START_SP @ sp
00025:
```

第 14, 15 行：对象定义。

第 16 - 24 行：为对象里的每个域赋值，例如第 16 行存储的是 __mmap_switched 的地址，第 17 行存储的是 __data_loc 的地址 ......

由上面对__switch_data 的定义可知，最终调用的是__mmap_switched

下面我们就来看__mmap_switched:

在 arch/arm/kernel/head-common.S 中:

```
00026: /*
00027:   * The following fragment of code is executed with the MMU on in MMU mode,
00028:   * and uses absolute addresses; this is not position independent.
00029:   *
00030:   *   r0  = cp#15 control register
00031:   *   r1  = machine ID
00032:   *   r9  = processor ID
00033:   */
00034:   .type   __mmap_switched, %function
00035: __mmap_switched:
00036:          adr r3, __switch_data + 4
00037:
00038:          ldmia     r3!, {r4, r5, r6, r7}
```

```
00039:      cmp r4, r5              @ Copy data segment if needed
00040: 1:   cmpne   r5, r6
00041:      ldrne   fp, [r4], #4
00042:      strne   fp, [r5], #4
00043:      bne 1b
00044:
00045:      mov fp, #0              @ Clear BSS (and zero fp)
00046: 1:   cmp r6, r7
00047:      strcc   fp, [r6],#4
00048:      bcc 1b
00049:
00050:      ldmia   r3, {r4, r5, r6, sp}
00051:      str r9, [r4]            @ Save processor ID
00052:      str r1, [r5]            @ Save machine type
00053:      bic r4, r0, #CR_A           @ Clear 'A' bit
00054:      stmia   r6, {r0, r4}            @ Save control register values
00055:      b   start_kernel
```

注意上面这些代码就已经跑在了 MMU 打开的情况下了。

第 34, 35 行: 函数 __mmap_switched 的定义。

第 36 行: 取 __switch_data + 4 的地址到 r3. 从上文可以看到这个地址就是第 17 行的地址.

第 38 行: 依次取出从第 17 行到第 20 行的地址,存储到 r4, r5, r6, r7 中. 并且累加 r3 的值.当执行完后, r3 指向了第 21 行的位置.

对照上文,我们可以得知:

r4 - __data_loc
r5 - __data_start
r6 - __bss_start
r7 - _end

这几个符号都是在 arch/arm/kernel/vmlinux.lds.S 中定义的变量:

```
00102: #ifdef CONFIG_XIP_KERNEL
00103:          __data_loc = ALIGN(4);              /* location in binary */
00104:          . = PAGE_OFFSET + TEXT_OFFSET;
00105: #else
00106:          . = ALIGN(THREAD_SIZE);
00107:          __data_loc = .;
00108: #endif
00109:
00110:          .data : AT(__data_loc) {
00111:                  __data_start = .;       /* address in memory */
00112:
00113:                  /*
00114:                   * first, the init task union, aligned
00115:                   * to an 8192 byte boundary.
```

```
00116:                         */
00117:                        *(.init.task)
            ......
00158:              .bss : {
00159:                                     __bss_start    =    .;                    /*    BSS
*/
00160:                        *(.bss)
00161:                        *(COMMON)
00162:                        _end = .;
00163:              }
```

对于这四个变量,我们简单的介绍一下:

__data_loc 是数据存放的位置

__data_start 是数据开始的位置

__bss_start 是 bss 开始的位置

_end 是 bss 结束的位置, 也是内核结束的位置

其中对第 110 行的指令讲解一下: 这里定义了.data 段,后面的 AT(__data_loc) 的意思是这部分的内容是在__data_loc 中存储的(要注意,储存的位置和链接的位置是可以不相同的).

关于 AT 详细的信息请参考 ld.info

第 38 行: 比较 __data_loc 和 __data_start

第 39 - 43 行: 这几行是判断数据存储的位置和数据的开始的位置是否相等,如果不相等,则需要搬运数据,从 __data_loc 将数据搬到 __data_start. 其中 __bss_start 是 bss 的开始的位置,也标志了 data 结束的位置,因而用其作为判断数据是否搬运完成.

第 45 - 48 行: 是清除 bss 段的内容,将其都置成 0. 这里使用 _end 来判断 bss 的结束位置.

第 50 行: 因为在第 38 行的时候,r3 被更新到指向第 21 行的位置.因而这里取得 r4, r5, r6, sp 的值分别是:

r4 - processor_id

r5 - __machine_arch_type

r6 - cr_alignment

sp - init_thread_union + THREAD_START_SP

processor_id 和 __machine_arch_type 这两个变量是在 arch/arm/kernel/setup.c 中 第 62, 63 行中定义的.

cr_alignment 是在 arch/arm/kernel/entry-armv.S 中定义的:

```
00182:              .globl          cr_alignment
00183:              .globl          cr_no_alignment
00184: cr_alignment:
00185:              .space           4
00186: cr_no_alignment:
```

```
00187:          .space          4
```

init_thread_union 是 init 进程的基地址. 在 arch/arm/kernel/init_task.c 中:

```
00033: union thread_union init_thread_union
00034:          __attribute__((__section__(".init.task"))) =
00035:                   { INIT_THREAD_INFO(init_task) };
```

对照 vmlnux.lds.S 中的 的 117 行,我们可以知道 init task 是存放在 .data 段的开始 8k, 并且是 THREAD_SIZE(8k)对齐的

第 51 行: 将 r9 中存放的 processor id (在 arch/arm/kernel/head.S 75 行) 赋值给变量 processor_id

第 52 行: 将 r1 中存放的 machine id (见"启动条件"一节)赋值给变量 __machine_arch_type

第 53 行: 清除 r0 中的 CR_A 位并将值存到 r4 中. CR_A 是在 include/asm-arm/system.h 21 行定义, 是 cp15 控制寄存器 c1 的 Bit[1](alignment fault enable/disable)

第 54 行: 这一行是存储控制寄存器的值.

从上面 arch/arm/kernel/entry-armv.S 的代码我们可以得知.

这一句是将 r0 存储到了 cr_alignment 中,将 r4 存储到了 cr_no_alignment 中.

第 55 行: 最终跳转到 start_kernel

arm Linux 系统启动之----start_kernel 函数 :
http://blog.csdn.net/skywalkzf/article/details/6412973

linux 字符设备驱动总结之：全自动创建设备及节点 ：
http://blog.csdn.net/zyhui65/article/details/8053116

附录 7 linux 字符设备驱动总结之：全自动创建设备及节点

摘自：http://blog.csdn.net/zyhui65/article/details/8053116

/**

linux 字符设备驱动总结之：全自动创建设备及节点

看了 LDD3，深入浅出 LDD，以及各个博文，还是需要总结下的。

张永辉 2012 年 10 月 9 日

**/

概览：

第一步：注册设备号　　　　信息#tail -f /var/log/message

注册函数：

register_chrdev_region() 或　　　　查看#lsmod

alloc_chrdev_region() 或　　　　查看#cat /proc/devices

register_chrdev()

注销函数：

unregist_chrdev_region() 或

unregister_chrdev()

第二步：初始化 cdev 并添加到系统

初始化 cdev

静态初始化 cdev_init() 或

动态初始化 cdev_alloc()

添加到系统函数

cdev_add()

从系统删除函数

cdev_del()

第三步：创建设备节点

创建类

class_create()　　将放于/sysfs　　查看#ls /sys/class

删除类

class_destroy()

创建节点

device_create() 或 class_device_create()　将存放于/dev　查看#ls /dev

删除节点

device_destroy() 或 class_device_destroy()

/***

第一步：注册设备号

***/

Linux 内核中所有已分配的字符设备编号都记录在一个名为 chrdevs 散列表里。

该散列表中的每一个元素是一个 char_device_struct 结构，它的定义如下：

```
static struct char_device_struct
{
    struct char_device_struct *next;      // 指向散列冲突链表中的下一个元素的指针
    unsigned     int major;               // 主设备号
    unsigned     int baseminor;           // 起始次设备号
    int minorct;                          // 设备编号的范围大小
    char     name[64];                    // 处理该设备编号范围内的设备驱动的名称
    struct file_operations *fops;         // 没有使用
    struct cdev *cdev;                    // 指向字符设备驱动程序描述符的指针
}*chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

1 每一个主设备有一个会分配一个此结构，可以有多个次设备号。次设备是依次递增的。

2 内核提供了 5 个函数来来管理字符设备编号。

register_chrdev_region()　　指定初始值

alloc_chrdev_region()　　动态分配

register_chrdev()　　指定设备号

他们都会调用 __register_chrdev_region() 来注册一组设备编号范围(一个char_device_struct结构),我们使用其中一个即可。

unregist_chrdev_region()　　释放都用此函数

unregister_chrdev()　　都调用了 __unregister_chrdev_region() 来注销设备

注册：

register_chrdev_region(dev_t first,unsigned int count,char *name)

first :要分配的设备编号范围的初始值(次设备号常设为 0);

count :连续编号范围.

Name　:编号相关联的设备名称. (/proc/devices);

int alloc_chrdev_region(dev_t *dev,unsigned int firstminor,unsigned int count,char *name);

*dev　　:存放返回的设备号

firstminor　:第一个次设备号的号数,常为 0;

int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)

major ：要注册的设备号，　若为 0 则自动分配一个

name　：设备名

*fops ：以后再聊

释放:

```
void unregister_chrdev(unsigned int major, const char *name);
void unregister_chrdev_region(dev_t from, unsigned count);
```

3 示例：略

4 参考：感谢原著 (有此 6 个函数的源码及解说)。

http://blog.csdn.net/iLetLet/article/details/6180314

/***

第二步：初始化 cdev 并添加到系统

***/

1.内核中每个字符设备都对应一个 cdev 结构的变量，定义如下：

```
linux-2.6.22/include/linux/cdev.h
struct cdev
{
    struct kobject kobj;                    //每个 cdev 都是一个 kobject
    struct module *owner;                   //指向实现驱动的模块
    const struct file_operations *ops;     //操纵这个字符设备文件的方法
    struct list_head list;                  //与 cdev 对应的字符设备文件的 inode->i_devices 的链表头
    dev_t dev;                              //起始设备编号
    unsigned int count;                     //设备范围号大小
};
```

2. 初始化 cdev ：有两种定义初始化方式：

方式 1：静态内存定义初始化：

```
struct cdev my_cdev;
cdev_init(&my_cdev, &fops);
my_cdev.owner = THIS_MODULE;
```

方式 2：动态内存定义初始化：

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &fops;
my_cdev->owner = THIS_MODULE;
```

下面是 2 函数的具体代码：

```
struct cdev *cdev_alloc(void)           //它主要完成了空间的申请和简单的初始化操作；
{
    struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
    if (p)
    {
        INIT_LIST_HEAD(&p->list);
        kobject_init(&p->kobj, &ktype_cdev_dynamic);
    }
    return p;
}

void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

```
{
    memset(cdev, 0, sizeof *cdev);   //主要是对空间起到一个清零作用并较之 cdev_alloc 多了一个 ops 的赋值操作
    INIT_LIST_HEAD(&cdev->list);
    kobject_init(&cdev->kobj, &ktype_cdev_default);
    cdev->ops = fops;
}
```

3. 添加 cdev 到系统

为此可以调用 cdev_add() 函数。传入 cdev 结构的指针，起始设备编号，以及设备编号范围。

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
    p->dev = dev;
    p->count = count;
    return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

释放时使用 cdev_del()函数来释放 cdev 占用的内存。

```
void cdev_del(struct cdev *p)
{
    cdev_unmap(p->dev, p->count);     //释放 cdev_map 散列表中的对象
    kobject_put(&p->kobj);            //释放 cdev 结构本身。
}
```

4.关于 kobject_init() kobj_map()

内核中所有都字符设备都会记录在一个 kobj_map 结构的 cdev_map 变量中。

这个结构的变量中包含一个散列表用来快速存取所有的对象。

kobj_map() 函数就是用来把字符设备编号和 cdev 结构变量一起保存到 cdev_map 这个散列表里。

当后续要打开一个字符设备文件时，通过调用 kobj_lookup() 函数，根据设备编号就可以找到 cdev 结构变量，从而取出其中的 ops 字段。

/**

第三步：创建设备节点

**/

方法一：利用 mknod 命令手动创建设备节点。

方法二：实际上 Linux 内核为我们提供了一组函数,可以在模块加载的时候在/dev 目录下创建相应设备节点,在卸载时可删除该节点。

原理：

1 内核中定义了 struct class 结构体，它对应一个类。

2 先调用 class_create()函数，可以用它来创建一个类，这个类将存放于 sysfs 下面.

3 再调用 device_create()函数，从而在/dev 目录下创建相应的设备节点。

4 卸载模块对应的函数是 device_destroy 和 class_destroy()

注：2.6 以后的版本使用 device_create(),之前的版本使用的 class_device_create()。

详解：

1:class 结构：

```
include/linux/device.h
struct class
{
```

```
        const      char              *name;
        struct module             *owner;
        struct kset               subsys;
        struct list_head        devices;
        struct list_head        interfaces;
        struct kset               class_dirs;
        struct semaphore sem;            /* locks     children, devices, interfaces */
        struct class_attribute     *class_attrs;
        struct device_attribute *dev_attrs;

        int       (*dev_uevent)     (struct device *dev,struct kobj_uevent_env *env);
        void      (*class_release)(struct class *class);
        void      (*dev_release)    (struct device      *dev);
        int       (*suspend)          (struct    device *dev, pm_message_t state);
        int       (*resume)            (struct device *dev);
    };
```

2:class_create()

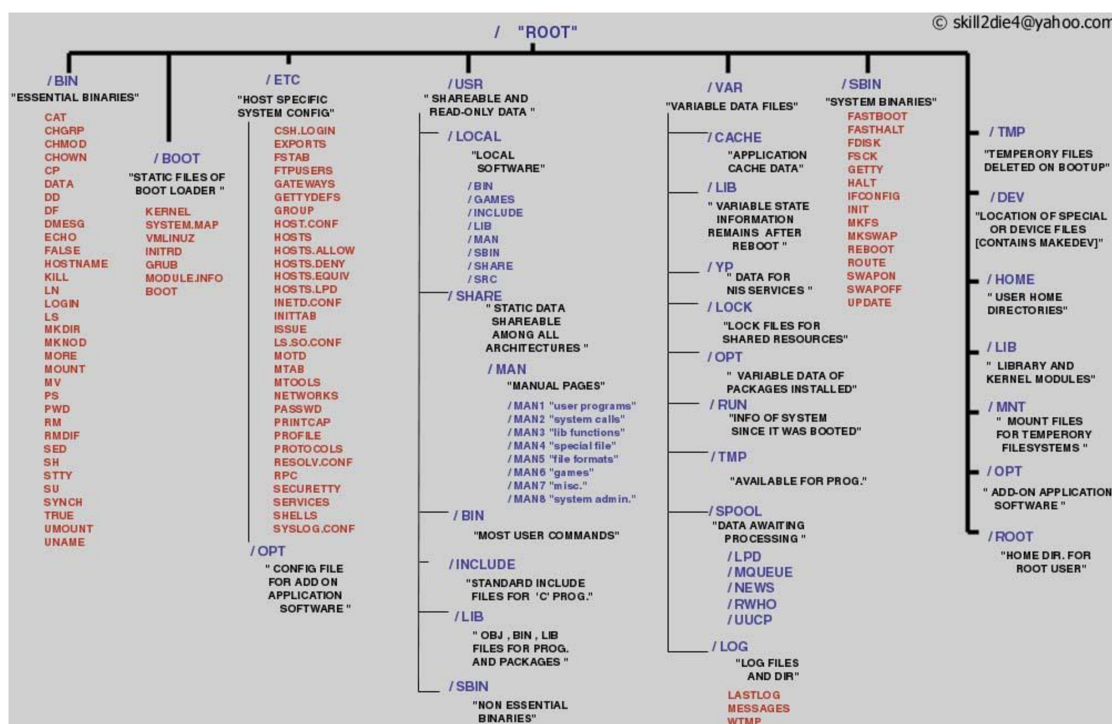
class_create()在/drivers/base/class.c 中实现：

```
    struct class       *class_create(struct module *owner, //    指定类的所有者是哪个模块
                                              const char *name)        //    指定类名
    {
        struct class *cls;
        int retval;
        cls =      kzalloc(sizeof(*cls), GFP_KERNEL);
        if (!cls)
        {
            retval =        -ENOMEM;
            goto        error;
        }

        cls->name      = name;
        cls->owner = owner;
        cls->class_release = class_create_release;

        retval = class_register(cls);
        if (retval)
            goto error;
        return cls;
        error:
            kfree(cls);
            return ERR_PTR(retval);
    }
```

3:device_create()函数在/drivers/base/core.c 中实现：

```
struct device *device_create(struct class *class,      //指定所要创建的设备所从属的类
                                    struct devicev *parent, //这个设备的父设备,如果没有就指定为 NULL
                                    dev_t devt,                  //设备号
                                    const char *fmt,             //设备名称
                                    ...)                         //从设备号
{
      va_list vargs;
      struct    device *dev;
      va_start(vargs, fmt);
      dev = device_create_vargs(class, parent, devt,    NULL, fmt, vargs);
      va_end(vargs);
      return    dev;
}
```

附录 8　LINUX 下的文件结构介绍



/bin 二进制可执行命令

/dev 设备特殊文件

/etc 系统管理和配置文件

/etc/rc.d 启动的配置文件和脚本

/home 用户主目录的基点，比如用户 user 的主目录就是/home/user，可以用~user 表示

/lib 标准程序设计库，又叫动态链接共享库，作用类似 windows 里的.dll 文件

/sbin 系统管理命令，这里存放的是系统管理员使用的管理程序

/tmp 公用的临时文件存储点

/root 系统管理员的主目录（呵呵，特权阶级）

/mnt 系统提供这个目录是让用户临时挂载其他的文件系统。

/lost+found 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows 下叫什么.chk）就在这里

/proc 虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。

/var 某些大文件的溢出区，比方说各种服务的日志文件

/usr 最庞大的目录，要用到的应用程序和文件几乎都在这个目录。其中包含：

/usr/x11r6 存放 x window 的目录

/usr/bin 众多的应用程序

/usr/sbin 超级用户的一些管理程序

/usr/doc linux 文档

/usr/include linux 下开发和编译应用程序所需要的头文件

/usr/lib 常用的动态链接库和软件包的配置文件

/usr/man 帮助文档

/usr/src 源代码，linux 内核的源代码就放在/usr/src/linux 里

/usr/local/bin 本地增加的命令

/usr/local/lib 本地增加的库

通常情况下，根文件系统所占空间一般应该比较小，因为其中的绝大部分文件都不需要经常改动，而且包括严格的文件和一个小的不经常改变的文件系统不容易损坏。 除了可能的一个叫/ｖｍｌｉｎｕｚ标准的系统引导映像之外，根目录一般不含任何文件。所有其他文件在根文件系统的子目录中。

1. /bin 目录

/ｂｉｎ目录包含了引导启动所需的命令或普通用户可能用的命令(可能在引导启动后)。这些命令都是二进制文件的可执行程序(ｂｉｎ是ｂｉｎａｒｙ－－二进制的简称)，多是系统中重要的系统文件。

2. /sbin 目录

/ｓｂｉｎ目录类似/bin ，也用于存储二进制文件。因为其中的大部分文件多是系统管理员使用的基本的系统程序，所以虽然普通用户必要且允许时可以使用，但一般不给普通用户使用。

3. /etc 目录

/ｅｔｃ目录存放着各种系统配置文件，其中包括了用户信息文件/ｅｔｃ/ｐａｓｓｗｄ，系统初始化文件/ｅｔｃ/ｒｃ等。ｌｉｎｕｘ正是*这些文件才得以正常地运行。

4. /root 目录

/root 目录是超级用户的目录。

5. /lib 目录

/ｌｉｂ目录是根文件系统上的程序所需的共享库，存放了根文件系统程序运行所需的共享文件。这些文件包含了可被许多程序共享的代码，以避免每个程序都包含有相同的子程序的副本，故可以使得可执行文件变得更小，节省空间。

6. /lib/modules 目录

/lib/modules 目录包含系统核心可加载各种模块，尤其是那些在恢复损坏的系统时重新引导系统所需的模块(例如网络和文件系统驱动)。

7. /dev 目录

/ｄｅｖ目录存放了设备文件，即设备驱动程序，用户通过这些文件访问外部设备。比如，用户可以通过访问/ｄｅｖ/ｍｏｕｓｅ来访问鼠标的输入，就像访问其他文件一样。

8. /tmp 目录

/tmp 目录存放程序在运行时产生的信息和数据。但在引导启动后，运行的程序最好使用/ｖａｒ/ｔｍｐ来代替/tmp ，因为前者可能拥有一个更大的磁盘空间。

9. /boot 目录

/ｂｏｏｔ目录存放引导加载器(bootstrap loader)使用的文件，如ｌｉｌｏ，核心映像也经常放在这里，而不是放在根目录中。但是如果有许多核心映像，这个目录就可能变得很大，这时使用单独的文件系统会更好一些。还有一点要注意的是，要确保核心映像必须在ｉｄｅ硬盘的前１０２４柱面内。

10. /mnt 目录

/ｍｎｔ目录是系统管理员临时安装(ｍｏｕｎｔ)文件系统的安装点。程序并不自动支持安装到/mnt 。/mnt 下面可以分为许多子目录，例如/mnt/dosa 可能是使用ｍｓｄｏｓ文件系统的软驱，而/mnt/exta 可能是使用ｅｘｔ２文件系统的软驱，/mnt/cdrom 光驱等等。

11. /proc, /usr,/var,/home 目录

其他文件系统的安装点。

下面详细介绍；

/etc 文件系统

/etc 目录包含各种系统配置文件，下面说明其中的一些。其他的你应该知道它们属于哪个程序，并阅读该程序的ｍａｎ页。许多网络配置文件也在/etc 中。

1. /etc/rc 或/etc/rc.d 或/etc/rc?.d

启动、或改变运行级时运行的脚本或脚本的目录。

2. /etc/passwd

用户数据库，其中的域给出了用户名、真实姓名、用户起始目录、加密口令和用户的其他信息。

3. /etc/fdprm

软盘参数表，用以说明不同的软盘格式。可用 setfdprm 进行设置。更多的信息见ｓｅｔｆｄｐｒｍ的帮助页。

4. /etc/fstab

指定启动时需要自动安装的文件系统列表。也包括用 swapon -a 启用的ｓｗａｐ区的信息。

5. /etc/group

类似/etc/passwd ，但说明的不是用户信息而是组的信息。包括组的各种数据。

6. /etc/inittab

init 的配置文件。

7. /etc/issue

包括用户在登录提示符前的输出信息。通常包括系统的一段短说明或欢迎信息。具体内容由系统管理员确定。

8. /etc/magic

“ｆｉｌｅ”的配置文件。包含不同文件格式的说明，“ｆｉｌｅ”基于它猜测文件类型。

9. /etc/motd

ｍｏｔｄ是 message of the day 的缩写，用户成功登录后自动输出。内容由系统管理员确定。常用于通告信息，如计划关机时间的警告等。

10. /etc/mtab

当前安装的文件系统列表。由脚本（ｓｃｒｉｔｐ）初始化，并由 mount 命令自动更新。当需要一个当前安装的文件系统的列表时使用(例如 df 命令)。

11. /etc/shadow

在安装了影子（ｓｈａｄｏｗ）口令软件的系统上的影子口令文件。影子口令文件将／ｅｔｃ／ｐａｓｓｗｄ文件中的加密口令移动到／ｅｔｃ／ｓｈａｄｏｗ中，而后者只对超级用户（ｒｏｏｔ）可读。这使破译口令更困难，以此增加系统的安全性。

12. /etc/login.defs

ｌｏｇｉｎ命令的配置文件。

13. /etc/printcap

类似/etc/termcap ，但针对打印机。语法不同。

14. /etc/profile 、/ e t c / c s h . l o g i n、/etc/csh.cshrc

登录或启动时 b o u r n e 或 c shells 执行的文件。这允许系统管理员为所有用户建立全局缺省环境。

15. /etc/securetty

确认安全终端，即哪个终端允许超级用户(r o o t)登录。一般只列出虚拟控制台，这样就不可能(至少很困难)通过调制解调器(m o d e m)或网络闯入系统并得到超级用户特权。

16. /etc/shells

列出可以使用的 s h e l l。chsh 命令允许用户在本文件指定范围内改变登录的 s h e l l。提供一台机器 f t p 服务的服务进程 ftpd 检查用户 s h e l l 是否列在/etc/shells 文件中，如果不是，将不允许该用户登录。

17. /etc/termcap

终端性能数据库。说明不同的终端用什么“转义序列”控制。写程序时不直接输出转义序列(这样只能工作于特定品牌的终端)，而是从/etc/termcap 中查找要做的工作的正确序列。

这样，多数的程序可以在多数终端上运行。

/dev 文件系统

/dev 目录包括所有设备的设备文件。设备文件用特定的约定命名，这在设备列表中说明。设备文件在安装时由系统产生，以后可以用/dev/makedev 描述。/ d e v / m a k e d e v.local 是系统管理员为本地设备文件(或连接)写的描述文稿(即如一些非标准设备驱动不是标准 makedev 的一部分)。下面简要介绍/ d e v 下一些常用文件。

1. /dev/console

系统控制台，也就是直接和系统连接的监视器。

2. /dev/hd

i d e 硬盘驱动程序接口。如： / d e v / h d a 指的是第一个硬盘， h a d 1 则是指/ d e v / h d a 的第一个分区。如系统中有其他的硬盘，则依次为/ d e v / h d b、/ d e v / h d c、. . . . . . ；如有多个分区则依次为 h d a 1、h d a 2 . . . . . .

