

Linux 开发指南

基于 EasyARM-i.MX280A/283A/287A 开发套件

UM15052701 V1.03 Date:2015/09/01

产品用户手册

类别	内容
关键词	Linux 开发、EasyARM-i.MX280A、EasyARM-i.MX283A、EasyARM-i.MX287A、i.MX28
摘要	本文主要介绍 EasyARM-i.MX280A、EasyARM-i.MX283A 、EasyARM-i.MX287A 学习套件下的 Linux 软件开发



修订历史

版本	日期	原因
V1.00	2015/01/19	创建文档
V1.01	2015/03/25	<ul style="list-style-type: none">1、修正发现的错别字和歧义语句2、把加载蜂鸣器的驱动改为加载 lradc 驱动3、添加无法通过 NFS 挂载根文件系统的注意事项4、把 lradc 驱动改为最新的驱动
V1.02	2015/05/27	<ul style="list-style-type: none">1、添加对 EasyARM-i.MX280A 的相关描述2、修正 V1.01 版本的冗余描述
V1.03	2015/09/01	<ul style="list-style-type: none">1、修正发现的错别字和歧义语句2、更新了关于光盘资料路径的描述3、裁剪了部分冗余描述4、更正了 SPI 接口部分的描述



目 录

第一篇 Linux 基础	7
第 1 章 Linux 操作系统简介	8
1.1 Linux 内核	8
1.1.1 简介	8
1.1.2 特点	9
1.1.3 内核版本号	11
1.1.4 组成部分	11
1.2 Linux 发行版	14
1.3 嵌入式 Linux	16
1.3.1 嵌入式 Linux 的特点	16
1.3.2 嵌入式 Linux 的产品形态	16
第 2 章 安装 Linux 操作系统	18
2.1 获得 Linux 环境的三种方式	18
2.2 发行版选择和 ISO 下载	18
2.3 VMware Player 软件	19
2.3.1 下载和安装	19
2.3.2 设置虚拟化支持	22
2.4 使用现成的虚拟机	23
2.5 创建和配置虚拟机	26
2.5.1 创建虚拟机	26
2.5.2 虚拟机设置	29
2.6 安装 Ubuntu	31
2.6.1 实体机安装前准备	31
2.6.2 虚拟机安装前准备	33
2.6.3 正式安装 Ubuntu	35
2.7 初识 Ubuntu	39
2.7.1 Ubuntu 桌面	39
2.7.2 输入法	40
2.7.3 系统设置	40
2.7.4 搜索软件和文件	41
2.7.5 打开终端	42
2.7.6 安装软件	43
第 3 章 开始使用 Linux	45
3.1 Linux Shell	45
3.1.1 Shell 是什么？	45
3.1.2 Shell 的种类和特点	45
3.2 Linux 常见命令	46
3.2.1 导航命令	47
3.2.2 目录操作命令	49
3.2.3 文件操作命令	52
3.2.4 网络操作命令	61



3.2.5	安装和卸载文件系统.....	62
3.2.6	使用内核模块和驱动.....	64
3.2.7	重启和关机.....	66
3.2.8	其它命令.....	66
3.3	Shell 文件	68
3.4	Linux 环境变量	69
3.4.1	环境变量.....	69
3.4.2	修改环境变量.....	69
第 4 章	Linux 文件系统	71
4.1	Linux 目录结构	71
4.1.1	Linux 目录树	71
4.1.2	Linux 目录树标准	72
4.2	Linux 的文件	72
4.2.1	Linux 文件结构	72
4.2.2	Linux 文件名称	74
4.2.3	文件类型.....	74
4.3	Linux 文件系统	75
4.3.1	Ext3 文件系统特点	75
4.3.2	Ext4 文件系统特点	76
4.3.3	其它文件系统.....	77
第 5 章	Vi 编辑器	80
5.1	Vi/Vim 编辑器.....	80
5.2	Vi 的模式.....	80
5.3	Vim 的安装.....	80
5.4	启动和关闭 Vi.....	81
5.5	光标移动.....	81
5.6	文本编辑.....	82
5.6.1	文本输入.....	82
5.6.2	文本处理.....	83
5.7	配置 vi.....	85
5.8	文件对比.....	87
第 6 章	嵌入式 Linux 开发环境构建	88
6.1	嵌入式 Linux 开发模型	88
6.1.1	交叉编译.....	88
6.1.2	交叉编译器.....	88
6.2	安装交叉编译器.....	89
6.2.1	解压工具链压缩包.....	89
6.2.2	设置环境变量.....	90
6.3	SSH 服务器	92
6.3.1	SSH 能做什么？	92
6.3.2	安装 SSH 服务器	93
6.3.3	测试 SSH 服务	93
6.3.4	用 Putty 测试	95
6.3.5	用 SSHSecure Shell 测试	97



6.4	NFS 服务器	100
6.4.1	NFS 能做什么？	100
6.4.2	安装 NFS 软件包	101
6.4.3	添加 NFS 共享目录.....	101
6.4.4	启动 NFS 服务	102
6.4.5	测试 NFS 服务器	102
6.5	TFTP 服务器	103
6.5.1	TFTP 能做什么？	103
6.5.2	安装配置 tftp 软件	103
6.5.3	配置 tftp 服务器.....	103
6.5.4	启动 tftp 服务	104
6.5.5	测试 tftp 服务器	104
第二篇	EasyARM-i.MX28xA 开发平台	105
第 7 章	开发套件介绍.....	106
7.1	开发套件简介.....	106
7.2	硬件资源.....	107
7.3	软件资源.....	108
7.4	开发所需配件.....	108
7.5	产品组装.....	109
第 8 章	入门实操.....	111
8.1	开机和登录.....	111
8.1.1	启动方式设置.....	111
8.1.2	供电连接.....	112
8.1.3	串口硬件连接.....	112
8.1.4	Windows 环境串口登录.....	115
8.1.5	Linux 环境串口登录	120
8.2	关机和重启.....	125
8.3	查看系统信息.....	125
8.3.1	查看系统内核版本.....	125
8.3.2	查看内存使用情况.....	125
8.3.3	查看磁盘使用情况.....	125
8.3.4	查看 CPU 等信息	125
8.4	设置开机自动启动.....	126
8.4.1	开机启动脚本.....	126
8.4.2	添加开机自动执行命令	126
8.4.3	禁止开机启动图形界面.....	127
8.5	加载驱动模块.....	127
8.5.1	在 shell 终端加载和使用驱动模块	127
8.5.2	在脚本文件加载和使用驱动模块	127
8.6	网络设置.....	128
8.7	通过 SSH 登录系统	130
8.8	TF 卡使用	131
8.9	U 盘使用	132
8.10	USB Device 使用	132



8.10.1	把 TF 卡作为虚拟 U 盘的储存空间	133
8.10.2	使用普通文件作为虚拟 U 盘的储存空间	133
8.11	LED 使用	134
8.11.1	LED 的操作接口	134
8.11.2	触发条件设置	135
8.12	蜂鸣器使用	136
8.13	LCD 背光控制	136
8.14	触摸屏校准	136
8.15	GPIO 操作	137
8.16	进阶操作	138
8.16.1	挂载 NFS 目录	138
8.16.2	使用 NFS 根文件系统	138
8.16.3	使用 TFTP 启动内核	141
8.16.4	内存文件系统	142
第 9 章	系统固件烧写	143
9.1	NAND Flash 存储器分区	143
9.2	烧写流程图	143
9.3	格式化 NAND Flash	144
9.3.1	通过 USB Boot 引导格式化 NAND Flash	144
9.3.2	通过 SD Boot 方式格式化 NAND Flash	146
9.4	TF 卡烧写方案	148
9.4.1	TF 卡烧写用的固件	148
9.4.2	制作 TF 启动卡	149
9.4.3	固件烧写步骤	150
9.5	USB 烧写方案	151
9.5.1	执行 USB 烧写	152
9.6	使用网络升级内核或文件系统	155
9.6.1	网络升级用的固件	155
9.6.2	升级步骤	156
9.6.3	故障排除	158
第三篇	基本固件编译及其简单应用	160
第 10 章	u-boot 编译及其简单应用	161
10.1	U-Boot 简介	161
10.2	U-Boot 源代码目录	161
10.3	编译 U-Boot	161
10.4	U-Boot 基本命令	163
10.4.1	预设的组合命令	165
10.5	U-Boot Tools	166
第 11 章	内核编译及其驱动要点	167
11.1	编译内核	167
11.1.1	解压内核文件	167
11.1.2	设置内核对应的型号	167
11.1.3	备份内核配置文件	168
11.1.4	编译内核	168



11.2	生成 imx28_ivt_linux.sb 内核固件.....	168
11.3	配置内核.....	169
11.4	内核 GPIO 使用方法	174
11.5	LED 驱动.....	176
11.5.1	驱动加载.....	176
11.5.2	卸载驱动.....	178
11.5.3	open 调用的实现.....	178
11.5.4	write 调用的实现.....	178
11.5.5	ioctl 函数的实现.....	179
11.5.6	close 调用的实现.....	179
11.5.7	编译驱动代码.....	180
11.5.8	测试程序.....	181
11.6	GPIO 中断示例程序	182
11.7	设置 LCD 的时序.....	184
第 12 章	根文件系统的打包及其简单应用	186
12.1	Linux 根文件系统	186
12.2	FHS 标准	186
12.2.1	顶层目录.....	186
12.2.2	“/usr” 目录.....	187
12.3	BusyBox.....	187
12.4	NFS 根文件系统	187
12.5	生成文件系统映像.....	188
12.5.1	生成 rootfs.ubifs 固件.....	188
12.5.2	生成 rootfs.tar.bz2 固件.....	189
12.6	修改预置的开机启动设置.....	189
第四篇	Linux 应用开发	190
第 13 章	功能部件应用编程	191
13.1	GPIO 应用编程	191
13.1.1	驱动加载.....	193
13.1.2	使用驱动接口.....	194
13.2	ADC 接口	197
13.2.1	LRADC	197
13.2.2	HSADC	200
13.3	串口编程.....	202
13.3.1	访问串口设备.....	203
13.3.2	配置串口属性.....	205
13.3.3	操作示例.....	210
13.3.4	串口数据监控.....	211
13.4	I ² C 接口	212
13.4.1	open 调用	213
13.4.2	ioctl 调用	213
13.4.3	write 调用	214
13.4.4	read 调用	214
13.4.5	close 调用.....	215



13.4.6 应用程序读写 DS2460 例程.....	215
13.5 PWM 接口	217
13.5.1 PWM 周期计算	218
13.5.2 PWM 占空比设置与输出	218
13.5.3 系统命令操作 PWM3 示例	218
13.5.4 系统命令操作 PWM4 示例	219
13.6 SPI 接口	220
13.6.1 open 调用	220
13.6.2 ioctl 调用	221
13.6.3 示例代码.....	223
13.7 CAN 接口	228
13.7.1 使用 CAN 设备	228
13.7.2 socket CAN 编程指南	233
第 14 章 嵌入式 Linux Qt 编程	238
14.1 背景知识.....	238
14.2 Qt 介绍.....	238
14.2.1 Qt 简介	238
14.2.2 Qt/E 简介	238
14.3 编译环境的搭建 (Qt-4.7.3)	239
14.3.1 交叉编译工具链的 Qt 库替换	239
14.3.2 目标板文件系统的 Qt 库替换	240
14.4 Qt 开发体验	240
14.4.1 编译 helloworld 程序	240
14.4.2 在目标板上运行 helloworld 程序	242
14.5 qmake 与 pro 文件	245
14.5.1 pro 文件例程	246
14.5.2 pro 文件常见配置	247
14.6 Qt 编程简单入门	247
14.6.1 例程讲解	247
14.6.2 信号和槽机制	249
14.7 Qt SDK 的使用	250
14.7.1 Qt SDK 简介	250
14.7.2 Qt SDK 安装	251
14.7.3 Qt Creator 配置	252
14.7.4 Qt Creator 使用例程	256
14.8 zylauncher 图形框架	262
表格索引	267
程序清单索引	269
参考文献	271
免责声明	272



第一篇 Linux 基础

本篇主要讲述进行嵌入式 Linux 开发所必备的基础知识，以实用和够用为标准进行介绍，与嵌入式 Linux 开发不相关的知识都不在讲述之列。特别是 Linux 命令部分，并没有介绍全部的 Linux 命令，而仅仅精选嵌入式 Linux 开发中的常用命令进行介绍。

本篇一共分为 6 章，从 Linux 操作系统开始，循序渐进地介绍，到最后讲述嵌入式 Linux 开发环境的构建，为嵌入式 Linux 开发做准备。各章标题和内容概要如下：

第 1 章：Linux 操作系统简介，主要介绍 Linux 内核和发行版等知识，属于常识性内容，作为一般性了解即可。

第 2 章：安装 Linux 操作系统，以 Ubuntu 为例讲述 Linux 操作系统安装过程，这部分内容属于实操性内容，建议跟着做一遍。

第 3 章：开始使用 Linux，主要介绍嵌入式 Linux 开发相关的操作和命令，掌握这部分内容是基础也是必备技能，需要多加操作和练习，做到熟练掌握。

第 4 章：Linux 文件系统，介绍 Linux 文件系统的一些常识性内容，做一般性了解即可。

第 5 章：Vi 编辑器，讲述 Vi 编辑器的基本使用。掌握一款 Linux 下的文本编辑器是进行 Linux 开发的一项必备技能，需要多加练习，熟练运用。

第 6 章：嵌入式 Linux 开发环境构建，这部分内容也是实操性内容，需要深刻理解，建议照着做一遍。

整个第一篇的内容，都没有什么难点，但对于习惯了 Windows 操作，或者刚接触 Linux 的初学者来说，可能会对 Linux 的操作方式有点不习惯，特别是命令行操作。只要多加练习，很快就可以度过适应期，习惯并喜欢上 Linux “简单就是美”的设计哲学和操作方式。



第1章 Linux 操作系统简介

本章首先对 Linux 发展简史进行简要介绍，然后对 Linux 内核进行了介绍，重点介绍了 Linux 内核的特点和功能，接着对 Linux 发行版进行介绍，并列举了一些典型的发行版；最后对嵌入式 Linux 进行了简要介绍，包括嵌入式 Linux 的特点和产品形态。

1.1 Linux 内核

1.1.1 简介

Linux 是全球最受欢迎的开源操作系统。它是一个由 C 语言编写的，符合 POSIX 标准的类 UNIX 系统。

➤ 词条 POSIX

POSIX 是 Portable Operating System Interface 的缩写，表示可移植操作系统接口，该标准规定了操作系统应该为应用程序提供的接口标准。

➤ 词条 UNIX

UNIX 是一个强大的多用户、多任务分时操作系统，支持多种处理器架构，于 1969 年在 AT&T 的贝尔实验室开发。UNIX 是商业操作系统，需要收费。

20 世纪九十年代，由于当时 UNIX 的商业化，Andrew Tanenbaum 教授开发了 Minix 操作系统，用于教学和科研，并发布在 Internet 上，免费给全世界的学生使用。Minix 具有 UNIX 的很多特点，但是不完全兼容。1991 年，芬兰大学生 Linus Torvalds 为了给 Minix 用户设计一个比较有效的 UNIX PC 版本，写了一个“类 Minix”的操作系统，并发布到了 Minix 新闻组，在众多支持者的帮助下，Linus 推出了 Linux 第一个稳定版本。1991 年 11 月份，Linux 0.10 版本推出，次年 12 月份，Linux 0.11 版本推出，并在发布网上免费供人们使用。Linux 0.13 版本发布时，Linux 已经非常接近于一种可靠、稳定的操作系统，Linus 决定将 0.13 版本改称为 0.95 版本，到 1994 年 3 月，Linux 发布了 1.0 版本。

➤ Linus 当时提交到 Minix 新闻组的原名并不是 Linux，而是 Freax，取自“Free”和“Unix”两个单词，为“免费的 Unix”之意。但当时的管理员并不喜欢“Freax”这个名称，并以“Linus’ s Minix”之意，将 Freax 放到了一个名为“Linux”的目录下，之后便一直用 Linux 这个名称。

Linux 诞生、发展和壮大于网络，目前依然掌控于 Linux 社区，遍布全球数以万计的黑客和志愿者参与 Linux 开发，也有商业公司为 Linux 贡献代码。Linux 内核核心开发队伍的领导者目前是 Linus 本人。

➤ Linus 其人

Linus Torvalds (1969.12.28 -)，芬兰赫尔辛基人。在 1991 年他还是一名大学生的时候，开发了 Linux 操作系统，在众多黑客的帮助和他的主持下，Linux 操作系统蓬勃发展，他本人至今依然是 Linux 内核项目的核心和领导人物。他本人获奖无数，主要有：

- 2014 年，Linus 获得 2014 IEEE 计算机先驱奖；
- 2012 年，芬兰千禧年科技奖；
- 2012 年，首批入驻“互联网名人堂”；
- 2011 年，首届 iTechLaw 成就奖；
- 2004 年，被评为世界最有影响力的人之一；
- 1998 年，电子前哨基金会先锋奖。

除 Linux 操作系统之外，Linus 还创建了目前最流行的版本控制系统 Git。



Linux 遵循 GPL 协议，允许任何人对代码进行修改或发行，包括商业行为。只要其遵守该 GPL 协议，所有基于 Linux 的软件也必须以 GPL 协议的形式发表，并提供源代码。

➤ 词条 GPL

GPL 是 GNU General Public License 的缩写，非正式中文翻译为“GNU 通用公共许可证”。只有 GPL 英文原版才具有法律效力。

在软件中采用了使用 GPL 协议的产品，该软件产品也必须采用 GPL 协议，即必须开源，这是 GPL 所谓的“传染性”。

获取 Linux 内核源码的网址为：<http://www.kernel.org>，在这里能够下载各版本的内核源码，包括测试版和最新稳定版。

Linux 的吉祥物是一只名叫 Tux 的企鹅，看起来像穿了一件晚礼服的企鹅，如右图。

Linux 吉祥物创作于 1996 年，据说 Linus 被澳大利亚国家动物园的一只小企鹅轻轻咬了一下，于是就有了用企鹅做吉祥物的想法。

Tux 全称 tuxedo，但大多数人更倾向于另一种说法，说是 Tux 名字来源于“Torvalds Unix”。

Linux 发音[’ li:nəks]，这也是 Linus 本人的发音，在不同语言里发音有差异，国内很大一部分人发音[’ li:njuks]。



1.1.2 特点

1. Linux 内核的重要特点

Linux 是一个开放自由的操作系统内核，具有一些鲜明的特点：

Linux 是一个一体化内核；

注：“一体化内核”也称“宏内核”，是相对于“微内核”而言的。几乎所有的嵌入式和实时系统都采用微内核，如 VxWorks、uC/OS-II、PSOS 等。

可移植性强。尽管 Linus 最初只为在 X86 PC 上实现一个“类 UNIX”，后来随着加入者的努力，Linux 目前已经成为支持硬件平台最广泛的操作系统；

注：已经在 X86、IA64、ARM、MIPS、AVR32、M68K、S390、Blackfin、M32R 等众多架构处理器上运行。

是一个可裁剪操作系统内核。Linux 极具伸缩性，内核可以任意裁剪，可以大至几十或者上百兆，可以小至几百 K，运行的设备从超级计算机、大型服务器到小型嵌入式系统、掌上移动设备或者嵌入式模块，都可以运行；

模块化。Linux 内核采用模块化设计，很多功能模块都可以编译为模块，可以在内核运行中动态加载/卸载而无需重启系统；

网络支持完善。Linux 内核集成了完整的 POSIX 网络协议栈，网络功能完善；

稳定性强。运行 Linux 的内核的服务器可以做到几年不用复位重启；

安全性好。Linux 源码开放，由众多黑客参与 Linux 的开发，一旦发现漏洞都能及时修复；

支持的设备广泛。Linux 源码中，设备驱动源码占了很大比例，几乎能支持任何常见设备，无论是很老旧的设备还是最新推出的硬件设备，几乎都能找到 Linux 下的驱动。

2. Linux 操作系统的特点

以 Linux 内核为核心的操作系统具有如下特点：



- 开放性

遵循世界标准规范，特别是遵循开放系统互连（OSI）国际标准。凡遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互连。

- 词条 OSI

OSI 是 Open System Interconnection 的缩写，意为开放系统互联，该模型由 ISO（国际标准化组织）制定。模型把网络通信分为 7 层：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

- 词条 ISO

ISO 是 International Organization for Standardization 的缩写，即国际标准化组织，该组织是由国家标准机构组成的世界范围的联合会，现有 140 个成员国。ISO 中央办事机构设在瑞士的日内瓦。

- 多用户

Linux 操作系统是一个真正的多用户操作系统；系统资源可以被不同用户各自拥有使用，即每个用户对自己的资源有特定的权限，互不影响。

经常有初学者将 Linux 的多用户与 Windows 的多用户弄混淆，实际上两者的差别是很大的。Windows 桌面同一时刻只允许一个用户登录，其余用户必须锁定；而 Linux 则允许多个用户同时登录。

- 多任务

多任务是现代计算机的最主要的一个特点。它是指计算机同时执行多个程序，而且各个程序的运行互相独立。Linux 系统调度每一个进程平等地访问处理器。

多任务实际上很常见，例如我们在编写文档的时候，还可以一边听歌，甚至还可以从网上下载资料。这至少就有文档处理、音乐播放和网络下载三个任务，相互互不影响，并且是同时运行的。

- 良好的用户界面

Linux 向用户提供了两种界面：用户界面和系统调用。

Linux 的传统用户界面是基于文本的命令行界面，即 Shell，它既可以联机使用，又可存在文件上脱机使用。Shell 有很强的程序设计能力，用户可方便地用它编制程序，从而为用户扩充系统功能提供了更高级的手段。

Linux 还为用户提供了图形用户界面。它利用鼠标、菜单、窗口、滚动条等设施，给用户呈现一个直观、易操作、交互性强的友好的图形化界面。

系统调用是提供给用户编程时使用的界面。用户可以在编程时直接使用系统提供的系统调用命令。系统通过这个界面为用户程序提供低级、高效率的服务。

- 设备独立性

Linux 操作系统把所有外部设备统一当作成文件来看待，只要安装它们的驱动程序，任何用户都可以像使用文件一样，操纵、使用这些设备，而不必知道它们的具体存在形式。

Linux 的设备独立性使得它具有高度适应能力，能够适应随时增加的新设备。

设备独立性主要是对应用程序开发者来说的。例如，对应用开发者来说，系统自带的串口与 USB 串口的操作方式是一样的，都是串口设备，而不用关心这个串口设备实际对应的物理硬件是什么。

现代计算机都实现了设备独立特性。



- 完善的网络功能

Linux 内置完整的 POSIX 网络协议栈，在通信和网络功能方面优于其它操作系统。Linux 为用户提供了完善的、强大的网络功能：

- 1) 支持 Internet。Linux 免费提供了大量支持 Internet 的软件，使得用户能用 Linux 与世界上的其他人通过 Internet 网络进行通信。
- 2) 网络文件传输。用户能通过一些 Linux 命令完成内部信息或文件的传输。
- 3) 远程访问功能。Linux 系统既允许本身通过网络访问远程的系统，也允许远程系统通过网络访问自身。

- 可靠的系统安全

Linux 采取了许多安全技术措施，包括对读、写进行权限控制、带保护的子系统、审计跟踪、核心授权等，为网络多用户环境中的用户提供了必要的安全保障。

- 模块化

运行时可以根据系统的需要来加载程序而无需重启系统。Linux 的模块化极大地提高了 Linux 的可裁剪性和灵活性。

- 良好的可移植性

Linux 是一种可移植的操作系统，能够在从微型计算机到大型计算机的任何环境和任何平台上运行。目前已经成为支持平台最广泛的操作系统。

➤ Linux 内核移植分 3 个层次：体系结构级别移植、处理器级别移植和板级移植。对大多数开发者而言，只需进行板级移植。

1.1.3 内核版本号

Linux 内核版本由 Linus 所领导的内核开发小组控制，版本号有严格规定。

Linux 内核版本号通常由 3 个数字组成，以 2.6.28 为例，2 为主版本号，6 为次版本号，28 为修订号。次版本号为偶数则表示这是一个稳定版本，如 2.6.17，为奇数则表示是一个开发版本，有可能是不稳定的，如 2.5.6。

另外，还可能见到如 2.6.27.8 这样的版本号，末尾的 .8 表示这是 2.6.27 版本的第 8 个升级版本，也是可用的稳定版本。

1.1.4 组成部分

Linux 内核由进程管理、内存管理、文件系统、网络协议、进程间通信、设备驱动等模块组成，如图 1.1 所示。



图 1.1 Linux 内核组成部分

1. 进程管理

进程管理负责控制进程对 CPU 的访问，如任务的创建、调度和终止等。任务调度是进程管理最核心的工作，由 Linux 内核调度器来完成。Linux 内核调度器根据进程的优先级选择最值得运行的进程。

一个进程的可能状态有如下几种：

- 运行态——已经获得了资源，并且进程正在被 CPU 执行。进程既可运行在内核态，也可运行在用户态。
- 内核态，内核和驱动所运行时的状态，程序处于特权阶级，能够访问系统的任何资源，好比社会的统治者。
- 用户态，用户程序运行的状态，处于非特权阶级，不能随意访问系统资源，必须通过驱动程序方可访问，用户态程序可通过系统调用进入内核态。用户态程序有如社会的被统治者，处于被管理的非特权阶级，只有通过某种途径才能进入特权阶级。
- 就绪态——当系统资源已经可用，但由于前一个进程还没有执行完释放 CPU，准备进入运行状态。
- 可中断睡眠状态——当进程处于可中断等待状态时，系统不会调度该程序执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以被唤醒进入就绪状态或者运行态。
- 不可中断睡眠状态——处于中断等待状态，但是该进程只能被使用 `wake_up()` 函数明确唤醒的时候才可进入就绪状态。
- 暂停状态——当进程收到 `SIGSTOP`、`SIGSTP`、`SIGTTIN` 或者 `SIGTTOU` 就会进入暂停状态，收到 `SIGCONT` 信号即可进入运行态。
- 僵死态——进程已经停止运行，但是其父进程还没有询问其状态。

各状态之间的转换关系和转换条件如图 1.2 所示。

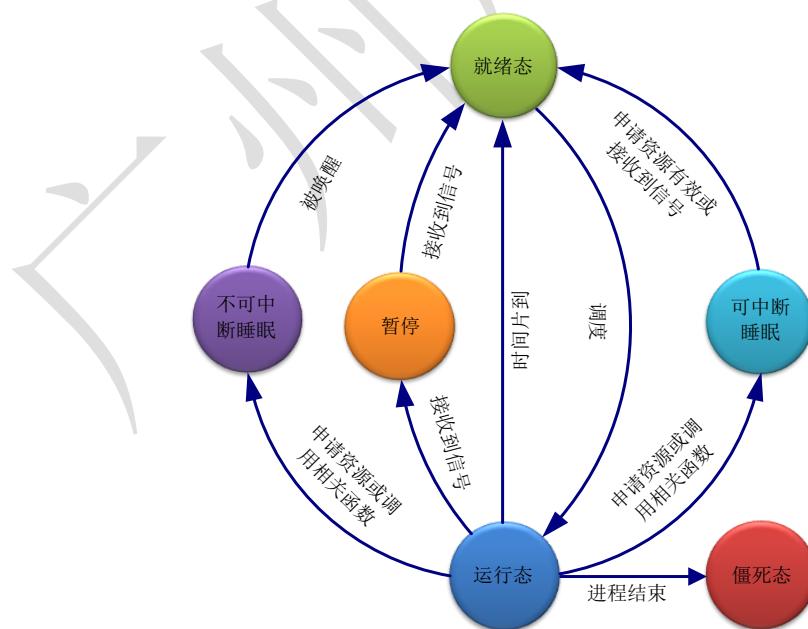


图 1.2 Linux 进程状态和转换

进程和状态的转换有点抽象，用生活中一个比较接近的例子类比一下，或许能有助于理解。Linux 内核调度器好比是生产线的主管，而进程则好比是生产线上的工人。主管 24 小