3. /dev/sd

ｓｃｓｉ磁盘驱动程序接口。如有系统有ｓｃｓｉ硬盘，就不会访问/ｄｅｖ/ｈａｄ，而会访问/ｄｅｖ/ｓｄａ。

4. /dev/fd

软驱设备驱动程序。如：/ｄｅｖ/ｆｄ０指系统的第一个软盘，也就是通常所说的 a：盘，/ｄｅｖ/ｆｄ１指第二个软盘，．．．．．．而/ｄｅｖ/ｆｄ１ｈ１４４０则表示访问驱动器１中的４．５高密盘。

5. /dev/st

ｓｃｓｉ磁带驱动器驱动程序。

6. /dev/tty

提供虚拟控制台支持。如：/ｄｅｖ/ｔｔｙ１指的是系统的第一个虚拟控制台，/ｄｅｖ/ｔｔｙ２则是系统的第二个虚拟控制台。

7. /dev/pty

提供远程登陆伪终端支持。在进行ｔeｌｎｅｔ登录时就要用到/ｄｅｖ/ｐｔｙ设备。

8. /dev/ttys

计算机串行接口，对于ｄｏｓ来说就是“ｃｏｍ１”口。

9. /dev/cua

计算机串行接口，与调制解调器一起使用的设备。

10. /dev/null

“黑洞”，所有写入该设备的信息都将消失。例如：当想要将屏幕上的输出信息隐藏起来时，只要将输出信息输入到/ｄｅｖ/ｎｕｌｌ中即可。

/usr 文件系统

/usr 是个很重要的目录，通常这一文件系统很大，因为所有程序安装在这里。/usr 里的所有文件一般来自ｌｉｎｕｘ发行版(ｄｉｓｔｒｉｂｕｔｉｏｎ)；本地安装的程序和其他东西在/usr/local 下，因为这样可以在升级新版系统或新发行版时无须重新安装全部程序。/usr 目录下的许多内容是可选的，但这些

功能会使用户使用系统更加有效。/ｕｓｒ可容纳许多大型的软件包和它们的配置文件。下面列出一些重要的目录(一些不太重要的目录被省略了)。

1. /usr/x11r6

包含ｘwiｎｄｏｗ系统的所有可执行程序、配置文件和支持文件。为简化ｘ的开发和安装，ｘ的文件没有集成到系统中。ｘwiｎｄｏｗ系统是一个功能强大的图形环境，提供了大量的图形工具程序。用户如果对microsoft wiｎｄｏｗｓ或ｍａｃｈｉｎｔｏｓｈ比较熟悉的话，就不会对ｘwiｎｄｏｗ系统感到束手无策了。

2. /usr/x386

类似/ｕｓｒ/ｘ11r6，但是是专门给ｘ11 release 5的。

3. /usr/bin

集中了几乎所有用户命令，是系统的软件库。另有些命令在/bin 或/usr/local/bin 中。

4. /usr/sbin

包括了根文件系统不必要的系统管理命令，例如多数服务程序。

5. /usr/man、/ｕｓｒ/ｉｎｆｏ、/ｕｓｒ/ｄｏｃ

这些目录包含所有手册页、ｇｎｕ信息文档和各种其他文档文件。每个联机手册的“节”都有两个子目录。例如：/ｕｓｒ/ｍａｎ/ｍａｎ１中包含联机手册第一节的源码(没有格式化的原始文件)，/ｕｓｒ/ｍａｎ/ｃａｔ１包含第一节已格式化的内容。l联机手册分为以下九节：内部命令、系统调用、库函数、设备、文件格式、游戏、宏软件包、系统管理和核心程序。

6. /usr/include

包含了ｃ语言的头文件，这些文件多以.ｈ结尾，用来描述ｃ语言程序中用到的数据结构、子过程和常量。为了保持一致性，这实际上应该放在/usr/lib 下，但习惯上一直沿用了这个名

字。

7. /usr/lib

包含了程序或子系统的不变的数据文件，包括一些ｓｉｔｅ－ｗｉｄｅ配置文件。名字ｌｉｂ来源于库(library)；编程的原始库也存在/usr/lib 里。当编译程序时，程序便会和其中的库进行连接。也有许多程序把配置文件存入其中。

8. /usr/local

本地安装的软件和其他文件放在这里。这与/ｕｓｒ很相似。用户可能会在这发现一些比较大的软件包，如ｔｅｘ、ｅｍａｃｓ等。

/var 文件系统

/var 包含系统一般运行时要改变的数据。通常这些数据所在的目录的大小是要经常变化或扩充的。原来/ｖａｒ目录中有些内容是在/ｕｓｒ中的，但为了保持/ｕｓｒ目录的相对稳定，就把那些需要经常改变的目录放到/ｖａｒ中了。每个系统是特定的，即不通过网络与其他计算机共享。

下面列出一些重要的目录(一些不太重要的目录省略了)。

1. /var/catman

包括了格式化过的帮助(ｍａｎ)页。帮助页的源文件一般存在/ｕｓｒ/ｍａｎ/ｍａｎ中；有些ｍａｎ页可能有预格式化的版本，存在/ｕｓｒ/ｍａｎ/ｃａｔ中。而其他的ｍａｎ页在第一次看时都需要格式化，格式化完的版本存在/var/man 中，这样其他人再看相同的页时就无须等待格式化了。(/var/catman 经常被清除，就像清除临时目录一样。)

2. /var/lib

存放系统正常运行时要改变的文件。

3. /var/local

存放/usr/local 中安装的程序的可变数据(即系统管理员安装的程序)。注意，如果必要，即使本地安装的程序也会使用其他/var 目录，例如/var/lock 。

4. /var/lock

锁定文件。许多程序遵循在/var/lock 中产生一个锁定文件的约定，以用来支持他们正在使用某个特定的设备或文件。其他程序注意到这个锁定文件时，就不会再使用这个设备或文

件。

5. /var/log

各种程序的日志(ｌｏｇ)文件，尤其是 login (/var/log/wtmp log 纪录所有到系统的登录和注销) 和 syslog (/var/log/messages 纪录存储所有核心和系统程序信息)。/var/log 里的文件经常不确定地增长，应该定期清除。

6. /var/run

保存在下一次系统引导前有效的关于系统的信息文件。例如，/var/run/utmp 包含当前登录的用户的信息。

7. /var/spool

放置“假脱机(ｓｐｏｏｌ)”程序的目录，如ｍａｉｌ、ｎｅｗｓ、打印队列和其他队列工作的目录。每个不同的ｓｐｏｏｌ在/var/spool 下有自己的子目录，例如，用户的邮箱就存放在/var/spool/mail 中。

8. /var/tmp

比/tmp 允许更大的或需要存在较长时间的临时文件。注意系统管理员可能不允许/var/tmp 有很旧的文件。

/proc 文件系统

/proc 文件系统是一个伪的文件系统，就是说它是一个实际上不存在的目录，因而这是一个非常特殊的目录。它并不存在于某个磁盘上，而是由核心在内存中产生。这个目录用于提

供关于系统的信息。下面说明一些最重要的文件和目录(/proc 文件系统在 proc man 页中有更详细的说明)。

1. /proc/x

关于进程 x 的信息目录，这一 x 是这一进程的标识号。每个进程在/proc 下有一个名为自己进程号的目录。

2. /proc/cpuinfo

存放处理器(ｃｐｕ)的信息，如ｃｐｕ的类型、制造商、型号和性能等。

3. /proc/devices

当前运行的核心配置的设备驱动的列表。

4. /proc/dma

显示当前使用的ｄｍａ通道。

5. /proc/filesystems

核心配置的文件系统信息。

6. /proc/interrupts

显示被占用的中断信息和占用者的信息，以及被占用的数量。

7. /proc/ioports

当前使用的ｉ／ｏ端口。

8. /proc/kcore

系统物理内存映像。与物理内存大小完全一样，然而实际上没有占用这么多内存；它仅仅是在程序访问它时才被创建。(注意：除非你把它拷贝到什么地方，否则/proc 下没有任何

东西占用任何磁盘空间。)

9. /proc/kmsg

核心输出的消息。也会被送到ｓｙｓｌｏｇ。

10. /proc/ksyms

核心符号表。

11. /proc/loadavg

系统“平均负载”； 3 个没有意义的指示器指出系统当前的工作量。

12. /proc/meminfo

各种存储器使用信息，包括物理内存和交换分区(ｓｗａｐ)。

13. /proc/modules

存放当前加载了哪些核心模块信息。

14. /proc/net

网络协议状态信息。

15. /proc/self

存放到查看/proc 的程序的进程目录的符号连接。当 2 个进程查看/proc 时，这将会是不同的连接。这主要便于程序得到它自己的进程目录。

16. /proc/stat

系统的不同状态，例如，系统启动后页面发生错误的次数。

17. /proc/uptime

系统启动的时间长度。

18. /proc/version

核心版本。

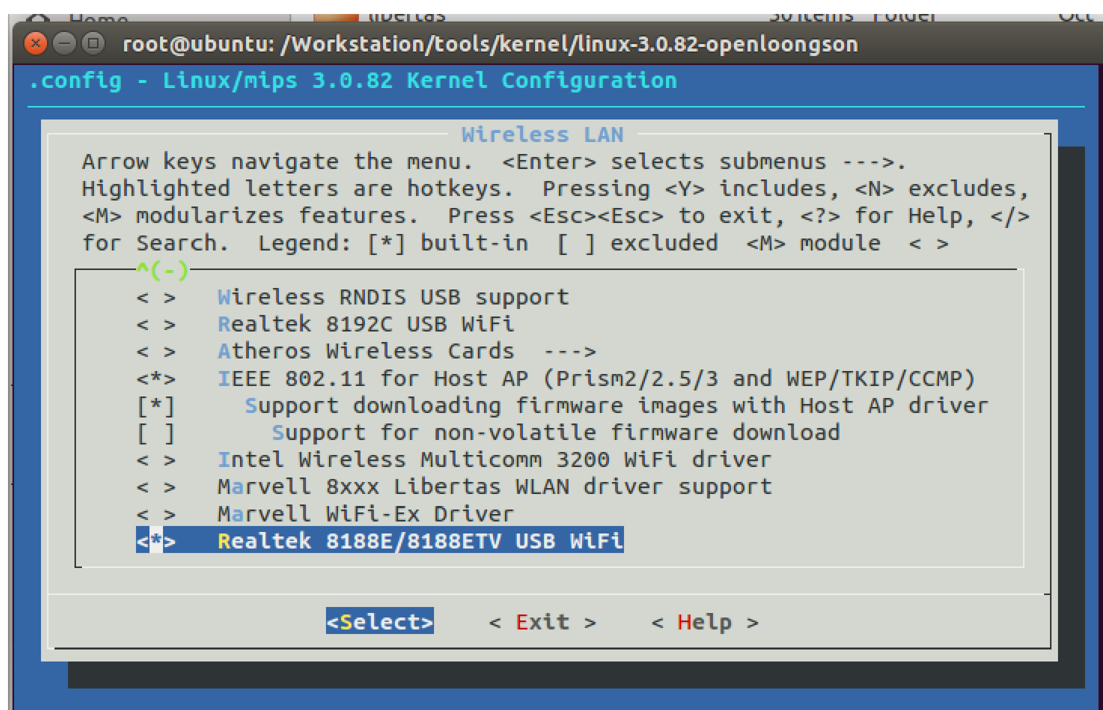
附录 9.无线网卡（RTL8188EU）驱动编译、使用 DHCP 配置无线网络

1 驱动编译进入内核

内核主界面下进入：

（1）WIFI 设备支持配置及打开 CONFIG_WIRELESS_EXT=y CONFIG_WEXT_PRIV=y

```
Device Drivers    --->
[*] Network device support    --->
[*]   Wireless LAN    --->
---   Wireless LAN
<*>      IEEE 802.11 for Host AP (Prism2/2.5/3 and WEP/TKIP/CCMP)
<*>      Realtck 8188E/8188ETV USB WiFi
```

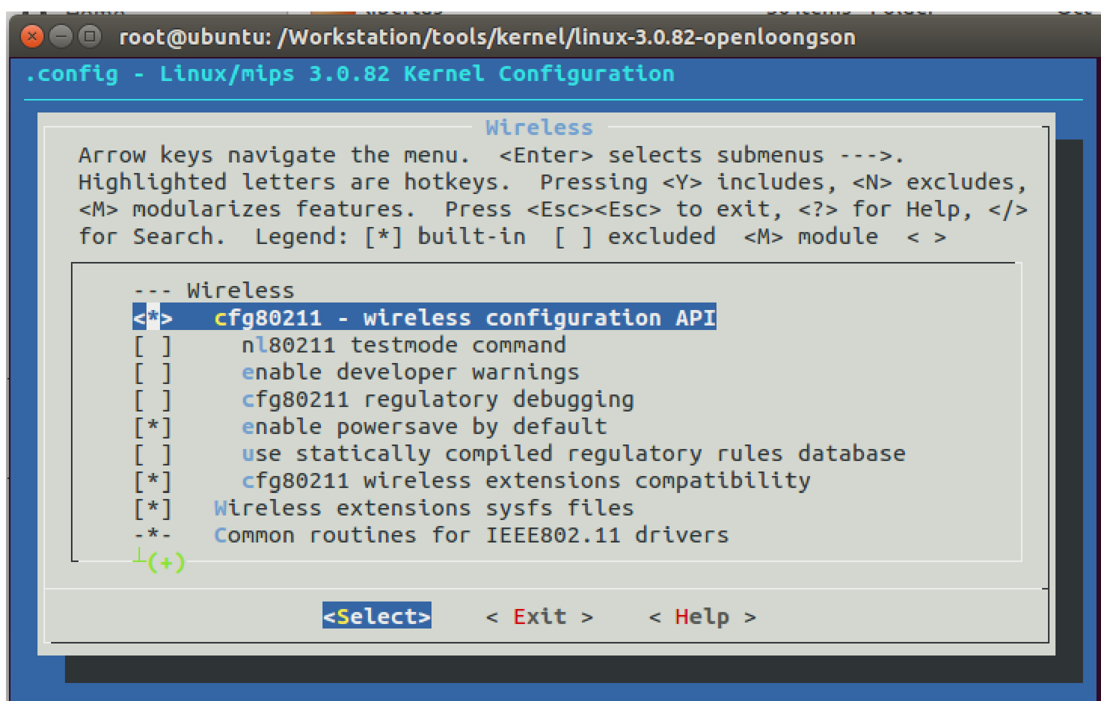


（2）Wifi 增加 802.11 协议栈的支持

[*]Networking support --->

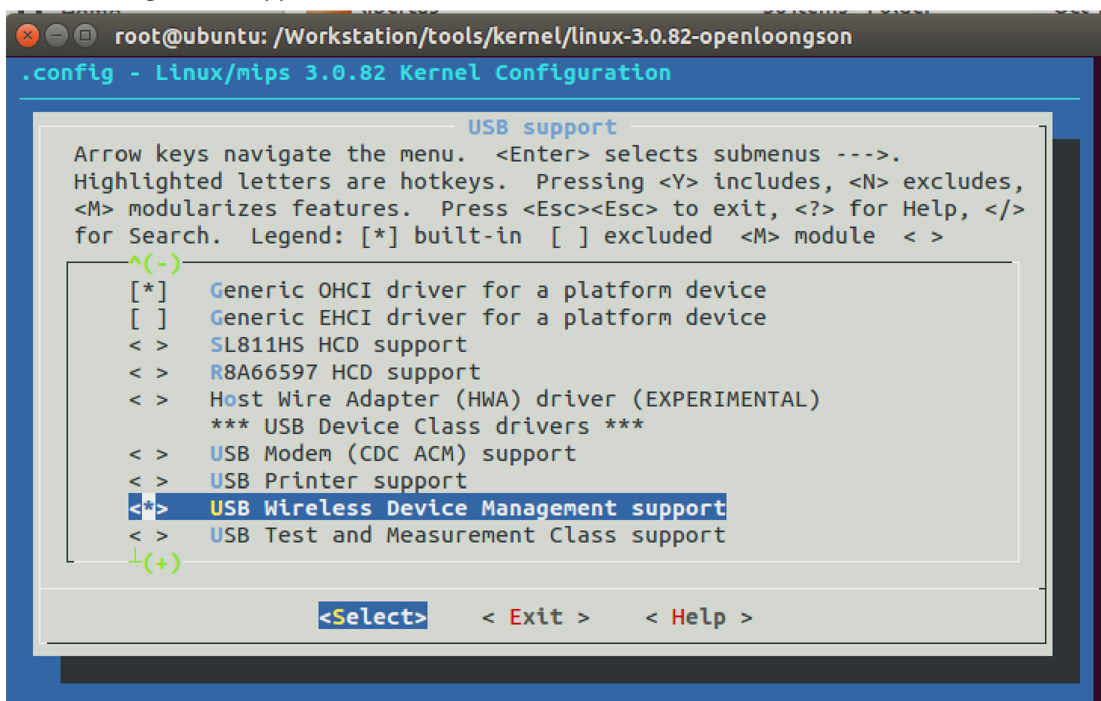
-*-   Wireless --->

选中 <*>   cfg80211 - wireless configuration API



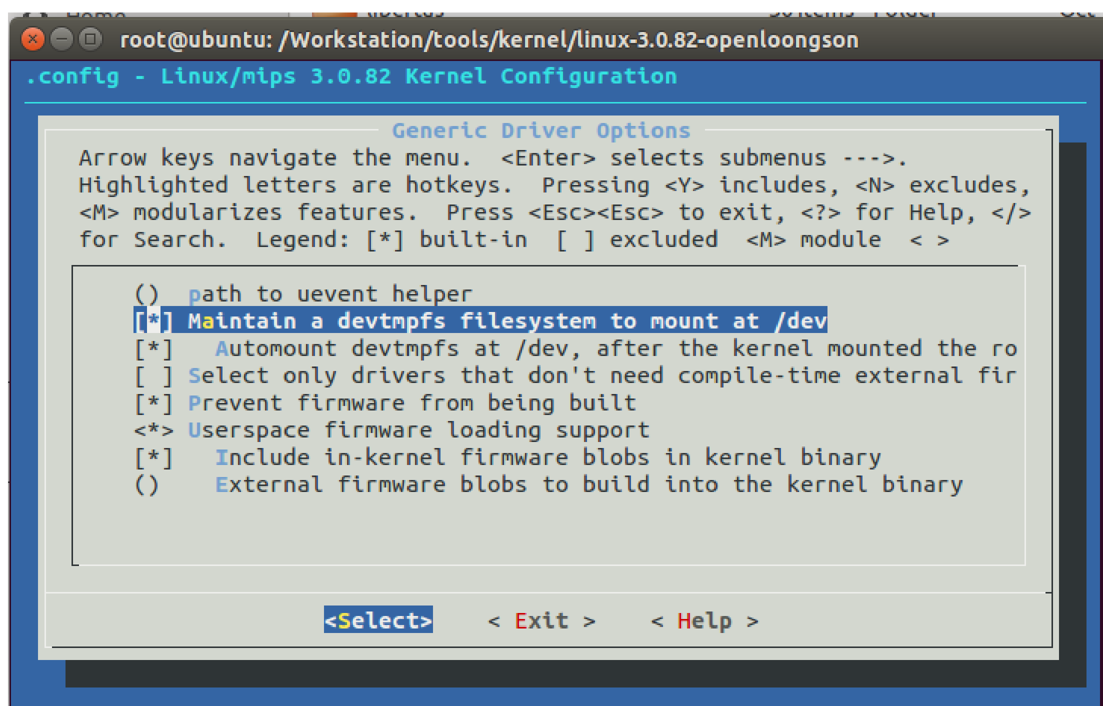
（3） USB 支持 WIFI 的配置

USB 支持 WIFI 的配置选项位于 Device Drivers >USB support 配置菜单下 USB Wireless DeviceManagement support。



（4） 用户空间的 mdev 和 firmware 支持配置

进入 Device Driver > Generic Driver Options 配置菜单，按照下图所示配置用户空间的 mdev 和 firmware 支持



内核 RAW socket 支持宏 CONFIG_PACKET=y

编译好的内核下载到开发板后，用命令 lsusb，查看无线网卡的 USB ID 号可看到：

```
[root@Loongson:/]#PHY: 0:13 - Link is Up - 100/Full
lsusb
Bus 001 Device 001: ID 1d6b:0002
Bus 002 Device 001: ID 1d6b:0001
Bus 003 Device 001: ID 1d6b:0002
Bus 001 Device 002: ID 0bda:8179
[root@Loongson:/]#
```

硬件 id 为 0bda:8179 的即为无线网卡。

2 下载 wifitools.tar.gz 使用

（1）下载并编译 wireless_tools

从网站 http://www.linuxfromscratch.org/blfs/view/svn/basicnet/wireless_tools.html

下载 wireless_tools.29.tar.gz

```
#tar xvfj wireless_tools.29.tar.bz2
#cd wireless_tools.29
```

Makefile 修改

```
CC= mipsel-linux-gcc
AR = mipsel-linux-ar
RANLIB = mipsel-linux-ranlib
```

运行 make

```
#make
```

复制文件到开发板 iwpriv,iwconfig、iwlist, iwevent, iwspy, iwgetid, ifrename,libiw.so.29。

解压，将 wireless-tools 文件夹内除了 libiw.so.29 外复制进/bin 文件夹内，将 libiw.so.29 复制进/lib 文件夹，将 wpa_supplicant 文件夹内所有文件也复制进/bin 内。

使用命令 chmod u+x ***修改以上文件权限。

出现以下信息，其中设备名为 wlan0 的网卡设备即是驱动识别出来的无线网卡，驱动程序安装成功。

```
[root@Loongson:/]# #iwconfig wlan0
```

```
RTL871X: rtw_wx_get_rts, rts_thresh=2347
RTL871X: rtw_wx_get_frag, frag_len=2346
wlan0         unassociated    Nickname:"<WIFI@REALTEK>"
              Mode:Auto    Frequency=2.412 GHz    Access Point: Not-Associated
              Sensitivity:0/0
              Retry:off    RTS thr:off    Fragment thr:off
              Encryption key:off
              Power Management:off
              Link Quality:0    Signal level:0    Noise level:0
              Rx invalid nwid:0    Rx invalid crypt:0    Rx invalid frag:0
              Tx excessive retries:0    Invalid misc:0    Missed beacon:0
```

查看 wifi 信号强度,速度,频段等信息用的 iwconfig iwlist 程序

3 建立连接

```
Wireless-tools（支持很多无线网卡，仅能访问 WEP 加密 AP）
```

启动无线网卡：ifconfig wlan0 up 192.168.1.123，配置无线网卡 IP 地址。

```
[root@Loongson:/]#ifconfig wlan0 up 192.168.1.123
RTL871X: +871x_drv - drv_open, bup=0
RTL871X: rtl8188e_FirmwareDownload: fw_ver=11 fw_subver=1 sig=0x88e1
RTL871X: rtl8188e_FirmwareDownload writeFW_retry:0, time after fwdl_start_time:56ms
RTL871X: _FWFreeToGo: Checksum report OK! REG_MCUFWDL:0x00030004
RTL871X: =====> _8051Reset88E(): 8051 reset success .
RTL871X: _FWFreeToGo: Polling FW ready success!! REG_MCUFWDL:0x000300c6
==> rtl8188e_iol_efuse_patch
RTL871X: pDM_Odm TxPowerTrackControl = 1
RTL871X: rtl8188eu_hal_init in 1048ms
RTL871X: MAC Address = 00:0b:81:96:76:24
RTL871X: -871x_drv - drv_open, bup=13. 扫描 AP（假设为 MYESSID）：iwlist wlan0 scanning
```

扫描无线网络：iwlist wlan0 scanning

```
[root@Loongson:/]#iwlist wlan0 scanning
RTL871X: survey done event(2d)
RTL871X: rtw_wx_get_scan: Njtech RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan: sundm
RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan: Njtech RTL871X: rtw_wx_get_scan: ssid = 0
RTL871X: rtw_wx_get_scan: CMCC-EDU RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan:
Njtech RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan: newstart RTL871X: rtw_wx_get_scan:
ssid = 0 RTL871X: rtw_wx_get_scan: 602 RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan:
CMCC-EDU RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan: FTTBEST RTL871X:
rtw_wx_get_scan: ssid = 0 RTL871X: rtw_wx_get_scan: Njtech RTL871X: rtw_wx_get_scan: ssid = 0 RTL871X:
rtw_wx_get_scan: Njtech RTL871X: rtw_wx_get_scan: ssid = 0 wlan0        Scan completed :
              Cell 01 - Address: 0A:69:6C:2F:AC:C1
                        ESSID:"Njtech"
                        Protocol:IEEE 802.11bgn
                        Mode:Master
                        Frequency:2.412 GHz (Channel 1)
                        Encryption key:off
                        Bit Rates:144 Mb/s
                        Quality=0/100    Signal level=44/100
                        Extra:fm=0001
              Cell 02 - Address: E0:06:E6:C8:6C:9D
                        ESSID:"sundm"
                        Protocol:IEEE 802.11bgn
                        Mode:Master
                        Frequency:2.437 GHz (Channel 6)
                        Encryption key:on
                        Bit Rates:72 Mb/s
                        Extra:rsn_ie=30140100000fac040100000fac040100000fac020000
                        IE: IEEE 802.11i/WPA2 Version 1
                            Group Cipher : CCMP
                            Pairwise Ciphers (1) : CCMP
                            Authentication Suites (1) : PSK
                        Quality=0/100    Signal level=84/100
                        Extra:fm=0003
```

至此，已经检测出能够成功加载驱动。

4 使用 wpa_supplicant 连接无线网络

修改 wpa_supplicant 文件夹中的 wpa_supplicant.conf 内容为：

```
network={
ssid=”your wireless”
Psk=”your password”
}
network={
        ssid="*******" # your wireless
        psk="*****" # your password
        priority=1    #设置优先级
        key_mgmt=WPA-PSK #加密类型
}
```

通过 wpa_supplicant 的配置文件来实现连接。

将文件 wpa_supplicant.conf 传入开发板的 bin 文件夹：

在开发板上运行以下命令：

```
ifconfig eth0 down    //关闭有线网络
ifconfig wlan0 up    192.168.1.123 //打开无线网络 配置无线网络 IP 地址
wpa_supplicant -d -Dwext -iwlan0 -c/bin/wpa_supplicant.conf & //后台运行 wpa_supplicant 程序，以支持 WEP，WPA/WPA2 和 WAPI 无线协议和加密认证
```

这里的& 表示在后台运行。-d 增加调试信息输出。-i 所需要配置的网口名称。-Dwext 指使用的驱动。 -c 指定配置文件。

wpa 可执行文件和配置文件命令格式：

```
wpa_supplicant [-BddhKLqqstuvW] [-P<pid file>] [-g<global ctrl>] \
        -i<ifname> -c<config file> [-C<ctrl>] [-D<driver>] [-p<driver_param>] \
        [-b<br_ifname>] [-f<debug file>] \
        [-o<override driver>] [-O<override ctrl>] \
        [-N -i<ifname> -c<conf> [-C<ctrl>] [-D<driver>] \
        [-p<driver_param>] [-b<br_ifname>] ...]
驱动（一般常使用的就是 wext）：
  wext = Linux wireless extensions (generic)
  hostap = Host AP driver (Intersil Prism2/2.5/3)
  atmel = ATMEL AT76C5XXx (USB, PCMCIA)
  wired = Wired Ethernet driver
各个选项及其含义:
  -b = optional bridge interface name
      增加网桥名称
  -B = run daemon in the background
      后台执行
  -c = Configuration file
      附加配置文件，即根据配置文件执行操作
  -C = ctrl_interface parameter (only used if -c is not)
      控制网口参数
  -i = interface name
      网口名称
  -d = increase debugging verbosity (-dd even more)
      增加调试信息输出
  -D = driver name (can be multiple drivers: nl80211,wext)
      驱动名称
  -g = global ctrl_interface
      全局网口配置
  -K = include keys (passwords, etc.) in debug output
      在 debug 输出中包含 keys
  -t = include timestamp in debug messages
      将各个 debug 信息前输出时间标签
  -h = show this help text
      显示帮助信息
```

```
  -L = show license (GPL and BSD)
       显示 license
  -o = override driver parameter for new interfaces
       覆盖 driver 参数
  -O = override ctrl_interface parameter for new interfaces
       覆盖 ctrl_interface 参数
  -p = driver parameters
       携带驱动参数
  -P = PID file
       进程文件
  -q = decrease debugging verbosity (-qq even less)
        在 debug 中不输出指定参数
  -v = show version
       显示版本信息
  -W = wait for a control interface monitor before starting
       启动前等待控制接口
  -N = start describing new interface
       启动对新接口的描述
4
命令举例：
wpa_supplicant -d -Dwext -i wlan0 -c /data/misc/wifi/wpa_supplicant.conf -t -d /data/misc/wifi/log.txt
使用 linux 通用驱动，网络接口为 wlan0，读取文件为/data/misc/wifi/wpa_supplicant.conf， 输出带时间戳的调试信息到/data/misc/wifi/log.txt 中，执行无线配置
```

运行命令后，产生错误：

```
ioctl[SIOCSIWAP]: Operation not permitted
WEXT: Failed to set bogus BSSID/SSID to disconnect
```

分析后，问题在于网友提供的 wpa_supplicant 有问题，于是重新编译 wpa_supplicant。

5 安装 wpa_supplicant

以下在虚拟机上编译。

（1） 首先安装支持库 libnl

在 http://www.openssl.org/source/下载最新源码 libnl-1.1.tar.gz

解压后，配置安装信息：

执行./configure -prefix=/usr/local/arm/libnl1.1，配置 libnl 安装路径

执行 make CC= mipsel-linux-gcc，完成编译，

执行 make install，将 libnl 库安装至/usr/local/arm/libnl1.1 路径下,生成二个目录 include lib，在 lib 中得到 libnl 库：libnl.so 等文件。

将/usr/local/arm/libnl1.1/lib 下所有文件拷贝至开发板的/lib 目录下，确保 hostapd 在开发板上运行，能够正确找到 libnl 库的位置。

（2）安装支持库 libopenssl

在 http://www.openssl.org/source/下载最新源码 openssl-1.0.1c.tar.gz

解压后，

配置安装信息：

```
./config no-asm shared --prefix=/usr/local/arm/openssl/openssl-install --cross-compile-prefix=/opt/gcc-4.3-ls232/bin/mipsel-linux-
```

no-asm 意思是关于汇编的模块部进行编译，因为部分汇编会报错

shared 意思是编译成动态链接库

--prefix=/usr/local/arm/openssl/openssl-install 意思是 指定 make install 的安装路径

--cross-compile-prefix=/opt/gcc-4.3-ls232/bin/mipsel-linux-指定交叉编译工具链路径

然后运行

```
make
make install
```

在/usr/local/arm/openssl/openssl-install 下生成四个目录 bin include lib ssl，在 lib 中得到 libopenssl 库：libcrypto.so 等文件。

（3）编译 wpa_supplicant

在 http://wireless.kernel.org/en/users/Documentation/wpa_supplicant 下载最新源码 wpa_supplicant-2.2.tar.gz

解压后，拷备配置文件

```
cp defconfig .config
```

修改.config 加上一行: CONFIG_LIBNL32=y

目的是：Use libnl 3.2 libraries (if this is selected, CONFIG_LIBNL20 is ignored)

并添加：

```
CFLAGS += -I/usr/local/arm/openssl/openssl-install/include/
CFLAGS += -I/usr/local/arm/libnl1.1/include/
LDFLAGS +=    -I/usr/local/arm/openssl/openssl-install/include/
LDFLAGS +=    -I/usr/local/arm/libnl1.1/include/
LIBS += -L/usr/local/arm/openssl/openssl-install/lib
LIBS += -L/usr/local/arm/libnl1.1/lib
```

修改 Makefile:

```
CC = mipsel-linux-gcc
```

Make 后出现错误：cannot find -lcryptocollect2: ld returned 1 exit statusMakefile:1622: recipe for target 'wpa_passphrase' failed ， 提示找不到 lib。

修改 Makefile:

```
CC = mipsel-linux-gcc -L/usr/local/arm/openssl/openssl-install/lib
```

-L 参数跟着的是库文件所在的目录名，把所有目标文件链接成可执行文件。

库文件中，分为两大类分别是动态链接库（通常以.so 结尾）和静态链接库（通常以.a 结尾），二者的区别仅在于程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。

静态库链接时搜索路径顺序：

1. ld 会去找 GCC 命令中的参数-L
2. 再找 gcc 的环境变量 LIBRARY_PATH
3. 再找内定目录 /lib /usr/lib /usr/local/lib 这是当初 compile gcc 时写在程序内的

动态链接时、执行时搜索路径顺序:

1. 编译目标代码时指定的动态库搜索路径
2. 环境变量 LD_LIBRARY_PATH 指定的动态库搜索路径
3. 配置文件/etc/ld.so.conf 中指定的动态库搜索路径
4. 默认的动态库搜索路径/lib
5. 默认的动态库搜索路径/usr/lib

有关环境变量：

LIBRARY_PATH 环境变量：指定程序静态链接库文件搜索路径

LD_LIBRARY_PATH 环境变量：指定程序动态链接库文件搜索路径

6 继续使用 wpa_supplicant 连接无线网络

编译结束后，在当前目录下生成三个文件：wpa_supplicant 、wpa_passphrase、wpa_cli。拷备到开发板，运行 wpa_supplicant。将这三个文件传入开发板/usr/bin 下，运行 wpa_supplicant，启动 wpa_supplicant 守护进程。以下显示调试信息。

```
[root@Loongson:/]#wpa_supplicant -d -Dwext -iwlan0 -c/bin/wpa_supplicant.conf &
[root@Loongson:/]#wpa_supplicant v2.2
random: Trying to read entropy from /dev/random
Successfully initialized wpa_supplicant
```

```
Initializing interface 'wlan0' conf '/bin/wpa_supplicant.conf' driver 'wext' ctr
l_interface 'N/A' bridge 'N/A'
Configuration file '/bin/wpa_supplicant.conf' -> '/bin/wpa_supplicant.conf'
Reading configuration file '/bin/wpa_supplicant.conf'
eapol_version=1
ap_scan=1
fast_reauth=1
Priority group 1
    id=0 ssid='sundm75_502'
RTL871X: [rtw_wx_set_pmkid] IW_PMKSA_FLUSH!
rfkill: Cannot oRTL871X: set_mode = IW_MODE_INFRA
pen RFKILL control device
WEXT:RTL871X: hw_var_set_opmode()-3378 mode = 2
 RFKILL status nRTL871X: set bssid:00:00:00:00:00:00
ot available
SIOCGIWRANGE: WE(compiled)=22 WE(sRTL871X: =>rtw_wx_set_essid
ource)=16 enc_caRTL871X: ssid=g▒isQ▒J▒)˛▒▒▒F|▒T▒vZ.c3▒↛, len=32
pa=0xf
   capabiRTL871X: set ssid [g▒isQ▒J▒)˛▒▒▒F|▒T▒vZ.c3▒↛] fw_state=0x00000008
lities: key_mgmtRTL871X: Set SSID under fw_state=0x00000008
 0xf enc 0x1f flRTL871X: <=rtw_wx_set_essid, ret 0
ags 0x0
ioctl[SIOCSIWAP]: Operation not permitted
WEXT: Failed to clear BSSID selection on disconnect
netlink: Operstate: ifindex=3 linkmode=1 (userspRTL871X: [rtw_wx_set_pmkid] IW_P
MKSA_FLUSH!
ace-control), operstate=5 (IF_OPER_DORMANT)
Add interface wlan0 to a new radio N/A
wlan0: Own MAC address: 00:0b:81:96:76:24
wpa_driver_wext_set_key: alg=0 key_idx=0 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=1 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=2 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=3 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_countermeasures
wlan0: RSN: flushing PMKID list in the driver
wlan0: Setting scan request: 0.100000 sec
EAPOL: SUPP_PAE entering state DISCONNECTED
EAPOL: Supplicant port status: Unauthorized
EAPOL: KEY_RX entering state NO_KEY_RECEIVE
EAPOL: SUPP_BE entering state INITIALIZE
EAP: EAP entering state DISABLED
wlan0: Added interface wlan0
wlan0: State: DISCONNECTED -> DISCONNECTED
wpa_driver_wext_set_operstate: operstate 0->0 (DORMANT)
netlink: Operstate: ifindex=3 linkmode=-1 (no change), operstate=5 (IF_OPER_DORM
ANT)
wlan0: State: DISCONNECTED -> SCANNING
wlan0: Starting AP scan for wildcard SSID
wlan0: Add radio work 'scan'@0x58eb50
wlan0: First radio work item in the queue - schedule start immediately
random: Got 17/20 bytes from /dev/random
RTM_NEWLINK: operstate=0 ifi_flags=0x1043 ([UP][RUNNING])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b06 len=8
RTM_NEWLINK: operstate=0 ifi_flags=0x1043 ([UP][RUNNING])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b1a len=40
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
wlan0: Starting radio work 'scan'@0x58eb50 after 0.010730 second wait
Scan requested (ret=0) - scan timeout 10 seconds
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
```

```
Wireless event: cmd=0x8b19 len=8
wlan0: Event SCAN_RESULTS (3) received
ioctl[SIOCGIWSCAN]: Resource temporarily unavailable
wlan0: Failed to get scan results
wlan0: Failed to get scan results - try scanning again
wlan0: Setting scan request: 1.000000 sec
wlan0: Radio work 'scan'@0x58eb50 done in 0.005621 seconds
EAPOL: disable timer tick
wlan0: Starting AP scan for wildcard SSID
wlan0: Add radio work 'scan'@0x58eb50
wlan0: First radio work item in the queue - schedule start immediately
wlan0: Starting radio work 'scan'@0x58eb50 after 0.000267 second wait
Scan requested (ret=0) - scan timeout 30 seconds
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b19 len=8
wlan0: Event SCAN_RESULTS (3) received
ioctl[SIOCGIWSCAN]: Resource temporarily unavailable
wlan0: Failed to get scan results
wlan0: Failed to get scan results - try scanning again
wlan0: Setting scan request: 1.000000 sec
wlan0: Radio work 'scan'@0x58eb50 done in 0.003203 seconds
RTL871X: survey done event(f)
RTL871X: rtw_select_and_join_from_scanned_queue: return _FAIL(candidate == NULL)
RTL871X: try_to_join, but select scanning queue fail, to_roaming:0
RTL871X: indicate disassoc
RTL871X: rtl8188e_set_FwJoinBssReport_cmd mstatus(0)
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])RTL871X: rtw_wx_get_scan: sundm7
5_502
RTM_NEWLINK, IRTL871X: rtw_wx_get_scan: ssid = 0 FLA_IFNAME: InteRTL871X: rtw_wx
_get_scan: rface 'wlan0' adRTL871X: rtw_wx_get_scan: ssid = 0 ded
WEXT: if_reRTL871X: rtw_wx_get_scan: FAST_C930BC moved already clRTL871X: rtw_wx
_get_scan: ssid = 0 eared - ignore eRTL871X: rtw_wx_get_scan: 17-1-801 vent
Wireless eRTL871X: rtw_wx_get_scan: ssid = 0 vent: cmd=0x8b15 len=20
Wireless event: new AP: 00:00:00:00:00:00
wlan0: Event DISASSOC (1) received
wlan0: DRTL871X: wpa_set_auth_algs, AUTH_ALG_OPEN_SYSTEM
isassociation noRTL871X: set_mode = IW_MODE_INFRA
tification
wlanRTL871X:
 wpa_ie(length:22):
0: Auto connect RTL871X: 0x30 0x14 0x01 0x00 0x00 0x0f 0xac 0x04
enabled: try to RTL871X: 0x01 0x00 0x00 0x0f 0xac 0x04 0x01 0x00
reconnect (wps=0RTL871X: 0x00 0x0f 0xac 0x02 0x00 0x00 0x00 0x00
 wpa_state=3)
wRTL871X: hw_var_set_opmode()-3378 mode = 2
lan0: Do not reqRTL871X: SetHwReg8188EU, 4091, RCR= 700060ca
uest new immediaRTL871X: rtw_wx_set_auth: IW_AUTH_KEY_MGMT case
te scan
wlan0: RTL871X: rtw_wx_set_auth: IW_AUTH_KEY_MGMT bwapipsk 0
Disconnect eventRTL871X: =>rtw_wx_set_essid
 - remove keys
RTL871X: ssid=sundm75_502, len=11
wlan0: State: SCRTL871X: set ssid [sundm75_502] fw_state=0x00000008
ANNING -> DISCONRTL871X: Set SSID under fw_state=0x00000008
NECTED
wpa_drivRTL871X: [by_bssid:0][assoc_ssid:sundm75_502][to_roaming:0] new candidat
e: sundm75_502(08:57:00:7c:fd:90) rssi:-58
er_wext_set_operRTL871X: rtw_select_and_join_from_scanned_queue: candidate: sund
m75_502(08:57:00:7c:fd:90, ch:1)
state: operstateRTL871X: link to Artheros AP
RTL871X: rtw_joinbss_cmd: smart_ps=2

netlink: OperstRTL871X: <=rtw_wx_set_essid, ret 0
ate: ifindex=3 lRTL871X: set bssid:08:57:00:7c:fd:90
inkmode=-1 (no cRTL871X: Set BSSID under fw_state=0x00000088
```

```
hange), operstatRTL871X: set ch/bw before connected
e=5 (IF_OPER_DORRTL871X: start_join_set_ch_bw: ch=1, bwmode=1, ch_offset=1
MANT)
EAPOL: External notification - portEnabled=0
EAPOL: External notification - portValid=0
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b1RTL871X: update_mgnt_tx_rate(): rate = 2
9 len=8
RTL871X: ### rtl8188e_set_FwMediaStatus_cmd: MStatus=1 MACID=0

Received 1234 bytes of scan results (4 BSSes)
wlan0: BSS: Start scan result update 1
wlan0: BSS: Add new id 0 BSSID 08:57:00:7c:fd:90 SSID 'sundm75_502'
wlan0: BSS: Add new id 1 BSSID e4:d3:32:c9:30:bc SSID 'FAST_C930BC'
wlan0: BSS: Add new id 2 BSSID c0:61:18:c8:08:7c SSID ''
wlan0: BSS: Add new id 3 BSSID 24:69:68:16:4f:1b SSID '17-1-801'
BSS: last_scan_res_used=4/32
wlan0: New scan results available (own=0 ext=0)
wlan0: Selecting BSS from priority group 1
wlan0:    0:    08:57:00:7c:fd:90    ssid='sundm75_502'    wpa_ie_len=22    rsn_ie_len=20    caps=
0x11 level=84
wlan0:    selected based on RSN IE
wlan0:    selected BSS 08:57:00:7c:fd:90 ssid='sundm75_502'
wlan0:    Considering    connect    request:    reassociate:    0    selected:    08:57:00:7c:fd:90
bssid:    00:00:00:00:00:00    pending:    00:00:00:00:00:00    wpa_state:    DISCONNECTED
ssid=0x57fd40    current_ssid=(nil)
wlan0: Request association with 08:57:00:7c:fd:90
wlan0: Add radio work 'connect'@0x58e400
wlan0: First radio work item in the queRTL871X: link to Artheros AP
ue - schedule stRTL871X: start auth
RTL871X: issue_auth

wlan0: Starting radio work 'conRTL871X: OnAuthClient
nect'@0x58e400 aRTL871X: auth success, start assoc
fter 0.000336 seRTL871X: network.SupportedRates[0]=82
cond wait
wlan0RTL871X: network.SupportedRates[1]=84
: Trying to assoRTL871X: network.SupportedRates[2]=8B
ciate with 08:57RTL871X: network.SupportedRates[3]=96
:00:7c:fd:90 (SSRTL871X: network.SupportedRates[4]=0C
ID='sundm75_502'RTL871X: network.SupportedRates[5]=12
RTL871X: network.SupportedRates[6]=18

wlan0: CancelliRTL871X: network.SupportedRates[7]=24
RTL871X: network.SupportedRates[8]=30

wlan0: WPA: cleRTL871X: network.SupportedRates[9]=48
aring own WPA/RSRTL871X: network.SupportedRates[10]=60
N IE
wlan0: AutRTL871X: network.SupportedRates[11]=6C
omatic auth_alg RTL871X: bssrate_len = 12
selection: 0x1
RSN: PMKSA cache search - network_ctx=0x57fd40 tRTL871X: OnAssocRsp
ry_opportunisticRTL871X: report_join_res(4)
=0
RSN: Search RTL871X: rtw_joinbss_update_network
for BSSID 08:57:RTL871X: +rtw_update_ht_cap()
00:7c:fd:90
RSNRTL871X: rtw_joinbss_update_stainfo
: No PMKSA cacheRTL871X: ### Set STA_(0) info
  entry found
wlRTL871X: assoc success
an0: RSN: using IEEE 802.11i/D9.0
wlan0: WPA: SRTL871X: HW_VAR_BASIC_RATE: BrateCfg(0x15f)
elected cipher suites: group 16 pairwise 16 key_mgmt 2 proto 2
```

```
WPA: set AP WPA IE - hexdump(len=24): dd 16 00 5RTL871X: WMM(0): 0, a42b
0 f2 01 01 00 00 50 f2 04 01 00 00 50 f2 04 01 0RTL871X: WMM(1): 0, a44f
0 00 50 f2 02
WRTL871X: WMM(2): 0, 5e4322
PA: set AP RSN IRTL871X: WMM(3): 0, 2f3222
E - hexdump(len=RTL871X: wmm_para_seq(0): 0
22): 30 14 01 00RTL871X: wmm_para_seq(1): 1
 00 0f ac 04 01 RTL871X: wmm_para_seq(2): 2
00 00 0f ac 04 0RTL871X: wmm_para_seq(3): 3
1 00 00 0f ac 02RTL871X: HTOnAssocRsp
 00 00
wlan0: WPA: using GTK CCMP
wlan0: WPA: using PTK CCMP
wlan0: WPA: using KEY_MGMT WPA-PSK
WPA: Set own WPA IE default - hexdump(len=22): 30 14 01 00 00 0f ac 04 01 00RTL8
71X: send eapol packet
 00 0f ac 04 01 00 00 0f ac 02 00 00
wlan0: State: DISCONNECTED -> ASSOCIATING
wpa_driver_wext_set_operstate: operstate 0->0 (DORMANT)
netlink: Operstate: ifindex=3 linkmode=-1 (no change), operstate=5 (IF_OPER_DORM
ANT)
UpdateHalRAMask8188EUsb => mac_id:0, networkType:0x0b, mask:0x000fffff
        ==> rssi_level:0, rate_bitmap:0x000ff015
Limit connectionRTL871X: ### MacID(1),Set Max Tx RPT MID(2)
 to BSSID 08:57:RTL871X: ### rtl8188e_set_FwMediaStatus_cmd: MStatus=1 MACID=0
00:7c:fd:90 freq=2412 MHz based on scan results (bssid_set=0)
wpa_driver_wext_aRTL871X: rtl8188e_set_FwJoinBssReport_cmd mstatus(1)
ssociate
wpa_driver_wext_set_drop_unencrypted
RTL871X: SetFwRsvdPagePkt
wpa_driver_wext_RTL871X: SetFwRsvdPagePkt: Set RSVD page location to Fw
set_psk
wlan0: Setting authentication timeout: RTL871X:
rtl8188e_set_FwJoinBssReport_cmd
                : 1 Download RSVD success! DLBcnCount:1, poll:1
10 sec 0 usec
EAPOL: External nRTL871X: Set RSVD page location to Fw.
otification - EARTL871X: =>mlmeext_joinbss_event_callback
P success=0
EAPOL: External notification - EAP fail=0
EAPOL: External notification - portControl=Auto
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b06 len=8
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b04 len=12
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b1a len=19
RTM_NEWLINK: operstate=0 ifi_flags=0x1003 ([UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
Wireless event: cmd=0x8b15 len=20
Wireless event: new AP: 08:57:00:7c:fd:90
wlan0: Event ASSOC (0) received
wlan0: State: ASSOCIATING -> ASSOCIATED
wpa_driver_wext_set_operstate: operstate 0->0 (DORMANT)
netlink: Operstate: ifindex=3 linkmode=-1 (no change), operstate=5 (IF_OPER_DORM
ANT)
wlan0: Associated to a new BSS: BSSID=08:57:00:7c:fd:90
wlan0: Associated with 08:57:00:7c:fd:90
wlan0: WPA: Association event - clear replay counter
wlan0: WPA: Clear old PTK
```

```
EAPOL: External notification - portEnabled=0
EAPOL: External notification - portValid=0
EAPOL: External notification - EAP success=0
EAPOL: External notification - portEnabled=1
EAPOL: SUPP_PAE entering state CONNECTING
EAPOL: enable timer tick
EAPOL: SUPP_BE entering state IDLE
wlan0: Setting authentication timeout: 10 sec 0 usec
wlan0: Cancelling scan request
RTM_NEWLINK: operstate=0 ifi_flags=0x11003 ([UP][LOWER_UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
wlan0: RX EAPOL from 08:57:00:7c:fd:90
wlan0: Setting authentication timeout: 10 sec 0 usec
wlan0: IEEE 802.1X RX: version=2 type=3 length=95
wlan0:     EAPOL-Key type=2
wlan0:     key_info 0x8a (ver=2 keyidx=0 rsvd=0 Pairwise Ack)
wlan0:     key_length=16 key_data_length=0
  replay_counter - hexdump(len=8): 00 00 00 00 00 00 00 01
  key_nonce - hexdump(len=32): 64 af b0 19 44 75 c6 ce d8 1d 04 0b 7f ab aa 9f 3
c 11 3f 7c 2e cf 84 1c 45 63 88 44 08 ee 07 83
  key_iv - hexdump(len=16): 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  key_rsc - hexdump(len=8): 00 00 00 00 00 00 00 00
  key_id (reserved) - hexdump(len=8): 00 00 00 00 00 00 00 00
  key_mic - hexdump(len=16): 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
wlan0: State: ASSOCIATED -> 4WAY_HANDSHAKE
wlan0: WPA: RX message 1 of 4-Way Handshake from 08:57:00:7c:fd:90 (ver=2)
RSN: msg 1/4 key data - hexdump(len=0):
WPA: Renewed SNonce - hexdump(len=32): c8 d8 b1 45 d8 7d 5e 6d 82 5b 04 f1 c6 82 84 4c 3a 14 15 42 94 08
b8 0a c1 a3 c6 ca be 32 56 11
WPA: PTK derivation - A1=00:0b:81:96:76:24 A2=08:57:00:7c:fd:90
WPA: Nonce1 - hexdump(len=32): c8 d8 b1 45 d8 7d 5e 6d 82 5b 04 f1 c6 82 84 4c 3a 14 15 42 94 08 b8 0a c1 a3
c6 ca be 32 56 11
WPA: Nonce2 - hexdump(len=32): 64 af b0 19 44 75 c6 ce d8 1d 04 0b 7f ab aa 9f 3c 11 3f 7c 2e cf 84 1c 45 63
88 44 08 ee 07 83
WPA: PMK - hexdump(len=32): [REMOVED]
WPA: PTK - hexdump(len=48): [REMOVED]
WPA: WPA IE for msg 2/4 - hexdump(len=22): 30 14 01 00 00 0f ac 04 01 00 00 0f ac 04 01 00 00 0f ac 02 00 00
WPA: Replay Counter - hexdump(len=8): 00 00 00 00 00 00 00 01
wlan0: WPA: Sending EAPOL-Key 2/4
WPA: KCK - hexdump(len=16): [REMOVED]
WPA: Derived Key MIC - hexdump(len=16): d4 2e 7b 4d 3b ef 8d 13 f3 cd f8 bd 0e c5 ef 01
wlan0: RX EAPOL from 08:57:00:7c:fd:90
wlan0: IEEE 802.1X RX: version=2 type=3 length=175
wlan0:     EAPOL-Key type=2
wlan0:     key_info 0x13ca (ver=2 keyidx=0 rsvd=0 Pairwise Install Ack MIC Secure Encr)
wlan0:     key_length=16 key_data_length=80
  replay_counter - hexdump(len=8): 00 00 00 00 00 00 00 02
  key_nonceRTL871X: send eapol packet
 - hexdump(len=32): 64 af b0 19 RTL871X:   ~~~~set sta key:unicastkey
44 75 c6 ce d8 1RTL871X: set pairwise key to hw: alg:4(WEP40-1 WEP104-5 TKIP-2 AES-4) camid:4
d 04 0b 7f ab aa[CCMP]   GTK key_len=16 @@@@@@@@@@@@@@@@@@
 9f 3c 11 3f 7c RTL871X:   ~~~~set sta key:groupkey
2e cf 84 1c 45 6RTL871X: ==> rtw_set_key algorithm(4),keyid(2),key_mask(0)
3 88 44 08 ee 07 83
  key_iv - hexdump(len=16): 00 00 00 00 00 RTL871X: set group key to hw: alg:4(WEP40-1 WEP104-5 TKIP-2
AES-4) keyid:2
00 00 00 00 00 00 00 00 00 00 00
  key_rsc - hexdump(len=8): d3 0e 00 00 00 00 00 00
  key_id RTL871X: SetHwReg8188EU, 4087, RCR= 700060ce
(reserved) - hexdump(len=8): 00 00 00 00 00 00 00 00
  key_mic - hexdump(len=16): 92 a6 ed de 4d 4b b9 b7 b8 72 4e 5a f3 65 0e 19
RSN: encrypted key data - hexdump(len=80): d9 d6 3a 4d 31 62 0a cb 52 e8 41 b6 be ee cb 31 80 45 8e 60 d3 b0
de b8 ee 37 f6 05 1e 7a 30 c3 da 0d 58 42 02 ca 7b 7c 87 92 34 94 67 d7 57 8b f5 90 47 65 71 53 9b 5a 83 f0 91 a7
0d 92 56 70 20 c8 09 68 4e 22 31 9b 86 3f 14 21 8f 78 48 e3
WPA: decrypted EAPOL-Key key data - hexdump(len=72): [REMOVED]
wlan0: State: 4WAY_HANDSHAKE -> 4WAY_HANDSHAKE
```

```
wlan0: WPA: RX message 3 of 4-Way Handshake from 08:57:00:7c:fd:90 (ver=2)
WPA: IE KeyData - hexdump(len=72): 30 14 01 00 00 0f ac 04 01 00 00 0f ac 04 01 00 00 0f ac 02 00 00 dd 16 00 50 f2 01 01 00 00 50 f2 04 01 00 00 50 f2 04 01 00 00 50 f2 02 dd 16 00 0f ac 01 02 00 88 54 b4 09 1f a8 fe ab f3 7f 1d b8 18 92 7d 87 dd 00
WPA: RSN IE in EAPOL-Key - hexdump(len=22): 30 14 01 00 00 0f ac 04 01 00 00 0f ac 04 01 00 00 0f ac 02 00 00
WPA: WPA IE in EAPOL-Key - hexdump(len=24): dd 16 00 50 f2 01 01 00 00 50 f2 04 01 00 00 50 f2 04 01 00 00 50 f2 02
WPA: GTK in EAPOL-Key - hexdump(len=24): [REMOVED]
wlan0: WPA: Sending EAPOL-Key 4/4
WPA: KCK - hexdump(len=16): [REMOVED]
WPA: Derived Key MIC - hexdump(len=16): 01 9b 30 e1 d9 67 f0 86 c7 97 96 07 87 8b 24 80
wlan0: WPA: Installing PTK to the driver
wpa_driver_wext_set_key: alg=3 key_idx=0 set_tx=1 seq_len=6 key_len=16
EAPOL: External notification - portValid=1
wlan0: State: 4WAY_HANDSHAKE -> GROUP_HANDSHAKEUpdateHalRAMask8188EUsb => mac_id:0, networkType:0x0b, mask:0x000fffff
        ==> rssi_level:2, rate_bitmap:0x000ff000

RSN: received GTK in pairwise handshake - hexdump(len=18): [REMOVED]
WPA: Group Key - hexdump(len=16): [REMOVED]
wlan0: WPA: Installing GTK to the driver (keyidx=2 tx=0 len=16)
WPA: RSC - hexdump(len=6): d3 0e 00 00 00 00
wpa_driver_wext_set_key: alg=3 key_idx=2 set_tx=0 seq_len=6 key_len=16
wlan0: WPA: Key negotiation completed with 08:57:00:7c:fd:90 [PTK=CCMP GTK=CCMP]
wlan0: Cancelling authentication timeout
wlan0: State: GROUP_HANDSHAKE -> COMPLETED
wlan0: Radio work 'connect'@0x58e400 done in 0.916960 seconds
wlan0: CTRL-EVENT-CONNECTED - Connection to 08:57:00:7c:fd:90 completed [id=0 id_str=]
wpa_driver_wext_set_operstate: operstate 0->1 (UP)
netlink: Operstate: ifindex=3 linkmode=-1 (no change), operstate=6 (IF_OPER_UP)
EAPOL: External notification - portValid=1
EAPOL: External notification - EAP success=1
EAPOL: SUPP_PAE entering state AUTHENTICATING
EAPOL: SUPP_BE entering state SUCCESS
EAP: EAP entering state DISABLED
EAPOL: SUPP_PAE entering state AUTHENTICATED
EAPOL: Supplicant port status: Authorized
EAPOL: SUPP_BE entering state IDLE
EAPOL authentication completed - result=SUCCESS
RTM_NEWLINK: operstate=1 ifi_flags=0x11043 ([UP][RUNNING][LOWER_UP])
RTM_NEWLINK, IFLA_IFNAME: Interface 'wlan0' added
WEXT: if_removed already cleared - ignore event
EAPOL: startWhen --> 0
EAPOL: disable timer tick
```