时不间断的工作，工人的工作时间是朝九晚五，其余时间在等待区排队等候。

早上工人到达工厂，还没到9点上班时间，工人可以在等待区休息，这个状态可以称之为“就绪态”；但是9点一到，工人则必须上生产线工作，这个工作状态可称之为“运行态”；下午5点一到，到了工人下班时间，工人离开生产线又回到等待区排队等候，处于“就绪态”。

如果工人上班的时候，收到主管的命令，说是“你暂时不用工作了，到休息室休息等待”，工人此时的这个状态，可以称之为“暂停”状态，过了一段时间，主管通知工人说是“休息结束，要准备工作了”，工人不能直接回生产线岗位，而是必须先到等待区排队等待，轮到后才上生产线工作。

如果有一天工人精神状态不好，向主管申请要睡觉休息，理由可以是“某种配件不到，我无法工作”，也可以是“我就是困了，想睡觉”，工人最后可能得到两种批准结果：一是主管批准了，但是附加了一个条件说“等我叫醒你，你必须醒来上班”，然后工人就去享受他的安稳觉了，工人进入“不可中断睡眠”状态；另一种是主管也批准了，但是附加了另一个条件，说“在你睡觉的时候，如果配件到了，你就得立马给我起来上班”，工人也去睡觉去了，但此时工人睡得并不安心，因为这不是一个安稳觉，是“可中断睡眠”。无论工人睡得是安稳觉，还是不安稳觉，醒来都不能直接上生产线，而是回到等待区，等待轮值。

还有一种情况，工人干完活到点下班了，但主管对他不闻不问，也不安排新的工作，这是一种非正常状况，工人进入了“僵死态”。

2. 内存管理

内存管理的主要作用是控制和管理多个进程，使之能够安全的共享主内存区域。当CPU提供内存管理单元（MMU）时，内存管理为各进程实现虚拟地址到内存物理地址的转换。在32位系统上，Linux内核将4G空间分为1G内核空间（3~4G）和3G（0~3G）用户空间，通过内存管理，每个进程都可以使用3G的用户空间。

3. 文件系统

Linux内核支持众多的逻辑文件系统，如Ext2、Ext3、Ext4、btrfs、NFS、VFAT等。VFS则是Linux基于各种逻辑文件系统抽象出的一种内存中的文件系统，隐藏了各种硬件设备细节，为用户提供统一的操作接口，使用户访问各种不同文件系统和设备时，不用区分具体的逻辑文件系统。例如，Linux下硬盘通常是Ext3/4格式，而U盘通常是FAT32格式，但是用户在使用中根本感觉不到差异，也不用区分文件系统的具体差别。

4. 网络接口

Linux对网络支持相当完善，网络接口提供了对各种网络标准的存取和各种网络硬件的支持，接口可分为网络协议和网络驱动程序。网络协议部分负责实现每一种可能的网络传输协议。网络设备驱动程序负责与硬件设备通讯，每一种可能的硬件设备都有相应的设备驱动程序。

5. 进程间通信

支持进程间各种通信机制，如管道、命名管道、信号、消息队列、内存共享、信号量和套接字等。

管道通常用于具有亲缘关系的父子进程或者兄弟进程间通信，是半双工的，数据只能往一个方向流动，先入先出，与自来水管很相似。如果双方互通时，需要建立两个管道。

命名管道则突破了进程间的亲缘关系限制，即非父子、兄弟进程之间也可相互通信。

信号是软件中断，用于在多个进程之间传递异步信号。日常生活中信号的例子很多了，如一对很亲密的哑巴情侣，在很多时候只需要一个简单的眼神，对方就能知道他（她）需要



什么，并做出回应，这个眼神，就是一个“信号”。

信号能传递的信息有限，而消息队列则正好弥补了这点。例如情侣的一个眼神，对方可能知道情侣的需求，但是如果情侣有一大堆需求，仅仅靠一个眼神就比较费力了。情侣就把自己的需求写在了一张纸条上，递交给对方，对方根据纸条的内容，逐一满足情侣的需求。

共享内存常用于不同进程间进行大量数据传递。Linux 下每个进程都有自己的独立空间，各自都不能直接访问其它进程的空间。好比这对情侣都有自己的小金库，有时候需要给对方一部分钱用，但他们不能直接相互转账，必须先将钱存到他们俩合开的一个公共账户上面，然后再使用。这个公共账户就是这对情侣的“共享内存”。

信号量用于进程同步。只有获得了信号量的进程才可以运行，没有获得信号量的进程则只能等待。就像十字路口的红绿灯，只有在绿灯亮（获得了绿灯）的时候才能通行，否则只能等待。

套接字（Socket）起源于 BSD，也常称“BSD 套接字”，用于多个进程间通信，可以基于文件，也可基于网络。Socket 本意是“插座”，套接字设计就是通过某些参数设定，然后将一个“插座”与另外一个“插座”连接起来。可能还有点抽象，看一个例子可能就好理解了。把套接字理解为固定电话的插口，现在要打电话出去，必须要知道打给谁，往哪里打；另外电话另一端必须有人在听才可以通话，否则也不能打电话。

6. 设备驱动

Linux 内核中很大一部分是 Linux 驱动，Linux 驱动实现对外部设备的访问和管理。

1.2 Linux 发行版

由 Linus 主持开发的 Linux 仅仅是一个内核，提供硬件抽象层、磁盘及文件系统控制、多任务等功能，并不是一个完整的操作系统。一套基于 Linux 内核的完整操作系统叫作 Linux 操作系统，也称 GNU/Linux。据不完全统计，目前大大小小应用于不同场合的 Linux 发行版已经超过 400 余种，桌面/服务器上常见的也就十来种，如 Redhat、Mandriva、Fedora、SuSe、Debian、Ubuntu 等。

一个完整的 Linux 发行版，是以 Linux 内核为基础，外加众多外围应用程序和文档组成，一个典型的 GNU/Linux 发行版基本系统结构如图 1.3 所示。不同软件厂商发布的 Linux 发行版各自包含的外围软件也不一样，发布版的镜像大小差别也很大。

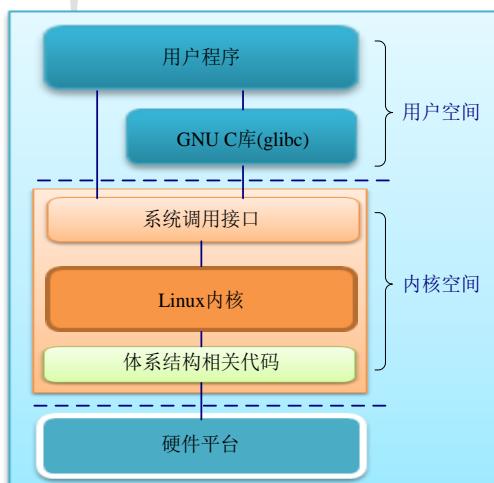


图 1.3 GNU/Linux 操作系统基本体系结构



Linux 内核为一些软件厂商提供了内核，促使了发行版的诞生；发行版的流行使得 Linux 更加广为人知，并吸引更多的人为开发内核做贡献，促进了内核的快速发展。不同发行版之间功能定位、用户群体都有差异，几乎每个发行版都拥有相当大数量的固定用户群或者忠实追随者。Linux 社区各大发行版之间的争论一直没有停止过，甚至有时候还有不同发行版用户之间的口水战，但是这并不妨碍 Linux 内核的发展。

Linux 发行版的版本号是发行厂商自定义的代号，与 Linux 内核版本号没有任何直接关系，并且各发行版的命名规则也各不相同，如 Fedora 20、Ubuntu 14.04 等。常见的 Linux 发行版有：

1. RedHat

RedHat	由 Redhat 公司发行，曾经是最流行的 Linux 发行版，一度几乎成为了 Linux 的代名词。由于其良好的兼容性和完善的开发工具，目前依然是不少工程师进行嵌入式 Linux 开发的首选平台	 redhat.
优点	拥有数量庞大的用户，优秀的社区技术支持	
缺点	已经停止开发，新硬件支持不佳或者不能支持	
官方主页	http://www.redhat.com	

2. Fedora

Fedora	RadHat Linux 发行版至 9.0 版本后就停止了开发，取而代之的是 Fedora Core Linux。与 RedHat 的公司维护不同，Fedora 是一个由 Redhat 赞助，由社区维护的发行版	
优点	拥有数量庞大的用户，优秀的社区技术支持，许多创新	
缺点	免费版版本生命周期太短	
官方主页	http://www.redhat.com	

3. Mandriva

Mandriva	Mandriva 原名 Mandrake，基于 RedHat 开发，继承 RedHat 的大部分优良特性	
优点	友好的操作界面，图形配置工具，庞大的社区支持，NTFS 分区大小变更	
缺点	部分版本 bug 较多，最新版本只先发布给 Mandrake 俱乐部的成员	
官方主页	http://www2.mandriva.com	

4. Debian

Debian	是最具有 Linux 精神，最严谨、组织发展最整齐的 Linux，以稳定性著称	
优点	遵循 GNU 规范，100% 免费，优秀的网络和社区资源，强大的 apt-get	
缺点	安装相对不易，目前发展较为缓慢，stable 分支的软件极度过时	
官方主页	http://www.debian.org	

5. Ubuntu

Ubuntu	基于 Debian 开发，堪称最完美的 Linux 操作系统。每 6 个月发布一个新版本	
优点	人气颇高的论坛提供优秀的资源和技术支持，固定的版本更新周期	
缺点	还未建立成熟的商业模式	
官方主页	http://www.ubuntu.com	

6. SuSe

SuSe	在德国和欧洲很流行，已经被 Novell 收购	



优点	专业，易用的 YaST 软件包管理系统	
缺点	FTP 发布通常要比零售版晚 1~3 个月	
官方主页	http://www.novell.com/linux	

7. Gentoo

Gentoo	全部源代码级安装，不适合初学者	 gentoo linux
优点	高度的可定制性，完整的使用手册，媲美 Ports 的 Portage 系统	
缺点	编译耗时多，安装缓慢	
官方主页	http://www.gentoo.org	

8. Slackware

Slackware	历史最悠久的 Linux 发行版本	
优点	非常稳定、安全，高度坚持 UNIX 的规范	
缺点	所有的配置均通过编辑文件来进行，自动硬件检测能力较差	
官方主页	http://www.slackware.com	

9. 红旗 Linux

红旗 Linux	由中科红旗软件公司开发，是国内比较优秀的 Linux 发行版	
优点	办公软件较齐全，适合办公	
缺点	开发软件少，不适合于嵌入式 Linux 系统开发	
官方主页	http://www.redflag-linux.com	

1.3 嵌入式 Linux

1.3.1 嵌入式 Linux 的特点

嵌入式 Linux 是对运行在嵌入式设备上的 Linux 的统称，严格说来，每种不同应用的嵌入式 Linux 都可以称为是一个发行版。嵌入式 Linux 往往针对于某个特殊领域，专门为实现某些特定的功能而开发，**一般说来，嵌入式 Linux 所运行的程序相对来说比较单一，功能定位也比较明确，如嵌入式网关、路由器等。**

将标准 Linux 应用到嵌入式领域，往往是根据实际需要裁减内核，内核一般从几百 K 到几兆字节不等。所使用的文件系统也不是桌面 Linux 这样复杂庞大的软件包，一般也是用源码或者其它工具定制，文件系统的大小也可以从几兆到几十兆，或者上百兆不等。

Linux 在嵌入式领域的分化，一般是两个方向，小型化和实时化。

小型化，一般就是根据需要，将不需要的功能和服务去掉，尽可能的减小内核和系统的体积，以节省硬件资源和成本，如 ETLinux、uLinux、ThinLinux 等。

实时化一般是通过修改源代码，为 Linux 内核增加比校准内核更好的实时性，以满足一些对实时性有要求的特定领域的应用，如 RTLinux、RTAI 等。

1.3.2 嵌入式 Linux 的产品形态

与其它嵌入式系统产品一样，嵌入式 Linux 产品在物理形态上与普通 Linux 设备有很大差异，不同产品之间物理形态也是各不相同。与桌面 Linux 相比，嵌入式 Linux 产品往往没



有硕大的显示器，或者鼠标键盘这样的外设。

嵌入式 Linux 产品既可以作为一个独立形态的产品出现，如手持机、交换机、路由器等；也有可能以某种特殊功能设备的形式出现，通过某种通信接口参与系统集成，例如协议转换器；或者甚至以电路板或者模块的形式出现在某种设备的电路板上，如嵌入式工业交换机模块。无论如何，它们的共性都是运行了经过高度裁剪的、具备特定功能的嵌入式 Linux 操作系统。图 1.4 列举了生活中一些常见的嵌入式 Linux 产品。



图 1.4 生活中常见的嵌入式 Linux 产品

无论最终产品以何种形态出现，在开发阶段，串口和网口几乎是必不可缺的外设接口。嵌入式 Linux 的默认终端通常是调试串口，系统输出信息通过串口输出，也通过串口接收各种命令。而网口则常用于数据传输和程序调试，特别是在内核开发阶段以及应用程序开发阶段，网络几乎也是必须的。



第2章 安装 Linux 操作系统

学习 Linux，必须要有一个 Linux 环境。本章先介绍获得 Linux 环境的 3 种方式，然后以 Ubuntu 发行版为例讲解 Linux 操作系统的安装和设置，图文并茂，清晰明了的展示 Ubuntu 操作系统安装的全过程，引领读者完成 Ubuntu 操作系统的安装。本章最后对 Ubuntu 桌面进行了粗略介绍。

2.1 获得 Linux 环境的三种方式

学习 Linux，必须先获得一个 Linux 主机环境，通常情况下，可以通过以下三种方式获得 Linux 环境。

1. 双系统安装

如果没有闲置的计算机，或者现有 Windows 系统的计算机有足够的硬盘空间，可以考虑划分一部分硬盘空间，用于安装 Linux 操作系统，最终形成双系统计算机。

优点：经济实惠，且对计算机硬件要求不太高。

缺点：**安装双系统比较危险，一不小心有可能造成整个硬盘数据丢失；**在开发过程使用到 Windows 工具时，需进行系统切换，不是很方便。

2. 全新硬盘安装

如果有足够的计算机可用，可以选择一台计算机全新安装 Linux 操作系统。

优点：不用考虑多系统并存的问题，且对计算机硬件要求不太高。

缺点：在嵌入式开发过程中，通常还会用到 Windows 下的工具，还需另外一台计算机安装 Windows 系统。

3. 安装虚拟机

如果计算机配置较高，可以考虑虚拟机方案。在 Windows 下安装虚拟机软件，然后通过虚拟机软件创建一台虚拟电脑，最后在虚拟电脑中安装 Linux 操作系统。常用的虚拟机软件有 VMware、Virtual Box 和 Virtual PC 等，不同虚拟机软件的使用方法稍有不同。下文均以 VMware 为例进行介绍。

优点：安装和使用 Linux 都很方便；还可同时使用 Windows 系统。

缺点：对计算机硬件要求高，特别是内存，推荐 4GB 及以上。

在 Windows 下使用虚拟机，除了可以继续使用 Windows 下的工具之外，还有下列好处：

- 一台电脑可以同时存放多台虚拟机，这样就可以存在多个不同版本的 Linux 系统；
- 在硬件允许的情况下，甚至可以同时运行多台虚拟机；
- 安装好的虚拟机可以任意复制和拷贝，方便在不同电脑之间迁移和扩散。

2.2 发行版选择和 ISO 下载

在第一章介绍 Linux 发行版的时候提到，Linux 有众多发行版，就算是常用的发行版也有十来种。不同发行版之间，在安装和使用上都有差异，选择一个合适的发行版，是能促进 Linux 的学习的。

首先要考虑该发行版的流行度，越流行的发行版，用户越多，遇到问题寻求技术支持也很方便，如果选择小众的发行版，寻求技术支持就不那么方便了。

其次要考虑该发行版使用的难易程度，通常来说，越简单易用的发行版越流行。

进行嵌入式 Linux 开发，还必须考虑嵌入式 Linux 开发工具的问题。最好选择处理器半



导体厂商以及开发平台厂商所选择的发行版，这样能够直接使用半导体或者开发平台原厂提供的各种工具，减少开发过程中的障碍。

基于以上 3 个理由，我们选择了 Ubuntu 发行版，下面的安装和使用都以 Ubuntu 为例进行介绍。Ubuntu 本身又有很多版本，我们选择的确切版本是 Ubuntu 12.04.5，是目前来说最适合于嵌入式 Linux 开发的 Ubuntu LTS（长期支持）版本。

Ubuntu 12.04 下载地址：www.ubuntu.com/download/alternative-downloads，网页界面截图如图 2.1 所示。

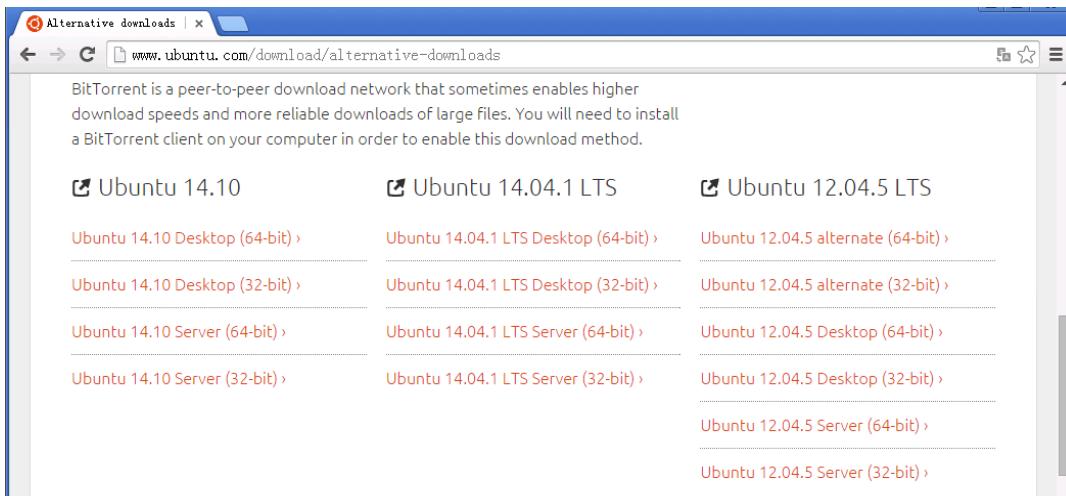


图 2.1 Ubuntu 镜像下载网页界面

建议选择 Desktop 版本，到底是 32-bit 版本还是 64-bit 版本，需要根据计算机硬件来决定，在硬件允许的情况下，推荐选择 64-bit 版本。

下载 ISO 文件后，如果进行虚拟安装，则可以直接使用 ISO 文件；如果进行物理实体安装，则可将 ISO 刻成启动光盘，或者用 unetbootin-windows 软件制作成 USB 启动盘备用。

用从 Ubuntu 官网下载的 ISO 镜像，安装后只能得到纯净的 Ubuntu 系统，如果从 www.zlg.cn/linux 下载经过重新打包的 Ubuntu 镜像，安装后将会得到已经构建好嵌入式 Linux 开发环境的 Ubuntu 系统。

如果使用虚拟机，还可以选择下载已经安装好的 Ubuntu 虚拟机文件，请参考 2.4 小节。

2.3 VMware Player 软件

2.3.1 下载和安装

打开 VMware 官方网站(www.vmware.com)，进入下载专区，下载非商用的 VMware Player 软件。在下载页面中选择下载 VMware Player for Windows 32-bit and 64-bit 软件，如图 2.2 所示。



图 2.2 VMware Player 下载页面

截止到本书完稿时，VMware Player 已经更新到了 7.0 版本，7.0 版本没有 32 位系统支持了，32 位系统请选择 6.0 版本下载使用。

文件下载完成后，得到 VMware-player-6.0.2-1744117.exe 程序安装文件（**具体文件名以实际下载到的文件为准**）。双击该程序安装文件，在弹出的对话框中选择“下一步”，如图 2.3 所示。



图 2.3 安装 VMware Player



在弹出的“许可协议”对话框中选择“我接受许可协议中的条款”，如图 2.4 所示。

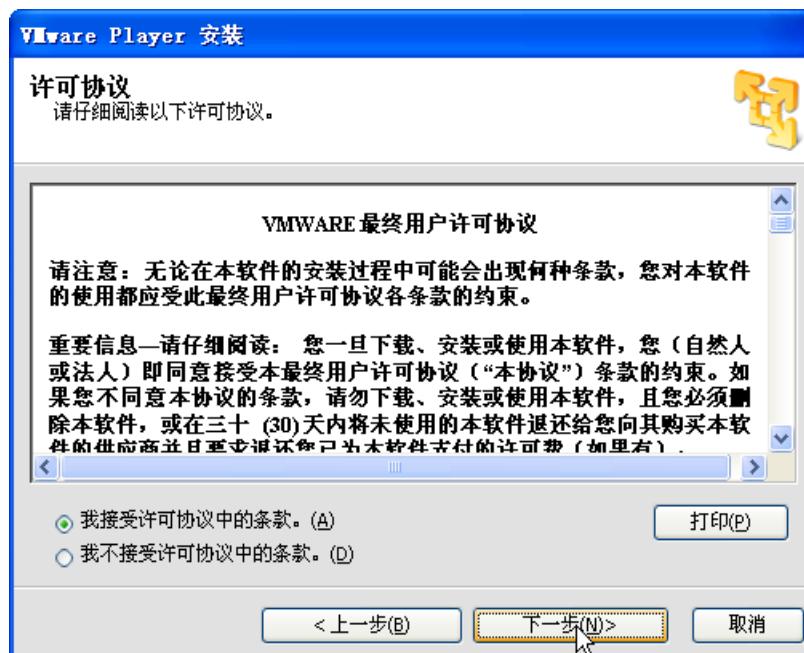


图 2.4 接受许可协议

然后按默认设置一直点击“下一步”直至如图 2.5 所示界面。

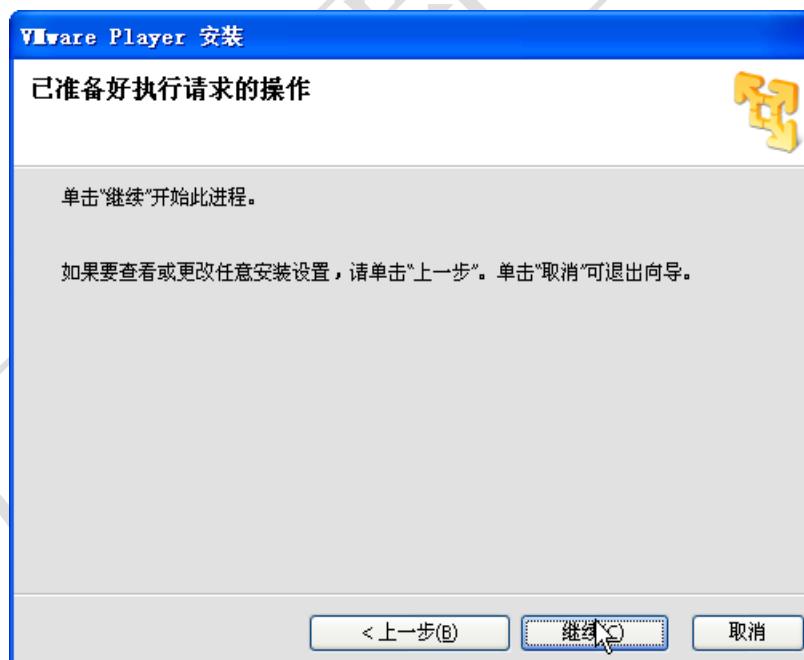


图 2.5 准备安装

此时点击“继续”按钮即可进行 VMware Player 软件的安装，安装完成时如图 2.6 所示。



图 2.6 完成安装

2.3.2 设置虚拟化支持

对于大多数 PC 而言，主板设置默认支持虚拟化，无需进行这步操作，但是对于一些笔记本电脑，默认关闭了虚拟化支持，需要使能才能正常使用虚拟机。

设置虚拟化支持，需要进入系统 BIOS 进行操作。不同品牌的笔记本进入 BIOS 的方法也存在差异，有的是在刚启动时持续按 F2 键进入 BIOS，有的是 F10 键，具体请参考对应品牌电脑的主板说明。

当进入 BIOS 系统，找到 Intel Virtualization Technology 选项，将其配置为 Enable，如图 2.7 所示。注意，不同 PC 的 BIOS 中对应的选项位置及描述可能不同，请以实际情况为准。

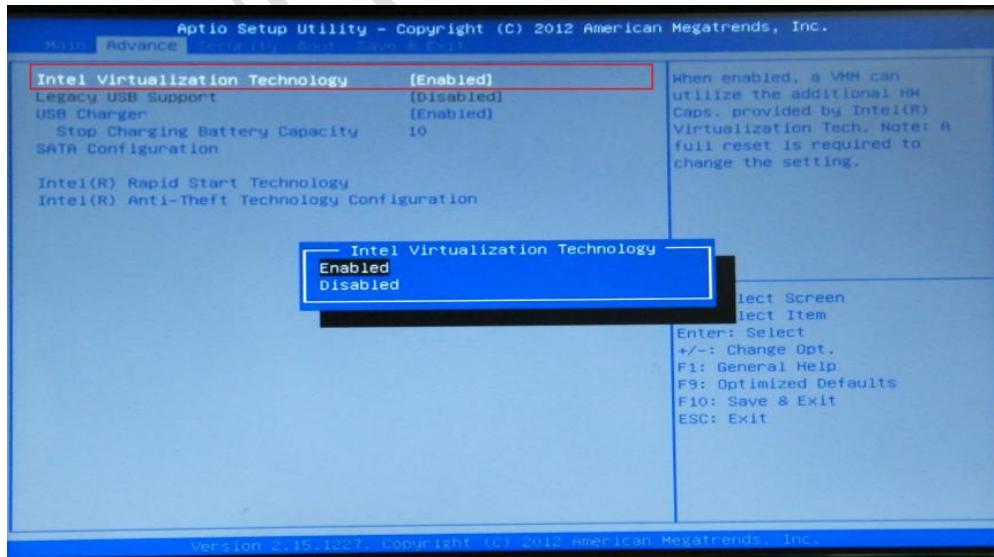


图 2.7 使能 Intel Virtualization Technology

设置好虚拟化支持后，保存并退出 BIOS，重启电脑。



2.4 使用现成的虚拟机

前面已经提到过，虚拟机可以在不同电脑之间迁移和扩散。如果觉得安装 Linux 操作系统麻烦，或者暂时不想安装，可以直接使用已经安装好的虚拟机镜像。打开 <http://www.zlg.cn/linux>，下载已经安装好的 Ubuntu 12.04 虚拟机镜像，存放到有足够空闲空间（建议 40GB 以上）的硬盘解压，将得到我们已经安装好的虚拟机，如图 2.8 所示。