以上 CTRL-EVENT-CONNECTED 说明，授权完成，联网进入路由器成功。

关闭 eth0，否则会默认使用 eth0 来连接网络。**添加网关**，否则无法连通网络。

```
# ifconfig eth0 down
Ignore event for foreign ifindex 2

# route add default gw 192.168.1.253
#route add default netmask 255.255.255.0
# ping www.baidu.com
```

记住：当出现『SIOCADDRT: Network is unreachable』这个错误时，肯定是由于 gw 后面接的 IP 无法直接网域沟通 (Gateway 并不在你的网域内)，需要检查一下是否输入错误。

在移植无线网卡的过程中出现这个错误，原因是路由器的无线网加密方式为 wpa,所以需要移植 wpa_supplicant 工具.如果不移植 wpa_supplicant，可以设置路由器的加密方式为 wep，密钥设置为 64 位，如设置为 128 位会报错。

总结，开机后，开启无线网卡并配置的命令：

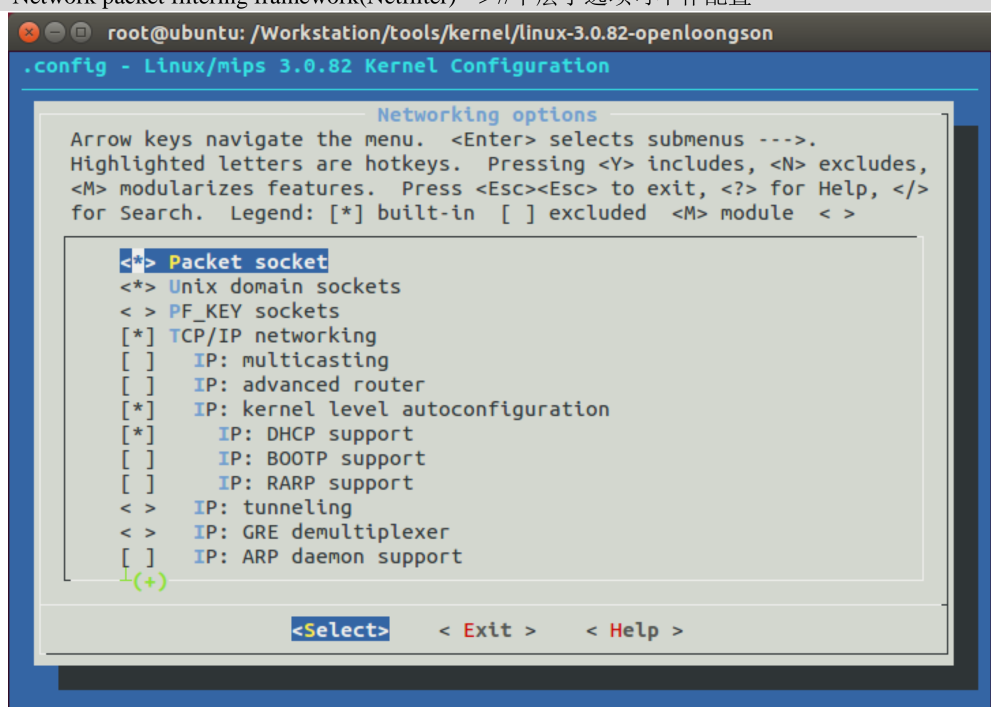
```
#ifconfig eth0 down
#wpa_supplicant -d -Dwext -iwlan0 -c/bin/wpa_supplicant.conf &
#ifconfig wlan0 192.168.1.111
#route add default gw 192.168.1.1
#ping 221.226.0.186
```

7 配置 DHCP

首先在内核和根文件系统的配置中打开 DHCP。

（1）配置 Linux 内核使能 DHCP 相关选项：

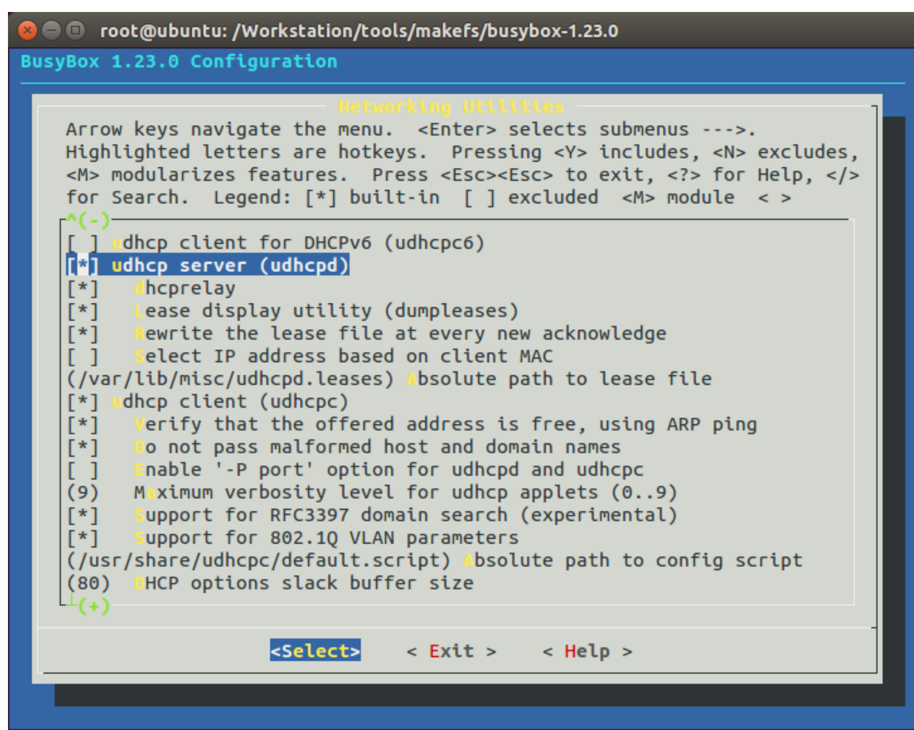
```
[*]Networking support -->
Networking support
Networking options -->
[*]      Packet socket
[*]      IP:kernel level autoconfiguration
[*]         IP:DHCP support
[*]      Network packet filtering framework(Netfilter) --> //下层子选项可不作配置
```



重新编译并下载进开发板。

（2）配置 Busybox，使能以下选项：

```
Networking Utilities -->
[*]udhcp server (udhcpd)
[*]      dhcprelay
[*]      Lease display utility (dumpleases)
[*]      Rewrite the lease file at every new acknowledge
[*]udhcp client (udhcpc)
[*]      Verify that the offered address is free,using ARP ping
[*]      Do not pass malformed host and domain names
```



重新编译并下载进开发板。

（3） 运行命令 udhcpc

```
[root@Loongson:/]#udhcpc -i wlan0
udhcpc (v1.22.1) started
Sending discover...
Sending select for 192.168.1.111...
Lease of 192.168.1.111 obtained, lease time 7200
[root@Loongson:/]#
```

以上只是自动获得了 IP 地址，并没有配置到网卡上。

需要把/busybox-1.23.0/examples/udhcpc 下的脚本 simple.script 改名为 default.script，放在开发板上的/usr/share/udhcpc/目录下，且用命令#chmod +x usr/share/udhcpc/default.script 增加该文件的执行权限，才能将获取的 IP 写到指定的网卡中。

```
[root@Loongson:/]#chmod +x usr/share/udhcpc/default.script
[root@Loongson:/]#udhcpc -i wlan0
udhcpc (v1.22.1) started
Setting IP address 0.0.0.0 on wlan0
Sending discover...
Sending discover...
Sending select for 192.168.1.101...
Lease of 192.168.1.101 obtained, lease time 7200
Setting IP address 192.168.1.101 on wlan0
Deleting routers
route: SIOCDELRT: No such process
Adding router 192.168.1.253
Recreating /etc/resolv.conf
 Adding DNS server 192.168.1.253
```

以上内容为 DHCP 配置 IP 地址为 192.168.1.101，路由为 192.168.1.253，DNS 为 192.168.1.253。

现在可 ping 通网关。

```
[root@Loongson:/]#ping 192.168.1.253
PING 192.168.1.253 (192.168.1.253): 56 data bytes
64 bytes from 192.168.1.253: seq=0 ttl=64 time=5.932 ms
64 bytes from 192.168.1.253: seq=1 ttl=64 time=2.230 ms
```

8 DHCP 配置成开机启动

建立一个脚本文件 wifiup，开机后自动执行。

```
#!/bin/sh

#Set ip
ifconfig eth0 down
ifconfig wlan0 up
sleep 2s
wpa_supplicant -Dwext -iwlan0 -c/bin/wpa_supplicant.conf &
sleep 5s
udhcpc -i wlan0
```

附录 10. 错误集锦

1. intel_rapl 错误

intel_rapl: no valid rapl domains found in package 0

将该模块列入不装入名单。编辑文件 sudo vim /etc/modprobe.d/blacklist.conf，在末尾加入 blacklist i2c_piix4 和 blacklist intel_rapl

2. rc-local.service

```
[FAILED] Failed to start /etc/rc.local Compatibility.
See 'systemctl status rc-local.service' for details.
[  OK  ] Started LSB: Tool to automatically collect and submit kernel crash signa
[  OK  ] Started LSB: Starts or stops the xinetd daemon..
[  OK  ] Started Login Service.
[FAILED] Failed to start LSB: HPA's tftp server.
See 'systemctl status tftpd-hpa.service' for details.
         Starting Authenticate and Authorize Users to Run Privileged Tasks...
         Starting Wait for Plymouth Boot Screen to Quit...
```

```
root@ubuntu:~# systemctl status rc-local.service
● rc-local.service - /etc/rc.local Compatibility
   Loaded: loaded (/lib/systemd/system/rc-local.service; static; vendor preset: enabled)
  Drop-In: /lib/systemd/system/rc-local.service.d
           └─debian.conf
   Active: failed (Result: exit-code) since Thu 2016-07-07 13:36:11 CST; 22min ago
  Process: 907 ExecStart=/etc/rc.local start (code=exited, status=1/FAILURE)

Jul 07 13:36:11 ubuntu systemd[1]: Starting /etc/rc.local Compatibility...
Jul 07 13:36:11 ubuntu systemd[1]: rc-local.service: Control process exited...=1
Jul 07 13:36:11 ubuntu systemd[1]: Failed to start /etc/rc.local Compatibility.
Jul 07 13:36:11 ubuntu systemd[1]: rc-local.service: Failed with result 'ex...'.
Hint: Some lines were ellipsized, use -l to show in full.
root@ubuntu:~#
```

设置开机启动脚本

sudo touch /etc/rc.d/rc.local

sudo vim /etc/rc.d/rc.local

在/etc/rc.d/rc.local 文件中写入，然后使用:wq 命令 保存并退出.
#!/bin/bash
  # 在这个文件中写入开机启动需要执行的命令
赋予可执行权限:
sudo chmod+x /etc/rc.d/rc.local
设置开机启动:
sudo systemctl enable rc-local.service
如果出现以下错误提示:
[root@dev-zhanghua ~]# systemctl enable rc-local.service
The unit files have no [Install] section. They are not meant to be enabled
using systemctl.
Possible reasons for having this kind of units are:
1) A unit may be statically enabled by being symlinked from another unit's
  .wants/ or .requires/ directory.
2) A unit's purpose may be to act as a helper for some other unit which has
  a requirement dependency on it.
3) A unit may be started when needed via activation (socket, path, timer,
  D-Bus, udev, scripted systemctl call, ...).

sudo vim /usr/lib/systemd/system/rc-local.service
在 rc-local.service 文件末尾加入:
[Install]
WantedBy=multi-user.target
并重新设置开机启动:
sudo systemctl enable rc-local.service
查看状态，如果出现以下提示，设置成功!
[zhanghua@dev-zhanghua ~]$ systemctl status rc-local.service
● rc-local.service - /etc/rc.d/rc.local Compatibility
  Loaded: loaded (/usr/lib/systemd/system/rc-local.service; enabled)
  Active: active (running) since Wed 2015-04-15 11:37:40 CST; 1h 44min ago

root@ubuntu:/# systemctl enable rc-local.service
Created symlink from /etc/systemd/system/multi-user.target.wants/rc-local.service to /lib/systemd/system/rc-local.service.

附录 11 git 命令

http://blog.csdn.net/five3/article/details/8904635
安装完成后对 git 进行配置，需要确保连接 github.com 的账号，需要命令

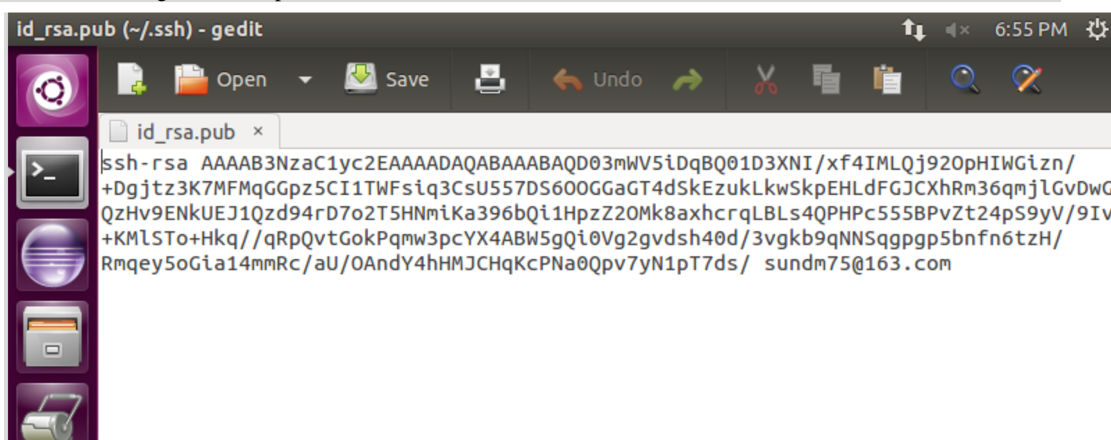
```
git config --global user.name "XXX"
git config --global user.eamil "邮箱地址"
```

配置完成后，需要创建验证的公钥（与 windows 下相同），每个用户需要独立的公钥来确定。使用命令 ssh-keygen -C '你的邮箱地址' -t rsa ，此时会在用户目录~/.ssh/下建立相应 的密钥文件

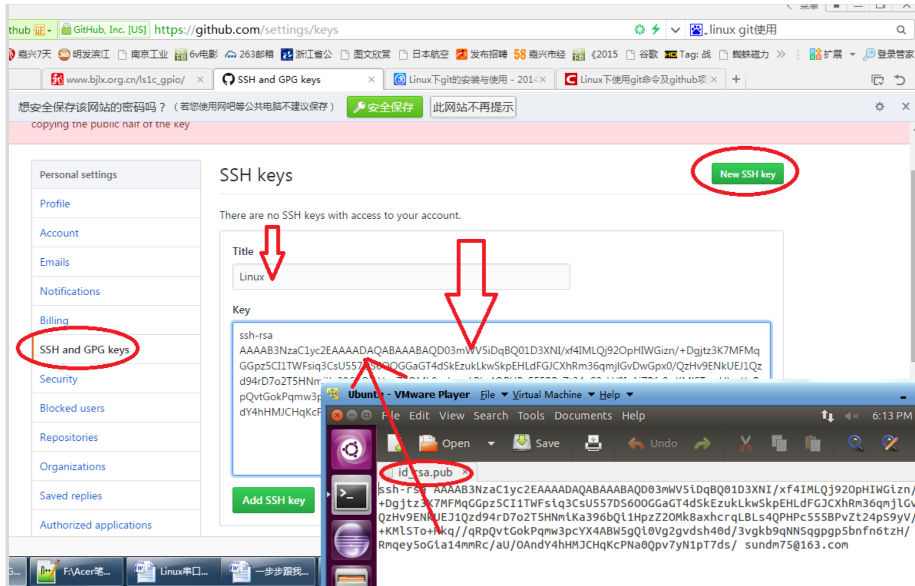
```
ssh-keygen -C '你的邮箱地址' -t rsa
```

之后使用命令 cd ~/.ssh 进入文件夹，使用 gedit id_rsa.pub 打开 id_rsa.pub 文件。文件里面的内容就是 ssh 公钥，将其全部复制。

```
root@ubuntu:/# ssh-keygen -C 'sundm75@163.com' -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:rppQREFMOwNeapt48EcC0thjmnfcWKl5H89YBHpojBc sundm75@163.com
The key's randomart image is:
+---[RSA 2048]----+
|o+.+*.E...       |
|oo==.+o+   .     |
|.+=oB** ..       |
|o=.**=o.. .      |
|..=.o. .S*       |
| . o     .o o    |
|   .       .     |
|     . . .       |
|       o..       |
+----[SHA256]-----+
root@ubuntu:/# cd ~/.ssh
root@ubuntu:~/.ssh# gedit id_rsa.pub
```



打开 github.com 网址，选择增加 ssh 公钥一项，标题的内容随意输入，下面将复制的内容粘贴进去之后就完成了。



可以使用命令 ssh –T git@github.com 来测试是否成功。

```
root@ubuntu:~/.ssh# ssh -T git@github.com
The authenticity of host 'github.com (192.30.253.113)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.253.113' (RSA) to the list of known hosts.
Hi sundm75! You've successfully authenticated, but GitHub does not provide shell access.
```

同步 github 到本地

1、复制项目到本地：

[plain] view plain copy

```
git clone git://github.com:xxxx/test.git ##以 gitreadonly 方式克隆到本地，只可以读
git clone git@github.com:xxx/test.git  ##以 SSH 方式克隆到本地，可以读写
git clone https://github.com/xxx/test.git ##以 https 方式克隆到本地，可以读写
git fetch git@github.com:xxx/xxx.git  ##获取到本地但不合并
git pull git@github.com:xxx/xxx.git ##获取并合并内容到本地
```

本地提交项目到 github

1、本地配置

[plain] view plain copy

```
git config --global user.name 'onovps'
git config --global user.email 'onovps@onovps.com' #全局联系方式，可选
```

2、新建 Git 项目并提交到 Github。

[plain] view plain copy

```
mkdir testdir & cd testdir
touch README.md
git init #初始化一个本地库
git add README.md #添加文件到本地仓库
git rm README.md #本地倒库内删除
git commit -m "first commit" #提交到本地库并备注，此时变更仍在本地。
git commit -a  ##自动更新变化的文件，a 可以理解为 auto
git remote add xxx git@github.com:xxx/xxx.git  #增加一个远程服务器的别名。
git remote rm xxx   ##删除远程版本库的别名
```

git push -u remotename master #将本地文件提交到 Github 的 remoname 版本库中。此时才更新了本地变更到 github 服务上。

分支版本操作

1、创建和合并分支

[plain] view plain copy

```
git branch #显示当前分支是 master
git branch new-feature  #创建分支
git checkout new-feature  #切换到新分支
vi page_cache.inc.php
git add page_cache.inc.php
git commit -a -m "added initial version of page cache"
git push origin new-feature  ##把分支提交到远程服务器，只是把分支结构和内容提交到远程，并没有发生和主干的合并行为。
```

2、如果 new-feature 分支成熟了，觉得有必要合并进 master

[plain] view plain copy

```
git checkout master  #切换到新主干
git merge new-feature  ##把分支合并到主干
git branch #显示当前分支是 master
git push  #此时主干中也合并了 new-feature 的代码
```

附录 12 在 pmon 中使命令 devcp 可以进行坏块处理和支持 yaffs2 烧写

http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=26307305&id=3031400

一、修改的文件：

1、拷贝目录文件：sys/dev/nand/yaf-nand

2、修改 conf/files，增加一行：file sys/dev/nand/yaf-nand/nand_util.c nand

3、替换 pmon/fs/mtd.c 这个文件

4、替换 pmon/cmds/mycmd.c 这个文件

5、增加 include/linux/mtd/compat.h 这个头文件

二、修改说明：

由于 nandflash 会有坏块的出现，所以当遇到坏块的时候，要跳过，直到不是坏块为止，对应于前面的 1、2、3 点。

而 yaffs2 的文件系统镜像跟其他的文件系统镜像的不同在于，它每 2KB 的数据之后会跟着 64B 的 oob 区数据。而 devcp 这个命令默认每次只会读 2KB 的数据就写入 nandflash，这就导致了 64B 的 oob 区数据也被当成了正常数据被烧到 main 区。解决方法是，每次读 2KB+64B 的数据出来，把 2KB 的数据写入到相应的 main 区，多出 64B 的 oob 数据也要写到 nandflash 相应的 oob 区，对应于前面的第 4 点。

三、使用命令：devcp tftp://192.168.1.xx/yaffs2.img /dev/mtd1 yaf nw，具体使用请参考广州龙芯用户手册。

附录 13 QT 安装使用

Qt 5 以后版本默认包含了所有需要的工具。这里使用了 32 位 Linux 版本的 Qt 5.4.0。

将下载的安装文件 qt-opensource-linux-x86-5.4.0.run 拷贝到虚拟机某个目录下。在该目录下执行./qt-opensource-linux-x86-5.4.0.run ，安装 Qt，缺省安装在/opt/Qt5.4.0 下；如果 qt-opensource-linux-x86-5.4.0.run 的属性中拥有者没有运行权限，则可用 chmod 命令添加执行权限： chmod u+x qt-opensource-linux-x86-5.4.0.run

如果没有安装 g++，则通过命令安装:apt-get install gcc-c++ 。

附录 14 GDB 使用

本节摘自：

作者: liigo 原文链接: http://blog.csdn.net/liigo/archive/2006/01/17/582231.aspx

GDB 是一个由 GNU 开源组织发布的、UNIX/LINUX 操作系统下的、基于命令行的、功能强大的程序调试工具。一般来说，GDB 主要完成下面四个方面功能：

1.启动程序，可以按照自定义的要求随心所欲的运行程序。

2. 可让被调试的程序在所指定的调置的断点处停住。（断点可以是条件表达式）

3. 当程序被停住时，可以检查此时程序中所发生的事。

4. 动态的改变程序的执行环境。

GDB 中的命令固然很多，但只需掌握其中十个左右的命令，就大致可以完成日常的基本的程序调试工作。

命令	解释	示例
file <文件名>	加载被调试的可执行程序文件。 因为一般都在被调试程序所在目录下执行 GDB，因而文本名不需要带路径。	(gdb) file gdb-sample
r	Run 的简写，运行被调试的程序。 如果此前没有下过断点，则执行完整个程序；如果有断点，则程序暂停在第一个可用断点处。	(gdb) r
c	Continue 的简写，继续执行被调试程序，直至下一个断点或程序结束。	(gdb) c
b <行号> b <函数名称> b *<函数名称> b *<代码地址> d [编号]	b: Breakpoint 的简写，设置断点。两可以使用“行号”“函数名称”“执行地址”等方式指定断点位置。 其中在函数名称前面加“*”符号表示将断点设置在“由编译器生成的 prolog 代码处”。如果不了解汇编，可以不予理会此用法。 d: Delete breakpoint 的简写，删除指定编号的某个断点，或删除所有断点。断点编号从 1 开始递增。	(gdb) b 8 (gdb) b main (gdb) b *main (gdb) b *0x804835c (gdb) d
s, n	s: 执行一行源程序代码，如果此行代码中有函数调用，则进入该函数； n: 执行一行源程序代码，此行代码中的函数调用也一	(gdb) s (gdb) n

	并执行。 s 相当于其它调试器中的“Step Into (单步跟踪进入)”； n 相当于其它调试器中的“Step Over (单步跟踪)”。 这两个命令必须在有源代码调试信息的情况下才可以使用（GCC 编译时使用“-g”参数）。	
si, ni	si 命令类似于 s 命令，ni 命令类似于 n 命令。所不同的是，这两个命令（si/ni）所针对的是汇编指令，而 s/n 针对的是源代码。	(gdb) si (gdb) ni
p <变量名称>	Print 的简写，显示指定变量（临时变量或全局变量）的值。	(gdb) p i (gdb) p nGlobalVar
display ... undisplay <编号>	display，设置程序中断后欲显示的数据及其格式。 例如，如果希望每次程序中断后可以看到即将被执行的下一条汇编指令，可以使用命令 “display /i $pc” 其中 $pc 代表当前汇编指令，/i 表示以十六进行显示。当需要关心汇编代码时，此命令相当有用。 undispaly，取消先前的 display 设置，编号从 1 开始递增。	(gdb) display /i $pc (gdb) undisplay 1
i	Info 的简写，用于显示各类信息，详情请查阅“help i”。	(gdb) i r
q	Quit 的简写，退出 GDB 调试环境。	(gdb) q
help [命令名称]	GDB 帮助命令，提供对 GDB 名种命令的解释说明。 如果指定了“命令名称”参数，则显示该命令的详细说明；如果没有指定参数，则分类显示所有 GDB 命令，供用户进一步浏览和查询。	(gdb) help display

一个示例用的 C 语言代码小程序：

```c
//此程序仅作为示例代码
#include <stdio.h>

int nGlobalVar = 0;

int tempFunction(int a, int b)
{
	printf("tempFunction is called, a = %d, b = %d /n", a, b);
	return (a + b);
}

int main()
{
	int n;
	n = 1;
	n++;
	n--;
```

```
18
19          nGlobalVar += 100;
20          nGlobalVar -= 12;
21
22          printf("n = %d, nGlobalVar = %d /n", n, nGlobalVar);
23
24          n = tempFunction(1, 2);
25          printf("n = %d", n);
26
27          return 0;
28      }
```

将此代码复制出来并保存到文件 gdb-sample.c 中，然后切换到此文件所在目录，用 GCC 编译：

```
gcc gdb-sample.c -o gdb-sample -g
```

在上面的命令行中，使用 -o 参数指定了编译生成的可执行文件名为 gdb-sample，使用参数 -g 表示将源代码信息编译到可执行文件中。如果不使用参数 -g，会给后面的 GDB 调试造成不便。当然，如果没有程序的源代码，自然也无从使用 -g 参数，调试/跟踪时也只能是汇编代码级别的调试/跟踪。

下面“gdb”命令启动 GDB，将首先显示 GDB 说明：

```
root@ubuntu:~/Downloads# gdb
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.   Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

上面最后一行“(gdb) ”为 GDB 内部命令引导符，等待用户输入 GDB 命令。

下面使用“file”命令载入被调试程序 gdb-sample（这里的 gdb-sample 即前面 GCC 编译输出的可执行文件）：

```
(gdb) file gdb-sample
Reading symbols from gdb-sample...done.
```

提示已经加载成功。

下面使用“r”命令执行（Run）被调试文件，因为尚未设置任何断点，将直接执行到程序结束：

```
(gdb) r
Starting program: /root/Downloads/developapps/0.gdb/gdb-sample
n = 1, nGlobalVar = 88 /ntempFunction is called, a = 1, b = 2 /nn = 3[Inferior 1 (process 3817) exited normally]
```

下面使用“b”命令在 main 函数开头设置一个断点（Breakpoint）：

```
(gdb) b main
Breakpoint 1 at 0x8048450: file gdb-sample.c, line 19.
```

提示已经成功设置断点，并给出了该断点信息：在源文件 gdb-sample.c 第 19 行处设置断点；这是本程序的第一个断点（序号为 1）；断点处的代码地址为 0x8048450（此值可能仅在本次调试过程中有效）。回过头去看源代码，第 19 行中的代码为“n = 1”，恰好是 main

函数中的第一个可执行语句（前面的“int n;”为变量定义语句，并非可执行语句）。

再次使用“r”命令执行（Run）被调试程序：

```
(gdb) r
Starting program: /root/Downloads/developapps/0.gdb/gdb-sample

Breakpoint 1, main () at gdb-sample.c:19
19          n = 1;
```

程序中断在 gdb-sample.c 第 19 行处，即 main 函数是第一个可执行语句处。

上面最后一行信息为：下一条将要执行的源代码为“n = 1;”，它是源代码文件 gdb-sample.c 中的第 19 行。

下面使用“s”命令（Step）执行下一行代码（即第 19 行“n = 1;”）：

```
(gdb) s
20          n++;
```

上面的信息表示已经执行完“n = 1;”，并显示下一条要执行的代码为第 20 行的“n++;”。

既然已经执行了“n = 1;”，即给变量 n 赋值为 1，那我们用“p”命令（Print）看一下变量 n 的值是不是 1 ：

```
(gdb) p n
$1 = 1
```

果然是 1。($1 大致是表示这是第一次使用“p”命令——再次执行“p n”将显示“$2 = 1”——此信息应该没有什么用处。)

下面分别在第 26 行、tempFunction 函数开头各设置一个断点（分别使用命令“b 26”“b tempFunction”）：

```
(gdb) b 26
Note: breakpoints 2 and 3 also set at pc 0x804847c.
Breakpoint 4 at 0x804847c: file gdb-sample.c, line 26.
(gdb) b tempFunction
Breakpoint 5 at 0x8048423: file gdb-sample.c, line 12.
```

使用“c”命令继续（Continue）执行被调试程序，程序将中断在第二 个断点（26 行），此时全局变量 nGlobalVar 的值应该是 88；再一次执行“c”命令，程序将中断于第三个断点（12 行，tempFunction 函数开头处），此时 tempFunction 函数的两个参数 a、b 的值应分别是 1 和 2：

```
(gdb) r
Starting program: /root/Downloads/developapps/0.gdb/gdb-sample

Breakpoint 1, main () at gdb-sample.c:19
19          n = 1;
(gdb) c
Continuing.

Breakpoint 2, main () at gdb-sample.c:26
26          printf("n = %d, nGlobalVar = %d /n", n, nGlobalVar);
(gdb) p nGlobalVar
$8 = 88
(gdb) c
Continuing.

Breakpoint 5, tempFunction (a=1, b=2) at gdb-sample.c:12
12          printf("tempFunction is called, a = %d, b = %d /n", a, b);
(gdb) p a
$9 = 1
(gdb) p b
$10 = 2
(gdb)
```

再一次执行“c”命令（Continue），因为后面再也没有其它断点，程序将一直执行到结束：

```
(gdb) c
Continuing.
n = 1, nGlobalVar = 88 /ntempFunction is called, a = 1, b = 2 /nn = 3[Inferior 1 (process 3904) exited normally]
```

有时候需要看到编译器生成的汇编代码，以进行汇编级的调试或跟踪，又该如何操作呢？

这就要用到 display 命令“display /i $pc”了（此命令前面已有详细解释），此后程序再中断时，就可以显示出汇编代码了：

```
(gdb) display /i $pc
(gdb) r
Starting program: /root/Downloads/developapps/0.gdb/gdb-sample

Breakpoint 1, main () at gdb-sample.c:19
19          n = 1;
1: x/i $pc
=> 0x8048450 <main+9>:    movl    $0x1,0x1c(%esp):
```

看到了汇编代码，“n = 1;”对应的汇编代码是“movl $0x1,0xfffffffc(%ebp)”。

并且以后程序每次中断都将显示下一条汇编指定（“si”命令用于执行一条汇编代码——区别于“s”执行一行 C 代码）：

```
(gdb) si
21          n--;
1: x/i $pc
=> 0x804845d <main+22>:  subl    $0x1,0x1c(%esp)
(gdb) si
23          nGlobalVar += 100;
1: x/i $pc
=> 0x8048462 <main+27>:  mov     0x804a024,%eax
(gdb) si
0x08048467      23          nGlobalVar += 100;
1: x/i $pc
=> 0x8048467 <main+32>:  add     $0x64,%eax
```

接下来试一下命令“b *<函数名称>”。

为了更简明，有必要先删除目前所有断点（使用“d”命令——Delete breakpoint）：

```
(gdb) d
Delete all breakpoints? (y or n) y
```

当被询问是否删除所有断点时，输入“y”并按回车键即可。

下面使用命令“b *main”在 main 函数的 prolog 代码处设置断点（prolog、epilog，分别表示编译器在每个函数的开头和结尾自行插入的代码）：

```
(gdb) info b
Num     Type           Disp Enb Address    What
6       breakpoint     keep y   0x08048447 in main at gdb-sample.c:17
        breakpoint already hit 1 time
(gdb) r
Starting program: /root/Downloads/developapps/0.gdb/gdb-sample

Breakpoint 6, main () at gdb-sample.c:17
17      {
1: x/i $pc
=> 0x8048447 <main>:      push    %ebp
(gdb) si
0x08048448      17      {
1: x/i $pc
=> 0x8048448 <main+1>:    mov     %esp,%ebp
(gdb) si
0x0804844a      17      {
1: x/i $pc
=> 0x804844a <main+3>:    and     $0xfffffff0,%esp
(gdb) si
0x0804844d      17      {
1: x/i $pc
=> 0x804844d <main+6>:    sub     $0x20,%esp
(gdb) si
19          n = 1;
1: x/i $pc
=> 0x8048450 <main+9>:    movl    $0x1,0x1c(%esp)
```

此时可以使用"i r"命令显示寄存器中的当前值———"i r"即"Infomation Register"：

```
(gdb) i r
eax            0x1  1
ecx            0x1d38bee0      490258144
edx            0xbffff144      -1073745596
ebx            0xb7fc0000      -1208221696
esp            0xbffff0f0 0xbffff0f0
ebp            0xbffff118      0xbffff118
esi            0x0  0
edi            0x0  0
eip            0x8048450       0x8048450 <main+9>
eflags         0x286       [ PF SF IF ]
cs             0x73115
ss             0x7b123
ds             0x7b        123
es             0x7b123
fs             0x0  0
gs             0x33        51
```

当然也可以显示任意一个指定的寄存器值：

```
(gdb) i r eax
eax            0x1  1
```

最后一个要介绍的命令是"q"，退出（Quit）GDB 调试环境：

```
(gdb) q
The program is running. Exit anyway? (y or n) y
```

附录 15　makefile 经典教程

http://blog.csdn.net/haoel/article/details/2886

跟我一起写 Makefile

陈皓

0 Makefile 很重要

什么是 makefile？或许很多 Winodws 的程序员都不知道这个东西，因为那些 Windows 的 IDE 都为你做了这个工作，但我觉得要作一个好的和 professional 的程序员，makefile 还是要懂。这就好像现在有这么多的 HTML 的编辑器，但如果你想成为一个专业人士，你还是要了解 HTML 的标识的含义。特别在 Unix 下的软件编译，你就不能不自己写 makefile 了，**会不会写 makefile，从一个侧面说明了一个人是否具备完成大型工程的能力**。因为，makefile 关系到了整个工程的编译规则。一个工程中的源文件不计数，其按**类型、功能、模块**分别放在若干个目录中，makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。makefile 带来的好处就是——"自动化编译"，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 makefile 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 make，Visual C++的 nmake，Linux 下 GNU 的 make。可见，makefile 都成为了一种在工程方面的编译方法。

现在讲述如何写 makefile 的文章比较少，这是我想写这篇文章的原因。当然，不同产商的 make 各不相同，也有不同的语法，但其本质都是在"文件依赖性"上做文章，这里，我仅对 GNU 的 make 进行讲述，我的环境是 RedHat Linux 8.0，make 的版本是 3.80。必竟，这个 make 是应用最为广泛的，也是用得最多的。而且其还是最遵循于 IEEE 1003.2-1992 标准的（POSIX.2）。

在这篇文档中，将以 C/C++的源码作为我们基础，所以必然涉及一些关于 C/C++的编译的知识，相关于这方面的内容，还请各位查看相关的编译器的文档。这里所默认的编译器是 UNIX 下的 GCC 和 CC。

0.1 关于程序的编译和链接

在此，我想多说关于程序编译的一些规范和方法，一般来说，无论是 C、C++、还是 pas，首先要把源文件编译成**中间代码文件**，在 Windows 下也就是 .obj 文件，UNIX 下是 .o 文件，即 Object File，这个动作叫做**编译（compile）**。然后再把大量的 Object File 合成执行文件，这个动作叫作链接（link）。

编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在 C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。一般来说，每个源文件都应该对应于一个中间目标文件（O 文件或是 OBJ 文件）。

链接时，主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件（O 文件或是 OBJ 文件）来链接我们的应用程序。链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在 Windows 下这种包叫"**库文件"（Library File)**，也就是 .lib 文件，在 UNIX 下，是 Archive File，也就是 .a 文件。

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成 Object File。而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的 ObjectFile.

好，言归正传，GNU 的 make 有许多的内容，闲言少叙，还是让我们开始吧。

1 Makefile 介绍

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎么样的去编译和链接程序。

首先，我们用一个示例来说明 Makefile 的书写规则。以便给大家一个感兴认识。这个示例来源于 GNU 的 make 使用手册，在这个示例中，我们的工程有 8 个 C 文件，和 3 个头文件，我们要写一个 Makefile 来告诉 make 命令如何编译和链接这几个文件。我们的规则是：

1.如果这个工程没有编译过，那么我们的所有 C 文件都要编译并被链接。

2.如果这个工程的某几个 C 文件被修改，那么我们只编译被修改的 C 文件，并链接目标程序。

3.如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C 文件，并链接目标程序。

只要我们的 Makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

1.1 Makefile 的规则

在讲述这个 Makefile 之前，还是让我们先来粗略地看一看 Makefile 的规则。

```
target... : prerequisites ...
 command
 ...
 ...
```

----

target 也就是一个目标文件，可以是 **Object File**，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的"伪目标"章节中会有叙述。

prerequisites 就是，要生成那个 target 所需要的文件或是目标。

command 也就是 make 需要执行的命令。（任意的 Shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

说到底，Makefile 的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是 Makefile 的主线和核心，但要写好一个 Makefile 还不够，我会以后面一点一点地结合我的工作经验给你慢慢到来。内容还多着呢。：）

1.2 一个示例

正如前面所说的，如果一个工程有 3 个头文件，和 8 个 C 文件，我们为了完成前面所述的那三个规则，我们的 Makefile 应该是下面的这个样子的。

```
edit : main.o kbd.o command.o display.o ₩
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o ₩
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
```

```
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o ₩
           insert.o search.o files.o utils.o
```

反斜杠（₩）是换行符的意思。这样比较便于 Makefile 的易读。我们可以把这个内容保存在文件为"Makefile"或"makefile"的文件中，然后在该目录下直接输入命令"make"就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下"make clean"就可以了。

在这个 makefile 中，目标文件（target）包含：执行文件 edit 和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的**操作系统命令，一定要以一个 Tab 键作为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令**。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

1.3 make 是如何工作的

在默认的方式下，也就是我们只输入 make 命令。那么，

1. make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。
3. 如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。
4. 如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。（这有点像一个堆栈的过程）
5. 当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成 .o 文件，然后再用 .o 文件声明 make 的终极任务，也就是执行文件 edit 了。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make 执行。即命令——**“make clean”，以此来清除所有的目标文件，以便重编译**。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，我们的目标 file.o 会被重编译（也就是在这个依性关系后面所定义的命令），于是 file.o 的文件也是最新的啦，于是 file.o 的文件修改时间要比 edit 要新，所以 edit 也会被重新链接了（详见 edit 目标文件后定义的命令）。

而如果我们改变了“command.h”，那么，kdb.o、command.o 和 files.o 都会被重编译，并且，edit 会被重链接。

1.4 makefile 中使用变量

在上面的例子中，先让我们看看 edit 的规则：

```
edit : main.o kbd.o command.o display.o ₩
          insert.o search.o files.o utils.o
     cc -o edit main.o kbd.o command.o display.o ₩
          insert.o search.o files.o utils.o
```

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在 clean 中）。当然，我们的 makefile 并不复杂，所以在两个地方加也不累，但如果 makefile 变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了 makefile 的易维护，在 makefile 中我们可以使用变量。makefile 的变量也就是一个字符串，理解成 C 语言中的宏可能会更好。

比如，我们声明一个变量，叫 objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示 obj 文件就行了。我们在 makefile 一开始就这样定义：

```
objects = main.o kbd.o command.o display.o ₩
      insert.o search.o files.o utils.o
```

于是，我们就可以很方便地在我们的 makefile 中以 **"$(objects)"** 的方式来使用这个变量了，于是我们的改良版 makefile 就变成下面这个样子：

```
objects = main.o kbd.o command.o display.o \
                    insert.osearch.o files.o utils.o
  edit : $(objects)
                cc -o edit $(objects)
main.o : main.c defs.h
                cc -c main.c
kbd.o : kbd.c defs.h command.h
                cc -c kbd.c
command.o : command.c defs.h command.h
                cc -c command.c
display.o : display.c defs.h buffer.h
                cc -c display.c
insert.o : insert.c defs.h buffer.h
                cc -c insert.c
search.o : search.c defs.h buffer.h
```

```
                cc -c search.c
files.o : files.c defs.h buffer.h command.h
                cc -c files.c
utils.o : utils.c defs.h
                cc -c utils.c
clean :
                rm edit $(objects)
```

于是如果有新的 .o 文件加入，我们只需简单地修改一下 objects 变量就可以了。

关于变量更多的话题，我会在后续给你一一道来。

1.5 让 make 自动推导

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令。

只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，我们的 makefile 再也不用写得这么复杂。我们的是新的 makefile 又出炉了。

```
objects = main.o kbd.o command.o display.o \
                    insert.o search.o files.o utils.o


edit : $(objects)
                cc -o edit $(objects)


main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h


.PHONY : clean
clean :
```

```
rm edit $(objects)
```

这种方法，也就是 make 的"隐晦规则"。上面文件内容中，".PHONY"表示，clean 是个伪目标文件。

关于更为详细的"隐晦规则"和"伪目标文件"，我会在后续给你一一道来。

1.6 另类风格的 makefile

即然我们的 make 可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题，这个对于 make 来说很容易，谁叫它提供了自动推导命令和文件的功能呢？来看看最新风格的 makefile 吧。

```
objects = main.o kbd.o command.o display.o \
                    insert.o search.o files.o utils.o

edit : $(objects)
                cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
                rm edit $(objects)
```

这种风格，让我们的 makefile 变得很简单，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的.o 文件，那就理不清楚了。

1.7 清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个"修养"（呵呵，还记得我的《编程修养》吗）。一般的风格都是：

```
clean:
    rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

前面说过，.PHONY 意思表示 clean 是一个"伪目标"，。而在 rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认目标，相信谁也不愿意这样。不成文的规矩是——"clean 从来都是放在文件的最后"。

上面就是一个 makefile 的概貌，也是 makefile 的基础，下面还有很多 makefile 的相关细节，准备好了吗？准备好了就来。

2 Makefile 总述

2.1 Makefile 里有什么？

Makefile 里主要包含了**五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。**

1. 显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
2. 隐晦规则。由于我们的 make 有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make 所支持的。
3. 变量的定义。在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。
5. 注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用"#"字符，这个就像 C/C++中的"//"一样。如果你要在你的 Makefile 中使用"#"字符，可以用反斜框进行转义，如："₩#"。

最后，还值得一提的是，在 Makefile 中的命令，必须要以[Tab]键开始。

2.2Makefile 的文件名

默认的情况下，make 命令会在当前目录下按顺序找寻文件名为**“GNUmakefile”、“makefile”、“Makefile”**的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“GNUmakefile”，这个文件是 GNU 的 make 识别的。有另外一些 make 只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的 make 都支持**“makefile”和“Makefile”**这两种默认文件名。

当然，你可以使用别的文件名来书写 Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要**指定特定的 Makefile，你可以使用 make 的“-f”和“--file”参数**，如：make -f Make.Linux 或 make --file Make.AIX。

2.3 引用其它的 Makefile

在 Makefile 使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include 的语法是：

include<filename>filename 可以是当前操作系统 Shell 的文件模式（可以保含路径和通配符）

在 include 前面可以有一些空字符，但是绝不能是[Tab]键开始。include 和可以用一个或多个空格隔开。举个例子，你有这样几个 Makefile：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量$(bar)，其包含了 e.mk 和 f.mk，那么，下面的语句：

```
include foo.make *.mk $(bar)
```