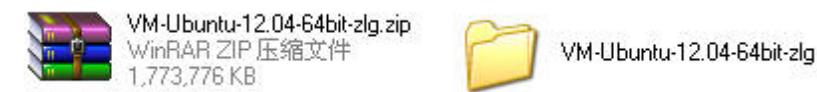


图 2.8 下载得到的虚拟机镜像和解压后的文件夹

下载页面同时提供了 64 位和 32 位虚拟机文件，请根据计算机硬件具体情况选择：32 位处理器的计算机只能使用 32 位镜像；而对于 64 位处理器的计算机，无论安装了 32 位还是 64 位操作系统，都可以任意选择。

打开 VMware Player 软件，点击“打开虚拟机”，选择打开已有的虚拟机，如图 2.9 所示。



图 2.9 选择“打开虚拟机”

在文件浏览器中，找到刚才虚拟机解压后得到的目录，打开选择打开虚拟机配置文件，如图 2.10 所示。

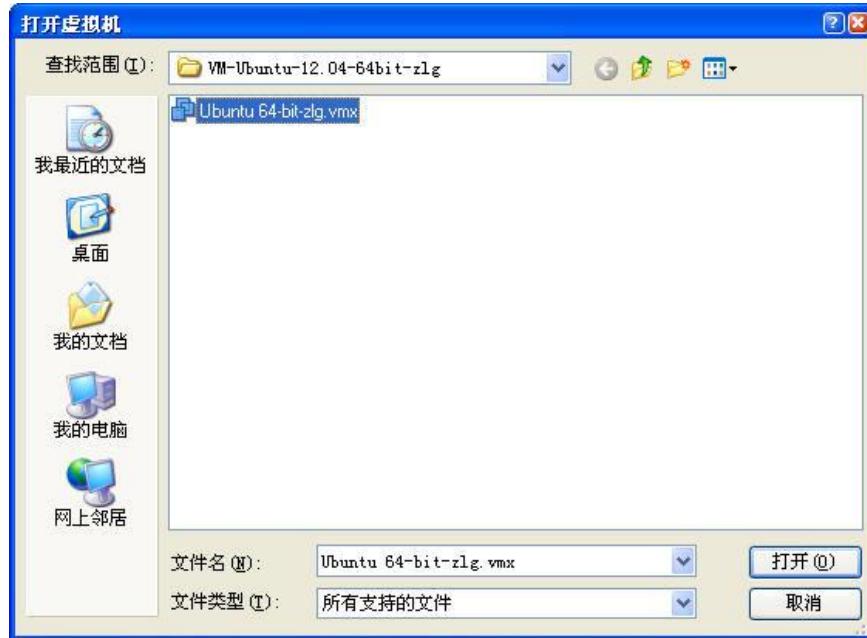


图 2.10 打开虚拟机配置文件

打开了虚拟机配置文件的 VMware Player 界面如图 2.11 所示，点击“播放虚拟机”可以启动虚拟机。



图 2.11 虚拟机装载成功后的界面

虚拟机文件被拷贝到新的位置，第一次运行虚拟机会出现如图 2.12 所示的对话框，选择“我已复制该虚拟机”即可。



图 2.12 选择 “I copied it”

之后虚拟机将会正常启动，启动成功后，可以看到 Ubuntu 桌面，如图 2.13 所示。

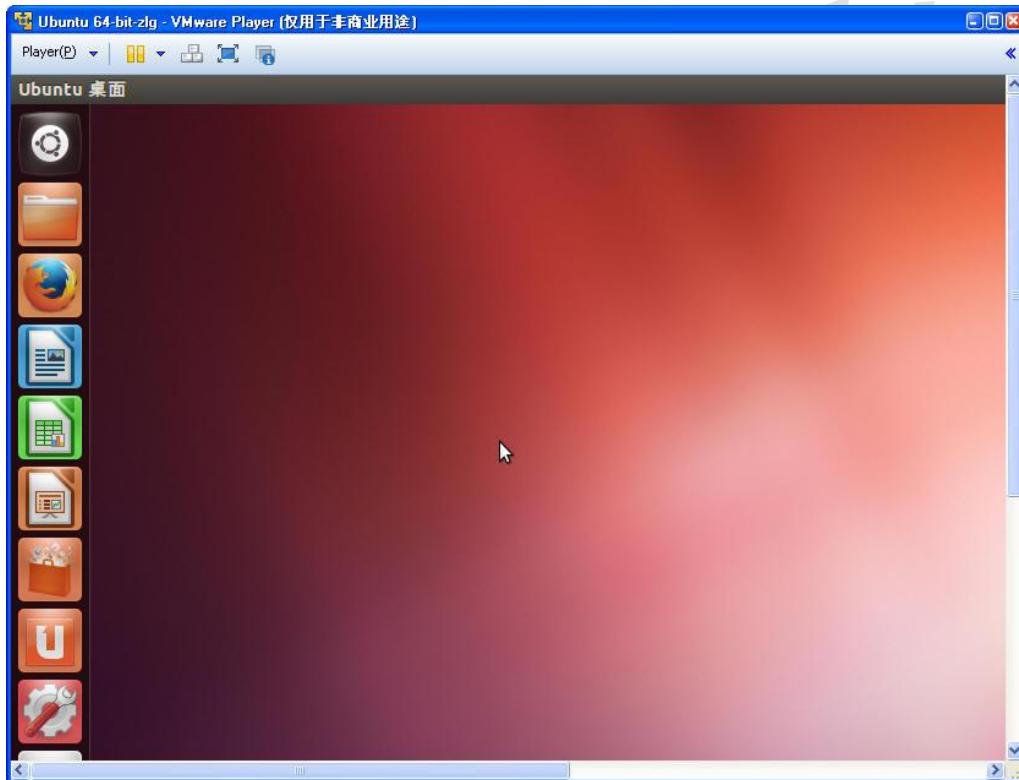


图 2.13 VMware Player 成功启动 Ubuntu 虚拟机

Ubuntu 系统在 VMware Player 中成功启动后，可以先阅读 2.7 小节，初步了解 Ubuntu 后，即可进入第 3 章，开始学习 Linux 命令。

如果以后想学习安装 Ubuntu，可以在另外的目录新建虚拟机，并安装新的 Ubuntu 系统。

在有些电脑上，特别是笔记本电脑，有可能出现启动登录后黑屏的状况，出现这种状况的原因有可能是 VMware 软件设置默认开启了“加速 3D 图形”选项，进入关闭即可。

先关闭虚拟机系统，打开虚拟机并装载虚拟机配置文件，在 VMware Player 主界面，选择“编辑虚拟机设置”，在“硬件”选项卡中选择“显示器”，将“加速 3D 图形”前面的勾去掉，如图 2.14 所示。

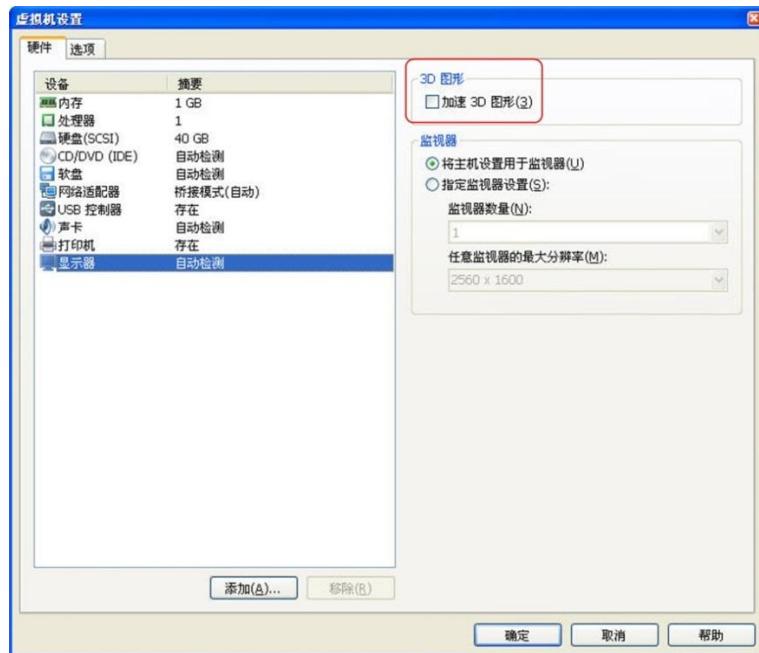


图 2.14 关闭 3D 图形加速

2.5 创建和配置虚拟机

2.5.1 创建虚拟机

双击桌面的 VMware Player 启动快捷方式图标^④ 打开 VMware Player 软件，运行界面如图 2.15 所示。点击“创建新虚拟机(N)”，可以创建一台虚拟机。



图 2.15 创建新虚拟机

在弹出的向导欢迎界面中选择“稍后安装操作系统(S)”，然后点击“下一步”按钮，如



图 2.16 所示。

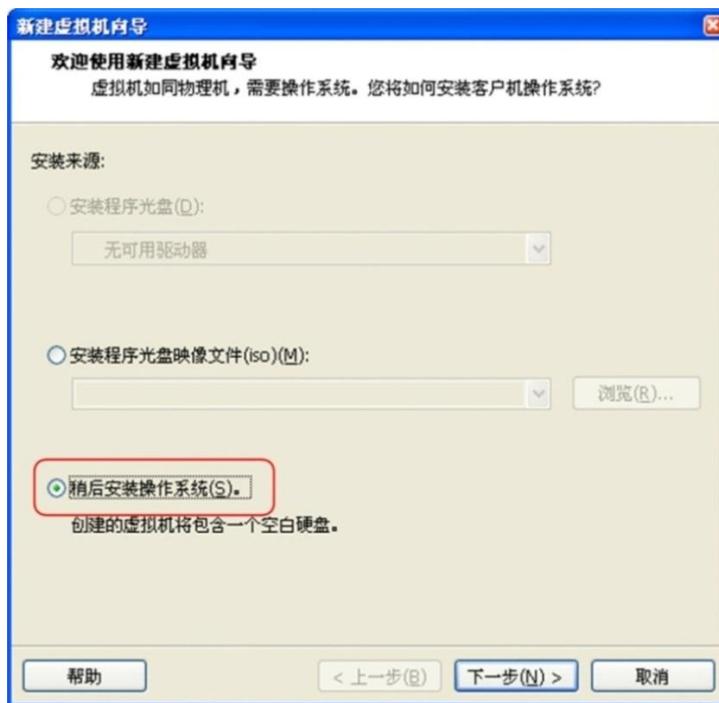


图 2.16 选择“稍后安装操作系统”

在图 2.17 所示的“选择客户机操作系统”界面，选择“Linux(L)”，并在版本下拉框中选择“Ubuntu 32 位”或者“Ubuntu 64 位”。请根据实际计算机硬件情况进行选择，图 2.17 的示例是安装 Ubuntu 64 位系统。



图 2.17 选择客户机操作系统

对于 64 位处理器的计算机，安装了 32 位操作系统，开启了虚拟化支持的话，在安装虚拟机的时候也可以选择 64 位 Linux 系统。



点击“下一步”，进入“命名虚拟机”设置界面，可设置虚拟机名称以及存储位置，如图 2.18 所示。名称可用默认名称，也可以更改为自己满意的名称；但存放位置则不推荐用默认值，必须放置到有足够空闲空间的硬盘分区上。



图 2.18 设置虚拟机名称及存储位置

设置好确认无误后，继续点击“下一步”，进入“指定磁盘容量”界面，如图 2.19 所示。

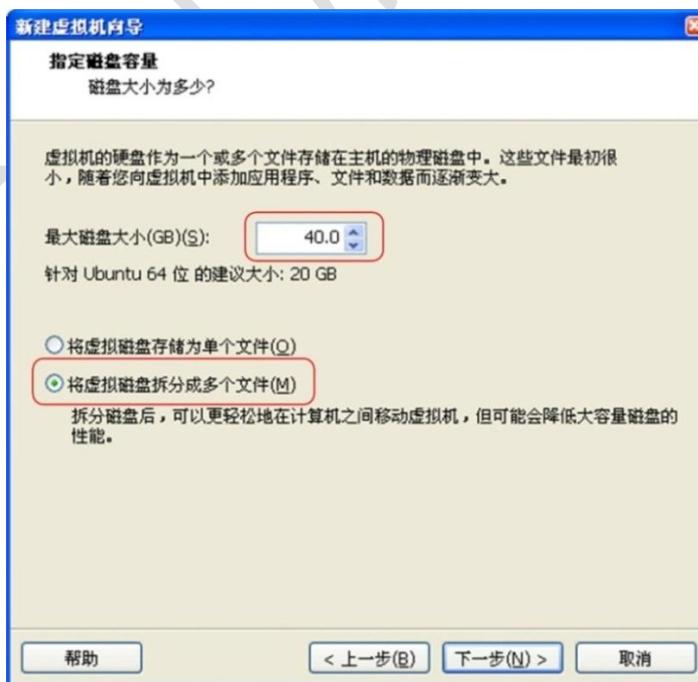


图 2.19 指定虚拟磁盘容量

磁盘容量设置，建议 40GB 以上。除了安装 Ubuntu 操作系统本身外，还会安装嵌入式 Linux 开发的各种工具，以及对应的源码等，都需要较大空间。



图 2.19 示例分配了 40GB 虚拟磁盘，会产生虚拟磁盘文件，但并不会立即占用 40GB 实际硬盘空间。虚拟磁盘文件会在使用过程中逐步增大，直到最大容量 40GB。尽管不会立即占用 40GB 硬盘空间，但是为了将来方便使用，必须保证放置虚拟机的磁盘有超过 40GB 的空闲空间。

由于虚拟磁盘文件大小会在使用中变化，分割成多个文件是比较好的选择。

确认设置无误后，点击“下一步”按钮，出现已经创建完毕的虚拟机的信息概览，如图 2.20 所示，点击完成即可。



图 2.20 完成虚拟机创建

2.5.2 虚拟机设置

创建得到的虚拟机，默认采用典型值，有的参数可能不是很合适，可以根据实际需要进行调整。点击图 2.20 界面的“自定义硬件”，可以对虚拟计算机硬件进行调整定制。

1. 内存调整

系统默认的内存值通常都比较小，建议适当增加，如在有 4GB 内存或以上的计算机上，给虚拟电脑的内存可以设置为 2GB。进入自定义硬件界面后，在“硬件”选项卡选中“内存”，得到如图 2.21 所示的界面，在这个界面可以设置内存大小。

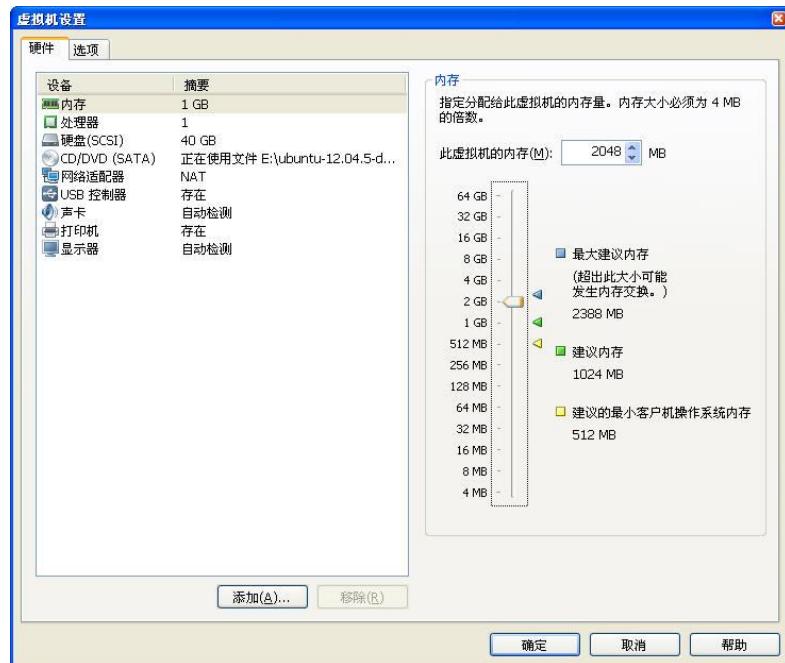


图 2.21 内存调整界面

2. 虚拟网卡设置

不少 VMware 用户都碰到过 VMware 的虚拟网卡的问题，这里重点介绍一下。进入自定义硬件界面后，在“硬件”选项卡选择“网络适配器”，得到如图 2.22 所示的网卡设置界面。

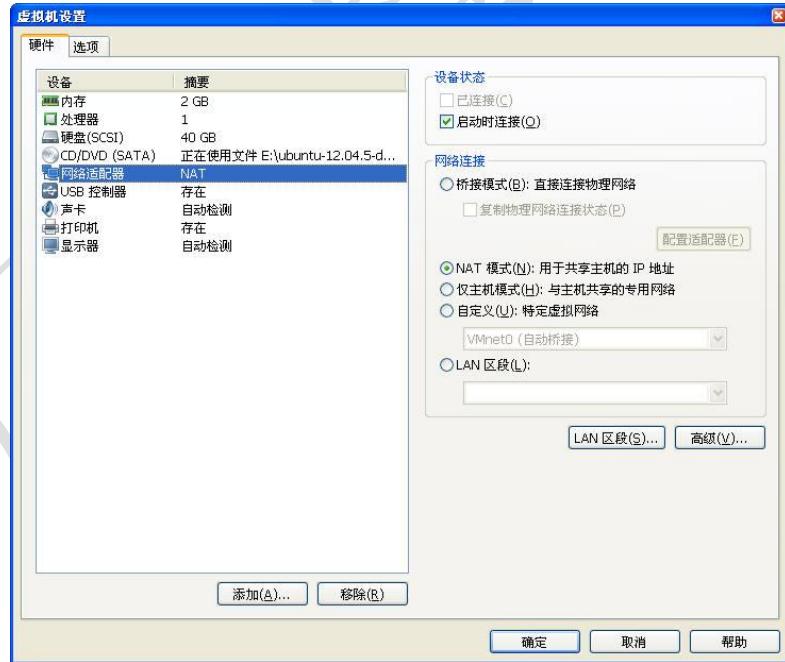


图 2.22 虚拟网卡设置

虚拟网卡有 3 种模式，分别如下：

- 桥接模式

在这种模式下，VMWare 虚拟出来的操作系统就像是局域网中的一台独立的主机，它可以访问网内任何一台机器。



在桥接模式下，虚拟系统和宿主机器的关系，就像连接在同一个 Hub 上的两台电脑。用户需要手工为虚拟系统配置 IP 地址、子网掩码，而且还要和宿主机器处于同一网段，这样虚拟系统才能和宿主机器进行通信。同时，由于这个虚拟系统是局域网中的一个独立的主机系统，那么就可以手工配置它的 TCP/IP 配置信息，以实现通过局域网的网关或路由器访问互联网。

➤ 在进行嵌入式 Linux 开发，要目标板通过 NFS 挂载虚拟机的 NFS 共享目录的话，必须将虚拟网卡配置为桥接模式。

- NAT 模式

使用 NAT 模式，就是让虚拟系统借助 NAT（网络地址转换）功能，通过宿主机器所在的网络来访问公网，也就是说，使用 NAT 模式可以实现在虚拟系统里访问互联网。NAT 模式下的虚拟系统的 TCP/IP 配置信息是由 VMnet8(NAT)虚拟网络的 DHCP 服务器提供的，虚拟机无法正常对主机所连网络中的其它主机提供普通的网络服务，如 TFTP、NFS 和 FTP 等。

采用 NAT 模式最大的优势是虚拟系统接入互联网非常简单，用户不需要进行任何其它的配置，只需要宿主机器能访问互联网即可。

- 仅主机模式

在某些特殊的网络调试环境中，要求将真实环境和虚拟环境隔离开，这时用户就可采用仅主机（Host-Only）模式。在 Host-Only 模式中，所有的虚拟系统是可以相互通信的，但虚拟系统和真实的网络是被隔离开的。

2.6 安装 Ubuntu

Ubuntu 的安装过程，无论在硬件实体安装还是虚拟机安装，大致过程是相同的。以下的安装过程都是在虚拟机中完成的，物理实体安装也是一样的。

2.6.1 实体机安装前准备

如果进行物理实体安装，需要制作启动盘。可将 ISO 刻成启动光盘，也可用 unetbootin 软件制作一个 USB 启动安装盘。这里讲述如何制作 USB 启动盘。

- 将 U 盘插入电脑（U 盘容量建议 2GB 以上），查看 U 盘对应的盘符；
- 打开 unetbootin-windows 软件界面，如图 2.23 所示。选中“光盘镜像”，并打开已经下载的 Ubuntu ISO 文件；
- 在驱动器一栏选择 U 盘对应的盘符，确定无误后点击“确定”，开始制作启动盘。



图 2.23 打开 unetbootin-windows 软件

制作时间较长，大约几分钟到十来分钟不等，取决于计算机性能以及 U 盘的读写性能，制作过程如图 2.24 所示。



图 2.24 制作 USB 启动盘

当 USB 启动盘制作完成后，将显示如图 2.25 所示界面。这时不要点击“现在重启”，直接点击“退出”即可。



图 2.25 USB 启动盘制作完成

当使用光盘安装时，需要在电脑的 BIOS 设置为从光驱启动，然后在光驱放入安装光盘，启动电脑进入 Ubuntu 安装程序。

当使用 USB 启动盘安装时，需要在电脑的 BIOS 设置为从 USB 启动，然后插入 USB 启动盘，启动电脑进入 Ubuntu 安装程序。

2.6.2 虚拟机安装前准备

在 VMware Player 中安装 Ubuntu，可以直接使用 ISO 文件，无需刻盘，也无需制作启动盘，只需将从 Ubuntu 官网下载的 ISO 文件加载进虚拟机即可。

打开 VMware Player 的主页界面，如图 2.26 所示。



图 2.26 VMware Player 的主页界面



在新建的虚拟机上点击“编辑虚拟机设置”进入虚拟机设置界面，选中“硬件”选项卡的“CD/DVD”，设置 ISO 光盘文件的路径，并请确认勾选“启动时连接”，如图 2.27 所示。

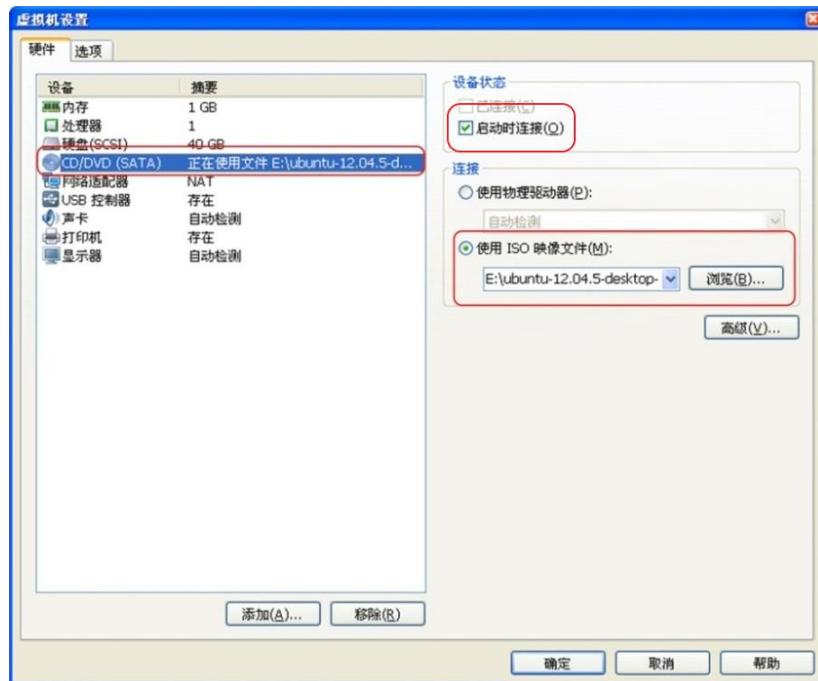


图 2.27 设置 ISO 文件

配置完虚拟机后，软件回到了 VMware Player 的主页界面，如图 2.28 所示，此时选中刚刚创建的虚拟机后，再点击位于右侧的“播放虚拟机”则可以启动该虚拟机并进入 Linux 系统的正式安装流程。