等价于：

```
include foo.make a.mk b.mk c.mk e.mk f.mk
```

make 命令开始时，会把找寻 include 所指出的其它 Makefile，并把其内容安置在当前的位置。就好像 C/C++的#include 指令一样。如果文件都没有指定绝对路径或是相对路径的话，make 会在当前目录下首先寻找，如果当前目录下没有找到，那么，make 还会在下面的几个目录下找：

1. 如果 make 执行时，有“-I”或“--include-dir”参数，那么 make 就会在这个参数所指定的目录下去寻找。

2. 如果目录/include（一般是：/usr/local/bin 或/usr/include）存在的话，make 也会去找。

如果有文件没有找到的话，make 会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成 makefile 的读取，make 会再重试这些没有找到，或是不能读取的文件，如果还是不行，make 才会出现一条致命信息。如果你想让 make 不理那些无法读取的文件，而继续执行，你可以在 include 前加一个减号“-”。如：

-include<filename>

其表示，无论 include 过程中出现什么错误，都不要报错继续执行。和其它版本 make 兼容的相关命令是 sinclude，其作用和这一个是一样的。

2.4 环境变量 MAKEFILES

如果你的当前环境中定义了环境变量 MAKEFILES，那么，make 会把这个变量中的值做一个类似于 include 的动作。这个变量中的值是其它的 Makefile，用空格分隔。只是，它和 include 不同的是，从这个环境变中引入的 Makefile 的"目标"不会起作用，如果环境变量中定义的文件发现错误，make 也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用 make 时，所有的 Makefile 都会受到它的影响，这绝不是你想看到的。在这里提这个事，只是为了告诉大家，也许有时候你的 Makefile 出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

2.5 make 的工作方式

GNU 的 make 工作时的执行步骤入下：（想来其它的 make 也是类似）

1. 读入所有的 Makefile。
2. 读入被 include 的其它 Makefile。
3. 初始化文件中的变量。
4. 推导隐晦规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

1-5 步为第一个阶段，6-7 为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make 会把其展开在使用的位置。但 make 并不会完全马上展开，make 使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

当然，这个工作方式你不一定要清楚，但是知道这个方式你也会对 make 更为熟悉。有了这个基础，后续部分也就容易看懂了。

3 Makefile 书写规则

规则包含两个部分，一个是**依赖关系**，一个是**生成目标的方法**。

在 Makefile 中，规则的顺序是很重要的，因为，**Makefile 中只应该有一个最终目标**，其它的目标都是被这个目标所连带出来的，所以一定要让 make 知道你的最终目标是什么。一般来说，定义在

Makefile 中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make 所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

3.1 规则举例

```
foo.o: foo.c defs.h       # foo 模块
        cc -c -g foo.c
```

看到这个例子，各位应该不是很陌生了，前面也已说过，foo.o 是我们的目标，foo.c 和 defs.h 是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（以 Tab 键开头）。这个规则告诉我们两件事：

1. 文件的依赖关系，foo.o 依赖于 foo.c 和 defs.h 的文件，如果 foo.c 和 defs.h 的文件日期要比 foo.o 文件日期要新，或是 foo.o 不存在，那么依赖关系发生。

2. 如果生成（或更新）foo.o 文件。也就是那个 cc 命令，其说明了，如何生成 foo.o 这个文件。（当然 foo.c 文件 include 了 defs.h 文件）

3.2 规则的语法

```
targets : prerequisites
  command
  ...
```

或是这样：

```
targets : prerequisites ; command
      command
      ...
```

targets 是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command 是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab 键]开头，如果和 prerequisites 在一行，那么可以用分号做为分隔。（见上）

prerequisites 也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘₩’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何成成目标文件。

一般来说，make 会以 UNIX 的标准 Shell，也就是/bin/sh 来执行命令。

3.3 在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make 支持三各通配符：“*”，“?”和“[...]”。这是和 Unix 的 B-Shell 是相同的。

"~"

波浪号（"~"）字符在文件名中也有比较特殊的用途。如果是"~/test"，这就表示当前用户的$HOME 目录下的 test 目录。而"~hchen/test"则表示用户 hchen 的宿主目录下的 test 目录。（这些都是 Unix 下的小知识了，make 也支持）而在 Windows 或是 MS-DOS 下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量"HOME"而定。

"*"

通配符代替了你一系列的文件，如"*.c"表示所以后缀为 c 的文件。一个需要我们注意的是，如果我们的文件名中有通配符，如："*"，那么可以用转义字符"₩"，如"₩*"来表示真实的"*"字符，而不是任意长度的字符串。

好吧，还是先来看几个例子吧：

```
clean:
    rm -f *.o
```

上面这个例子我不不多说了，这是操作系统 Shell 所支持的通配符。这是在命令中的通配符。

```
print: *.c
    lpr -p $?
    touch print
```

上面这个例子说明了通配符也可以在我们的规则中，目标 print 依赖于所有的[.c]文件。其中的"$?"是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了，通符同样可以用在变量中。并不是说[*.o]会展开，不！objects 的值就是"*.o"。Makefile 中的变量其实就是 C/C++中的宏。如果你要让通配符在变量中展开，也就是让 objects 的值是所有[.o]的文件名的集合，那么，你可以这样：

```
objects := $(wildcard *.o)
```

这种用法由关键字"wildcard"指出，关于 Makefile 的关键字，我们将在后面讨论。

3.4 文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当 make 需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉 make，让 make 在自动去找。

Makefile 文件中的特殊变量"VPATH"就是完成这个功能的，如果没有指明这个变量，make 只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make 就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

上面的的定义指定两个目录，"src"和"../headers"，make 会按照这个顺序进行搜索。目录由"冒号"分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用 make 的"vpath"关键字（注意，它是全小写的），这不是变量，这是一个 make 的关键字，这和上面提到的那个 VPATH 变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1. vpath < pattern> < directories>　为符合模式< pattern>的文件指定搜索目录<directories>。
2. vpath < pattern>　清除符合模式< pattern>的文件的搜索目录。
3. vpath　清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的< pattern>需要包含"%"字符。"%"的意思是匹配零或若干字符，例如，"%.h"表示所有以".h"结尾的文件。< pattern>指定了要搜索的文件集，而< directories>则指定了的文件集的搜索的目录。例如：

```
vpath %.h ../headers
```

该语句表示，要求 make 在"../headers"目录下搜索所有以".h"结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的< pattern>，或是被重复了的< pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。如：

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

其表示".c"结尾的文件，先在"foo"目录，然后是"blish"，最后是"bar"目录。

```
vpath %.c foo:bar
vpath %   blish
```

而上面的语句则表示".c"结尾的文件，先在"foo"目录，然后是"bar"目录，最后才是"blish"目录。

3.5 伪目标

最早先的一个例子中，我们提到过一个"clean"的目标，这是一个"伪目标"，

```
clean:
        rm *.o temp
```

正像我们前面例子中的"clean"一样，即然我们生成了许多文件编译文件，我们也应该提供一个清除它们的"目标"以备完整地重编译而用。（以"make clean"来使用该目标）

因为，我们并不生成"clean"这个文件。"伪目标"并不是一个文件，只是一个标签，由于"伪目标"不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个"目标"才能让其生效。当然，"伪目标"的取名不能和文件名重名，不然其就失去了"伪目标"的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记".PHONY"来显示地指明一个目标是"伪目标"，向 make 说明，不管是否有这个文件，这个目标就是"伪目标"。

```
.PHONY : clean
```

只要有这个声明，不管是否有"clean"文件，要运行"clean"这个目标，只有"make clean"这样。于是整个过程可以这样写：

```
.PHONY: clean
clean:
        rm *.o temp
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为"默认目标"，只要将其放在第一个。一个示例就是，如果你的 Makefile 需要一口气生成若干个可执行文件，但你只想简单地敲一个 make 完事，并且，所有的目标文件都写在一个 Makefile 中，那么你可以使用"伪目标"这个特性：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile 中的第一个目标会被作为其默认目标。我们声明了一个"all"的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如"all"这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。".PHONY : all"声明了"all"这个目标为"伪目标"。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o
```

```
cleandiff :
        rm *.diff
```

"makeclean"将清除所有要被清除的文件。"cleanobj"和"cleandiff"这两个伪目标有点像"子程序"的意思。我们可以输入"makecleanall"和"make cleanobj"和"makecleandiff"命令来达到清除不同种类文件的目的

3.6 多目标

Makefile 的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量"$@"（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

上述规则等价于：

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

其中，-$(subst output,,$@)中的"$"表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，"$@"表示目标的集合，就像一个数组，"$@"依次取出目标，并执于命令。

3.7 静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

```
<targets...>: <target-pattern>: <prereq-patterns ...>
        <commands>
...
```

targets 定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern 是指明了 targets 的模式，也就是的目标集模式。

prereq-parrterns 是目标的依赖模式，它对 target-parrtern 形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-parrtern>定义成"%.o"，意思是我们的集合中都是以".o"结尾的，而如果我们的<prereq-parrterns>定义成"%.c"，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的"%"（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

所以，我们的"目标模式"或是"依赖模式"中都应该有"%"这个字符，如果你的文件名中有"%"那么你可以使用反斜杠"₩"进行转义，来标明真实的"%"字符。

看一个例子：

```
objects = foo.o bar.o


all: $(objects)


$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从$object 中获取，"%.o"表明要所有以".o"结尾的目标，也就是"foo.o bar.o"，也就是变量$object 集合的模式，而依赖模式"%.c"则取模式"%.o"的"%"，也就是"foobar"，并为其加下".c"的后缀，于是，我们的依赖目标就是"foo.cbar.c"。而命令中的"$<"和"$@"则是自动化变量，"$<"表示所有的依赖目标集（也就是"foo.c bar.c"），"$@"表示目标集（也褪恰瓣 oo.o bar.o"）。于是，上面的规则展开后等价于下面的规则：

```
foo.o : foo.c
        $(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
        $(CC) -c $(CFLAGS) bar.c -o bar.o
```

试想，如果我们的"%.o"有几百个，那种我们只要用这种很简单的"静态模式规则"就可以写完一堆规则，实在是太有效率了。"静态模式规则"的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
files = foo.elc bar.o lose.o


$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

$(filter%.o,$(files))表示调用 Makefile 的 filter 函数，过滤"$filter"集，只要其中模式为"%.o"的内容。其的它内容，我就不用多说了吧。这个例字展示了 Makefile 中更大的弹性。

3.8 自动生成依赖性

在 Makefile 中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的 main.c 中有一句"#include "defs.h""，那么我们的依赖关系应该是：

main.o : main.c defs.h

但是，如果是一个比较大型的工程，你必需清楚哪些 C 文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改 Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用 C/C++编译的一个功能。大多数的 C/C++编译器都支持一个"-M"的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

cc -M main.c

其输出是：

main.o : main.c defs.h

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用 GNU 的 C/C++编译器，你得用"-MM"参数，不然，"-M"参数会把一些标准库的头文件也包含进来。

gcc-M main.c 的输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h ₩
    /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h ₩
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h ₩
    /usr/include/bits/types.h /usr/include/bits/pthreadtypes.h ₩
    /usr/include/bits/sched.h /usr/include/libio.h ₩
    /usr/include/_G_config.h /usr/include/wchar.h ₩
    /usr/include/bits/wchar.h /usr/include/gconv.h ₩
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h ₩
    /usr/include/bits/stdio_lim.h
```

gcc-MM main.c 的输出则是：

main.o: main.c defs.h

那么，编译器的这个功能如何与我们的 Makefile 联系在一起呢。因为这样一来，我们的 Makefile 也要根据这些源文件重新生成，让 Makefile 自已依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU 组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个"name.c"的文件都生成一个"name.d"的 Makefile 文件，[.d]文件中就存放对应[.c]文件的依赖关系。

于是，我们可以写出[.c]文件和[.d]文件的依赖关系，并让 make 自动更新或自成[.d]文件，并把其包含在我们的主 Makefile 中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

%.d: %.c

```
@set -e; rm -f $@; ₩
 $(CC) -M $(CPPFLAGS) $< > $@.
; ₩
 sed 's,$*₩.o[ :]*,₩1.o $@ : ,g' < $@.
> $@; ₩
 rm -f $@.
```

这个规则的意思是，所有的[.d]文件依赖于[.c]文件，"rm-f $@"的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件"$<"，也就是[.c]文件生成依赖文件，"$@"表示模式"%.d"文件，如果有一个 C 文件是 name.c，那么"%"就是"name"，"

"意为一个随机编号，第二行生成的文件有可能是"name.d.12345"，第三行使用 sed 命令做了一个替换，关于 sed 命令的用法请参看相关的使用文档。第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依赖关系：

```
main.o : main.c defs.h
```

转成：

```
main.o main.d : main.c defs.h
```

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主 Makefile 中。我们可以使用 Makefile 的"include"命令，来引入别的 Makefile 文件（前面讲过），例如：

```
sources = foo.c bar.c


include $(sources:.c=.d)
```

上述语句中的"$(sources:.c=.d)"中的".c=.d"的意思是做一个替换，把变量$(sources)所有[.c]的字串都替换成[.d]，关于这个"替换"的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为 include 是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标

4 Makefile 书写命令

每条规则中的命令和操作系统 Shell 的命令行是一致的。make 会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令

行之间中的空格或是空行会被忽略，但是如果该空格或空行是以 Tab 键开头的，那么 make 会认为其

是一个空命令。

我们在 UNIX 下可能会使用不同的 Shell，但是 make 的命令默认是被"/bin/sh"——UNIX 的标准 Shell 解释执行的。除非你特别指定一个其它的 Shell。Makefile 中，"#"是注释符，很像 C/C++中的"//"，其后的本行字符都被注释。

4.1 显示命令

通常，make 会把其要执行的命令行在命令执行前输出到屏幕上。当我们用"@"字符在命令行前，那么，这个命令将不被 make 显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

```
@echo 正在编译 XXX 模块......
```

当 make 执行时，会输出"正在编译 XXX 模块......"字串，但不会输出命令，如果没有"@"，那么，make 将输出：

```
echo 正在编译 XXX 模块......
正在编译 XXX 模块......
```

如果 make 执行时，带入 make 参数"-n"或"--just-print"，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的 Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而 make 参数"-s"或"--slient"则是全面禁止命令的显示。

4.2 命令执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make 会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是 cd 命令，你希望第二条命令得在 cd 之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

```
exec:
        cd /home/hchen
        pwd
```

示例二：

```
exec:
        cd /home/hchen; pwd
```

当我们执行"make exec"时，第一个例子中的 cd 没有作用，pwd 会打印出当前的 Makefile 目录，而第二个例子中，cd 就起作用了，pwd 会打印出"/home/hchen"。

make 一般是使用环境变量 SHELL 中所定义的系统 Shell 来执行命令，默认情况下使用 UNIX 的标准 Shell——/bin/sh 来执行命令。但在 MS-DOS 下有点特殊，因为 MS-DOS 下没有 SHELL 环境变量，当然你也可以指定。如果你指定了 UNIX 风格的目

录形式，首先，make 会在 SHELL 所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在 PATH 环境变量中所定义的所有路径中寻找。MS-DOS 中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如".exe"、".com"、".bat"、".sh"等后缀。

4.3 命令出错

每当命令运行完后，make 会检测每个命令的返回码，如果命令返回成功，那么 make 会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如 mkdir 命令，我们一定需要建立一个目录，如果目录不存在，那么 mkdir 就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用 mkdir 的意思就是一定要有这样的一个目录，于是我们就不希望 mkdir 出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在 Makefile 的命令行前加一个减号"-"（在 Tab 键之后），标记为不管命令出不出错都认为是成功的。如：

```
clean:
        -rm -f *.o
```

还有一个全局的办法是，给 make 加上"-i"或是"--ignore-errors"参数，那么，Makefile 中所有命令都会忽略错误。而如果一个规则是以".IGNORE"作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的 make 的参数的是"-k"或是"--keep-going"，这个参数的意思是，如果某规则中的命令出错了，那么就终目该规则的执行，但继续执行其它规则。

4.4 嵌套执行 make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的 Makefile，这有利于让我们的 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个 Makefile 中，这样会很难维护我们的 Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile 文件，来指明了这个目录下文件的编译规则。那么我们总控的 Makefile 可以这样书写：

```
subsystem:
        cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
        $(MAKE) -C subdir
```

定义$(MAKE)宏变量的意思是，也许我们的 make 需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入"subdir"目录，然后执行 make 命令。

我们把这个 Makefile 叫做"总控 Makefile"，总控 Makefile 的变量可以传递到下级的 Makefile 中（如果你显示的声明），但是不会覆盖下层的 Makefile 中所定义的变量，除非指定了"-e"参数。

如果你要传递变量到下级 Makefile 中，那么你可以使用这样的声明：

export<variable ...>

如果你不想让某些变量传递到下级 Makefile 中，那么你可以这样声明：

unexport<variable ...>

如：

示例一：

```
export variable = value
```

其等价于：

```
variable = value
export variable
```

其等价于：

```
export variable := value
```

其等价于：

```
variable := value
export variable
```

示例二：

```
export variable += value
```

其等价于：

variable += value

export variable

如果你要传递所有的变量，那么，只要一个 export 就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到下层 Makefile 中，特别是 MAKEFILES 变量，其中包含了 make 的参数信息，如果我们执行"总控 Makefile"时有 make 参数或是在上层 Makefile 中定义了这个变量，那么 MAKEFILES 变量将会是这些参数，并会传递到下层 Makefile 中，这是一个系统级的环境变量。

但是 make 命令中的有几个参数并不往下传递，它们是"-C","-f","-h""-o"和"-W"（有关 Makefile 参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

```
subsystem:
        cd subdir && $(MAKE) MAKEFLAGS=
```

如果你定义了环境变量 MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有"-t","-n",和"-q"参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在"嵌套执行"中比较有用的参数，"-w"或是"--print-directory"会在 make 的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级 make 目录是"/home/hchen/gnu/make"，如果我们使用"make -w"来执行，那么当进入该目录时，我们会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层 make 后离开目录时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用"-C"参数来指定 make 下层 Makefile 时，"-w"会被自动打开的。如果参数中有"-s"（"--slient"）或是"--no-print-directory"，那么，"-w"总是失效的。

4.5 定义命令包

如果 Makefile 中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以"define"开始，以"endef"结束，如：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

这里，"run-yacc"是这个命令包的名字，其不要和 Makefile 中的变量重名。在"define"和"endef"中的两行就是命令序列。这个命令包中的第一个命令是运行 Yacc 程序，因为 Yacc 程序总是生成"y.tab.c"的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

```
foo.c : foo.y
        $(run-yacc)
```

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包"run-yacc"中的"$^"就是"foo.y"，"$@"就是"foo.c"（有关这种以"$"开头的特殊变量，我们会在后面介绍），make 在执行命令包时，命令包中的每个命令会被依次独立执行。

5 使用变量

在 Makefile 中的定义的变量，就像是 C/C++语言中的宏一样，他代表了一个文本字串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在"目标"，"依赖目标"，"命令"或是 Makefile 的其它部分中。变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有":"、"#"、"="或是空字符（空格、回车等）。变量是大小写敏感的，"foo"、"Foo"和"FOO"是三个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。有一些变量是很奇怪字串，如"$<"、"$@"等，这些是自动化变量，我会在后面介绍。

一、变量的基础

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。

先看一个例子：

```
objects = program.o foo.o utils.o
program : $(objects)
cc -o program $(objects)

$(objects) : defs.h
```

变量会在使用它的地方精确地展开，就像 C/C++中的宏一样，例如：

```
foo = c
prog.o : prog.$(foo)
$(foo)$(foo) -$(foo) prog.$(foo)
```

展开后得到：

```
prog.o : prog.c
cc -c prog.c
```

当然，千万不要在你的 Makefile 中这样干，这里只是举个例子来表明 Makefile 中的变量在使用处展开的真实样子。可见其就是一个"替代"的原理。另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

二、变量中的变量

在定义变量的值时，我们可以使用其它变量来构造变量的值，在 Makefile 中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用"="号，在"="左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好

的值，其也可以使用后面定义的值。如：

```
foo = $(bar)

bar = $(ugh)

ugh = Huh?



all:

echo $(foo)
```

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，

$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

```
CFLAGS = $(include_dirs) -O

include_dirs = -Ifoo -Ibar
```

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这种形式也有不好的地方

，那就是递归定义，如：

```
CFLAGS = $(CFLAGS) -O
```

或：

```
A = $(B)

B = $(A)
```

这会让 make 陷入无限的变量展开过程中去，当然，我们的 make 是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的 make 运行时非常慢，更糟糕的是，他会使用得两个 make 的函数“wildcard”和“shell”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

为了避免上面的这种方法，我们可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：

```
x := foo
y := $(x) bar
x := later
```

其等价于：

```
y := foo bar
x := later
```

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，y 的值是“bar”，而不是“foo bar”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 make 的函数、条件表达式和一个系统变量“MAKELEVEL”的使用：

```
ifeq (0,${MAKELEVEL})
cur-dir := $(shell pwd)
whoami := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

关于条件表达式和函数，我们在后面再说，对于系统变量“MAKELEVEL”，其意思是，如果我们的make 有一个嵌套执行的动作（参见前面的“嵌套使用 make”），那么，这个变量会记录了我们的当前 Makefile 的调用层数。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：

```
nullstring :=
space := $(nullstring) # end of the line
```

nullstring 是一个 Empty 变量，其中什么也没有，而我们的 space 的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个 Empty 变量来标明变量的值开始了，而后面采用“#”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“#”的使用，注释符“#”的这种特性值得我们注意，如果我们这样定义一个变量：

```
dir := /foo/bar # directory to put the frobs in
```

dir 这个变量的值是“/foo/bar”，后面还跟了 4 个空格，如果我们这样使用这样变量来指定别的目录——“$(dir)/file”那么就完蛋了。

还有一个比较有用的操作符是“?=”，先看示例：

```
FOO ?= bar
```

其含义是，如果 FOO 没有被定义过，那么变量 FOO 的值就是“bar”，如果 FOO 先前被定义过，那么这条语将什么也不做，其等价于：

```
ifeq ($(origin FOO), undefined)
FOO = bar
endif
```

三、变量高级用法

这里介绍两种变量的高级使用方法，第一种是变量值的替换。

我们可以替换变量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

第二种高级用法是——“把变量的值再当成变量”。先看一个例子：

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

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

我们还可以使用更多的层次：

```
x = y
y = z
z = u
a := $($($(x)))
```

这里的$(a)的值是“u”，相关的推导留给读者自己去做吧。

让我们再复杂一点，使用上“在变量定义中使用变量”的第一个方式，来看一个例子：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(z)，也就是“Hello”。

再复杂一点，我们再加上函数：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

这个例子中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst 函数把“variable1”中的所有“1”字串替换成“2”字串，于是，“variable1”变成“variable2”，再取其值，所以，最终，$(a)的值就是$(variable2)的值—— “Hello”。（喔，好不容易）

在这种方式中，或要可以使用多个变量来组成一个变量的名字，然后再取其值：

```
first_second = Hello

a = first

b = second

all = $($a_$b)
```

这里的“$a_$b”组成了“first_second”，于是，$(all)的值就是“Hello”。

再来看看结合第一种技术的例子：

```
a_objects := a.o b.o c.o

1_objects := 1.o 2.o 3.o
```

```
sources := $($(a1)_objects:.o=.c)
```

这个例子中，如果$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这种技术和“函数”与“条件语句”一同使用的例子：

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

这个示例中，如果定义了“do_sort”，那么：foo := $(sort a d b g q c)，于是$(foo)的值就是“a b c d g q”，而如果没有定义“do_sort”，那么：foo := $(sort a d bg q c)，调用的就是 strip 函数。

当然，“把变量的值再当成变量”这种技术，同样可以用在操作符的左边：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
```

```
define $(dir)_print
lpr $($(dir)_sources)
endef
```

这个例子中定义了三个变量："dir"，"foo_sources"和"foo_print"。

四、追加变量值

我们可以使用"+="操作符给变量追加值，如：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

于是，我们的$(objects)值变成："main.o foo.o bar.o utils.o another.o"（another.o 被追加进去了）

使用"+="操作符，可以模拟为下面的这种例子：

```
objects = main.o foo.o bar.o utils.o
```

objects := $(objects) another.o

所不同的是，用"+="更为简洁。

如果变量之前没有定义过，那么，"+="会自动变成"="，如果前面有变量定义，那么"+="会继承于前次操作的赋值符。如果前一次的是":="，那么"+="会以":="作为其赋值符，如：

variable := value

variable += more

等价于：

variable := value

variable := $(variable) more

但如果是这种情况：

variable = value

variable += more

由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以 make 会自动为我们解决这个问题，我们不必担心这个问题。

五、override 指示符

如果有变量是通常 make 的命令行参数设置的，那么 Makefile 中对这个变量的赋值会被忽略。如果你想在 Makefile 中设置这类参数的值，那么，你可以使用“override”指示符。其语法是：

override <variable> = <value>

override <variable> := <value>

当然，你还可以追加：

override <variable> += <more text>

对于多行的变量定义，我们用 define 指示符，在 define 指示符前，也同样可以使用 ovveride 指示符，如：

```
override define foo
bar
endef
```

六、多行变量

还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过"命令包"的技术就是利用这个关键字）。

define 指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以 endef 关键字结束。其工作方式和"="操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用 define 定义的命令变量中没有以[Tab]键开头，那么 make 就不会把其认为是命令。

下面的这个示例展示了 define 的用法：

```
define two-lines

echo foo

echo $(bar)

endef
```

七、环境变量

make 运行时的系统环境变量可以在 make 开始运行时被载入到 Makefile 文件中，但是如果 Makefile 中已定义了这个变量，或是这个变量由 make 命令行带入，那么系统的环境变量的值将被覆盖。（如果 make 指定了“-e”参数，那么，系统环境变量将覆盖 Makefile 中定义的变量）

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的 Makefile 中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果 Makefile 中定义了 CFLAGS，那么则会使用 Makefile 中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。　　当 make 嵌套调用时（参见前面的“嵌套调用”章节），上层 Makefile 中定义的变量会以系统环境变量的方式传递到下层的 Makefile 中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层 Makefile 传递，则需要使用 exprot 关键字来声明。（参见前面章节）

当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的 Makefile 时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

八、目标变量

前面我们所讲的在 Makefile 中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。当然，“自动化变量”除外，如“$<”等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我样同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

```
<target ...> : <variable-assignment>

<target ...> : overide <variable-assignment>
```

<variable-assignment>可以是前面讲过的各种赋值表达式，如"="、":="、"+="或是"？="。第二个语法是针对于 make 命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如：

```
prog : CFLAGS = -g

prog : prog.o foo.o bar.o

$(CC) $(CFLAGS) prog.o foo.o bar.o



prog.o : prog.c

$(CC) $(CFLAGS) prog.c



foo.o : foo.c

$(CC) $(CFLAGS) foo.c



bar.o : bar.c

$(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的$(CFLAGS)的值是什么，在 prog 目标，以及其所引发的所有规则中（prog.o foo.o bar.o 的规则），$(CFLAGS)的值都是“-g”

九、模式变量

在 GNU 的 make 中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make 的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

```
%.o : CFLAGS = -O
```

同样，模式变量的语法和“目标变量”一样：

<pattern ...> : <variable-assignment>

<pattern ...> : override <variable-assignment>

override 同样是针对于系统环境传入的变量，或是 make 命令行指定的变量。

使用条件判断

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

一、示例

下面的例子，判断$(CC)变量是否"gcc"，如果是的话，则使用 GNU 函数编译目标。

```
libs_for_gcc = -lgnu

normal_libs =



foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs_for_gcc)

else

$(CC) -o foo $(objects) $(normal_libs)

endif
```

可见，在上面示例的这个规则中，目标"foo"可以根据变量"$(CC)"值来选取不同的函数库来编译程序。

我们可以从上面的示例中看到三个关键字：ifeq、else 和 endif。ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else 表示条件表达式为假的情况。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。

当我们的变量$(CC)值是"gcc"时，目标 foo 的规则是：

```
foo: $(objects)
```

$(CC) -o foo $(objects) $(libs_for_gcc)

而当我们的变量$(CC)值不是"gcc"时（比如"cc"），目标 foo 的规则是：

foo: $(objects)

$(CC) -o foo $(objects) $(normal_libs)

当然，我们还可以把上面的那个例子写得更简洁一些：

libs_for_gcc = -lgnu

normal_libs =

ifeq ($(CC),gcc)

libs=$(libs_for_gcc)

else

libs=$(normal_libs)

endif

foo: $(objects)

$(CC) -o foo $(objects) $(libs)

二、语法

条件表达式的语法为：

<conditional-directive>

<text-if-true>

endif

以及：

<conditional-directive>

<text-if-true>

else

<text-if-false>

endif

其中<conditional-directive>表示条件关键字，如“ifeq”。这个关键字有四个。

第一个是我们前面所见过的“ifeq”

```
ifeq (<arg1>, <arg2> )
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"
```

比较参数“arg1”和“arg2”的值是否相同。当然，参数中我们还可以使用 make 的函数。如：

```
ifeq ($(strip $(foo)),)
<text-if-empty>
endif
```

这个示例中使用了“strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件关键字是“ifneq”。语法是：

```
ifneq (<arg1>, <arg2> )
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"
```

其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。和“ifeq”类似。

第三个条件关键字是“ifdef”。语法是：

```
ifdef <variable-name>
```

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。注意，ifdef 只是测试一个变量是否有值，其并不会把变量扩展到当前位置。还是来看两个例子：

示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件关键字是“ifndef”。其语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也

一样，只要不是以[Tab]键开始就行了。

特别注意的是，make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“$@”等）放入条件表达式中，因为自动化变量是在运行时才有的。

而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

6 使用函数

在 Makefile 中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。

make 所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

一、函数的调用语法

函数调用，很像变量的使用，也是以“$”来标识的，其语法如下：

```
$(<function> <arguments> )
```

或是

${<function> <arguments>}

这里，<function>就是函数名，make 支持的函数不多。<arguments>是函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b,${x})”的形式。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)

的值是“a b c”，$(bar)的定义用，调用了函数“subst”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“

a,b,c”。

二、字符串处理函数

$(subst <from>,<to>,<text>)

名称：字符串替换函数——subst。

功能：把字串<text>中的<from>字符串替换成<to>。

返回：函数返回被替换过后的字符串。

示例：

$(subst ee,EE,feet on the street)，

把“feet on the street”中的“ee”替换成“EE”，返回结果是“fEEt on the strEEt

”。

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数——patsubst。

功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“₩”来转义，以“₩%”来表示真实含义的“%”字符）返回：函数返回被替换过后的字符串。

示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

备注：

这和我们前面"变量章节"说过的相关知识有点相似。如：

"$(var:<pattern>=<replacement>)"

相当于

"$(patsubst <pattern>,<replacement>,$(var))"，

而"$(var: <suffix>=<replacement>)"

则相当于

"$(patsubst %<suffix>,%<replacement>,$(var))"。

例如有：objects = foo.o bar.o baz.o，

那么，"$(objects:.o=.c)"和"$(patsubst %.o,%.c,$(objects))"是一样的。

$(strip <string>)

名称：去空格函数——strip。

功能：去掉<string>字串中开头和结尾的空字符。

返回：返回被去掉空格的字符串值。
示例：

```
$(strip a b c )
```

把字串“a b c ”去到开头和结尾的空格，结果是“a b c”。

```
$(findstring <find>,<in> )
```

名称：查找字符串函数——findstring。

功能：在字串<in>中查找<find>字串。

返回：如果找到，那么返回<find>，否则返回空字符串。

示例：

```
$(findstring a,a b c)
```

```
$(findstring a,b c)
```

第一个函数返回“a”字符串，第二个返回“”字符串（空字符串）

$(filter <pattern...>,<text>)

名称：过滤函数——filter。

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>的字串。

示例：

```
sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo
```

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

$(filter-out <pattern...>,<text>)

名称：反过滤函数——filter-out。

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

返回：返回不符合模式<pattern>的字串。

示例：

objects=main1.o foo.o main2.o bar.o

mains=main1.o main2.o

$(filter-out $(mains),$(objects)) 返回值是"foo.o bar.o"。

$(sort <list>)

名称：排序函数——sort。

功能：给字符串<list>中的单词排序（升序）。

返回：返回排序后的字符串。

示例：$(sort foo bar lose)返回"bar foo lose" 。

备注：sort 函数会去掉<list>中相同的单词。

$(word <n>,<text>)

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。

示例：$(word 2, foo bar baz)返回值是“bar”。

$(wordlist <s>,<e>,<text>)

名称：取单词串函数——wordlist。

功能：从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。

示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text>)

名称：单词个数统计函数——words。

功能：统计<text>中字符串中的单词个数。

返回：返回<text>中的单词数。

示例：$(words, foo bar baz)返回值是"3"。

备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words <text>),<text>)。

$(firstword <text>)

名称：首单词函数——firstword。

功能：取字符串<text>中的第一个单词。

返回：返回字符串<text>的第一个单词。

示例：$(firstword foo bar)返回值是"foo"。

备注：这个函数可以用 word 函数来实现：$(word 1,<text>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里，举一个现实中应用的例子。我们知道，make 使用"VPATH"变量来指定"依赖文件"的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数 CFLAGS，如：

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

如果我们的"$(VPATH)"值是"src:../headers"，那么"$(patsubst %,-I%,$(subst :

, ,$(VPATH)))"将返回"-Isrc -I../headers"，这正是 cc 或 gcc 搜索头文件路径的参数

。

三、文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（"/"）之

前的部分。如果没有反斜杠，那么返回“./”。

返回：返回文件名序列<names>的目录部分。

示例： $(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“/”）之后的部分。

返回：返回文件名序列<names>的非目录部分。

示例： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列<names>中取出各个文件名的后缀。

返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。

示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列<names>中取出各个文件名的前缀部分。

返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。

示例：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar h

acks”。

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀<suffix>加到<names>中的每个单词后面。

返回：返回加过后缀的文件名序列。

示例：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

示例：$(addprefix src/,foo bar)返回值是"src/foo src/bar"。

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

返回：返回连接过后的字符串。

示例：$(join aaa bbb , 111 222 333)返回值是"aaa111 bbb222 333"。

四、foreach 函数

foreach 函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile 中的foreach 函数几乎是仿照于 Unix 标准 Shell（/bin /sh）中的 for 语句，或是 C-Shell（/bin /csh）中的 foreach 语句而构建的。它的语法是：

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach 函数的返回值。

所以，<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用<var>

这个参数来依次枚举<list>中的单词。举个例子：

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为 foreach 函数的返回，所以，$(f

iles)的值是“a.o b.o c.o d.o”。

注意，foreach 中的<var>参数是一个临时的局部变量，foreach 函数执行完后，参数<var>的变量将不在作用，其作用域只在 foreach 函数当中。

五、if 函数

if 函数很像 GNU 的 make 所支持的条件语句——ifeq（参见前面所述的章节），if 函数的语法是：

```
$(if <condition>,<then-part> )
```

或是

```
$(if <condition>,<then-part>,<else-part> )
```

可见，if 函数可以包含"else"部分，或是不含。即 if 函数的参数可以是两个，也可以是三个。<condition>参数是 if 的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part> 会被计算。

而 if 函数的返回值是，如果<condition>为真（非空字符串），那个<then- part>会是整个函数的返回值，如果<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时如果<else-part>没有被定义，那么，整个函数返回空字串。

所以，<then-part>和<else-part>只会有一个被计算。

六、call 函数

call 函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用 call 函数来向这个表达式传递参数。其语法是：

```
$(call <expression>,<parm1>,<parm2>,<parm3>...)
```

当 make 执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。例如：

```
reverse = $(1) $(2)

foo = $(call reverse,a,b)
```

那么，foo 的值就是"a b"。当然，参数的次序是可以自定义的，不一定是顺序的，如：

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
```

此时的 foo 的值就是"b a"。

七、origin 函数

origin 函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的？其语法是：

```
$(origin <variable> )
```

注意，<variable>是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。

Origin 函数会以其返回值来告诉你这个变量的“出生情况”，下面，是 origin 函

数的返回值:

“undefined”

如果<variable>从来没有定义过，origin 函数返回这个值“undefined”。

“default”

如果<variable>是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。

“environment”

如果<variable>是一个环境变量，并且当 Makefile 被执行时，“-e”参数没有被打开。

“file”

如果<variable>这个变量被定义在 Makefile 中。

“command line”

如果<variable>这个变量是被命令行定义的。

“override”

如果<variable>是被 override 指示符重新定义的。

“automatic”

如果<variable>是一个命令运行中的自动化变量。关于自动化变量将在后面讲述。

这些信息对于我们编写 Makefile 是非常有用的，例如，假设我们有一个 Makefile 其包了一个定义文件 Make.def，在 Make.def 中定义了一个变量“bletch”，而我们的环境中也有一

个环境变量“bletch”，此时，我们想判断一下，如果变量来源于环境，那么我们就把之重定义了，如果来源于 Make.def 或是命令行等非环境的，那么我们就不重新定义它。于是

，在我们的 Makefile 中，我们可以这样写：

```
ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif
```

当然，你也许会说，使用 override 关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，我们用 override 是可以达到这样的效果，可是 override 过于粗

暴，它同时会把从命令行定义的变量也覆盖了，而我们只想重新定义环境传来的，而不想重新定义命令行传来的。

八、shell 函数

shell 函数也不像其它的函数。顾名思义，它的参数应该就是操作系统 Shell 的命令。它和反引号"`"是相同的功能。这就是说，shell 函数把执行操作系统命令后的输出作为函数

返回。于是，我们可以用操作系统命令以及字符串处理命令 awk，sed 等等命令来生成一个变量，如：

```
contents := $(shell cat foo)
```

```
files := $(shell echo *.c)
```

注意，这个函数会新生成一个 Shell 程序来执行命令，所以你要注意其运行性能，如果你的 Makefile 中有一些比较复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是 Makefile 的隐晦的规则可能会让你的 shell 函数执行的次数比你想像的多得多。

九、控制 make 的函数

make 提供了一些函数来控制 make 的运行。通常，你需要检测一些运行 Makefile 时的运行时信息，并且根据这些信息来决定，你是让 make 继续执行，还是停止。

$(error <text ...>)

产生一个致命的错误，<text ...>是错误信息。注意，error 函数不会在一被使用就会产生错误信息，所以如果你把其定义在某个变量中，并在后续的脚本中使用这个变量，那么也是可以的。例如：

示例一：

```
ifdef ERROR_001

$(error error is $(ERROR_001))

endif
```

示例二：

ERR = $(error found an error!)

.PHONY: err

err: ; $(ERR)

示例一会在变量 ERROR_001 定义了后执行时产生 error 调用，而示例二则在目录 err 被执行时才发生 error 调用。

$(warning <text ...>)

这个函数很像 error 函数，只是它并不会让 make 退出，只是输出一段警告信息，而 make 继续执行。

7 make 的运行

一般来说，最简单的就是直接在命令行下输入 make 命令，make 命令会找当前目录的 makefile 来执行，一切都是自动的。但也有时你也许只想让 make 重编译某些文件，而不是整个工程，而又有的时候你有几套编译规则，你想在不同的时候使用不同的编译规则，等等。本章节就是讲述如何使用 make 命令的。

一、make 的退出码

make 命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果 make 运行时出现任何错误，其返回 1。

2 —— 如果你使用了 make 的"-q"选项，并且 make 使得一些目标不需要更新，那么返回 2。

Make 的相关参数我们会在后续章节中讲述。

二、指定 Makefile

前面我们说过，GNU make 找寻默认的 Makefile 的规则是在当前目录下依次找三个文件——"GNUmakefile"、"makefile"和"Makefile"。其按顺序找这三个文件，一旦找到，就开始读取这个文件并执行。

当前，我们也可以给 make 命令指定一个特殊名字的 Makefile。要达到这个功能，我们要使用 make 的"-f"或是"--file"参数（"-- makefile"参数也行）。例如，我们有个 makefile 的名字是"hchen.mk"，那么，我们可以这样来让 make 来执行这个文件：

make –f hchen.mk

如果在 make 的命令行是，你不只一次地使用了"-f"参数，那么，所有指定的 makefile 将会被连在一起传递给 make 执行。

三、指定目标

一般来说，make 的最终目标是 makefile 中的第一个目标，而其它目标一般是由这个目标连带出来的。这是 make 的默认行为。当然，一般来说，你的 makefile 中的第一个目标是由许多个目标组成，你可以指示 make，让其完成你所指定的目标。要达到这一目的很简单，需在 make 命令后直接跟目标的名字就可以完成（如前面提到的"make clean"形式）任何在 makefile 中的目标都可以被指定成终极目标，但是除了以"- "打头，或是包含了"="的目标，因为有这些字符的目标，会被解析成命令行参数或是变量。甚至没有被我们明确写出来的目标也可以成为 make 的终极目标，也就是说，只要 make 可以找到其隐含规则推导规则，那么这个隐含目标同样可以被指定成终极目标。

有一个 make 的环境变量叫"MAKECMDGOALS"，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

```
sources = foo.c bar.c

ifneq ( $(MAKECMDGOALS),clean)

include $(sources:.c=.d)

endif
```

基于上面的这个例子，只要我们输入的命令不是"make clean"，那么 makefile 会自动包含"foo.d"和"bar.d"这两个 makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

```
.PHONY: all

all: prog1 prog2 prog3 prog4
```

从这个例子中，我们可以看到，这个 makefile 中有四个需要编译的程序——“prog1”，“prog2”，“prog3”和“prog4”，我们可以使用“make all”命令来编译所有的目标

（如果把 all 置成第一个目标，那么只需执行“make”），我们也可以使用“make prog2”来单独编译目标“prog2”。

即然 make 可以指定所有 makefile 中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的 makefile 根据指定的不同的目标来完成不同的事。在 Unix 世界中，软件

发布时，特别是 GNU 这种开源软件的发布时，其 makefile 都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的 makefile 中的目标。

“all” 这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

“clean” 这个伪目标功能是删除所有被 make 创建的文件。

“install” 这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。

“print” 这个伪目标的功能是例出改变过的源文件。

“tar” 这个伪目标功能是把源程序打包备份。也就是一个 tar 文件。

“dist” 这个伪目标功能是创建一个压缩文件，一般是把 tar 文件压成 Z 文件。或是 gz 文件。

“TAGS” 这个伪目标功能是更新所有的目标，以备完整地重编译使用。

“check”和“test” 这两个伪目标一般用来测试 makefile 的流程。

当然一个项目的 makefile 中也不一定要书写这样的目标，这些东西都是 GNU 的东西，但是我想，GNU 搞出这些东西一定有其可取之处（等你的 UNIX 下的程序文件一多时你就会发现这些功能很有用了），这里只不过是说明了，如果你要书写这种功能，最好使用这种名字命名你的目标，这样规范一些，规范的好处就是——不用解释，大家都明白。而且如果你的 makefile 中有这些功能，一是很实用，二是可以显得你的 makefile 很专业（不是那种初学者的作品）。

四、检查规则

有时候，我们不想让我们的 makefile 中的规则执行起来，我们只想检查一下我们的命令，或是执行的序列。于是我们可以使用 make 命令的下述参数：

“-n”

“--just-print”

“--dry-run”

“--recon”

不执行参数，这些参数只是打印命令，不管目标是否更新，把规则和连带规则下的命令打印出来，但不执行，这些参数对于我们调试 makefile 很有用处。

“-t”

“--touch”

这个参数的意思就是把目标文件的时间更新，但不更改目标文件。也就是说，make 假装编译目标，但不是真正的编译目标，只是把目标变成已编译过的状态。

“-q”

“--question”

这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出，当然也不会执行编译，如果目标不存在，其会打印出一条出错信息。

“-W <file>”

“--what-if=<file>”

“--assume-new=<file>”

“--new-file=<file>”

这个参数需要指定一个文件。一般是是源文件（或依赖文件），Make 会根据规则推导来运行依赖于这个文件的命令，一般来说，可以和“-n”参数一同使用，来查看这个依赖文件

所发生的规则命令。

另外一个很有意思的用法是结合“-p”和“-v”来输出 makefile 被执行时的信息(这个将在后面讲述)。

五、make 的参数

下面列举了所有 GNU make 3.80 版的参数定义。其它版本和产商的 make 大同小异，不过其它产商的 make 的具体参数还是请参考各自的产品文档。

“-b”

“-m”

这两个参数的作用是忽略和其它版本 make 的兼容性。

“-B”

“--always-make”

认为所有的目标都需要更新（重编译）。

“-C <dir>”

“--directory=<dir>”

指定读取 makefile 的目录。如果有多个“-C”参数，make 的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。如：“make –C ~hchen/test –C prog”

等价于“make –C ~hchen/test/prog”。

“—debug[=<options>]”

输出 make 的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —— 也就是 all，输出所有的调试信息。（会非常的多）

b —— 也就是 basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —— 也就是 verbose，在 b 选项的级别之上。输出的信息包括哪个 makefile 被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —— 也就是 implicit，输出所以的隐含规则。

j —— 也就是 jobs，输出执行规则中命令的详细信息，如命令的 PID、返回码等。

m —— 也就是 makefile，输出 make 读取 makefile，更新 makefile，执行 makefile 的信息。

“-d”

相当于“--debug=a”。

“-e”

“--environment-overrides”

指明环境变量的值覆盖 makefile 中定义的变量的值。

“-f=<file>”

“--file=<file>”

“--makefile=<file>”

指定需要执行的 makefile。

“-h”

“--help”

显示帮助信息。

“-i”

“--ignore-errors”

在执行时忽略所有的错误。

“-I <dir>”

“--include-dir=<dir>”

指定一个被包含 makefile 的搜索目标。可以使用多个“-I”参数来指定多个目录。

“-j [<jobsnum>]”

“--jobs[=<jobsnum>]”

指同时运行命令的个数。如果没有这个参数，make 运行命令时能运行多少就运行多少。如果有一个以上的“-j”参数，那么仅最后一个“-j”才是有效的。（注意这个参数在 MS-D

OS 中是无用的）

“-k”

“--keep-going”

出错也不停止运行。如果生成一个目标失败了，那么依赖于其上的目标就不会被执行了。

“-l <load>”

“--load-average[=<load]”

“—max-load[=<load>]”

指定 make 运行命令的负载。

“-n”

“--just-print”

“--dry-run”

“--recon”

仅输出执行过程中的命令序列，但并不执行。

“-o <file>”

“--old-file=<file>”

“--assume-old=<file>”

不重新生成的指定的<file>，即使这个目标的依赖文件新于它。

“-p”

“--print-data-base”

输出 makefile 中的所有数据，包括所有的规则和变量。这个参数会让一个简单的 makefile 都会输出一堆信息。如果你只是想输出信息而不想执行 makefile，你可以使用“make -q

p”命令。如果你想查看执行 makefile 前的预设变量和规则，你可以使用“make –p –f /dev/null“。这个参数输出的信息会包含着你的 makefile 文件的文件名和行号，所以，用

这个参数来调试你的 makefile 会是很有用的，特别是当你的环境变量很复杂的时候。

“-q”

“--question”

不运行命令，也不输出。仅仅是检查所指定的目标是否需要更新。如果是 0 则说明要更新，如果是 2 则说明有错误发生。

“-r”

“--no-builtin-rules”

禁止 make 使用任何隐含规则。

“-R”

“--no-builtin-variabes”

禁止 make 使用任何作用于变量上的隐含规则。

“-s”

“--silent”

“--quiet”

在命令运行时不输出命令的输出。

“-S”

“--no-keep-going”

“--stop”

取消“-k”选项的作用。因为有些时候，make 的选项是从环境变量“MAKEFLAGS”中继承下来的。所以你可以在命令行中使用这个参数来让环境变量中的“-k”选项失效。

“-t”

"--touch"

相当于 UNIX 的 touch 命令，只是把目标的修改日期变成最新的，也就是阻止生成目标的命令运行。

"-v"

"--version"

输出 make 程序的版本、版权等关于 make 的信息。

"-w"

"--print-directory"

输出运行 makefile 之前和之后的信息。这个参数对于跟踪嵌套式调用 make 时很有用。

"--no-print-directory"

禁止"-w"选项。

“-W <file>”

“--what-if=<file>”

“--new-file=<file>”

“--assume-file=<file>”

假定目标<file>需要更新，如果和“-n”选项使用，那么这个参数会输出该目标更新时的运行动作。如果没有“-n”那么就像运行 UNIX 的“touch”命令一样，使得<file>的修改时

间为当前时间。

“--warn-undefined-variables”

只要 make 发现有未定义的变量，那么就输出警告信息。

8 隐含规则

在我们使用 Makefile 时，有一些我们会经常使用，而且使用频率非常高的东西，比如，我们编译 C/C++的源程序为中间目标文件（Unix 下是[.o] 文件，Windows 下是[.obj]文件）。本章讲述的就是一些在 Makefile 中的“隐含的”，早先约定了的，不需要我们再写出来的规则。

“隐含规则”也就是一种惯例，make 会按照这种“惯例”心照不喧地来运行，那怕我们的 Makefile 中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就

不用写出来，make 会自动推导出这种规则，并生成我们需要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。

我们还可以通过“模式规则”的方式写下自己的隐含规则。用“后缀规则”来定义隐含规则会有许多的限制。使用“模式规则”会更回得智能和清楚，但“后缀规则”可以用来保

证我们 Makefile 的兼容性。

我们了解了“隐含规则”，可以让其为我们更好的服务，也会让我们知道一些“约定俗成”了的东西，而不至于使得我们在运行 Makefile 时出现一些我们觉得莫名其妙的东西。当

然，任何事物都是矛盾的，水能载舟，亦可覆舟，所以，有时候“隐含规则”也会给我们造成不小的麻烦。只有了解了它，我们才能更好地使用它。

一、使用隐含规则

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make 会试图去自动推导产生这个目标的规则和命令，如果 make 可以自动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自动推导。当然，隐含规则是 make 事先约定好的一些东西。例如，我们有下面的一个 Makefile：

```
foo : foo.o bar.o

cc –o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

我们可以注意到，这个 Makefile 中并没有写下如何生成 foo.o 和 bar.o 这两目标的规则和命令。因为 make 的"隐含规则"功能会自动为我们自动去推导这两个目标的依赖目标和生成
命令。

make 会在自己的"隐含规则"库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make 调用的隐含规则是，把 [.o]的目标的依赖文件置成[.c]，并使用 C 的编译命令"cc –c $(CFLAGS) [.c]"来生成[.o]的目标。也就是说，我们完全没有必要写下下面的两条规则：