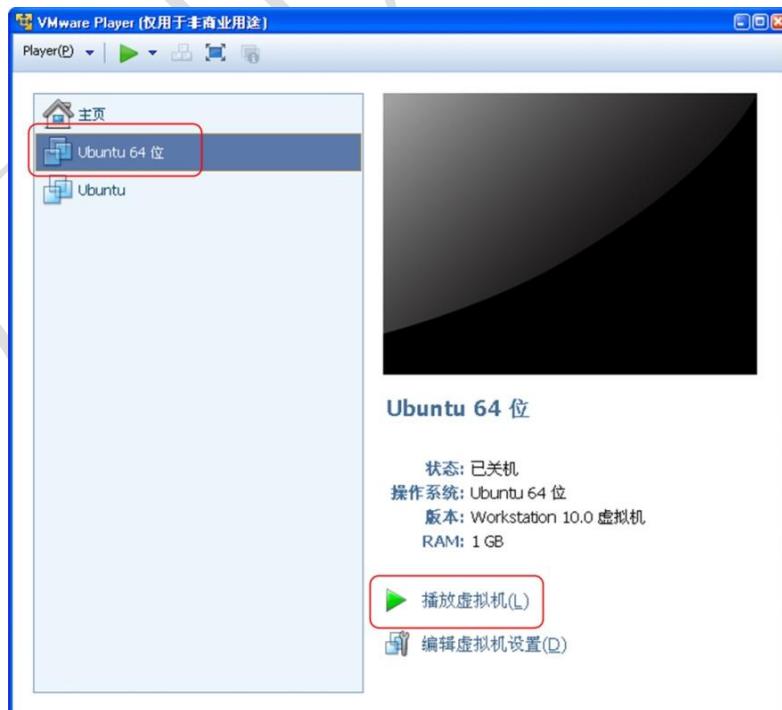


图 2.28 启动虚拟机



虚拟机启动后，可能会弹出一些警告对话框，通常无需理会，虚拟机正常启动后出现如图 2.29 所示的界面。

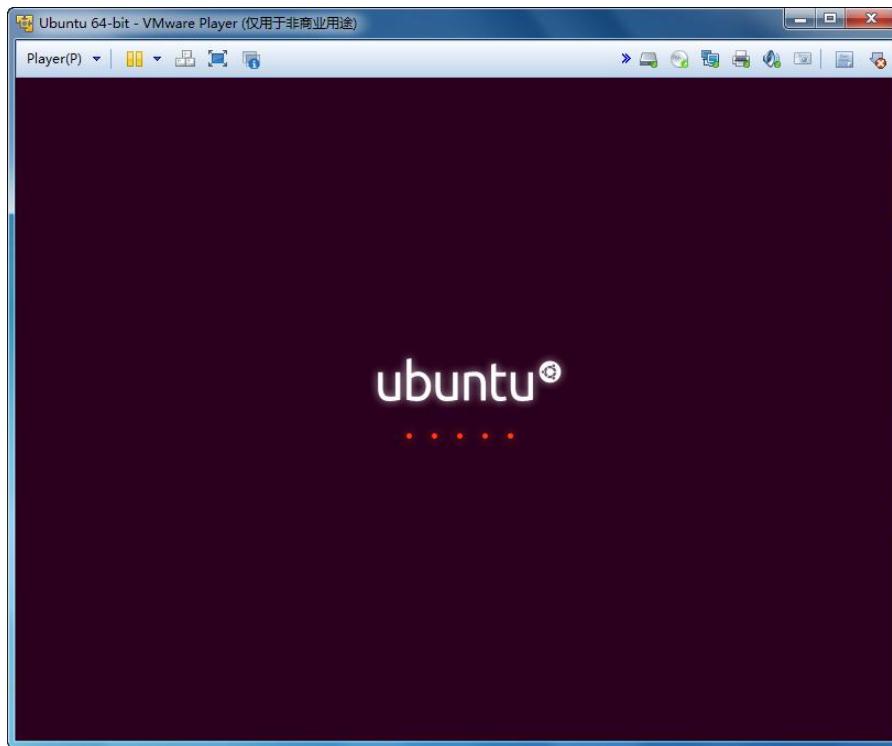


图 2.29 Ubuntu 安装镜像正常启动

2.6.3 正式安装 Ubuntu

Ubuntu 安装镜像正常启动后，会出现如图 2.30 所示的欢迎界面。请在左侧的列表选择“中文（简体）”，然后点击“安装 Ubuntu”按钮进入下一步安装。



图 2.30 选择系统语言

在图 2.31 所示的“准备安装 Ubuntu”界面，会给出当前安装环境的检测结果，包括系统空闲磁盘以及是否联网等信息，点击“继续”进行下一步安装。



图 2.31 点击“继续”按钮

紧接着出现图 2.32 所示的“安装类型”选择界面，如果在全新硬盘或虚拟机安装 Ubuntu，可以选择“清除整个磁盘并安装 Ubuntu”，**但如果用户是双系统安装，必须选择“其它选项”，具体操作请参考其它资料。**



图 2.32 清除整个磁盘

请点击“继续”按钮，出现“清除整个磁盘并安装 Ubuntu”的界面，点击“现在安装”按钮开始安装。

注意：物理实体安装会清空整个物理硬盘，但是虚拟机安装仅仅是清空创建虚拟机时创建的虚拟磁盘，并不会清空物理硬盘上的其它数据，无需担心。



图 2.33 点击“现在安装”按钮

紧接着会出现如图 2.34 所示的地理位置选择界面，请请选择“shanghai”或者“Chongqing”。

当然也可以选择其它地方，如 Hongkong 等，但是会影响到系统时间和语言。很遗憾，这里并没有“Beijing”可选。

进入键盘布局设置界面。



图 2.34 选择系统时间所在时区

然后点击“继续”按钮，进入如图 2.35 所示的“键盘布局”设置界面。选择“汉语”并点击“继续”。

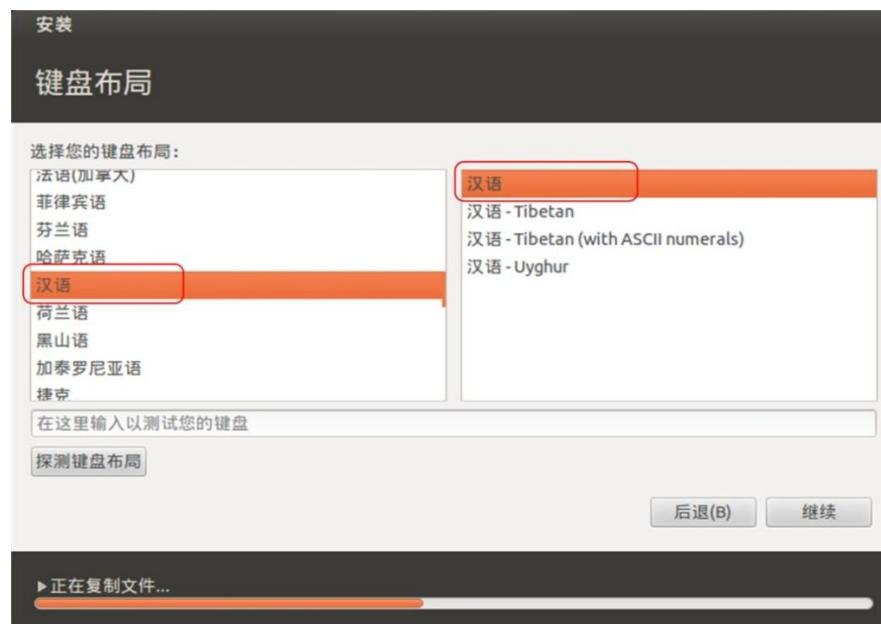


图 2.35 选择键盘布局

将进入“您是谁”的用户信息设置界面。在这里填写用户的相关信息，包括计算机名、用户名、用户密码等信息。

为方便后面的描述，这里设置了计算机名为 Linux-host、用户名为 vmuser、用户密码为 vmuser，如图 2.36 所示。为方便系统启动后，开机能自动进入桌面，这里设置登录方式为“自动登录”。

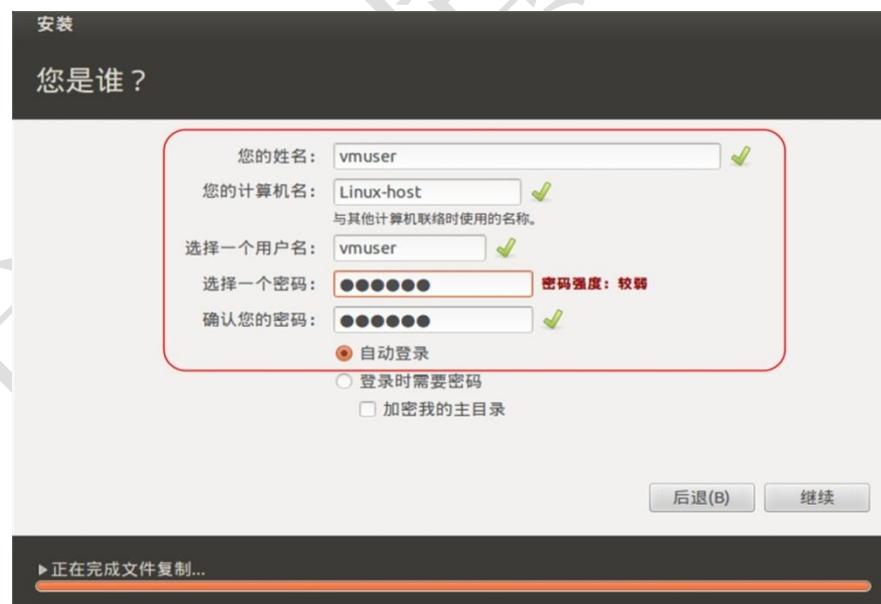


图 2.36 设置用户信息

设置完成后，点击“继续”按钮然后等待系统安装完成。整个过程时间大约 30 分钟到 1 个小时不等，取决于计算机情况。

系统安装完成后，将出现如图 2.37 所示的“安装完成”对话框，点击“现在重启”按钮完成重启。

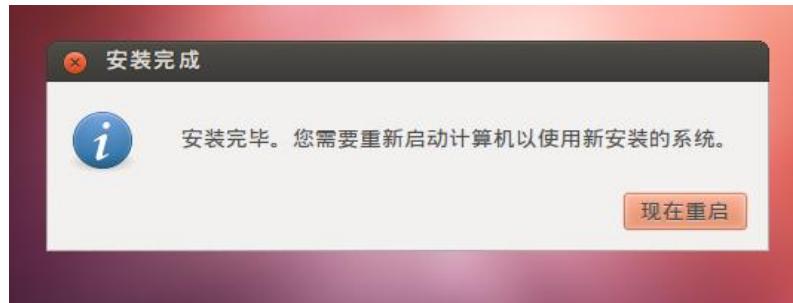


图 2.37 系统安装完成

2.7 初识 Ubuntu

2.7.1 Ubuntu 桌面

Ubuntu 启动后，进入桌面系统，桌面环境如图 2.38 所示。

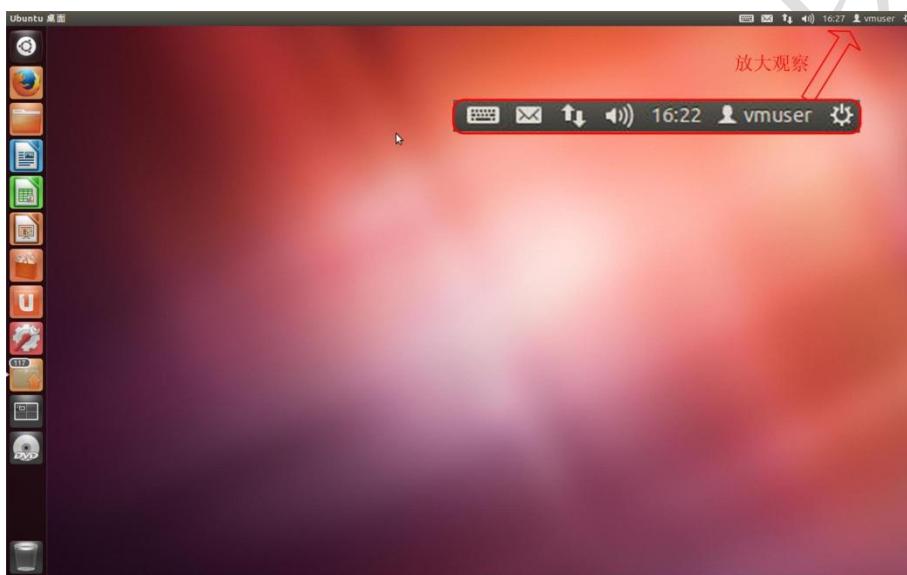


图 2.38 Ubuntu 桌面环境

在桌面的右上角显示的是输入法、时间、登录用户名的信息。

桌面的右侧是任务栏。在任务栏上，可以看到 Ubuntu 为用户准备了一些常用的软件：

 火狐浏览器，上网用：

 文件浏览器：用于浏览计算机上的文件。

 文档处理软件，类似 Windows Office 的 Word 软件；

 表格处理软件，类似 Windows Office 的 Excel 软件。

 演示文稿软件，类似 Windows Office 的 PowerPoint。



软件中心，为用户提供海量的软件下载、安装；



系统设置。

2.7.2 输入法

在桌面的右上角点击 图标打开输入法菜单，如图 2.39 所示。



图 2.39 输入法菜单

在该菜单可以看到系统默认设置了两种汉字输入法。中/英文输入法切换的默认快捷键是“Ctrl+空格”；中文之间输入法切换的默认快捷键是“Alt+Shift”。若用户需要设置输入法和输入法切换的快捷键，请点击“首选项”菜单项进行设置。

2.7.3 系统设置

在任务栏点击 图标即可打开系统设置窗口，如图 2.40 所示。用户可以在这里对系统进行设置，这和 Windows 的“控制板面”类似。



图 2.40 系统设置窗口

如果用户是使用虚拟机安装 Ubuntu，系统启动后，桌面有可能不是默认全屏显示。这时可以在系统设置里点击“显示”图标，进入“显示”配置界面设置分辨率。

2.7.4 搜索软件和文件

在 Ubuntu 的桌面环境，用户可以用 Dash 工具查找软件、文件和目录。在任务栏点击图标，即可打开 Dash 的主页，如图 2.41 所示。



图 2.41 Dash 主页



在 Dash 的主页显示了用户最近打开的程序和文件的图标。Dash 的主页还有一个“搜索”输入框，用于搜索安装程序或者文件，支持模糊查找。Dash 主页下方有多个快捷方式方便用户搜索，功能如表 2.1 所列。

表 2.1 快捷方式说明

图标	应用	应用说明
	主页	—
	应用程序搜索	支持用户按类别（办公、附件、互联网、教育……）搜索程序
	文件搜索	支持用户按文件大小、修改时间和类型（目录、视频、文档、图片、演示文稿……）搜索文件
	视频文件搜索	空
	音频文件搜索	支持用户按音乐风格搜索音频文件

这些快捷方式简单易用，这里不再多述。

2.7.5 打开终端

在 Dash 的搜索输入框输入 “terminal”，即可搜索到终端程序。在实际应用中，并不需要写全，输入前面几个字母，系统就能自动列出相关软件，如输入 “te”，即可出现包含终端在内的程序，如图 2.42 所示。

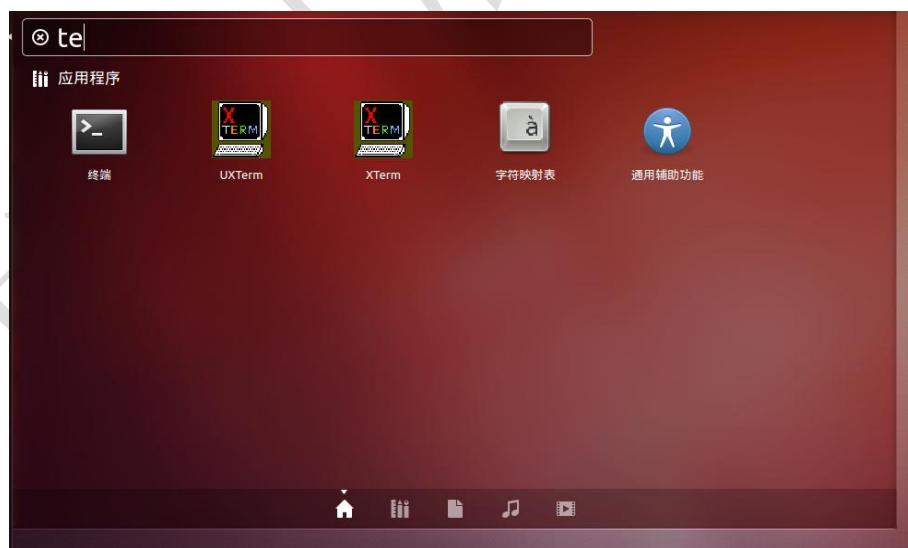


图 2.42 搜索终端程序

小贴士：在中文版 Ubuntu 中，搜索支持中文拼音单字匹配，例如“终端”，输入“终”的拼音“zhong”或者“端”的拼音“duan”，都可以搜索到终端程序。如果记不起某个程序英文怎么写，而知道大概中文，就可以采用这种方法。

点击终端图标即可打开终端的窗口，如图 2.43 所示。按“Ctrl+Alt+T”组合键也可以打开终端窗口。终端窗口的大小，可以由用户用鼠标拖伸，或最大化。

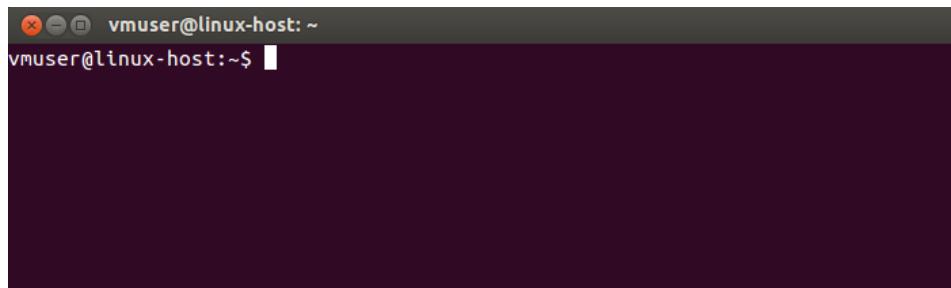


图 2.43 终端界面

按“Ctrl+Shift+T”键可以在终端窗口再打开一个终端的标签，如图 2.44 所示。按“Alt+1”或“Alt+2”切换终端标签。

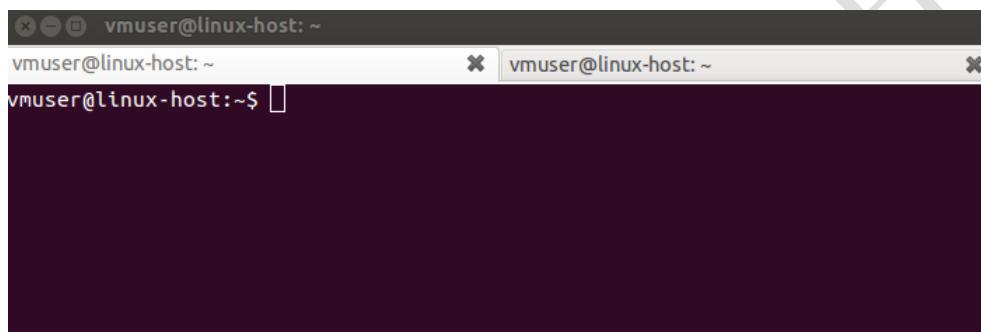


图 2.44 再打开一个终端

2.7.6 安装软件

在 Ubuntu 一般使用 `apt-get` 命令安装软件。但前提是电脑需要连接到互联网。`apt-get` 命令在执行时会在网上下载指定的软件包，然后完成安装。

例如，安装 vim 的方法是：

```
$ sudo apt-get install vim
```

若要卸载安装好的 vim 方法是：

```
$ sudo apt-get remove vim
```

Ubuntu 也提供很好的图形界面让用户比较方便查找、安装自己所需的软件。在 Ubuntu 桌面的左侧任务栏，点击  图标，即可打开如图 2.45 所示的软件中心。



图 2.45 Ubuntu 软件中心

用户可以很方便地在这里查找软件、下载软件、完成安装，或者卸载安装好的软件。这和“app store”类似。



第3章 开始使用 Linux

本章重点介绍 Linux 的常用操作和命令。在介绍命令之前，先对 Linux 的 Shell 进行了简单介绍，然后按照大多数用户的使用习惯，对各种操作和相关命令进行了分类介绍。对相关命令的介绍都力求通俗易懂，都给出操作实例，使读者能够照着实际操作，并得到正确结果。命令是 Linux 操作系统的利器，务必掌握好，当然不可能一下子熟练掌握，但是只要多加练习，就可熟能生巧，运用自如。最后对 Linux 的环境变量也进行了必要的介绍。

3.1 Linux Shell

3.1.1 Shell 是什么？

前面已经提到过，Linux 系统为用户提供了多种用户界面，包括 Shell 界面、系统调用和图形界面。其中 Shell 界面是 UNIX/Linux 系统的传统界面，也可以说是最重要的用户界面，无论是服务器、桌面系统还是嵌入式应用，都离不开 Shell。

Shell，英文本意是外壳，Linux Shell 就是 Linux 操作系统的外壳，为用户提供使用操作系统的接口，是用户与 Linux 系统进行交互的重要接口。登录 Linux 系统或者打开 Linux 的终端，都将会启动 Linux 所使用的 Shell。

Linux Shell 一个命令解释器，是 Linux 下最重要的交互界面，从标准输入接收用户命令，将命令进行解析并传递给内核，内核则根据命令，作出相应的动作，如果有反馈信息，则输出到标准输出上，示意过程如图 3.1 所示。嵌入式 Linux 的标准输入和输出都是串口终端。

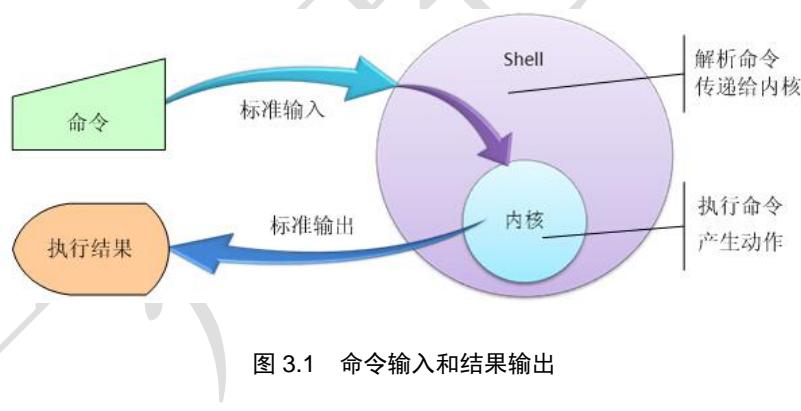


图 3.1 命令输入和结果输出

Shell 既能解释自身的内建命令，也能解释外部命令，如系统某个目录下的可执行程序。Shell 首先判断是否是自己的内建命令，然后再检查是不是系统的应用程序，如果不是内建命令，在系统中也找不到这个应用程序，则提示错误信息，如果找到了应用程序，则将程序分解为系统调用传递给内核。

Shell 也是一种解释型的程序设计语言，并且支持绝大多数高级语言的程序元素，如变量、数组、函数以及程序控制等。Shell 编程简单易学，任何在 Shell 提示符中输入的命令都可以放到一个可执行的 Shell 程序文件中。Shell 文件其实就是众多 Linux 命令的集合，也称为 Shell 脚本文件。

3.1.2 Shell 的种类和特点

Linux Shell 有多种 Shell，比较通用且有标准的主要分为两类：Bourne Shell(sh)和 C Shell(csh)，各自包括几种具体的 Shell，具体如表 3.1 所列。



表 3.1 常见 Linux Shell

类别	名称	说明
Bourne Shell	Bourne shell (sh)	由贝尔实验室开发, UNIX 最初使用的 Shell
	Bourne Again shell (bash)	GNU 操作系统上默认的 Shell
	Korn shell (ksh)	
	POSIX shell (sh)	Korn Shell 的变种
C Shell	C Shell (csh)	目前使用较少
	TENEX/TOPS C shell (tcsh)	

Bourne Shell 是 UNIX 最初使用的 shell, 在每种 UNIX 上都可以使用。Bourne Shell 的优点是在 Shell 编程方面很好, 缺点是用户的交互不如其他几种 Shell。

Bourne Again Shell 简称 Bash, 是 Bourne Shell 的扩展, 与 Bourne Shell 完全向后兼容, 在 Bourne Shell 的基础上增加了很多新特性。Bash 提供了命令补全、命令编辑和命令历史表等功能, 还包含了很多 C shell 和 Korn shell 中的优点, 使用灵活, 界面友好, 编程方便, 是 GNU/Linux 操作系统的默认 Shell。

Korn Shell 由 AT&T 的 Bell 实验室 David Korn 开发, 吸收了所有 C Shell 的交互式特性, 并融入了 Bourne shell 的语法, 与 Bourne shell 完全兼容。

C Shell 由 Bill Joy 在 BSD 系统上开发, 增强了用户交互功能, 并将编程语法变成了 C 语言风格, 还增加了命令历史、别名、文件名替换、作业控制等功能。目前使用较少。

在不同发行版中, 所采用的默认 Shell 也有所不同, 如 Redhat 和 Fedora 中默认 Shell 为 bash, Ubuntu 中用了 dash。无论用哪种 Shell, 登录系统后系统将运行一个 Shell 进程。根据不同用户, Shell 提供不同的命令提示符, root 用户的提示符为 “#”, 普通用户的命令提示符为 “\$”, 在命令提示符下输入命令即可操纵系统。

尽管不同发行版的默认 Shell 有可能不同, 但是所采用的 Shell 一般都具有如下特性:

- 具有内置命令可供用户直接使用;
- 支持复合命令: 把已有命令组合成新的命令;
- 支持通配符 (*、?、[]);
- 支持 TAB 键补齐;
- 支持历史记录;
- 支持环境变量;
- 支持后台执行命令或者程序;
- 支持 Shell 脚本程序;
- 具有模块化编程能力, 如顺序流控制、条件控制和循环控制等;
- Ctrl+C 能终止进程。

3.2 Linux 常见命令

本节将介绍在嵌入式 Linux 开发过程中常用的一些操作和相关命令, 以便读者进一步加深对 Linux 的了解。命令是 Linux 最重要的人机交互界面之一, 学习和掌握 Linux 命令是学习 Linux 必须经历的阶段。在 Shell 下, 一些命令加上一些参数, 或者几个简单命令进行组合, 可以完成在图形界面下需要经过复杂操作才能完成的任务。“简单就是美”在 Linux 的命令中得到了很好的体现。

Linux 的命令通常会有很多选项和参数, 但日常操作中用到的都不多, 在这里也仅仅择



取常用的进行介绍，更多或者完整的 Linux 命令请参考 Linux 命令手册或者其它资料。在接触具体的命令之前，先对 Linux 命令的特点做一个概括，也是使用 Linux 命令的一些注意事项：

- 大多数命令都有各种参数和选项；
- 大多数命令的参数可以组合使用（相斥参数除外）；
- 用“命令 --help”或者“man 命令”可以获取相应命令的详细用法；
- 命令/工具不同版本所支持的参数可能会有所差异；
- 命令区分大小写，包括参数；
- Shell 支持 TAB 键命令补齐，输入命令开头字母，按 TAB 键能补齐命令。

3.2.1 导航命令

打开 Linux 的虚拟终端后，一般都停在用户主目录下。当前目录下有什么？如何进入到其它目录？进入其它目录后，如何才能知道当前的确切位置？像这类操作通常称之为导航。Linux 下，能帮助进行导航的命令有 3 个：ls、cd 和 pwd。

1. 查看当前目录的内容

打开 Linux 虚拟终端后，查看当前目录下的内容，几乎是所有 Linux 使用者的习惯。查看当前目录下有什么文件和目录，然后再进行其它操作。查看当前目录下的内容的命令是 ls，简单的输入 ls 就可以了，参考图 3.2。



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ ls
examples.desktop 公共的 模板 视频 图片 文档 下载 音乐 桌面
vmuser@Linux-host:~$
```

图 3.2 ls 命令结果

ls 命令应该是学习 Linux 的第一个命令。ls 命令支持选项，加上不同选项，可以按不同条件查看或者按不同方式排序结果。用法：

```
$ ls [选项]
```

下面给出一些常用选项和说明，如表 3.2 所列。

表 3.2 ls 命令常用选项

选项	说明	
空	按字母顺序列出当前目录下的所有非隐藏文件（包括目录）	
-a	按字母顺序列出当前目录下的所有文件，包括隐藏文件	
-l	列出当前目录下的所有文件，包括文件长度、拥有者、权限和时间戳等信息	
-t	按文件最后修改时间列出文件	
-F	按类型列出所有文件，在文件末尾用不同符号区分：	
	斜线(/)	表示目录
	星号(*)	表示可执行文件
	@符号	表示链接文件
--color	以不同颜色显示目录、普通文件、可执行文件、压缩文件以及链接文件等	

说明：

- (1) Linux 区分大小写，在输入的时候需要特别注意；
- (2) 各参数可以任意组合，如 ls -la；



(3) 支持通配符*、?等。

使用范例，以详细列表查看当前目录下的全部内容，可使用 ls -la 命令，结果如图 3.3 所示。

```
vmuser@Linux-host:~$ ls -la
总用量 192
drwxr-xr-x 23 vmuser vmuser 4096 1月 4 09:41 .
drwxr-xr-x  3 root   root   4096 11月 13 2014 ..
-rw-----  1 vmuser vmuser 8084 1月 4 09:40 .bash_history
-rw-r--r--  1 vmuser vmuser 220 11月 13 2014 .bash_logout
-rw-r--r--  1 vmuser vmuser 3486 11月 13 2014 .bashrc
drwx----- 17 vmuser vmuser 4096 11月 22 2014 .cache
drwx----- 11 vmuser vmuser 4096 1月 4 09:36 .config
drwx----- 3 vmuser vmuser 4096 11月 13 2014 .dbus
-rw-r--r--  1 vmuser vmuser 26 1月 4 08:37 .dmrc
-rw-r--r--  1 vmuser vmuser 8445 11月 13 2014 examples.desktop
drwxr-xr-x  2 vmuser vmuser 4096 11月 20 2014 .fontconfig
drwx----- 5 vmuser vmuser 4096 1月 4 08:37 .gconf
drwx----- 4 vmuser vmuser 4096 11月 13 2014 .gnome2
-rw-rw-r--  1 vmuser vmuser 195 1月 4 08:37 .gtk-bookmarks
dr-x----- 2 vmuser vmuser 0 1月 4 08:37 .gvfs
-rw----- 1 vmuser vmuser 7348 1月 4 08:37 .ICEauthority
-rw----- 1 vmuser vmuser 42 11月 26 2014 .lessht
drwxr-xr-x  3 vmuser vmuser 4096 11月 13 2014 .local
drwx----- 3 vmuser vmuser 4096 11月 13 2014 .mission-control
drwx----- 4 vmuser vmuser 4096 11月 22 2014 .mozilla
-rw-r--r--  1 vmuser vmuser 675 11月 13 2014 .profile
drwx----- 2 vmuser vmuser 4096 1月 4 08:37 .pulse
-rw----- 1 vmuser vmuser 256 11月 13 2014 .pulse-cookie
```

图 3.3 ls -la 命令结果

ls -la 结果中，以点号（.）开始的是隐藏文件。

在 Linux 下，隐藏一个文件只需将文件改名为点号（.）开始的文件名即可，而 Windows 下，通常需要修改文件属性。

2. 切换工作目录

得知所处目录下的内容后，可以根据需求进行操作。如果想进入到更深的目录中去，或者进入到系统其它目录中去，又该如何操作？这就要用到 cd 命令。cd 命令是 change directory 的缩写，用于改变工作目录，与 MS-DOS 的 cd 命令类似。用法：

\$cd 目标路径

Linux 下路径的表示方法，详见表 3.3。

表 3.3 Linux 下路径的表示方法

表示方法	说明
/	根目录
句点（.）	当前目录
句点 2（..）	上一层目录
~	当前用户的主目录，一般为/home/username，如当前登录用户为 user，则~表示/home/user 目录，cd 命令不加任何参数，将切换到用户主目录（~）
短横线（-）	上一次工作目录，cd - 可切换至切换之前的工作目录

说明：

(1) Linux 下目录、计算机名和域名之间都是用斜线（/）分开，而非反斜线（\）；



(2) Linux 下切换目录，可用相对路径，亦可用绝对路径。

假定当前在用户主目录 (~) 下，先进入目录 “/etc/network” 目录，然后切换到 “/etc/network/if-down.d” 目录，接下来在 “/etc/network/if-post-down.d” 和 “/etc/network/if-down.d” 目录间切换，操作过程的命令如下：

```
vmuser@Linux-host:~$ cd /etc/network/  
vmuser@Linux-host:/etc/network$ cd if-down.d/  
vmuser@Linux-host:/etc/network/if-down.d$ cd ..../if-post-down.d/  
vmuser@Linux-host:/etc/network/ if-post-down.d $ cd -
```

实际操作结果如图 3.4 所示。

```
vmuser@Linux-host: /etc/network/if-post-down.d  
vmuser@Linux-host:~$ ls  
examples.desktop 公共的 模板 视频 图片 文档 下载 音乐 桌面  
vmuser@Linux-host:~$ cd /etc/network/  
vmuser@Linux-host:/etc/network$ ls  
if-down.d if-post-down.d if-pre-up.d if-up.d interfaces run  
vmuser@Linux-host:/etc/network$ cd if-down.d/  
vmuser@Linux-host:/etc/network/if-down.d$ cd ..../if-post-down.d/  
vmuser@Linux-host:/etc/network/if-post-down.d$ cd -  
/etc/network/if-down.d  
vmuser@Linux-host:/etc/network/if-down.d$ cd -  
/etc/network/if-post-down.d  
vmuser@Linux-host:/etc/network/if-post-down.d$
```

图 3.4 cd 命令操作示例

3. 查看当前路径

掌握了前面介绍的 ls 和 cd 两条命令后，几乎可以走遍整个 Linux 文件系统中所允许访问的目录。但是如果将 Linux 的命令提示设置为只提示当前目录名而不是显示完整的路径，那么在 Shell 下，如果进入的目录较深，有时候可能因为不清楚当前所在路径而导致“迷路”。pwd 命令是一个导航辅助命令，功能是打印当前所在路径，告知用户当前所处位置。用法很简单，在 Shell 终端中输入 pwd 即可：

```
vmuser@Linux-host: drivers$ pwd
```

如图 3.5 所示是一个简单范例。

```
vmuser@Linux-host: /etc/network/if-down.d  
vmuser@Linux-host:~$ cd /etc/network/if-down.d/  
vmuser@Linux-host:/etc/network/if-down.d$ pwd  
/etc/network/if-down.d  
vmuser@Linux-host:/etc/network/if-down.d$
```

图 3.5 pwd 命令结果

3.2.2 目录操作命令

Linux 下目录和文件都被称为文件，一般情况下不区分文件和目录，只是在特殊情况下加以区分。

1. 创建目录

创建目录在日常研发过程中是再常用不过的了。在图形界面下，单击右键选择新建文件夹可以完成目录创建工作。**在命令行下，用 mkdir 命令可以更简单快速的创建一个或者多个目录，甚至多级目录。**



mkdir 用于创建一个或者多个目录，加上选项也可以创建多级目录，这样的快捷性是图形界面无法做到的。mkdir 支持的选项如表 3.4 所列。用法：

```
$mkdir [选项] [参数] 目录
```

表 3.4 mkdir 命令支持的选项

参数	说明
-m	创建目录的同时指定访问权限
-p	如果所创建目录的父目录不存在，则一同创建父目录

创建一个目录。假如要在当前目录创建 new_dir 这个目录，命令如下：

```
vmuser@Linux-host: ~$ mkdir new_dir
```

实际操作和结果如图 3.6 所示。

```
vmuser@Linux-host: ~$ mkdir new_dir
vmuser@Linux-host: ~$ ls
examples.desktop  new_dir  公共的  模板  视频  图片  文档  下载  音乐  桌面
vmuser@Linux-host: ~$
```

图 3.6 创建一个目录

操作完成后，可以在文件浏览器看到新创建的 new_dir 目录，如图 3.7 所示。



图 3.7 新创建的目录

创建多个目录。假如要在当前目录下一次性创建 dir1、dir2、dir3 这 3 个目录，只需在 mkdir 命令后面依次写目录名即可，命令如下：

```
vmuser@Linux-host: test$ mkdir dir1 dir2 dir3
```

实际操作和结果如图 3.8 所示。



```
vmuser@Linux-host:~$ ls
examples.desktop 公共的 模板 视频 图片 文档 下载 音乐 桌面
vmuser@Linux-host:~$ mkdir dir1 dir2 dir3
vmuser@Linux-host:~$ ls
dir1 dir3 公共的 视频 文档 音乐
dir2 examples.desktop 模板 图片 下载 桌面
vmuser@Linux-host:~$
```

图 3.8 创建多个目录

操作完成后，可以在文件浏览器看到新创建的目录，如图 3.8 所示。



图 3.9 创建多个目录的效果

创建多级目录。假如需要创建 dir1 目录，并在其中创建 apps 子目录，同时在 apps 目录下再创建 hello 子目录，只需加上-p 参数，操作命令如下：

```
vmuser@Linux-host: test$ mkdir -p dir1/apps/hello
```

实际操作和结果如图 3.10 所示。

```
vmuser@Linux-host:~/dir1$ ls
dir1 dir3 公共的 视频 文档 音乐
dir2 examples.desktop 模板 图片 下载 桌面
vmuser@Linux-host:~/dir1$ mkdir -p dir1/apps/hello
vmuser@Linux-host:~/dir1$ cd dir1/apps/hello/
vmuser@Linux-host:~/dir1/apps/hello$ pwd
/home/vmuser/dir1/apps/hello
vmuser@Linux-host:~/dir1/apps/hello$
```

图 3.10 创建多级目录

2. 删除目录

如果一个目录不再需要，可以将其删除。Linux 下有两个命令可用于删除目录，rmdir 和 rm。

用 rmdir 删除空目录



rmdir 命令只能删除空目录，也可删除多级空目录。用法：

```
vmuser@Linux-host: test$ rmdir dir1 dir2
```

注意：rmdir 命令只能删除空目录，无法删除非空目录。

使用范例，删除空目录：

```
vmuser@Linux-host: test$ rmdir dir1 dir2
```

rmdir 也支持参数-p，表示删除某个目录后，如果父目录也成了空目录，则连父目录一并删除。范例：

```
vmuser@Linux-host: test$ rmdir -p dir4/dir5/dir6/
```

用 rm 命令删除

用 rmdir 命令很安全，不会误删数据，但是实际上用的不是很多，更常用的是用 rm 命令。rm 命令既可以删除文件，也可以删除目录而不管目录是否非空。用法：

```
$rm [选项] 文件/目录
```

rm 命令支持选项，用户可以控制删除过程，常用选项如表 3.5 所列。

表 3.5 rm 命令选项

选项	说明
-f	强制删除文件或者目录，无需用户确认
-i	删除文件或者目录之前，需用户确认
-r	递归删除，删除指定目录以及子目录下的文件
-v	显示删除过程

注意：删除命令，无论是删除目录还是文件，一旦删除，都将不可恢复，并不像 Windows 下或者桌面下会移动到回收站暂存。特别是一般的嵌入式并不设定“回收站”，所以在删除的时候请特别小心。

为了确保不误删文件，可使用 alias 别名，将 rm 命令设置为“rm-i”，这样每次删除都会有确认过程。用法：alias rm="rm -i”。

使用示例，强制删除某些文件和目录：

```
vmuser@Linux-host: test$ rm -fr dir3 video1.mpeg
```

如果加上-i 参数，则需要用户确认：

```
vmuser@Linux-host: test$ rm -i config.gz  
rm: 是否删除有写保护的普通文件 “config.gz” ?
```

这样，只有用户输入 y 后方可删除，输入 n 则保留文件。

3.2.3 文件操作命令

1. 创建空文件

在一些时候，为了某种特殊要求，需要在系统中创建一个空文件。touch 命令可以完成这个功能，创建的文件大小为 0，命令如下：

```
vmuser@Linux-host: ~$ touch a
```

操作过程和结果如图 3.11 所示。

```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ touch a
vmuser@Linux-host:~$ ls -la a
-rw-rw-r-- 1 vmuser vmuser 0 1月 4 10:00 a
vmuser@Linux-host:~$
```

图 3.11 创建空文件

2. 创建一个有内容的文件

Linux 下创建文件，可以使用文本编辑器如 vi 等来操作。对于简单的内容，可以用普通命令来创建文件。用普通命令创建非空文件，需要用到 Linux Shell 重定向机制，首先来了解一下重定向。

Linux Shell 终端启动的时候会打开 3 个标准文件：标准输入（stdin）、标准输出（stdout）和标准错误（stderr）。Shell 从标准输入（通常是键盘）接收命令，命令执行结果信息打印到标准输出（通常是终端屏幕）上，如有错误信息，则打印到标准错误（通常是终端屏幕）上，如图 3.12 所示。

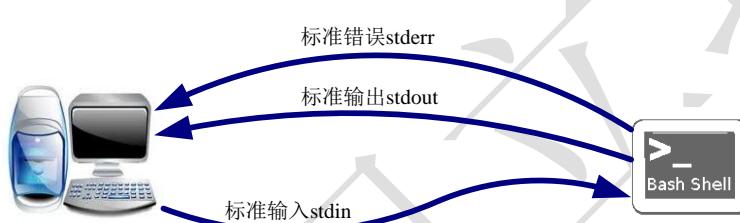


图 3.12 标准输入输出和标准错误

Shell 允许用户对输入输出进行重定向。输出重定向允许将输出信息从标准输出重定向到其它文件上，也可以重定向到某个设备如打印机上。如图 3.13 所示是将标准输出重定向到文件的示意图。

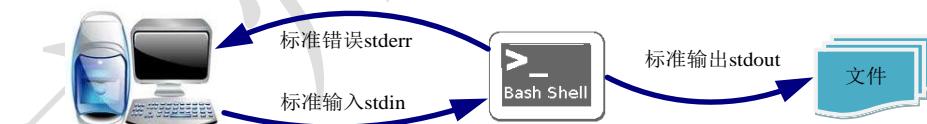


图 3.13 将标准输出重定向到文件

重定向在 Linux 下用“>”和“>>”表示，“>”表述输出到一个新文件中，而“>>”则表示输出到现有文件的末尾。如果文件已经存在，则直接操作文件，否则将创建新文件。

echo 命令将内容回显到标准输出；使用 echo 命令加上重定向可以创建一个带内容的非空文件，用法如下所示。**请注意，为了确保命令正确执行，“>”的左右两边最好留出一个空格：**

\$echo 内容或者“内容”	#输出到标准输出
\$echo 内容或者“内容” > 文件	#重定向到文件，如果文件不存在则创建新文件

回显内容如果不加引号，则用单空格替代多个连续空格，如果加了引号，则原封不动回显，图 3.14 所示操作过程显示了这些差异。



```
vmuser@Linux-host:~$ echo I am fine > a
vmuser@Linux-host:~$ echo "I am fine" >> a
I am fine
I am fine
vmuser@Linux-host:~$
```

图 3.14 输出重定向

可以看到，第一次输入的内容没有引号，连续空格被单空格替换了，而第二次加了引号，连续空格依然保留。

3. 查看文件类型

在 Windows 下，文件都有标准扩展，基本上可以根据文件扩展名来识别和判断文件类型，如.exe 是可执行文件，.c 是 C 代码文件、.zip 是压缩文件等。

Linux 与 Windows 不同，Linux 下的文件并没有标准扩展名，Linux 也不是根据扩展名来识别文件，而是根据文件头来识别文件类型。

尽管在大多数 Linux 发行版中，默认情况下都能以不同颜色显示目录以及不同类型的文件，但是根据颜色只能简单粗略判断常用类型文件。要准确确定一个文件的类型，必须依赖于 file 命令。file 命令能读取文件头并识别文件类型，包括目录。用法：

```
$ file 文件
```

说明：只能查看具有可读属性的文件。

file 命令支持通配符，如可以一次性查看当前目录下的全部文件类型，如图 3.15 所示。

```
vmuser@Linux-host:/etc/newt$ file *
palette:           symbolic link to `/etc/alternatives/newt-palette'
palette.original: ASCII text, with very long lines
palette.ubuntu:   ASCII text, with very long lines
vmuser@Linux-host:/etc/newt$
```

图 3.15 file 识别文件类型

file 命令还可以查看二进制可执行文件的详细信息，包括所运行的处理器体系结构。在 PC 机上用 gcc 编译得到的程序，用 file 命令查看：

```
vmuser@Linux-host: hello$ file hello
hello.x86: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, not stripped
```

而经过 arm-linux-gcc 交叉编译之后再次查看：

```
vmuser@Linux-host: hello$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses shared libs), for GNU/Linux 2.6.27,
not stripped
```

如果运行某个程序出现 cannot execute binary file 这个错误，很有可能是文件编译的目标体系结构与当前所运行的体系结构不一致，可用 file 命令确认。

4. 查看文件内容

准确判断文件类型后，对于 ASCII 码文件，无需使用特殊软件仅仅用 Linux 的命令就可以查看，如文本文件、C 代码文件、Shell 脚本文件等。Linux 下可以查看文件内容的命令有



好几个，如 more/less、head/tail、cat 等。

用 more 和 less 命令查看

more 和 less 两个命令都可用来浏览文本文件，可以分页查看文件内容，空格翻页。文件浏览完毕，按键盘 q 退出。用法：

```
$more/less 文件
```

相比来说 less 命令更加灵活，支持键盘的 Page Up 和 Page Down 键，可任意向前向后翻页浏览，并且还支持文本搜索。使用 less 打开文件后，输入/abc 可在文本中搜索字符串 abc，匹配的字符串高亮显示。如图 3.16 所示是用 less 命令打开文件后搜索 hello 字符串的截图。

```
vmuser@Linux-host: ~
static init __init hello_init(void)
{
    printk("hello, I'm ready! \n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk("I'll be leaving, bye!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

图 3.16 less 命令的字符串搜索

用 head/tail 命令查看

head 和 tail 这两个命令可分别查看文件头部和文件尾部，一般用于查看 ASCII 文件。默认显示 10 行，可加上参数指定显示内容的多少，支持的参数如表 3.6 所列。用法：

```
$head/tail [选项][参数] 文件
```

表 3.6 head 和 tail 支持的选项

参数	说明
-n [数字]	显示[数字]所指定的行数
-c [数字]	显示[数字]所指定的字节数

说明，数字的表示方法：b=512，kB=1000，K=1024，MB=1000*1000，M=1024*1024 等。

指定显示多少行，如查看文件头 20 行：

```
vmuser@Linux-host: hello$ head -n20 install.cf
```

指定显示多少字节，例如，指定查看 300 字节：

```
vmuser@Linux-host: hello$ head -c 300 install.cf
```

查看文件开头的 512 字节：

```
vmuser@Linux-host: hello$ head -c 1b install.cf
```

用 cat 命令查看

cat 命令可以将一个或者多个文件输出到标准输出上，可以用于文件查看。用法：

```
$ cat 文件
```

5. 文件合并

经常会有这么一种需求，将某个文件的内容添加到另外一个文件的末尾，或者要求对某



一个文件进行行编号，这样的工作在 Windows 下或者 Linux 图形界面下完成，都得花不少心思，基本上就得依赖于所使用的编辑软件。

尽管这样的工作比较复杂，但是在 Linux 命令行下，可以轻松解决，用 cat 命令可以几乎不费力就完成在图形界面下操作起来很复杂的工作。

cat 命令可以将一个或者多个文件输出到标准输出，如果将标准输出重定向到某个文件，则可以把多个文件合并为一个文件。用法：

```
$ cat [选项] 文件 1 文件 2 ... [>文件 3]
```

如果不加选项，则原封不动的显示各个文件，加上一些参数的话，可以对原文件进行一些处理，常用选项如表 3.2 所列。

表 3.7 cat 命令常用选项

选项	说明
-n	从 1 开始对输出行进行编号
-b	类似于-n，从 1 开始编号，但是忽略空白行
-s	遇到连续两行或以上的空白行，就替换为一行空白行

使用示例 1，查看 hello.c 文件并编号：

```
vmuser@Linux-host: hello$ cat -n hello.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5
6 int main(int argc, char **argv)
7 {
....
```

使用示例 2，查看 hello.c 文件并忽略空白行编号：

```
vmuser@Linux-host: hello$ cat -b hello.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>

5 int main(int argc, char **argv)
6 {
....
```

如果使用重定向符 (>) 可以将屏幕输出保存到另一个文件中，或者追加符 (>>) 可以将屏幕输出添加到某个文件末尾。示例，将 hello.c 和 Makefile 文件增加行号后合并为 test 文件：

```
vmuser@Linux-host: hello$ cat -n hello.c Makefile > test
```

说明：

- (1) 重定向符 (>) 可以将标准输出重定向到其它输出或者文件，文件不存在则会创建新文件；
- (2) 追加符 (>>) 则将标准输出追加到文件末尾，如果文件不存在则创建新文件；



6. 文件压缩/解压

在日常开发过程中，不可避免的会用到压缩文件，如现在不少开源软件都是以压缩包方式提供，下载后必须解压才能使用；另一方面，也经常需要制作压缩文件，例如将工作资料打包进行备份。无论是压缩还是解压，都可以使用 tar 工具来实现。

tar 是 UNIX 系统的一个文件打包工具，只是连续首尾相连的将文件堆放起来，并不具备压缩功能，但是加上选项，tar 可以调用其它压缩/解压工具，能够实现文件的压缩和解压。用法：

```
$tar [选项] 文件
```

tar 工具常用选项如表 3.8 所列。

表 3.8 tar 常用选项

选项	说明
-c	创建存档文件，与-x 相斥
-t	列出档案文件的文件列表
-x	解包存档文件，与-c 相斥
-A	合并存档文件
-d	比较存档文件与源文件
-r	追加文件到存档文件末尾
-u	更新存档文件
-f	指定存档文件，与其它选项同时使用时，必须在最后，如 tar -xjvf a.tar.bz2
-v	显示详细处理信息
-C	转到指定目录，常用于解开存档文件
-j	调用 bzip2 程序
-z	调用 gzip 程序
-Z	调用 compress 程序
--exclude=PATH	排除指定文件/目录，常用于打包文件

使用示例：

(1) 解压 a.tar.bz2 文件，并显示详细信息：

```
vmuser@Linux-host: ~$ tar -xjvf a.tar.bz2
```

(2) 解压 b.tar.gz 文件，并指定解压到/home/chenxibing/目录：

```
vmuser@Linux-host: ~$ tar -xzvf b.tar.gz -C /home/chenxibing
```

紧跟 tar 命令选项的“-”可以不需要，但是-C 的“-”是必须的，例如上一条命令等价于：

```
vmuser@Linux-host: ~$ tar xzvf b.tar.gz -C /home/chenxibing
```

(3) 将 drivers 目录的文件打包，创建一个.tar.bz2 压缩文件：

```
vmuser@Linux-host: ~$ tar -cjvf drivers.tar.bz2 drivers
```

7. 删除文件

删除文件用 rm 命令，用法与删除目录相同。

8. 文件改名和移动

在日常操作中，经常会将文件从一个目录移动到另外一个目录，或者对文件进行改名。



在 Linux 下，文件移动和改名都是通过 mv 命令实现的，且移动和改名可以同时实现。用法：

```
$mv 源文件/目录 目的文件/目录
```

若目的路径与源路径不相同，则进行移动操作，如相同则进行改名操作。

文件改名和移动的用法比较简单，图 3.17 所示示例中，先将目录 other 改名为 newdir，然后再将 newdir 移动到上一级目录并改名为 hello2。

```
vmuser@Linux-host:~/test/apps
vmuser@Linux-host:~$ mkdir -p test/apps
vmuser@Linux-host:~$ cd test/apps
vmuser@Linux-host:~/test/apps$ mkdir hello other
vmuser@Linux-host:~/test/apps$ ls
hello other
vmuser@Linux-host:~/test/apps$ mv other/ newdir
vmuser@Linux-host:~/test/apps$ ls
hello newdir
vmuser@Linux-host:~/test/apps$ mv newdir/ ../hello2
vmuser@Linux-host:~/test/apps$ ls ..
apps hello2
vmuser@Linux-host:~/test/apps$
```

图 3.17 改名和移动

严格来说，Linux 下的文件名是由“路径+文件名”组成的，不同目录的两个同名文件实际上不是一个文件，如/home/lpc3250/apps/hello.c 与 /home/lpc3250/drivers/hello.c 是两个不同文件。所以，Linux 下文件的改名和移动实际上是一回事。

说明：讲删除命令的时候，提到删除的文件不会在回收站暂存，在通用桌面 Linux，一般都设有回收站，在桌面下删除一般会暂存在回收站，在命令行下若要想将某个文件暂存回收站，只能用 mv 命令，将文件移动到回收站中。Linux 下的回收站，一般在主目录下，为隐藏文件.Trash，不同发行版回收站的路径也各不相同。Ubuntu 的回收站目录是“~/.local/share/Trash”。

Ubuntu 图形界面下的删除，实际上都是 mv 指令，将“删除”的文件移动到回收站，清空垃圾桶才是用 rm 命令彻底删除。

9. 文件复制

在图形界面下复制文件，无非是选中某个文件，然后选择复制操作，再进入将要复制到的目的目录，再粘贴。在命令行下无需这么复杂，只需输入简单的命令，就可以完成各种不同的文件复制操作。cp 命令用法：

```
$cp [选项] 源文件/目录 目的文件/目录
```

cp 命令支持多种选项，可实现多种不同操作，常用的选项见表 3.9。

表 3.9 cp 命令常用选项

选项	说明
-a	保留链接、文件属性并递归复制，等同于-dpR 组合，常用于复制目录
-d	复制时保留链接
-f	若目标文件已经存在，则直接删除而不提示
-i	若目标文件已经存在，需要用户确认操作，与-f 相反
-p	除复制文件内容外，把访问权限和修改时间也复制到新文件中
-r	递归复制，递归复制指定目录下的文件和目录
-v	显示文件复制过程

通过 cp 命令，可以在同一目录下将文件/目录复制为另外一个文件/目录，也可将文件/



目录复制到其它目录，还可用其它文件名，图 3.18 所示的范例演示了这些操作。

```
vmuser@Linux-host:~$ cp -av /etc/newt/ .
"/etc/newt/" -> "./newt"
"/etc/newt/palette/ubuntu" -> "./newt/palette/ubuntu"
"/etc/newt/palette.original" -> "./newt/palette.original"
"/etc/newt/palette" -> "./newt/palette"
vmuser@Linux-host:~$ cp -av newt/ 2_newt
"newt/" -> "2_newt"
"newt/palette/ubuntu" -> "2_newt/palette/ubuntu"
"newt/palette.original" -> "2_newt/palette.original"
"newt/palette" -> "2_newt/palette"
vmuser@Linux-host:~$ cp -av newt/ /tmp/
"newt/" -> "/tmp/newt"
"newt/palette/ubuntu" -> "/tmp/newt/palette/ubuntu"
"newt/palette.original" -> "/tmp/newt/palette.original"
"newt/palette" -> "/tmp/newt/palette"
vmuser@Linux-host:~$
```

图 3.18 文件复制

10. 创建链接

链接文件在 Linux 系统中很常见，特别是库文件目录以及/etc 下与启动级别相关的目录。例如 “/etc/rc5.d/S99rc.local” 文件，实际上是链接到 “/etc/init.d/rc.local” 文件的一个软链接，如图 3.19 所示。

```
vmuser@Linux-host:~$ ls -n /etc/rc5.d/S99rc.local
lrwxrwxrwx 1 0 0 18 11月 13 2014 /etc/rc5.d/S99rc.local -> ../../init.d/rc.local
vmuser@Linux-host:~$
```

图 3.19 软链接文件

Linux 创建链接的命令为 ln，用法：

\$ln 选项源文件/目录目标文件

Linux 下的链接分软链接和硬链接两种，默认创建硬链接，选项加上-s 则创建软链接。

硬链接通过索引节点进行链接，相当于源文件的镜像，占用源文件一样大小的空间，修改其中一个，另外一个都会进行同样的改动。给一个文件创建硬链接后，文件属性的硬连接数会增加。