```
foo.o : foo.c

cc –c foo.c $(CFLAGS)
```

bar.o : bar.c

cc –c bar.c $(CFLAGS)

因为，这已经是"约定"好了的事了，make 和我们约定好了用 C 编译器"cc"生成[.o]文件的规则，这就是隐含规则。

当然，如果我们为[.o]文件书写了自己的规则，那么 make 就不会自动推导并调用隐含规则，它会按照我们写好的规则忠实地执行。

还有，在 make 的"隐含规则库"中，每一条隐含规则都在库中有其顺序，越靠前的则是越被经常使用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make 也不会管。如下面这条规则（没有命令）：

foo.o : foo.p

依赖文件"foo.p"（Pascal 程序的源文件）有可能变得没有意义。如果目录下存在了"foo.c"文件，那么我们的隐含规则一样会生效，并会通过 "foo.c"调用 C 的编译器生成 f

oo.o 文件。因为，在隐含规则中，Pascal 的规则出现在 C 的规则之后，所以，make 找到可以生成 foo.o 的 C 的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“依赖规则”，而不写命令。

二、隐含规则一览

这里我们将讲述所有预先设置（也就是 make 内建）的隐含规则，如果我们不明确地写下规则，那么，make 就会在这些规则中寻找所需要规则和命令。当然，我们也可以使用 make 的参数“-r”或“--no-builtin-rules”选项来取消所有的预设置的隐含规则。

当然，即使是我们指定了“-r”参数，某些隐含规则还是会生效，因为有许多的隐含规则都是使用了“后缀规则”来定义的，所以，只要隐含规则中有“后缀列表 ”（也就一系统

定义在目标.SUFFIXES 的依赖目标），那么隐含规则就会生效。默认的后缀列表

是：.out,.a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .

h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el。具体的细节，我们会在后面讲述。

还是先来看一看常用的隐含规则吧。

1、编译 C 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

2、编译 C++程序的隐含规则。

“<n>.o” 的目标的依赖目标会自动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为 C++源文件的后缀，而

不是“.C”）

3、编译 Pascal 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.p”，并且其生成命令是“$(PC) –c $(PFLAGS)”。

4、编译 Fortran/Ratfor 程序的隐含规则。

“<n>.o”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”或“<n>.f”，并且其生成命令是:

“.f” “$(FC) –c $(FFLAGS)”

“.F” “$(FC) –c $(FFLAGS) $(CPPFLAGS)”

“.f” “$(FC) –c $(FFLAGS) $(RFLAGS)”

5、预处理 Fortran/Ratfor 程序的隐含规则。

“<n>.f”的目标的依赖目标会自动推导为“<n>.r”或“<n>.F”。这个规则只是转换 Ratfor 或有预处理的 Fortran 程序到一个标准的 Fortran 程序。其使用的命令是：

“.F” “$(FC) –F $(CPPFLAGS) $(FFLAGS)”

“.r” “$(FC) –F $(FFLAGS) $(RFLAGS)”

6、编译 Modula-2 程序的隐含规则。

“<n>.sym” 的目标的依赖目标会自动推导为“<n>.def”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>” 的目标的依赖目标会自动推导为“<n>.mod”，

并且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。

“<n>.o” 的目标的依赖目标会自动推导为“<n>.s”，默认使用编译品“as”，并且其生成命令是：“$(AS) $(ASFLAGS)”。“<n>.s” 的目标的依赖目标会自动推导为“<n>.S”

，默认使用 C 预编译器“cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。

8、链接 Object 文件的隐含规则。

“<n>” 目标依赖于“<n>.o”，通过运行 C 的编译器来运行链接程序生成（一般是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对

于只有一个源文件的工程有效，同时也对多个 Object 文件（由不同的源文件生成）的也有效。例如如下规则：

x : y.o z.o

并且“x.c”、“y.c”和“z.c”都存在时，隐含规则将执行如下命令：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

如果没有一个源文件（如上例中的 x.c）和你的目标名字（如上例中的 x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9、Yacc C 程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.y”（Yacc 生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10、Lex C 程序时的隐含规则。

“<n>.c”的依赖文件被自动推导为“n.l”（Lex 生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

11、Lex Ratfor 程序时的隐含规则。

“<n>.r”的依赖文件被自动推导为“n.l”（Lex 生成的文件），其生成命令是：“$(LEX

) $(LFALGS)”。

12、从 C 程序、Yacc 文件或 Lex 文件创建 Lint 库的隐含规则。

“<n>.ln”（lint 生成的文件）的依赖文件被自动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>.y”和“<n>.l”也是同样的规则。

三、隐含规则使用的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的 makefile 中改变这些变量的值，或是在 make 的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也可以利用 make 的"-R"或"--no– builtin-variables"参数来取消你所定义的变量

对隐含规则的作用。

例如，第一条隐含规则——编译 C 程序的隐含规则的命令是"$(CC) –c $(CFLAGS) $(CPPFLAGS)"。Make 默认的编译命令是"cc"，如果你把变量"$(CC)"重定义成"gcc"，把

变量"$(CFLAGS)"重定义成 "-g"，那么，隐含规则中的命令全部会以"gcc –c -g $(CPPFLAGS)"的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如"CC"；一种是参数

相的关，如"CFLAGS"。下面是所有隐含规则中会用到的变量：

1、关于命令的变量。

AR　函数库打包程序。默认命令是"ar"。

AS

汇编语言编译程序。默认命令是“as”。

CC

C 语言编译程序。默认命令是“cc”。

CXX

C++语言编译程序。默认命令是“g++”。

CO

从 RCS 文件中扩展文件程序。默认命令是“co”。

CPP

C 程序的预处理器（输出是标准输出设备）。默认命令是“$(CC) –E”。

FC

Fortran 和 Ratfor 的编译器和预处理程序。默认命令是“f77”。

GET

从 SCCS 文件中扩展文件的程序。默认命令是“get”。

LEX

Lex 方法分析器程序（针对于 C 或 Ratfor）。默认命令是“lex”。

PC

Pascal 语言编译程序。默认命令是“pc”。

YACC

Yacc 文法分析器（针对于 C 程序）。默认命令是“yacc”。

YACCR

Yacc 文法分析器（针对于 Ratfor 程序）。默认命令是“yacc –r”。

MAKEINFO

转换 Texinfo 源文件（.texi）到 Info 文件程序。默认命令是“makeinfo”。

TEX

从 TeX 源文件创建 TeX DVI 文件的程序。默认命令是“tex”。

TEXI2DVI

从 Texinfo 源文件创建军 TeX DVI 文件的程序。默认命令是“texi2dvi”。

WEAVE

转换 Web 到 TeX 的程序。默认命令是“weave”。

CWEAVE

转换 C Web 到 TeX 的程序。默认命令是“cweave”。

TANGLE

转换 Web 到 Pascal 语言的程序。默认命令是“tangle”。

CTANGLE

转换 C Web 到 C。默认命令是“ctangle”。

RM

删除文件命令。默认命令是“rm –f”。

2、关于命令参数的变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

ARFLAGS

函数库打包程序 AR 命令的参数。默认值是“rv”。

ASFLAGS

汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。

CFLAGS

C 语言编译器参数。

CXXFLAGS

C++语言编译器参数。

COFLAGS

RCS 命令参数。

CPPFLAGS

C 预处理器参数。（ C 和 Fortran 编译器也会用到）。

FFLAGS

Fortran 语言编译器参数。

GFLAGS

SCCS “get”程序参数。

LDFLAGS

链接器参数。（如：“ld”）

LFLAGS

Lex 文法分析器参数。

PFLAGS

Pascal 语言编译器参数。

RFLAGS

Ratfor 程序的 Fortran 编译器参数。

YFLAGS

Yacc 文法分析器参数。

四、隐含规则链

有些时候，一个目标可能被一系列的隐含规则所作用。例如，一个[.o]的文件生成，可能会是先被 Yacc 的[.y]文件先成[.c]，然后再被 C 的编译器生成。我们把这一系列的隐含规则

叫做“隐含规则链”。

在上面的例子中，如果文件[.c]存在，那么就直接调用 C 的编译器的隐含规则，如果没有[.c]文件，但有一个[.y]文件，那么 Yacc 的隐含规则会被调用，生成[.c]文件，然后，再调

用 C 编译的隐含规则最终由[.c]生成[.o]文件，达到目标。

我们把这种[.c]的文件（或是目标），叫做中间目标。不管怎么样，make 会努力自动推导生成目标的一切方法，不管中间目标有多少，其都会执着地把所有的隐含规则和你书写的规则全部合起来分析，努力达到目标，所以，有些时候，可能会让你觉得奇怪，怎么我的目标会这样生成？怎么我的 makefile 发疯了？

在默认情况下，对于中间目标，它和一般的目标有两个地方所不同：第一个不同是除非中间的目标不存在，才会引发中间规则。第二个不同的是，只要目标成功产生，那么，产生最终目标过程中，所产生的中间目标文件会被以"rm -f"删除。

通常，一个被 makefile 指定成目标或是依赖目标的文件不能被当作中介。然而，你可以明显地说明一个文件或是目标是中介目标，你可以使用伪目标".INTERMEDIATE"来强制声明。（如：.INTERMEDIATE ： mid ）

你也可以阻止 make 自动删除中间目标，要做到这一点，你可以使用伪目标".SECONDARY"来强制声明（如：.SECONDARY : sec）。你还可以把你的目标，以模式的方式来指定（如：%.o）成伪目标".PRECIOUS"的依赖目标，以保存被隐含规则所生成的中间文件。

在"隐含规则链"中，禁止同一个目标出现两次或两次以上，这样一来，就可防止在 make 自动推导时出现无限递归的情况。

Make 会优化一些特殊的隐含规则，而不生成中间文件。如，从文件"foo.c"生成目标程序"foo"，按道理，make 会编译生成中间文件"foo.o"，然后链接成"foo"，但在实际情况下，这一动作可以被一条"cc"的命令完成（cc –o foo foo.c），于是优化过的规

则就不会生成中间文件。

五、定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

有一点需要注意的是，"%"的展开发生在变量和函数的展开之后，变量和函数的展开发生在 make 载入 Makefile 时，而模式规则中的"%"则发生在运行时。

1、模式规则介绍

模式规则中，至少在规则的目标定义中要包含"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名的匹配，"%"表示长度任意的非空字符串。例如："%.c"表示以".c"结尾的文件名（文件名的长度至少为 3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名（文件名的长度至少为 5）。

如果"%"定义在目标中，那么，目标中的"%"的值决定了依赖目标中的"%"的值，也就是说，目标中的模式的"%"决定了依赖目标中"%"的样子。例如有一个模式规则如下：

%.o : %.c ; <command ......>

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make 会被要求去匹配当前目录下所有的文件名，一旦找到，make 就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

%.o : %.c

```
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

其中，"$@"表示所有的目标的挨个值，"$<"表示了所有依赖目标的挨个值。这些奇怪的变量我们叫"自动化变量"，后面会详细讲述。

下面的这个例子中有两个目标是模式的：

```
%.tab.c %.tab.h: %.y
bison -d $<
```

这条规则告诉 make 把所有的[.y]文件都以"bison -d <n>.y"执行，然后生成"<n>.tab.c"和"<n>.tab.h"文件。（其中，"<n>" 表示一个任意字符串）。如果我们的执行程序"foo"依赖于文件"parse.tab.o"和"scan.o"，并且文件"scan.o"依赖于文件"parse.tab.h"，如果"parse.y"文件被更新了，那么根据上述的规则，"bison -d parse.y"就会被执行一次，于是，"parse.tab.o"和"scan.o"的依赖文件就齐了。（假设，"parse.tab.o" 由"parse.tab.c"生成，和"scan.o"由"scan.c"生成，而"foo"由"parse.tab.o"和"scan.o"链接生成，而且 foo 和其[.o]文件的依赖关系也写好，那么，所有的目标都会得到满足）

3、自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@

表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$%

仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix 下是

[.a]，Windows 下是[.lib]），那么，其值为空。

$<

依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+

这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$*

这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是

"a.%.b"，那么，"$*"的值就是"dir /a.foo"。这个变量对于构造有关联的文件名是比

较有较。如果目标中没有模式的定义，那么"$*"也就不能被推导出，但是，如果目标文件的后缀

是 make 所识别的，那么"$*"就是除了后缀的那一部分。例如：如果目标是"foo.c"

，因为".c"是 make 所能识别的后缀名，所以，"$*"的值就是"foo"。这个特性是 GNU make 的，

很有可能不兼容于其它版本的 make，所以，你应该尽量避免使用"$*"，除非是在隐含规则或是静态

模式中。如果目标中的后缀是 make 所不能识别的，那么"$*"就是空值。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数

库文件叫"lib"，其由其它几个 object 文件更新。那么把 object 文件打包的比较有效

率的 Makefile 规则是：

lib : foo.o bar.o lose.o win.o

ar r lib $?

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是 GNU make 中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是 Directory，就是目录，"F"的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)

表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

$(@F)

表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(*D)"

"$(*F)"

和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(*D)"返回"dir"，而"$(*F)"返回"foo"

"$(%D)"

"$(%F)"

分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"

"$(+F)"

分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

"$(?D)"

"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于"$<"，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比如，"$(<)"就要比"$<"要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是"显式规则"和"静态模式规则"（参见前面"书写规则"一章）。其在隐含规则中并没有意义。

4、模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的"%"，或是没有前后缀，直接就是一个"%"。因为"%"代表一个或多个字符，所以在定义好了的模式中，我们把"%"所匹配的内容叫做"茎"，例如

"%.c"所匹配的文件"test.c"中"test"就是"茎"。因为在目标和依赖目标中同时有"%"时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行"茎"的传递时，我们需要知道这个步骤。例如有一个模式"e%t"，文件"src/eat" 匹配于该模式，于是"src/a"就是其"茎"，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式"c%r"，那么，目标就是"src/car"。（"茎"被传递）

5、重载内建隐含规则

你可以重载内建的隐含规则（或是定义一个全新的），例如你可以重新构造和内建隐含规则不同的命令，如：

```
%.o : %.c
$(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)
```

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

```
%.o : %.s
```

同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

六、老式风格的"后缀规则"

后缀规则是一个比较老式的定义隐含规则的方法。后缀规则会被模式规则逐步地取代。因为模式规则更强更清晰。为了和老版本的 Makefile 兼容，GNU make 同样兼容于这些东西。后缀规则有两种方式："双后缀"和"单后缀"。

双后缀规则定义了一对后缀：目标文件的后缀和依赖目标（源文件）的后缀。如".c.o"相当于"%o : %c"。单后缀规则只定义一个后缀，也就是源文件的后缀。如".c"相当于"% : %.c"。

后缀规则中所定义的后缀应该是 make 所认识的，如果一个后缀是 make 所认识的，那么这个规则就是单后缀规则，而如果两个连在一起的后缀都被 make 所认识，那就是双后缀规则。例如：".c"和".o"都是 make 所知道。因而，如果你定义了一个规则是".c.o"那么其就是双后缀规则，意义就是".c"是源文件的后缀，".o"是目标文件的后缀。如下示例：

```
.c.o:
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则不允许任何的依赖文件，如果有依赖文件的话，那就不是后缀规则，那些后缀统统被认为是文件名，如：

```
.c.o: foo.h

$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

这个例子，就是说，文件".c.o"依赖于文件"foo.h"，而不是我们想要的这样：

```
%.o: %.c foo.h

$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则中，如果没有命令，那是毫无意义的。因为他也不会移去内建的隐含规则。

而要让 make 知道一些特定的后缀，我们可以使用伪目标".SUFFIXES"来定义或是删除，如：

```
.SUFFIXES: .hack .win
```

把后缀.hack 和.win 加入后缀列表中的末尾。

.SUFFIXES: # 删除默认的后缀

.SUFFIXES: .c .o .h # 定义自己的后缀

先清楚默认后缀，后定义自己的后缀列表。

make 的参数"-r"或"-no-builtin-rules"也会使用得默认的后缀列表为空。而变量"SUFFIXE"被用来定义默认的后缀列表，你可以用".SUFFIXES"来改变后缀列表，但请不要改变变量"SUFFIXE"的值。

七、隐含规则搜索算法

比如我们有一个目标叫 T。下面是搜索目标 T 的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在 Makefile 被载入内存时，会被转换成模式规则。如果目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标 T，如果没有找到的话，那么进入第二次，第二次会把"member"当作 T 来搜索。

1、把 T 的目录部分分离出来。叫 D，而剩余部分叫 N。（如：如果 T 是"src/foo.o"，那么，D 就是"src/"，N 就是"foo.o"）

2、创建所有匹配于 T 或是 N 的模式规则列表。

3、如果在模式规则列表中有匹配所有文件的模式，如"%"，那么从列表中移除其它的模式。

4、移除列表中没有命令的规则。

5、对于第一个在列表中的模式规则：

1）推导其"茎"S，S 应该是 T 或是 N 匹配于模式中"%"非空的部分。

2）计算依赖文件。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把 D 加在第一个依赖文件的开头。

3）测试是否所有的依赖文件都存在或是理当存在。（如果有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）

4）如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。

6、如果经过第 5 步，没有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：

1）如果规则是终止规则，那就忽略它，继续下一条模式规则。

2）计算依赖文件。（同第 5 步）

3）测试所有的依赖文件是否存在或是理当存在。

4）对于不存在的依赖文件，递归调用这个算法查找他是否可以被隐含规则找到。

5）如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。

7、如果没有隐含规则可以使用，查看".DEFAULT"规则，如果有，采用，把".DEFAULT"的命令给T 使用。

一旦规则被找到，就会执行其相当的命令，而此时，我们的自动化变量的值才会生成。

9 使用 make 更新函数库文件

函数库文件也就是对 Object 文件（程序编译的中间文件）的打包文件。在 Unix 下，一般是由命令"ar"来完成打包工作。

一、函数库文件的成员

一个函数库文件由多个文件组成。你可以以如下格式指定函数库文件及其组成：

archive(member)

这个不是一个命令，而一个目标和依赖的定义。一般来说，这种用法基本上就是为了"ar"命令来服务的。如：

foolib(hack.o) : hack.o

ar cr foolib hack.o

如果要指定多个 member，那就以空格分开，如：

foolib(hack.o kludge.o)

其等价于：

foolib(hack.o) foolib(kludge.o)

你还可以使用 Shell 的文件通配符来定义，如：

```
foolib(*.o)
```

二、函数库成员的隐含规则

当 make 搜索一个目标的隐含规则时，一个特殊的特性是，如果这个目标是"a(m)"形式的，其会把目标变成"(m)"。于是，如果我们的成员是"%.o" 的模式定义，并且如果我们使用"make foo.a(bar.o)"的形式调用 Makefile 时，隐含规则会去找"bar.o"的规则，如果没有定义 bar.o 的规则，那么内建隐含规则生效，make 会去找 bar.c 文件来生成 bar.o，如果找得到的话，make 执行的命令大致如下：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

还有一个变量要注意的是"$%"，这是专属函数库文件的自动化变量，有关其说明请参见"自动化变量"一节。

三、函数库文件的后缀规则

你可以使用"后缀规则"和"隐含规则"来生成函数库打包文件，如：

```
.c.a:

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o

$(AR) r $@ $*.o

$(RM) $*.o
```

其等效于：

```
(%.o) : %.c

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o

$(AR) r $@ $*.o

$(RM) $*.o
```

四、注意事项

在进行函数库打包文件生成时，请小心使用 make 的并行机制（"-j"参数）。如果多个 ar 命令在同一时间运行在同一个函数库打包文件上，就很有可以损坏这个函数库文件。所以，在 make 未来的版本中，应该提供一种机制来避免并行操作发生在函数打包文件上。

但就目前而言，你还是应该不要尽量不要使用"-j"参数。

10 后序

终于到写结束语的时候了，以上基本上就是 GNU make 的 Makefile 的所有细节了。其它的产商的 make 基本上也就是这样的，无论什么样的 make，都是以文件的依赖性为基础的，其基本是都是遵循一个标准的。这篇文档中 80%的技术细节都适用于任何的 make，我猜测"函数"那一章的内容可能不是其它 make 所支持的，而隐含规则方面，我想不同的 make 会有不同的实现，我没有精力来查看 GNU 的 make 和 VC 的 nmake、BCB 的 make，或是别的 UNIX 下的 make 有些什么样的差别，一是时间精力不够，二是因为我基本上都是在 Unix 下使用 make，以前在 SCO Unix 和 IBM 的 AIX，现在在 Linux、Solaris、HP-UX、AIX 和 Alpha 下使用，Linux 和 Solaris 下更多一点。不过，我可以肯定的是，在 Unix 下的 make，无论是哪种平台，几乎都使用了 Richard Stallman 开发的 make 和 cc/gcc 的编译器，而且，基本上都是 GNU 的 make（公司里所有的 UNIX 机器上都被装上了 GNU 的东西，所以，使用 GNU 的程序也就多了一些）。GNU 的东西还是很不错的，特别是使用得深了以后，越来越觉得 GNU 的软件的强大，也越来越觉得 GNU 的在操作系统中（主要是 Unix，甚至 Windows）"杀伤力"。

对于上述所有的 make 的细节，我们不但可以利用 make 这个工具来编译我们的程序，还可以利用 make 来完成其它的工作，因为规则中的命令可以是任何 Shell 之下的命令，所以，在 Unix 下，你不一定只是使用程序语言的编译器，你还可以在 Makefile 中书写其它的命令，如：tar、awk、mail、sed、cvs、compress、ls、rm、yacc、rpm、 ftp......等等，等等，来完成诸如"程序打包"、"程序备份"、"制作程序安装包"、"提交代码"、"使用程序模板"、"合并文件"等等五花八门的功能，文件操作，文件管理，编程开发设计，或是其它一些异想天开的东西。比如，以前在书写银行交易程序时，由于银行的交易程序基本一样，就见到有人书写了一些交易的通用程序模板，在该模板中把一些网络通讯、数据库操作的、业务操作共性的东西写在一个文件中，在这些文件中用些诸如"@@@N、###N"奇怪字串标注一些位置，然后书写交易时，只需按照一种特定的规则书写特定的处理，最后在 make 时，使用 awk 和 sed，把模板中的"@@@N、###N"等字串替代成特定的程序，形成 C 文件，然后再编译。这个动作很像数据库的"扩展 C"语言（即在 C 语言中用"EXEC SQL"的样子执行 SQL 语句，在用 cc/gcc 编译之前，需要使用"扩展 C"的翻译程序，如 cpre，把其翻译成标准 C）。如果

你在使用 make 时有一些更为绝妙的方法，请记得告诉我啊。

回头看看整篇文档，不觉记起几年前刚刚开始在 Unix 下做开发的时候，有人问我会不会写 Makefile 时，我两眼发直，根本不知道在说什么。一开始看到别人在 vi 中写完程序后输入"!make"时，还以为是 vi 的功能，后来才知道有一个 Makefile 在作怪，于是上网查啊查，那时又不愿意看英文，发现就根本没有中文的文档介绍 Makefile，只得看别人写的 Makefile，自己瞎碰瞎搞才积累了一点知识，但在很多地方完全是知其然不知所以然。后来开始从事 UNIX 下产品软件的开发，看到一个 400 人年，近 200 万行代码的大工程，发现要编译这样一个庞然大物，如果没有 Makefile，那会是多么恐怖的一样事啊。于是横下心来，狠命地读了一堆英文文档，才觉得对其掌握了。但发现目前网上对 Makefile 介绍的文章还是少得那么的可怜，所以想写这样一篇文章，共享给大家，希望能对各位有所帮助。

现在我终于写完了，看了看文件的创建时间，这篇技术文档也写了两个多月了。发现，自己知道是一回事，要写下来，跟别人讲述又是另外一回事，而且，现在越来越没有时间专研技术细节，所以在写作时，发现在阐述一些细节问题时很难做到严谨和精练，而且对先讲什么后讲什么不是很清楚，所以，还是参考了一些国外站点上的资料和题纲，以及一些技术书籍的语言风格，才得以完成。整篇文档的提纲是基于 GNU 的 Makefile 技术手册的提纲来书写的，并结合了自己的工作经验，以及自己的学习历程。因为从来没有写过这么长，这么细的文档，所以一定会有很多地方存在表达问题，语言歧义或是错误。因些，我迫切地得等待各位给我指证和建议，以及任何的反馈。

最后，还是利用这个后序，介绍一下自己。我目前从事于所有 Unix 平台下的软件研发，主要是做分布式计算/网格计算方面的系统产品软件，并且我对于下一代的计算机革命——网格计算非常地感兴趣，对于分布式计算、P2P、Web Service、J2EE 技术方向也很感兴趣，同时，对于项目实施、团队管理、项目管理也小有心得，希望同样和我战斗在"技术和管理并重"的阵线上的年轻一代，能够和我多多地交流。我的 MSN 是：haoel@hotmail.com（常用），QQ 是：753640（不常用）。（注：请勿给我 MSN 的邮箱发信，由于 hotmail 的垃圾邮件导致我拒收这个邮箱的所有来信）

我欢迎任何形式的交流，无论是讨论技术还是管理，或是其它海阔天空的东西。除了政治和娱乐新闻我不关心，其它只要积极向上的东西我都欢迎！

最最后，我还想介绍一下 make 程序的设计开发者。

首当其冲的是： Richard Stallman

开源软件的领袖和先驱，从来没有领过一天工资，从来没有使用过 Windows 操作系统。对于他的事迹和他的软件以及他的思想，我无需说过多的话，相信大家对这个人并不比我陌生，这是他的主页：http://www.stallman.org/ 。