如图 3.20 所示的示例，hello.c 原有硬链接数是 1，创建硬链接 main.c 后，main.c 和 hello.c 文件大小一样，两者的硬连接数都增加为 2。

```
vmuser@Linux-host:~/hello
vmuser@Linux-host:hello$ ls
h hello.c libFOO.so
vmuser@Linux-host:hello$ ls -la hello.c
-rw-r--r-- 1 vmuser vmuser 104 2014-12-12 09:46 hello.c
vmuser@Linux-host:hello$ ln hello.c main.c
vmuser@Linux-host:hello$ ls -la hello.c main.c
-rw-r--r-- 2 vmuser vmuser 104 2014-12-12 09:46 hello.c
-rw-r--r-- 2 vmuser vmuser 104 2014-12-12 09:46 main.c
vmuser@Linux-host:hello$
```

图 3.20 创建硬链接

硬链接不能跨文件系统，只能在同一个文件系统内进行链接，且不能对目录文件建立硬



链接。给目录创建硬链接会出错，如图 3.21 所示。

```
vmuser@Linux-host:~$ ls
apps examples.desktop libhelloa test 模板 图片 下载 桌面
EasyARM hello libhelloso 公共的 视频 文档 音乐
vmuser@Linux-host:~$ ln EasyARM/ myboard
ln: "EasyARM/": 不允许将硬链接指向目录
vmuser@Linux-host:~$
```

图 3.21 创建目录硬链接错误

软链接和硬链接不同，软链接是产生一个新文件，这个文件指向另一个文件的位置，类似于 Windows 下的快捷方式。通常用的更多的是软链接，软链接可以跨文件系统，且可用于任何文件，包括目录文件。

假定为了使用方便，需要给 dir1 目录创建一个软链接 lpc，创建和结果如图 3.22 所示。

```
vmuser@Linux-host:~$ mkdir dir1
vmuser@Linux-host:~$ ln -s dir1 lpc
vmuser@Linux-host:~$ ls -l lpc
lrwxrwxrwx 1 vmuser vmuser 4 1月 4 10:38 lpc -> dir1
vmuser@Linux-host:~$
```

图 3.22 创建软链接

11. 改变文件和目录权限

Linux 系统是一个真正的多用户操作系统，系统的每个目录和文件对不同用户开放都有不同的权限。一个普通文件/bin/bash 的 ls -l 输出信息：

```
vmuser@Linux-host:~$ ls -l /bin/bash
-rw-r-xr-x 1 root root 917888 2010-08-11 04:47 /bin/bash
```

其中的 rwxr-xr-x 是权限信息，说明如图 3.23 所示。

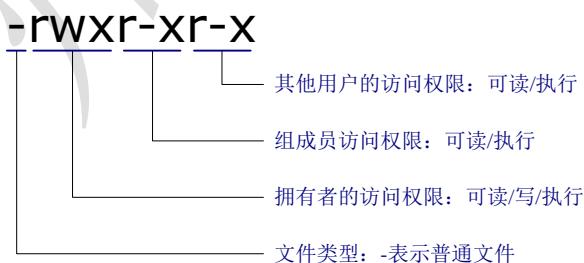


图 3.23 文件访问权限示例

输出信息第一列表示文件访问权限，该示例详细说明

第一个字符是-，表示这是一个普通文件，如果是 b 则表示是块设备，是 c 则表示是字符设备，是 d 则表示是目录，是 l 则表示是链接文件，p 表示命名管道，s 表示 Socket 文件。

接下来的 9 个字符 rwxr-xr-x，分成三组，各含义如表 3.10 所示。

表 3.10 文件权限说明



—		User (拥有者)			Group (群组成员)			Other (其它用户)		
权限		读	写	执行	读	写	执行	读	写	执行
字符		r	w	x	r	w	x	r	w	r
数字		4	2	1	4	2	1	4	2	1

权限字符中，ls -l 结果中，用 r/w/x 表示的则表明拥有相应的权限，用“-”表示的则表明没有相应的权限；拥有权限的用数字表示结果为“读/写/执行”3个数字相或得到，如 rwx 可用 7 表示，r-x 可用 5 表示，rwxr-xr-x 则可用 755 表示。

chmod 用于改变或者设置文件/目录的权限。用法：

```
$chmod [参数] 文件/目录
```

设置或者改变文件/目录的权限，可直接用数字表示，如将 hello 文件设置为任何人都可以读写并执行：

```
vmuser@Linux-host: hello$ chmod 777 hello
```

更常用的是用字符方式设定文件/目录的权限，分别用 u/g/o 表示文件的拥有者/组内用户/其它用户，用 rwx 分别表示读/写/执行权限，用 +/- 表示增加或去除某种权限。例如，将 hello 文件的其它用户权限可执行属性去掉：

```
vmuser@Linux-host: hello$ chmod o-x hello
```

如果同时设置 u/g/o，可用 a 表示，例如为 hello 增加全部用户可执行权限：

```
vmuser@Linux-host: ~$ chmod a+x hello
```

拥有可执行权限的文件，在 Linux 终端下通常呈现为绿色。如果在运行程序的时候遇到 permission denied 这样的错误提示，可在终端输入 chmod +x file，为将要运行的程序增加可执行权限。

类似的命令还有 chown 改变文件属主和 chgrp 改变文件群组，用法请参考其它资料。

3.2.4 网络操作命令

1. 网络配置

Linux 的网络功能很完善，在图形界面下有不少配置网卡的工具，在命令行下，也有不少用于配置网卡的工具和命令，用的最多的就是 ifconfig 命令，类似于 Windows 下的 ipconfig 命令，但是功能强大得多。

ifconfig 命令是 Linux 系统配置网卡的命令工具，可用于查看和更改网络接口的地址和参数，包括 IP 地址、广播地址、子网掩码和物理地址，也可激活和关闭网卡。用法：

```
$ifconfig 网络接口 [选项] 地址/参数
```

各参数等介绍如表 3.11 所列。

表 3.11 ifconfig 命令各选项参数

选项/参数	说明	示例
-a	查看系统拥有的全部网络接口	ifconfig -a
网络接口	指定操作某个网口	ifconfig eth0 192.168.1.136
broadcast	设置网口的广播地址	ifconfig eth0 broadcast 192.168.1.255
netmask	设置网口的子网掩码	ifconfig eth0 netmask 255.255.255.0

续上表



选项/参数	说明	示例
hw ether	设置网卡物理地址（如果驱动不支持则无效）	ifconfig eth0 hw ether 00:11:00:00:11:22
up	激活指定网卡	ifconfig eth0 up
down	关闭指定的网卡	ifconfig eth0 down

说明：

- (1) 使用 ifconfig 命令操作网口需要 root 权限；
- (2) 使用 ifconfig 修改网卡配置无需重启系统，也不能复位保存；
- (3) 可以同时配置网口的多个参数。

使用 ifconfig 同时配置网卡多个参数的范例：

```
vmuser@Linux-host: ~$ sudo ifconfig eth1 192.168.1.136 netmask 255.255.255.0 broadcast 192.168.1.255 up
```

2. ping 命令

有时候可能会遇到网络不通的故障，此时首先应该做的就是检查网络是否通畅。在 Linux 命令行下，可用 ping 命令来检查。用法：

\$ping IP 地址

如果没有进行特殊的路由设置，通常情况下只能 ping 同网段的主机，不能跨网段 ping 操作。

进行 ping 操作，如果能收到目标 IP 的返回信息，则表示网络通畅，例如：

```
vmuser@Linux-host: ~$ ping 192.168.1.100
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
64 bytes from 192.168.1.100: icmp_seq=1 ttl=128 time=0.206 ms
64 bytes from 192.168.1.100: icmp_seq=2 ttl=128 time=0.179 ms
```

3.2.5 安装和卸载文件系统

1. 文件系统挂载

Linux 允许多个文件系统存在于同一个系统中，也允许用户在系统运行中安装内核所支持的文件系统。例如，将一个 FAT 格式的 U 盘插入到 Linux 系统中。

往系统安装文件系统需要用到 mount 命令，并且需要 root 权限。用法：

mount [-参数] [设备名称] [挂载点]

mount 命令支持的参数较多，常用参数如表 3.12 所列。

表 3.12 mount 常用参数

参数	说明	
-a	挂载/etc/fstab 文件中列出的所有文件系统	
-r	以可读方式挂载	
-w	以可写方式挂载（默认）	
-v	显示详细安装信息	
-t <文件系统类型>	指定文件系统类型，常见的文件系统类型有：	
	ext/ext2/ext3/ext4	Linux 常用文件系统
	msdos	MS-DOS 的 FAT，即 FAT16
	vfat	Windows 系统的 FAT，FAT32



续上表

	nfs	网络文件系统
	ntfs	Windows2000/NT/XP 的 ntfs 文件系统
	auto	自动检测文件系统
-o <选项>	指定挂载时的一些选项，常用的有：	
	defaults	使用默认值（auto、nouser、rw、suid）
—	suid/nosuid	确认/不确认 uid 和 sgid 位
	user/nouser	允许/不允许一般用户挂载
	codepage=XXX	指定 codepage
	iocharset=XXX	指定字符集
	ro	以只读方式挂载
	rw	以读写方式挂载
	remount	重新安装已经安装了的文件系统
	loop	挂载 loopback 设备以及 ISO 文件

说明：

- (1) 挂载点必须是一个已经存在的目录；
- (2) 如果挂载点非空，则挂载后会导致挂载点之前的内容不可用，卸载后方可用；
- (3) 一个挂载点可被多个设备/文件重复挂载，只是后一次挂载将覆盖前一次内容，卸载后可用；
- (4) 使用多个-o 参数的时候，-o 只用一次，参数之间用半角逗号隔开。

假如需要在 Linux 系统中使用 FAT 的 U 盘，则需要进行挂载，实现文件系统安装：

```
#mount -t vfat /dev/sda1 /mnt
```

在进行嵌入式 Linux 开发过程中，mount 命令经常被使用，特别是进行 NFS 连接和调试的时候，通过 NFS 挂载，将远程主机 Linux 的某个共享目录挂载到嵌入式系统本地，当成本地设备进行操作。NFS 挂载范例：

```
root@EasyARM-iMX28x ~# mount -t nfs 192.168.1.138:/home/chenxibing/lpc3250 /mnt -o nolock
```

nolock 表示禁用文件锁，当连接到一个旧版本的 NFS 服务器时常加该选项。

此外，嵌入式开发中常用的文件系统还有 cramfs、jffs2、yaffs/yaffs2 以及 ubifs 等，特别是用于 NOR Flash 的 jffs2 和用于 NAND Flash 的 yaffs/yaffs2、ubifs 等，在进行系统操作中通常需要对各设备进行挂载或者卸载，需要在挂载的时候指定正确的文件系统类型。挂载 yaffs2 分区的命令示例：

```
#mount -t yaffs2 /dev/mtdblock2 /mnt
```

使用条件：需要内核已经支持 yaffs2 文件系统。

挂载 ubifs 分区的命令示例：

```
#mount -t ubifs ubi0:rootfs /mnt
```

使用条件：需要内核已经支持 ubifs 文件系统。

2. 文件系统卸载

当不再需要某个文件系统的时候，可以将其卸载。umount 用于卸载已经挂载的设备或者文件。用法：

```
#umount 挂载点
```



如果已经将 U 盘挂载到/mnt 目录下，用完后的卸载命令为：

```
root@EasyARM-iMX28x ~# umount /mnt
```

3.2.6 使用内核模块和驱动

1. 加载（插入）模块

Linux 是一个具有模块化特性操作系统，允许在内核运行中插入模块或者卸载不再需要的模块。能够动态加载和卸载模块是 Linux 引以为豪的特性之一，如果某些功能平时用不到，可以不用编进内核，而采取模块方式编译，在需要的时候再插入内核，不再需要的时候卸载，这样可以精简内核，提高效率，并提高系统的灵活性。Linux 中最常见的模块是内核驱动，掌握模块的加载和卸载，也是使用 Linux 必须掌握的知识点之一。

通过 insmod 命令可以往正在运行中的内核插入某些模块而无需重启系统。用法：

```
# insmod [选项]模块 [符号名称=值]
```

insmod 常用选项介绍如表 3.13 所列。

表 3.13 insmod 命令常用选项

选项	说明
-f	强制将模块载入，不检查目前 kernel 版本与模块编译时的 kernel 版本是否一致
-k	将模块设置为自动卸载
-p	测试模块是否能正确插入
-x	不导出模块符号
-X	导出模块所有外部符号（默认）
-v	显示执行过程

一般情况下，如果一个模块的版本与所运行的内核不一致，模块将无法插入系统。就算是同一版本内核编译得到，如果内核配置文件不同，也有可能无法插入。使用-f 选项强制插入后，可能会出现运行不正确的情况。

插入和卸载模块需要 root 权限。插入模块比较简单，如需要在开发套件的系统中插入 lradc.ko 驱动模块，可用：

```
root@EasyARM-iMX28x ~# insmod lradc.ko
```

注：开发套件的文件系统中默认不集成各种预编译的应用程序与驱动模块。本文示例的所有驱动均提供源码（可在“3、Linux\4、开发示例\6、驱动示例”目录下找到），读者需要自行编译后拷贝到开发套件中使用。

有些驱动/模块可以接受外部参数，在插入模块的时候为相应的符号赋值。一个模块/驱动能否接受外部参数，能够接受几个外部参数，取决于模块/驱动的具体实现，符号以及赋值请参考相应模块/驱动的说明。例如，pcm-8032a.ko 模块（并不实际存在）能接收 irq 和 addr 两个符号参数，插入模块的时候指定：

```
vmuser@Linux-host: ~$ sudo insmod pcm-8032a.ko irq=3 addr=0x300
```

2. 查看系统已经加载的模块

如果想要知道某个模块是否已经插入系统，或者想知道系统已经加载了哪些模块，可用 lsmod 命令查看。lsmod 命令用法：

```
vmuser@Linux-host: ~$ lsmod
```

lsmod 命令结果实际上就是列出/proc/modules 的内容，结果如图 3.24 所示。



```
vmuser@Linux-host:~$ lsmod
Module                  Size  Used by
bnep                   18258  2
rfcomm                 47864  0
bluetooth              247024  10 bnep,rfcomm
snd_ens1371             25783  2
gameport                19699  1 snd_ens1371
snd_ac97_codec           134918  1 snd_ens1371
ac97_bus                 12766  1 snd_ac97_codec
snd_pcm                  102477  2 snd_ens1371,snd_ac97_codec
snd_seq_midi              13324  0
snd_rawmidi               30417  2 snd_ens1371,snd_seq_midi
snd_seq_midi_event         14899  1 snd_seq_midi
snd_seq                   61930  2 snd_seq_midi,snd_seq_midi_event
snd_timer                  29989  2 snd_pcm,snd_seq
snd_seq_device              14497  3 snd_seq_midi,snd_rawmidi,snd_seq
coretemp                  13596  0
snd                      69533  11 snd_ens1371,snd_ac97_codec,snd_pcm,snd_rawm
idi,snd_seq,snd_timer,snd_seq_device
ppdev                     17113  0
psmouse                  97873  0
```

图 3.24 查看内核模块

3. 卸载驱动模块

当某个内核模块或者驱动不再需要被使用，则可以将其从系统中卸载，以释放所占用的资源。卸载模块用 `rmmmod` 命令，用法：

```
# rmmmod [选项]模块
```

`rmmmod` 命令常用可选项如表 3.14 所列。

表 3.14 `rmmmod` 常用可选项

选项	说明
-f	强制卸载正在被使用的模块，非常危险！需要内核支持（ <code>CONFIG_MODULE_FORCE_UNLOAD</code> 使能），否则无效
-w	通常情况下不能卸载正在被使用的模块，加上-w 选项，指定模块将会被孤立，直到不再被使用
-s	将错误信息写入 <code>syslog</code> ，而不是标准错误
-v	显示执行过程

如果一个模块正在被另外一个模块所依赖，或者正在被某个应用程序使用，则一般情况下无法卸载这个模块。如果内核支持强制卸载模块功能，加上-f 可以卸载，但是不要轻易使用，否则有可能会带来严重错误。假定系统的 `lradc.ko` 不再需要，卸载命令：

```
root@EasyARM-iMX28x ~# rmmmod lradc.ko
```

4. 自动处理可加载模块

前面提到的 `insmod/rmmmod` 分别用于加载和卸载模块，但是每次只能加载/卸载一个模块，如果一个模块依赖于多个模块，则需要进行多次操作，比较繁琐。`modprobe` 命令集加载/卸载功能于一身，并且可以自动解决模块的依赖关系，将某模块所依赖的其它模块全部加载。用法：

```
# modprobe [选项] 模块 [符号=值]
```

`modprobe` 也支持很多选项，常用选项如表 3.15 所列。



表 3.15 modprobe 常用选项

选项	说明
-C <文件>	不适用默认配置文件，使用指定文件作为配置文件
-i	忽略配置文件中的加载和卸载命令
-r	卸载指定模块，包括依赖模块
-f	强制安装
-l	显示所有匹配模块
-a	安装所有匹配的模块
--show-dependencies	显示模块的依赖关系
-v	显示执行过程
-q	不显示任何信息
-V	显示版本信息

modprobe 处理模块时忽略模块的路径，这要求系统模块和驱动是按照 make modules_install 方式安装的，即模块必须放在/lib/modules/\$(uname -r) 目录下，并且有正确的 /lib/modules/\$(uname -r)/modules.dep 文件，modprobe 根据该文件来寻找和解决依赖关系。

5. 创建设备节点

如果系统不能自动创建设备节点，加载驱动后，则需要为驱动建立对应的设备节点，否则无法通过驱动来操作设备。只有 root 用户才能创建设备节点，命令为 mknod，用法：

```
# mknod 设备名 设备类型 主设备号 次设备号
```

如需要创建一个字符设备 led，主设备号为 231，次设备号为 0，则命令如下：

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

3.2.7 重启和关机

重启系统用 reboot 命令，关机用 poweroff 命令，两者都需要 root 权限。

3.2.8 其它命令

1. 临时获取 root 权限

在普通用户权限下，Linux 下很多命令都是不能使用的，一般在/sbin 和/usr/sbin 目录下的命令，执行都需要 root 权限。sudo 命令则可以临时获取 root 权限，需要输入密码。用法：

```
$ sudo 命令
```

例如，当前登录用户是普通用户，想挂载一个新的文件系统，则可以这样操作：

```
$ sudo /sbin/mount [参数]
```

根据发行版的不同，普通用户无法搜索 root 用户的搜索目录，所以最好指定命令所在的绝对路径。

另外，通过普通命令还可操作只有 root 才能操作的文件。假定文件 root.ini 只有 root 用户才能修改，现在普通用户想修改，可以这样操作：

```
vmuser@Linux-host: ~$ sudo vim root.ini
```

说明：

(1) sudo 只能临时获取 root 权限，通常一段时间之内（如 5 分钟之内）再次使用 sudo，无需输入密码，超过这段时间则需再次输入密码。

(2) 使用 sudo 命令需要管理员将用户添加到 sudoer 组中，一般在/etc/sudoers 文件中修改。



另外 su 命令则是切换到 root 用户，只要不用 exit 退出，将都处在 root 权限下，这样比较危险，一般不推荐使用 su 命令。

2. 文件同步

经常会遇到这样的情形：刚刚修改了某个文件，突然断电，重启后发现刚刚做的修改并没有保存，或者被修改的文件已经损坏。这是由于在 Linux 中，对文件的操作都是先写入缓存，并没有立即写入磁盘，经系统调度后方可写入磁盘。如果修改了缓存，还没来得及写到磁盘就断电，自然会造成文件改变丢失。

要避免这种情况发生，就是修改文件后，立即强制进行一次文件同步操作，将缓存的内容写入磁盘，确保文件系统的完整性。能完成这样功能的命令是 sync。只需在关闭文本编辑器后在 Shell 输入 sync 即可。

3. 文件搜索

在嵌入式 Linux 开发过程中，对于大型工程，如 Linux 内核或者 U-Boot 移植中，经常会遇到记不清某个文件的位置的情形。如果用 cd 和 ls 命令在各个目录盲目查找，费时费力，效率低下，用 find 文件查找命令可以快速解决这样的问题。

find 是 Linux 下很常用的查找命令，功能很强大，用法也很复杂，这里仅仅介绍常用的简单用法。find 命令基本语法：

```
$ find 路径 -选项其它
```

最常用的是根据文件名来查找，加上-name 参数就可以了，还可以支持通配符，进行模糊搜索。例如只大概知道内核源码“arch/arm”目录下有文件名以 mux 开头的文件，但不知道确切文件名，可用下列命令搜索：

```
vmuser@Linux-host: ~$ find arch/arm/ -name mux*.c
```

命令和实际操作结果如图 3.25 所示。

```
chenxibing@linux-compiler: ~/work/am3352/m3352/linux-3.2.0-psp04.06.00.11
chenxibing@linux-compiler: linux-3.2.0-psp04.06.00.11$ find arch/arm/ -name mux*.c
arch/arm/mach-omap2/mux.c
arch/arm/mach-omap2/mux34xx.c
arch/arm/mach-omap2/mux2430.c
arch/arm/mach-omap2/mux33xx.c
arch/arm/mach-omap2/mux44xx.c
arch/arm/mach-omap2/mux2420.c
arch/arm/mach-davinci/mux.c
arch/arm/mach-omap1/mux.c
arch/arm/plat-omap/mux.c
chenxibing@linux-compiler: linux-3.2.0-psp04.06.00.11$
```

图 3.25 find 命令示例

可以看到，“arch/arm”目录下全部以 mux 开头的文件都被列了出来，在其中选择需要的文件进行查看和操作即可。

4. 字符串搜索

在嵌入式 Linux 开发中，还经常有另外一种查找情形。知道某个变量、函数名，但不知道在什么地方定义，或者知道某个关键字，想知道在哪些文件内用到了，这些都可以用 grep 命令完成。

grep 是 Linux 系统中一个强大的文本搜索工具，用法很多很复杂，这里也仅仅介绍简单的常用用法。语法格式：

```
$ grep 选项
```



只要提供查找的关键字，用 grep 命令就可以完成查找。例如，想知道 pcf8563 这个关键字在“arch/arm”目录下哪些地方用到了，可以输入下列命令：

```
vmuser@Linux-host: ~$ grep "pcf8563" -R arch/arm
```

关键字最好加上双引号，特别是包含空格的关键字，对于单个关键字倒是可以不用引号。“-R”表示进行递归查找，而不是仅仅在指定的目录下查找。

命令和实际操作结果如图 3.26 所示，可以看到，凡是用了“pcf8563”的地方都被列了出来，并且红色高亮显示。

```
chenxibing@linux-compiler: ~/work/am3352/m3352/linux-3.2.0-psp04.06.00.11$ grep "pcf8563" -R arch/arm/
Binary file arch/arm/boot/Image matches
arch/arm/mach-ixp4xx/nas100d-setup.c:           I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-ixp4xx/dsmg600-setup.c:           I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-pcm037.c:                 I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-pca100.c:                 I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-pcm038.c:                 I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-cpuimx35.c:               I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-pcm043.c:                 I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-cpuimx27.c:               I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-imx/mach-eukrea_cpuimx25.c:       I2C_BOARD_INFO("pcf8563", 0
x51),
Binary file arch/arm/mach-omap2/board-m3352.o matches
arch/arm/mach-omap2/board-m3352.c:                I2C_BOARD_INFO("pcf8563", 0x51),
Binary file arch/arm/mach-omap2/built-in.o matches
arch/arm/mach-lpc32xx/phy3250.c:                  I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-mx5/board-cpuimx51sd.c:             I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-mx5/board-cpuimx51.c:                I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-ks8695/board-ac5k.c:                 I2C_BOARD_INFO("pcf8563", 0x51),
arch/arm/mach-orion5x/mv2120-setup.c:              I2C_BOARD_INFO("pcf8563", 0x51),
chenxibing@linux-compiler: linux-3.2.0-psp04.06.00.11$
```

图 3.26 grep 查找关键字

3.3 Shell 文件

Shell 文件是以某种方式将一些命令放在一起得到的文件，常称为 Shell 脚本。Shell 文件通常以“#!/bin/sh”开始，#!后面指定解释器，如下是一个简单的 Shell 文件的内容：

```
#!/bin/sh
echo "hello, I am shell script"
```

假定此文件名为 a.sh，增加可执行权限后，在 Shell 中即可运行，将在终端打印“hello, I am shell script”字符串。

```
vmuser@Linux-host: ~$ chmod +x a.sh
vmuser@Linux-host: ~$ ./a.sh
hello, I am shell script
```

执行 Shell 脚本有多种方式：

点+斜线+文件名，这种方式要求文件必须有可执行权限；

点+空格+文件名，这种方式不要求文件一定具有可执行权限。

sh+空格+文件名，这种方式不要求文件一定具有可执行权限。

source+空格+文件名，这种方式不要求文件一定具有可执行权限。



3.4 Linux 环境变量

3.4.1 环境变量

Linux 是一个多用户操作系统，每个用户都有自己专有的运行环境。用户所使用的环境由一系列变量所定义，这些变量被称为环境变量。系统环境变量通常都是大写。

每个用户都可以根据需要修改自己的环境变量，以达到自己的使用要求。常见的环境变量如表 3.16 所列。

表 3.16 Linux 常见环境变量

变量	说明
PATH	决定了 Shell 将到哪些目录中寻找命令或程序，这个变量是在日常使用中经常需要修改的变量
TERM	指定系统终端
SHELL	当前用户 Shell 类型
HOME	当前用户主目录
LOGNAME	当前用户的登录名
USER	当前用户名
HISTSIZE	历史命令记录数
HOSTNAME	指主机的名称
LANGUAGE	语言相关的环境变量，多语言可以修改此环境变量
MAIL	当前用户的邮件存放目录
PS1	基本提示符，对于 root 用户是#，对于普通用户是\$
PS2	附属提示符，默认是“>”
LS_COLORS	ls 命令结果颜色显示

在 Shell 下通过美元符号 (\$) 来引用环境变量，使用 echo 命令可以查看某个具体环境变量的值。例如，查看 TERM 的值：

```
vmuser@Linux-host: ~$ echo $TERM
```

使用 env 或者 printenv 命令可以查看系统全部的环境变量设置，例如：

```
chenxibing@gitserver-zhiyuan:~$ env
TERM=xterm
SHELL=/bin/bash
USER=chenxibing
MAIL=/var/mail/chenxibing
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/chenxibing
LANG=zh_CN.UTF-8
SHLVL=1
HOME=/home/chenxibing
LANGUAGE=zh_CN:zh
LOGNAME=chenxibing
```

3.4.2 修改环境变量

登录用户可以根据需要修改和设置环境变量。Linux 下修改环境变量既可以在终端通过



Shell 命令修改，也可以通过修改系统的配置文件来进行。

(1) 通过 Shell 命令设置环境变量，常用于临时设置环境变量，一旦关闭当前终端或者新开一个终端，所设置的环境变量都将丢失。可直接用等号 (=) 为变量赋值，或者用 export 命令为变量赋值，用法：

```
$ 变量=$变量:新增加变量值  
$ export 变量=$变量:新增加变量值
```

新增加的变量值既可以放在变量原有值的末尾 (\$变量:新变量值)，也可以放在原有变量值的开头 (新变量值:\$变量)。

例如，需要为系统 PATH 变量增加 /opt/usr/bin 目录，操作示例：

```
vmuser@Linux-host: ~$ echo $PATH  
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/local/ruby/bin:/opt/mysql5/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin  
vmuser@Linux-host: ~$ export PATH=$PATH:/opt/usr/bin  
vmuser@Linux-host: ~$ echo $PATH  
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/local/ruby/bin:/opt/mysql5/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/usr/bin
```

(2) 修改系统配置文件。修改系统配置文件，可以达到永久改变环境变量的目的。修改某个配置文件后，在 Shell 下运行该文件即可使新的设置生效，或者重新登录使用新的变量。运行文件可用“source 文件”的方式操作。通常修改/etc/profile 文件或者~/.bashrc（有的发行版上为~/.bash_profile）文件：

- 修改/etc/profile 文件会影响使用本机的全部用户；
- 修改~/.bashrc 则仅仅影响当前用户；
- 推荐修改~/.bashrc 文件。



第4章 Linux 文件系统

本章主要讲述 Linux 文件系统结构和相关概念。理解这一章的内容，能够更加深刻地理解和掌握 Linux。因此尽管都是一些概念性的介绍而没有实操练习，但还是请务必仔细阅读。

4.1 Linux 目录结构

Linux 文件系统对文件的管理包括两方面，一方面是文件本身，另一方面是目录管理。先从目录入手，会显得比较直观和更加容易理解一些。

4.1.1 Linux 目录树

Linux 整个文件系统以根目录 (/) 为最顶层目录，下面包含众多和多级其它目录，形成了一个拓扑结构，整个目录结构看起来就像一棵倒挂着的树，称之为“Linux 目录树”，如图 4.1 所示。整个 Linux 有且只有这样一棵树。

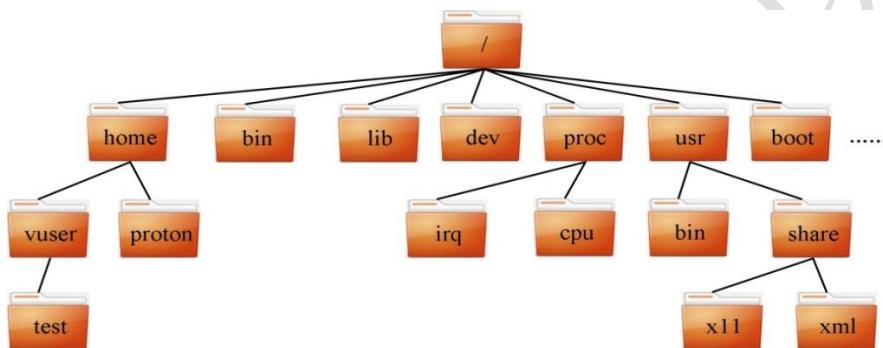


图 4.1 Linux 目录树

打开 Ubuntu 的文件浏览器，切换到根目录，实际呈现给我们的内容如图 4.2 所示。

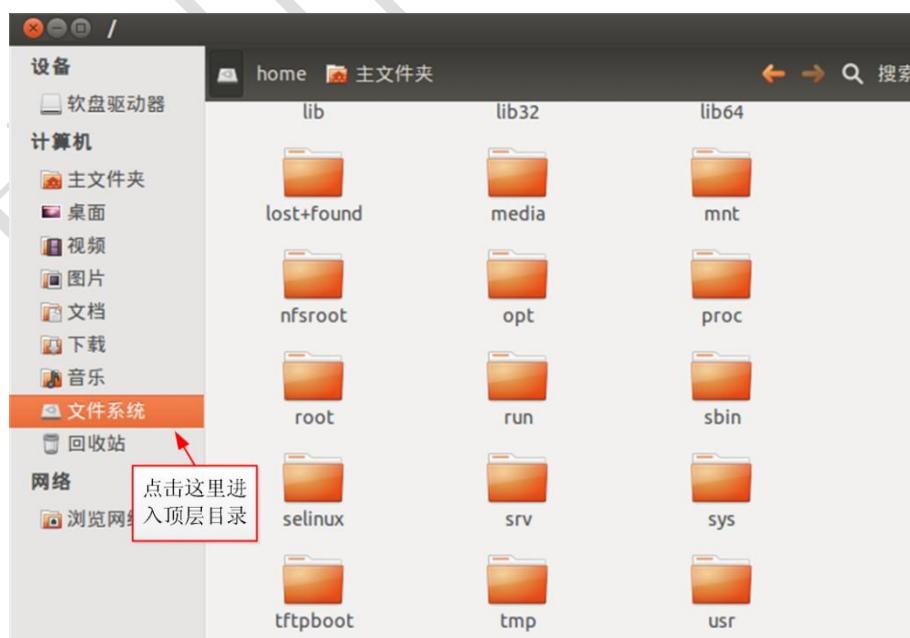


图 4.2 文件浏览器看到的根目录

这个目录树实际上是一个虚拟的概念，并不与任何文件、任何介质绑定，也没有容量，



甚至连读写规则都没有。只有将某个介质如磁盘或者光驱挂载（mount）到这棵树的某个目录后，这个目录下面才有文件。但是，此时这个目录依旧没有容量的概念，看到的容量仅仅是这个磁盘或者光驱这个设备的容量属性，并不是文件系统的属性。

由于这棵树是虚拟的，没有任何限制，所以很容易进行扩展。

4.1.2 Linux 目录树标准

理论上，Linux 目录树的目录结构是可以随意安排的，事实上很多 Linux 系统开发人员也这么做，但这就带来了不同开发人员之间不统一的情况存在，很容易出现混乱。后来这样的问题得到了重视，文件层次标准（FHS，Filesystem Hierarchy Standard）就在这种情况下出台的。FHS 对 Linux 根文件系统的基本目录结构做了比较详细的规定，尽管不是强制标准，但事实上，大部分 Linux 发行版都遵循这个标准。

Linux 目录树下各子目录的简单说明如表 4.1 所列。

表 4.1 FHS 顶层和/usr 目录

目录	说明	
bin	基本命令的程序文件，里面不能再包含目录	
boot	Boot Loader 静态文件	
dev	设备文件	
etc	系统配置文件，配置文件必须是静态文件，不能是二进制文件	
home	存放各用户的个人数据	
lib	基本的共享库和内核模块	
media	可移动介质的挂载点	
mnt	临时的文件系统挂载点	
opt	附加的应用程序软件包	
root	root 用户目录	
sbin	基本的系统命令二进制文件	
srv	系统服务的一些数据	
tmp	临时文件	
usr	/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/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
	/usr/local/bin	本地增加的命令
	/usr/local/lib	本地增加的库
var	可变数据	

4.2 Linux 的文件

4.2.1 Linux 文件结构

文件是数据的一种组织形式，是具有文件名的一组相关信息的集合，是文件系统中存储数据的一个命名的对象。Linux 下一切都是文件，无论程序、文档、数据库这样的普通文件，



还是链接文件和目录这样的特殊文件，甚至于连硬件设备都用文件来描述。Linux 下所有文件的描述结构都是相同的，包含索引节点和数据，如图 4.3 所示。

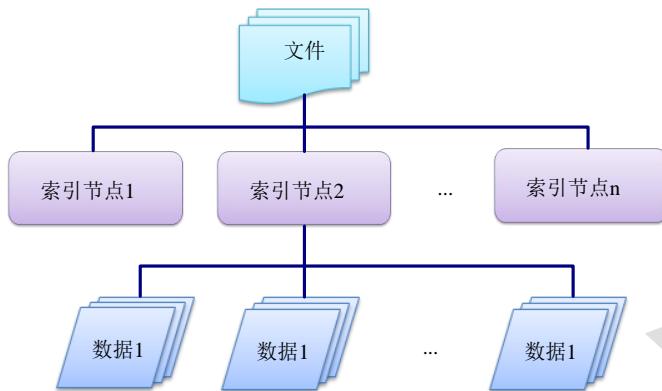


图 4.3 文件、记录和数据项的关系

索引节点：又称 I 节点，是 Linux 文件系统用来记录文件信息的一种数据结构，信息包括文件名、文件长度、文件权限、存放位置、所属关系、创建和修改时间等。文件系统维护了一个索引节点的数组，每个文件都与索引节点数组中的唯一元素对应，索引节点在数组中的索引号，称为索引节点号。每个文件都有一个索引号与之对应，而一个索引节点号，可以对应多个文件。

数据：文件的实际内容。可以是空的，也可以非常大，并且拥有自己的结构。

一个文件的索引节点、文件大小、属主等信息，在 Shell 下可用 ls 命令加上参数 -i 查看，例如：

```
vmuser@Linux-host: ~$ ls -li examples.desktop
785326 -rw-r--r-- 1 vmuser root 179 2010-10-15 09:07 examples.desktop
```

各信息说明如表 4.2 所列。

表 4.2 examples.desktop 文件详细信息说明

信息	说明	
examples.desktop	文件名	
785326	索引节点号	
- (第 1 个)	-表示是普通文件，可能出现的有：	
	-	普通文件
	c	字符设备
	b	块设备
	d	目录
	l	链接文件
	s	Socket 文件
rw-r--r--	文件访问权限：644	
1	1 表示文件没有硬链接；如果文件有 1 个硬链接则为 2；对于目录，则表示该目录包含的目录文件数（包含隐藏的(.)和(..)）	
chenxibing	文件拥有者	
root	文件所在群组	
179	文件大小	



续上表

信息	说明
2010-10-15 09:07	文件修改时间

文件数据信息则需要用相关的编辑器或者软件才能查看。

4.2.2 Linux 文件名称

Linux 的文件名保存在目录文件中，命名应当遵循以下规则：

区分大小写；

不能以“+”和“-”开头；

不能包含<>、|`“‘ \$! % &*?\()[]等在 Shell 中有特殊含义的字符；

不能包含空格；

长度不能超过 255 个字符。

Linux 系统中，文件名以点号(.)开始的文件是隐藏文件，用 ls 命令不加参数-a 将看不到这类文件。“同名”的隐藏文件与非隐藏文件是不同的，如.file 与 file 是两个不同的文件。

4.2.3 文件类型

Linux 系统中的文件可以分为如下几类：普通文件、目录文件、设备文件和符号链接文件。

1. 普通文件

普通文件又称常规文件，包含各种长度的字符串。常见的 C 程序文件、脚本文件、数据库文件等都是文件。普通文件在 Linux Shell 下用 ls 命令查看，得到的信息中，第一个字符是“-”。如：

```
-rw----- 1 chenxibing member 10732 2011-01-08 13:28 .bash_history
```

2. 目录文件

目录文件是一种特殊文件，利用它可以构成文件系统的分层树形结构。在 Linux Shell 下用 ls 命令查看，第一个字符用 d 表示。如：

```
drwxr-xr-x 3 chenxibing member 4096 2010-11-27 14:02 abing  
drwxr-xr-x 4 chenxibing member 4096 2010-10-19 14:00 git
```

3. 设备文件

设备是一种特别文件，除了存放在文件 i 节点中的信息外，它们不包含任何数据。有效地设备文件与相应的设备对应，通过设备文件，可以操作与之对应的硬件设备。

设备文件包括字符设备和块设备文件。字符设备文件按照字符操作设备，如键盘、终端等；块设备文件以块为单位操作设备，如磁盘、光盘等。Linux 系统的设备文件都放在/dev 目录下，用 ls -la 命令可以查看各设备的属性：

```
crw-rw-rw- 1 root root 1, 8 2011-01-08 15:12 random  
crw-r--r-- 1 root root 10, 135 2011-01-08 15:12 rtc  
brw-rw---- 1 root disk 8, 0 2011-01-08 15:12 sda  
brw-rw---- 1 root disk 8, 1 2011-01-08 07:12 sda1
```

输出信息中，以第一个字符用 b 表示的是块设备，用 c 表示的是字符设备。

4. 符号链接文件

链接文件是一种特殊文件，提供对其它文件的参照，它的数据是它所链接的文件的路径



名。用 ln 命令可以创建一个文件的软硬链接或者一个目录的软连接。链接文件常用于不同目录下的文件共享。链接文件在 ls 命令下输出结果中第一个字符为字母 l，并显示为“文件->目标”的方式，如链接文件 git 链接到/var/server/repo-git/chenxibing，查看结果为：

```
lrwxrwxrwx 1 chenxibing root          31 2010-10-22 08:37 git -> /var/server/repo-git/chenxibing
```

4.3 Linux 文件系统

Linux 最初是基于 X86 设计的，保存文件的物理设备是磁盘或者磁带。Linux 最初用于管理磁盘文件的文件系统是基于 Minix 的，存在文件管理效率不高的问题；后来在 Minix 的基础上进行了扩展，设计了专门用于 Linux 的 Ext 扩展文件系统（Extended file system），并添加到内核中，并成为了 Linux 事实上的标准文件系统，Linux 的发布和安装都基于 Ext 文件系统。

Ext 扩展文件系统经过发展，历经第二代扩展文件系统（Ext2，the Second Extended file system）、第三代扩展文件系统（Ext3，the Third Extended file system）和第四代扩展文件系统（Ext4，the fourth Extended file system）目前最新和流行最广的是 Ext4。

Ext2 属于非日志型文件系统，而 Ext3/4 文件系统是日志型文件系统。日志型文件系统用独立的日志文件跟踪磁盘内容的变化，比传统文件系统安全。

4.3.1 Ext3 文件系统特点

Ext3 从 Ext2 发展而来，并且完全兼容 Ext2 文件系统，且比 Ext2 可靠。在文件大小、数量和文件名方面有如下限制：

- 最大文件大小：2TB；
- 最大文件数量：可变；
- 最长文件名限制：255 字节；
- 最大卷大小：16TB；
- 文件名允许的字符数：除 NUL 和 “/” 外的所有字符。

整体上，Ext3 具有下面的一些特点：

1. 高可用性

使用了 Ext3 文件系统后，即使在非正常关机后，系统也不需要检查文件系统。即使发生了宕机，也只需要数十秒钟即可恢复 Ext3 文件系统。

2. 数据的完整性

Ext3 文件系统能够极大地提高文件系统的完整性，避免意外宕机对文件系统的破坏。Ext3 文件系统为用户提供了 2 种模式来保证数据的完整性。其中一种是“同时保持文件系统及数据的一致性”模式。如果采用这种方式，用户永远不会看到由于非正常关机而存储在磁盘上的垃圾文件。

3. 文件系统的速度

尽管使用 Ext3 文件系统时，可能在存储前需要多次写数据，但是从总体上看来，Ext3 比 Ext2 的性能要好一些。因为 Ext3 的日志功能对磁盘的驱动器读写头进行了优化，所以文件系统整体读写性能并没有降低。

4. 数据转换

Ext3 兼容 Ext2，从 Ext2 文件系统转换成 Ext3 文件系统非常容易，只需要简单地键入两条命令即可完成整个转换过程，用户不用花时间备份、恢复、格式化分区等。并且 Ext3 文件系统可以不经任何更改，而直接加载成为 Ext2 文件系统。



5. 多种日志模式

Ext3 有多种日志模式，系统管理人员可以根据系统的实际工作要求，在系统的工作速度与文件数据的一致性之间作出选择：

- 一种工作模式是对所有的文件数据及 metadata（定义文件系统中数据的数据，即数据的数据）进行日志记录（data=journal 模式），这种模式数据一致性好；
- 另一种工作模式则是只对 metadata 记录日志，而不对数据进行日志记录，就是所谓的 data=ordered 或者 data=writeback 模式，这种模式工作速度快。

4.3.2 Ext4 文件系统特点

Ext4 在 Ext3 的基础上进行了改进，修改了一部分重要的数据结构。Ext4 在性能和可靠性方面都有更好的表现，功能方面也更加丰富。

Ext4 兼容 Ext3，从 Ext3 迁移到 Ext4，无需格式化磁盘或者重装系统。

与 Ext3 相比，Ext4 具有下列特点：

1. 支持更大的文件系统和文件

Ext3 支持最大的文件系统是 16TB，Ext4 支持到了 1EB（1,048,576TB，1EB=1024PB，1PB=1024TB）；Ext3 支持的最大文件是 2TB，而 Ext4 支持到了 16TB。

2. 无限数量的子目录

Ext3 只支持 32,000 个子目录，而 Ext4 支持无限数量的子目录。

3. Extents

Ext3 采用间接块映射，当操作大文件时，效率极其低下。比如一个 100MB 大小的文件，在 Ext3 中要建立 25,600 个数据块（每个数据块大小为 4KB）的映射表。而 Ext4 引入了现代文件系统中流行的 extents 概念，每个 extent 为一组连续的数据块，上述文件则表示为“该文件数据保存在接下来的 25,600 个数据块中”，提高了不少效率。

4. 多块分配

当写入数据到 Ext3 文件系统中时，Ext3 的数据块分配器每次只能分配一个 4KB 的块，写一个 100MB 文件就要调用 25,600 次数据块分配器，而 Ext4 的多块分配器“multiblock allocator”（mballoc）支持一次调用分配多个数据块。

5. 延迟分配

Ext3 的数据块分配策略是尽快分配，而 Ext4 和其它现代文件操作系统的策略是尽可能地延迟分配，直到文件在 cache 中写完才开始分配数据块并写入磁盘，这样就能优化整个文件的数据块分配，与前两种特性搭配起来可以显著提升性能。

6. 快速 fsck

以前执行 fsck 第一步就会很慢，因为它要检查所有的 inode，而现在 Ext4 给每个组的 inode 表中都添加了一份未使用 inode 的列表，今后 fsck Ext4 文件系统就可以跳过它们而只去检查那些在用的 node 了。

7. 日志校验

日志是最常用的部分，也极易导致磁盘硬件故障，而从损坏的日志中恢复数据会导致更多的数据损坏。Ext4 的日志校验功能可以很方便地判断日志数据是否损坏，而且它将 Ext3 的两阶段日志机制合并成一个阶段，在增加安全性的同时提高了性能。

8. 无日志模式（No Journaling）

日志总归有一些开销，Ext4 允许关闭日志，以便某些有特殊需求的用户可以借此提升



性能。

9. 在线碎片整理

尽管延迟分配、多块分配和 extents 能有效减少文件系统碎片，但碎片还是不可避免会产生。Ext4 支持在线碎片整理，并将提供 e4defrag 工具进行个别文件或整个文件系统的碎片整理。

10. inode 相关特性

Ext4 支持更大的 inode，较之 Ext3 默认的 inode 大小 128 字节，Ext4 为了在 inode 中容纳更多的扩展属性（如纳秒时间戳或 inode 版本），默认 inode 大小为 256 字节。Ext4 还支持快速扩展属性（fast extended attributes）和 inode 保留（inodes reservation）。

11. 持久预分配（Persistent preallocation）

P2P 软件为了保证下载文件有足够的空间存放，常常会预先创建一个与所下载文件大小相同的空文件，以免未来的数小时或数天之内磁盘空间不足导致下载失败。Ext4 在文件系统层面实现了持久预分配并提供相应的 API（libc 中的 posix_fallocate()），比应用软件自己实现更有效率。

12. 默认启用 barrier

磁盘上配有内部缓存，以便重新调整批量数据的写操作顺序，优化写入性能，因此文件系统必须在日志数据写入磁盘之后才能写 commit 记录，若 commit 记录写入在先，而日志有可能损坏，那么就会影响数据完整性。Ext4 默认启用 barrier，只有当 barrier 之前的数据全部写入磁盘，才能写 barrier 之后的数据。

4.3.3 其它文件系统

Linux 支持多种文件系统，且同时存在于一个运行的系统中。查看 “/proc/filesystems” 文件，可以看到系统支持的全部文件系统。

```
vmuser@Linux-host: ~$ cat /proc/filesystems
nodev    sysfs
nodev    rootfs
nodev    ramfs
nodev    bdev
nodev    proc
nodev    cgroup
nodev    cpuset
nodev    tmpfs
nodev    devtmpfs
nodev    debugfs
nodev    securityfs
nodev    sockfs
nodev    pipefs
nodev    anon_inodes
nodev    devpts
        ext3
        ext2
        ext4
nodev    hugetlbfs
```



```
vfat  
nodev  encryptfs  
        fuseblk  
nodev  fuse  
nodev  fusectl  
nodev  pstore  
nodev  efivarfs  
nodev  mqueue  
nodev  rpc_pipefs  
nodev  binfmt_misc  
nodev  nfs  
nodev  nfs4  
nodev  nfsd
```

可以看到，Linux 支持很多种文件系统，这里不再一一介绍，仅对其中两个很具有代表性的 proc 文件系统和 sysfs 文件系统进行简单说明。

1. proc 文件系统

proc 是 Linux 系统中的一种特殊文件系统，是内核和内核模块用来向进程发送消息的机制，只存在于内存中，实际上是一个伪文件系统。用户和应用程序可通过 /proc 获得系统的信息，还可以改变内核的某些参数。/proc 子目录和所包含的内容说明如表 4.3 所列。

表 4.3 /proc 子目录内容说明

/proc 下的子目录	所包含的内容
/proc/cpuinfo	CPU 的信息 (型号, 家族, 缓存大小等)
/proc/meminfo	物理内存、交换空间等的信息
/proc/mounts	已加载的文件系统的列表
/proc/devices	可用设备的列表
/proc/filesystems	被支持的文件系统
/proc/modules	已加载的模块
/proc/version	内核版本
/proc/cmdline	系统启动时输入的内核命令行参数

2. sysfs 文件系统

sysfs 是 Linux 2.6 引入的一个新型文件系统，是一个基于内存的文件系统，它的作用是将内核信息以文件的方式提供给用户程序使用。该文件系统的目录层次结构严格按照内核的数据结构组织。除了二进制文件外（只有特殊场合才使用），sysfs 文件内容均以 ASCII 格式保存，且一个文件只保存一个数据，另外，一个文件不可大于一个内存页（通常为 4096 字节）。

sysfs 提供一种机制，使得可以显式的描述内核对象、对象属性及对象间关系。sysfs 有两组接口，一组针对内核，用于将设备映射到文件系统中，另一组针对用户程序，用于读取或操作这些设备。表 4.4 描述了内核中的 sysfs 要素及其在用户空间的表现。

表 4.4 sysfs 内部结构与外部表现

sysfs 在内核中的组成要素	在用户空间的显示
内核对象 (kobject)	目录



对象属性 (attribute)	文件
对象关系 (relationship)	链接 (Symbolic Link)

sysfs 产生了一个包含所有系统硬件的层次视图，把连接在系统上的设备和总线组织成为一个分级的文件，向用户空间导出内核数据结构和以及它们的属性。sysfs 清晰的展示了设备驱动模型中各组件的关系，顶层目录包括 block、device、bus、drivers、class、power 和 firmware 等，各目录和所包含的内容如表 4.5 所列。

表 4.5 sysfs 目录结构

/sys 下的子目录	所包含的内容
/sys/devices	这是内核对系统中所有设备的分层次表达模型，也是/sys 文件系统管理设备的最重要的目录结构
/sys/dev	这个目录下维护一个按字符设备和块设备的主次号码 (major:minor) 链接到真实的设备 (/sys/devices 下) 的符号链接文件
/sys/bus	这是内核设备按总线类型分层放置的目录结构， devices 中的所有设备都是连接于某种总线之下，在这里的每一种具体总线之下可以找到每一个具体设备的符号链接，它也是构成 Linux 统一设备模型的一部分
/sys/class	这是按照设备功能分类的设备模型，如系统所有输入设备都会出现在/sys/class/input 之下，而不论它们是以何种总线连接到系统。它也是构成 Linux 统一设备模型的一部分
/sys/kernel	这里是内核所有可调整参数的位置，目前只有 uevent_helper、kexec_loaded、mm 和新式的 slab 分配器等几项较新的设计在使用它，其它内核可调整参数仍然位于 sysctl(/proc/sys/kernel) 接口中
/sys/module	这里有系统中所有模块的信息，不论这些模块是以内联(inlined)方式编译到内核映像文件(vmlinuz)中还是编译为外部模块 (ko 文件)，都可能会出现在/sys/module 中：
	编译为外部模块(ko 文件)在加载后会出现对应的/sys/module/<module_name>/，并且在这个目录下会出现一些属性文件和属性目录来表示此外部模块的一些信息，如版本号、加载状态、所提供的驱动程序等
	编译为内联方式的模块则只在当它有非 0 属性的模块参数时会出现对应的 /sys/module/<module_name>，这些模块的可用参数会出现在 /sys/modules/<modname>/parameters/<param_name> 中：
/sys/module	如/sys/module/printk/parameters/time 这个可读写参数控制着内联模块 printk 在打印内核消息时是否加上时间前缀
	所有内联模块的参数也可以由 “<module_name>.<param_name>=<value>” 的形式写在内核启动参数上，如启动内核时加上参数 "printk.time=1" 与向 "/sys/module/printk/parameters/time" 写入 1 的效果相同
	没有非 0 属性参数的内联模块不会出现于此
/sys/power	这里是系统中电源选项，这个目录下有几个属性文件可以用于控制整个机器的电源状态，如可以向其中写入控制命令让机器关机、重启等



第5章 Vi 编辑器

本章主要介绍 UNIX 世界的文本编辑利器 Vi/Vim。熟练掌握 Vi/Vim 编辑器，能极大提高文本或者编写的效率，为开发工作带来极大便利。从接触 Vi/Vim 到熟练掌握，需要一个过程，但只要常加练习，都可以熟能生巧。

5.1 Vi/Vim 编辑器

Linux 用户经常需要对系统配置文件进行文本编辑，所以至少掌握一种文本编辑器，首选编辑器是 vi/vim。几乎任何一个发行版都有 vi 或者 vim 编辑器，在嵌入式 Linux 通常也会集成 vi 编辑器。

Vi 编辑器是 Linux 和 Unix 上最基本的文本编辑器，工作在字符模式下，支持众多的命令，是一款功能强大，效率很高的文本编辑器。Vi 编辑器可以对文本进行编辑、删除、查找和替换、文本块操作等，全部操作都是在命令模式下进行的。Vi 有两种工作模式：命令模式和输入模式。嵌入式 Linux 系统中集成的 vi 编辑器通常是由 Busybox 构建的，只支持了部分 Vi 命令，很多完整版 Vi 中的命令在嵌入式中将不可用。

Vim 是 Vi 的加强版，比 Vi 更容易使用。vi 的命令几乎全部都可以在 vim 上使用，安装了 Vim 的系统，在命令行输入 vi，实际启动的是 Vim 编辑器。下面的介绍不对 Vi 和 Vim 加以区分。

5.2 Vi 的模式

Vi 的工作模式可分为命令模式和输入模式，两者之间可以任意切换：

- 命令模式，从键盘上输入的任何字符都被作为编辑命令来解释，vi 下很多操作如配置编辑器、文本查找和替换、选择文本等都是在命令模式下进行的；
- 输入模式，从键盘上输入的所有字符都被插入到正在编辑的缓冲区中，被当作正文。

启动 Vi 后处于命令模式，在命令模式下，输入编辑命令，将进入输入模式；在输入模式下，按 ESC 键将进入命令模式，Vi 的关系转换如图 5.1 所示。

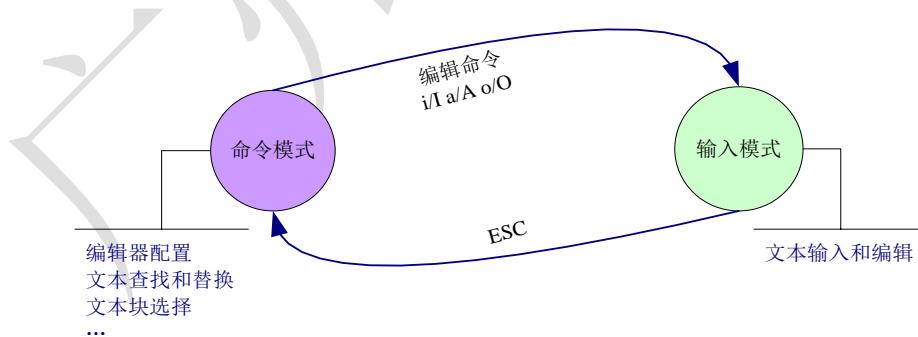


图 5.1 vi 模式转换关系

这里指的编辑命令是指：插入（i 或者 I）、附加（a 或者 A）以及打开（o 或者 O）命令。

5.3 Vim 的安装

Ubuntu 默认安装了 Vi 编辑器，但没有安装 Vim，可用 apt-get install 命令进行安装：

```
vmuser@Linux-host:$ sudo apt-get install vim
```



5.4 启动和关闭 Vi

1. 启动 vi

在 Linux Shell 终端，输入 vi 或者“vi 文件名”即可启动 Vi 编辑器，默认进入命令模式。

```
vmuser@Linux-host:$ vi
```

刚启动的 Vi 界面如图 5.2 所示。

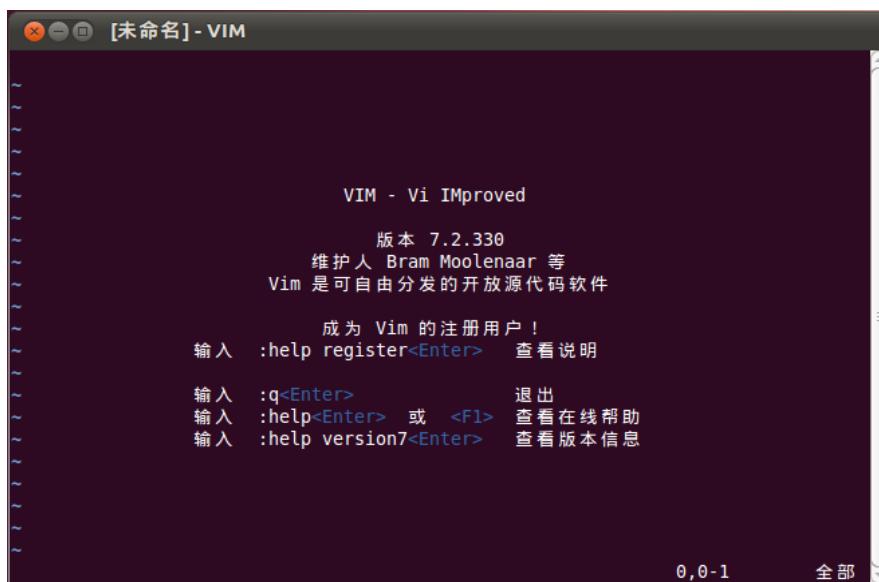


图 5.2 vi 编辑器启动界面

2. 退出 Vi

在命令模式下输入如表 5.1 所示的命令都可以退出 Vi 编辑器，回到 Shell 界面。

表 5.1 退出 Vi 的命令

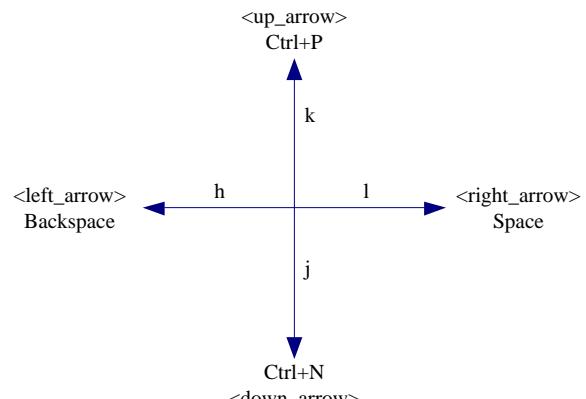
命令	说明
:q	退出未被编辑过的文件
:q!	强行退出 vi, 丢弃所做改动
:x	存盘退出 vi
:wq	存盘退出 vi
ZZ	等同于:wq

5.5 光标移动

Vi 编辑器的整个文本编辑都用键盘而非鼠标来完成，传统的光标移动方式是在命令模式下输入 h、j、k、l 完成光标的移动，后来也支持键盘的方向键以及 Page Up 和 Page Down 翻页键了，并且这些键可在命令模式和编辑模式下使用。光标移动示意图如右图所示。

总结一下，在命令模式下光标移动的方法：

- 上：k、Ctrl+P、<up_arrow>
- 下：j、Ctrl+N、<down_arrow>
- 左：h、Backspace、<left_arrow>





- 右: l、Space、<right_arrow>

无论在编辑模式下还是命令模式下，

都支持 Page Up 和 Page Down 翻页。

另外，vi 支持命令快速光标定位，常用命令如表 5.2 所列。

表 5.2 光标快速定位

命令	说明
G	将光标定位到最后一行
nG	将光标定位到第 n 行
gg	将光标定位到第 1 行
ngg	将光标定位到第 n 行
:n	将光标定位到第 n 行

5.6 文本编辑

5.6.1 文本输入

在命令模式下输入编辑命令 (i/I、a/A、o/O)，就可以进入输入模式，Vi 左下角将会提示“插入”字样，如图 5.3 所示。

在输入模式下，任何从键盘输入的字符都将被当成正文。



图 5.3 输入模式的 vi

说明：波浪线（~）开始的行表示空行。

进入输入模式的编辑命令有 a/A、i/I 和 o/O，它们之间的差异如表 5.3 所列。

表 5.3 vi 的编辑命令

命令	说明
a	在当前光标位置后面开始插入
A	在当前行行末开始插入
i	在当前光标前开始插入



I	在当前光标行行首开始插入
o	从当前光标开始下一行开始插入
O	从当前光标开始前一行开始插入

在输入模式下，可以使用键盘上的功能键对文本进行操作，如用退格键删除文本、用方向键移动光标，也可使用翻页键翻页等。

5.6.2 文本处理

使用 Vi 能进行高效的文本编辑处理，主要得益于 Vi 提供了丰富的文本处理命令，可在命令模式下进行快速的文本复制、粘贴、删除、剪切、查找、替换、撤销和恢复等操作。

1. 文本块选定

将光标移到将要选定的文本块开始处，按 ESC 进入命令模式，再按 v，进入可视状态（视图左下角提示“可视”字样），然后移动光标至文本块结尾，被选定的文本块高亮显示，如图 5.4 所示。

The screenshot shows a terminal window titled "vmuser@Linux-host: ~". Inside, a C program is displayed in the Vi editor. The entire code block from the first include directive to the end of the main function is highlighted in red, indicating it is selected. The status bar at the bottom right shows "20,0-1" and "顶端".

```
// Purpose of this utility is to timestamp each line coming over stdin
// USAGES:
//   tstamp.exe < /dev/ttys0
//   <commands> | tstamp.exe
//

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>
#include <sys/signalf.h>

#define MAX_BUF_SIZE (10*1024)

char buf[MAX_BUF_SIZE];

// Global variables that control process shutdown.
sig_atomic_t graceful_quit = 0;

// Signal handler for SIGINT.
void SIGINT_handler (int signum)
{
-- 可视 --
```

图 5.4 选定文本块

连接两次 ESC 可以取消所选定文本块。

2. 复制和粘贴

如果已经选定文本块，按 y，即可将所选定文本复制到缓冲区，将光标移到将要粘贴的地方，按 p，就可完成文本粘贴。

Vi 提供了很多简便快捷的复制方法，在命令模式下，连接 yy，即可复制光标所在行的内容，连接 yny 即可复制从光标所在行开始的 n 行。例如：y5y，复制光标开始的 5 行内容。

3. 剪切和删除

最后一次剪切和删除的内容都能够被粘贴到其它位置。常用的剪切和删除命令如表 5.4 所示。

表 5.4 剪切和删除命令

命令	说明
x 或 nx	剪切从光标所在的位置开始的一个或 n 个字符



X 或 nX	剪切光标前的一个或 n 个字符
dd	删除光标所在的行
D	删除从光标位置开始至行尾
dw	删除从光标位置至该词末尾的所有字符
d0	删除从光标位置开始至行首
dnd	删除光标所在行开始的 n 行
dnG	将光标所在行至第 n 行删除

4. 文本查找

在命令模式下，输入“/字符串”即可从光标位置开始向下查找字符串，如输入“/text”，即从光标所在位置向下开始查找 text 字符串。输入“?字符串”则从光标位置开始向上查找字符串。无论向下还是向上查找，查找下一个，按键盘 n 键即可。如图 5.5 所示是在 Vi 中搜索字符串 signum 得到的结果。

```
12 #include <sys	signal.h>
13
14 #define MAX_BUF_SIZE    (10*1024)
15
16 char buf[MAX_BUF_SIZE];
17
18 // Global variables that control process shutdown.
19 sig_atomic_t graceful_quit = 0;
20
21 // Signal handler for SIGINT.
22 void SIGINT_handler (int signum)
23 {
24     assert (signum == SIGINT);
25     graceful_quit = 1;
26 }
27
28 // Signal handler for SIGQUIT.
29 void SIGQUIT_handler (int signum)
30 {
31     assert (signum == SIGQUIT);
32     graceful_quit = 1;
33 }
34
/signum
```

图 5.5 搜索字符串

默认情况下搜索到的字符串不会高亮显示，在命令模式下输入“:set hlsearch”可以实现高亮显示。

用“/字符串”或者“?字符串”方式搜索，将以局部匹配方式显示搜索结果，例如搜索字符串 abc，字符串 abcd 也将被显示在搜索结果中。如图 5.5 所示，搜索 signum，而 signum_not 也被搜索到了。

如果不希望将 abcd 列入显示结果中，则可用全局匹配搜索。方法，先将光标移动到字符串 abc，然后按“SHIFT+*”，完成搜索。如图 5.6 所示，用这种方法搜索字符串 signum，字符串 signum_not 已经不在搜索结果中。



```
9 #include <assert.h>
10 #include <time.h>
11 #include <sys/time.h>
12 #include <sys/signals.h>
13
14 #define MAX_BUF_SIZE    (10*1024)
15
16 char buf[MAX_BUF_SIZE];
17
18 // Global variables that control process shutdown.
19 sig_atomic_t graceful_quit = 0;
20
21 // Signal handler for SIGINT.
22 void SIGINT_handler (int signum)
23 {
24     assert (signum != SIGINT);
25     graceful_quit = 1;
26 }
27
28 // Signal handler for SIGQUIT.
29 void SIGQUIT_handler (int signum)
30 {
31     assert (signum == SIGQUIT);
32 }
```

图 5.6 全局匹配搜索

5. 文本替换

文本替换的命令稍微复杂一些，在命令模式下，输入：

```
:%s /old/new/g
```

能够将文本内全部的字符串 old 替换为 new。为了安全起见，可以在替换命令尾部加上 c，这样每次替换前都需要确认一下。

6. 撤销和恢复

在命令模式下输入 u，可撤销所做的更改，恢复编辑前的状态。这里的编辑以保存命令为界，例如，打开一篇文本，在编辑过程中被保存了 3 次，则可撤销 3 次。最多能撤销的次数由 Vi 的 undolevels 决定，一般是 500。不小心多按了 u 时，可以用 Ctrl+R 来恢复。

注意，一旦文本被关闭，再次打开将无法使用 u 撤销所做更改。

5.7 配置 vi

Vi 编辑器支持很多配置选项，如设置和取消行号、设置 TAB 键字符数、设置语法高亮等，在命令模式下输入“:set”可以对 Vi 进行配置。在一般模式下，输入“:set number”可以显示行号，输入“:set nonumber”取消显示行号。常用的配置命令如表 5.5 所示。

表 5.5 配置 vi 命令

命令	说明
:set number	显示行号
:set ignorecase	不区分大小写
:set tabstop=n	按下 tab 键则实际输入 n 个空格
:set hlsearch	搜索时高亮显示
:syntax on	开启语法高亮

灵活利用 Vi 的这些配置特性，在编程过程中能带来极大便利，如在 C 编程中，设置语法高亮，能极大减少编码中的书写错误。设置语法高亮和没有设置语法高亮的效果对比如图 5.7 所示。



```
// tstamp.exe < /dev/ttyS0
// <commands> | tstamp.exe
//
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>
#include <sys	signal.h>

#define MAX_BUF_SIZE    (10*1024)

char buf[MAX_BUF_SIZE];

// Global variables that control process shutdown.
sig_atomic_t graceful_quit = 0;

// Signal handler for SIGINT.
void SIGINT_handler (int signum)
{
    assert (signum == SIGINT);
    graceful_quit = 1;
}
```

图 5.7 开启和不开启语法高亮对比

再例如显示行号在编写代码的时候也非常有帮助，但是在用鼠标选择代码片段并复制的时候，如果开启了显示行号的话，会连同行号一起复制，如果不需要行号，可以在命令模式下输入“:set nu!”将行号关闭。显示与不显示行号的效果如图 5.8 所示。

```
1 // Purpose of this utility is to timestamp each line coming over stdin
2 // USAGES:
3 //   tstamp.exe < /dev/ttyS0
4 //   <commands> | tstamp.exe
5 //
6
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <assert.h>
10 #include <time.h>
11 #include <sys/time.h>
12 #include <sys	signal.h>
13
14 #define MAX_BUF_SIZE    (10*1024)
15
16 char buf[MAX_BUF_SIZE];
17
18 // Global variables that control process shutdown.
19 sig_atomic_t graceful_quit = 0;
20
21 // Signal handler for SIGINT.
22 void SIGINT_handler (int signum)
23 {
:set nu!
```

图 5.8 显示和不显示行号的效果对比

在 Vi 内执行配置命令的效果是临时的，关闭 Vi，再次打开 Vi，需要重新配置。Vi 有自己的配置文件，可以是“/etc/vim/vimrc”或者“~/.vimrc”。两者的区别是前者全局的，影响登录本机的全部用户，后者仅仅对当前用户有效。

把配置命令放在配置内，每次启动 Vi 就会自动载入配置文件中的设置。如程序清单 5.1 所示是一个简单的 Vi 配置文件范例。

程序清单 5.1 vi/vim 配置文件范例

```
" 在窗口标题栏显示文件名称
set title

" 编辑的时候将所有的 tab 设置为空格
set tabstop=4

" 设置自动对齐空格数
set shiftwidth=4
```



```
"显示行号  
set number  
"搜索时高亮显示  
set hlsearch  
"不区分大小写  
set ignorecase  
"语法高亮  
syntax on
```

说明：以“开始的是注释。

Vi 的功能远远不止这些，更多的命令请参考 Vi 的用户手册以及其它资料，熟练掌握 Vi 的使用，能极大提高 Linux 下的文本编辑效率。

5.8 文件对比

在编码过程中，经常会用到文件对比功能。Vim 包含了文件对比工具 vimdiff。用 vimdiff 工具可以很容易实现文件对比。

用法：

```
vmuser@Linux-host: ~$ vimdiff file1 file2 file3
```

vimdiff 可以同时进行 2 个以上文件的对比，但大多数情况下是进行两个文件的对比。如图 5.9 所示是两个有差异的文件的对比效果。

```
+ --- 7 行: Purpose of this utility is to timestamp | + --- 7 行: Purpose of this utility is to timestamp |
#include <stdlib.h> #include <stdlib.h>
#include <assert.h> #include <assert.h>
#include <time.h> #include <time.h>
#include <sys/time.h> #include <sys/time.h>
#include <sys/signalf.h> #include <sys/signalf.h>

char buf[MAX_BUF_SIZE];
// Global variables that control process shutdown.
sig_atomic_t graceful_quit = 0;

// Signal handler for SIGINT.
void SIGINT_handler (int signum)
{
    assert (signum == SIGINT);
    graceful_quit = 1;
}

tstamp.c          12,1      顶端 tstamp2.c        25,2-5      顶端
"tstamp2.c" [已转换] 110L, 2166C
```

图 5.9 vimdiff 文件对比



第6章 嵌入式 Linux 开发环境构建

本章首先讲述在 Linux 环境下进行嵌入式 Linux 开发的基本方法，然后对嵌入式开发用到的软件进行介绍，包括如何安装和测试。这一章是进行嵌入式 Linux 开发必不可少的，是进行嵌入式 Linux 开发的基础，请务必仔细理解，并进行正确的设置。

如果用从 Ubuntu 官网下载的 ISO 镜像，安装后只能得到纯净的 Ubuntu 系统，进行嵌入式 Linux 开发，必须按照本章步骤搭建嵌入式 Linux 开发环境。

如果从广州致远电子有限公司官网下载经过重新打包的 Ubuntu 镜像，安装后将会得到已经构建好嵌入式 Linux 开发环境的 Ubuntu 系统，则可以跳过本章安装部分内容，直接进行测试。

如果直接使用我们已经安装好后打包的 Ubuntu 虚拟机，该虚拟机也已经安装了完整的嵌入式 Linux 开发环境，也可以跳过本章的安装。

6.1 嵌入式 Linux 开发模型

6.1.1 交叉编译

由于嵌入式系统资源匮乏，一般不能像 PC 一样安装本地编译器和调试器，不能在本地编写、编译和调试自身运行的程序，而需借助其它系统如 PC 来完成这些工作，这样的系统通常被称为宿主机。

宿主机通常是 Linux 系统，并安装交叉编译器、调试器等工具；宿主机也可以是 Windows 系统，安装嵌入式 Linux 集成开发环境。在宿主机上编写和编译代码，通过串口、网口或者硬件调试器将程序下载到目标系统里面运行，系统示意图如图 6.1 所示。

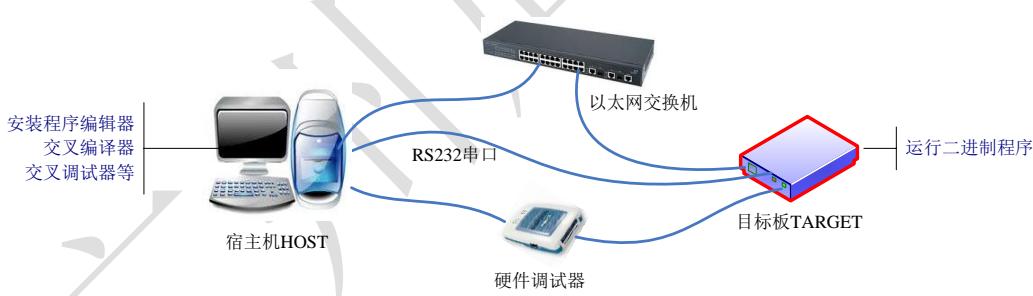


图 6.1 嵌入式 Linux 开发环境

所谓的交叉编译，就是在宿主机平台上使用某种特定的交叉编译器，为某种与宿主机不同平台的目标系统编译程序，得到的程序在目标系统上运行而非在宿主机本地运行。这里的平台包含两层含义：一是核心处理器的架构，二是所运行的系统。这样，交叉编译有 3 种情形：

- 目标系统与宿主机处理器相同，运行不同的系统；
- 目标系统与宿主机处理器不同，运行相同的系统；
- 目标系统与宿主机处理器不同，运行不同的系统。

实际上，在 PC 上进行非 Linux 的嵌入式开发，哪怕使用 IDE 集成环境如 Keil、ADS、Realview，都是交叉编译和调试的过程，只是 IDE 工具隐藏了细节，没有明确提出这个概念而已。

6.1.2 交叉编译器



交叉编译器是在宿主机上运行的编译器，但是编译后得到的二进制程序却不能在宿主机上运行，而只能在目标机上运行。交叉编译器命名方式一般遵循“处理器-系统-gcc”这样的规则，一般通过名称便可以知道交叉编译器的功能。例如下列交叉编译器：

- arm-none-eabi-gcc，表示目标处理器是 ARM，不运行操作系统，仅运行前后台程序；
- arm-uclinuxabi-gcc，表示目标处理器是 ARM，运行 uClinux 操作系统；
- arm-none-linux-gnueabi-gcc，表示目标处理器是 ARM，运行 Linux 操作系统；
- mips-linux-gnu-gcc，表示目标处理器是 MIPS，运行 Linux 操作系统。

进行 ARM Linux 开发，通常选择 arm-linux-gcc 交叉编译器。ARM-Linux 交叉编译器可以自行从源代码编译，也可以从第三方获取。在能从第三方获取交叉编译器的情况下，请尽量采用第三方编译器而不要自行编译，一是编译过程繁琐，不能保证成功，二是就算编译成功，也不能保证交叉编译器的稳定性，编译器的不稳定性会对后续的开发带来无限隐患。而第三方提供的交叉编译器通常都经过比较完善的测试，确认是稳定可靠的。

6.2 安装交叉编译器

6.2.1 解压工具链压缩包

光盘提供了 gcc-4.4.4-glibc-2.11.1-multilib-1.0.tar.bz2 这样的交叉编译工具发布包，解压命令：

```
vmuser@Linux-host: ~$ tar xjvf gcc-4.4.4-glibc-2.11.1-multilib-1.0.tar.bz2
```

如果希望解压到一个指定的目录，可以先将 gcc-4.4.4-glibc-2.11.1-multilib-1.0.tar.bz2 压缩包复制到目标目录，然后进入目标目录再运行解压命令，也可以在任意目录解压，通过-C 指定目标目录。假定希望解压到 “/opt/” 目录，则命令如下：

```
vmuser@Linux-host: ~$ tar xjvf gcc-4.4.4-glibc-2.11.1-multilib-1.0.tar.bz2-C /opt/
```

1. 确定交叉编译器的实际目录

在完成解压后，如果不设置环境变量，如果不指定交叉编译器的完整路径，系统是无法调用交叉编译器的。

假如交叉工具链安装在

“/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/” 目录下，用 ls 命令可以看到该目录下的各种文件：

```
vmuser@Linux-host: ~$ ls /opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/
arm-fsl-linux-gnueabi-addr2line          arm-fsl-linux-gnueabi-gprof
arm-fsl-linux-gnueabi-ar                  arm-fsl-linux-gnueabi-ld
arm-fsl-linux-gnueabi-as                  arm-fsl-linux-gnueabi-ldd
arm-fsl-linux-gnueabi-c++                arm-fsl-linux-gnueabi-nm
arm-fsl-linux-gnueabi-cc                  arm-fsl-linux-gnueabi-objcopy
arm-fsl-linux-gnueabi-c++filt            arm-fsl-linux-gnueabi-objdump
arm-fsl-linux-gnueabi-cpp                arm-fsl-linux-gnueabi-populate
arm-fsl-linux-gnueabi-ct-ng.config       arm-fsl-linux-gnueabi-ranlib
arm-fsl-linux-gnueabi-g++                arm-fsl-linux-gnueabi-readelf
arm-fsl-linux-gnueabi-gcc                arm-fsl-linux-gnueabi-run
arm-fsl-linux-gnueabi-gcc-4.4.4           arm-fsl-linux-gnueabi-size
arm-fsl-linux-gnueabi-gccbug             arm-fsl-linux-gnueabi-strings
arm-fsl-linux-gnueabi-gcov              arm-fsl-linux-gnueabi-strip
```



```
arm-fsl-linux-gnueabi-gdb
```

有不少初学者不知道自己已经将交叉编译器安装在哪个目录下了，确认方法就是看 arm-fsl-linux-gnueabi-* 这些文件到底在哪个目录。确定了在哪个目录后，接下去的事情就好办了。

2. 全路径引用

如果不想添加设置交叉编译器的路径到系统环境变量中，则必须在每次使用交叉编译器的地方写明交叉编译器的全路径，例如：

```
export CC=/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/arm-fsl-linux-gnueabi-make CROSS_COMPILE=$CC ARCH=arm uImage
```

如果系统安装了多个不同版本的同名编译器，就可以采用这种方法。不过前提是必须对自己安装的交叉编译器路径有清醒的认识。

6.2.2 设置环境变量

如果系统只有一个交叉编译器，还是强烈推荐设置系统环境变量，毕竟每次都设置全路径，稍微显得有点麻烦。设置系统环境变量后，只需在 Linux 终端输入 arm-fsl-linux-gnueabi-gcc，就可以调用交叉编译器，简单方便。

设置系统环境变量有 3 种方法，下面分别讲述。

1. 临时设置

临时设置系统环境变量，是通过 export 命令，将交叉编译器的路径添加到系统 PATH 环境变量中。用法（多个值之间用冒号隔开）：

```
vmuser@Linux-host: ~$ export PATH=$PATH:/交叉编译器路径
```

紧接前面这个示例，在添加交叉编译器路径前，先查看系统 PATH 的值：

```
vmuser@Linux-host:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

添加工具链路径：

```
vmuser@Linux-host: ~$ export PATH=$PATH:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/
```

再次查看 PATH 的值：

```
vmuser@Linux-host:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/  
arm-fsl-linux-gnueabi/bin/
```

可以看到，交叉编译器的路径已经被添加到系统 PATH 变量中。此时在终端输入 arm-fsl-linux-gnueabi-，然后按键盘 TAB 键，可以看到很多 arm-fsl-linux-gnueabi- 开头的命令被列了出来，说明系统已经能够正确找到交叉编译器了。

```
vmuser@Linux-host:~$ arm-fsl-linux-gnueabi-  
arm-fsl-linux-gnueabi-addr2line      arm-fsl-linux-gnueabi-gcc          arm-fsl-linux-gnueabi-objcopy  
arm-fsl-linux-gnueabi-ar            arm-fsl-linux-gnueabi-gcc-4.4.4    arm-fsl-linux-gnueabi-objdump  
arm-fsl-linux-gnueabi-as            arm-fsl-linux-gnueabi-gccbug        arm-fsl-linux-gnueabi-populate  
arm-fsl-linux-gnueabi-c++           arm-fsl-linux-gnueabi-gcov         arm-fsl-linux-gnueabi-ranlib  
arm-fsl-linux-gnueabi-cc            arm-fsl-linux-gnueabi-gdb          arm-fsl-linux-gnueabi-readelf  
arm-fsl-linux-gnueabi-c++filt       arm-fsl-linux-gnueabi-gprof        arm-fsl-linux-gnueabi-run  
arm-fsl-linux-gnueabi-cpp           arm-fsl-linux-gnueabi-ld           arm-fsl-linux-gnueabi-size
```



arm-fsl-linux-gnueabi-ct-ng.config

arm-fsl-linux-gnueabi-ldd

arm-fsl-linux-gnueabi-strings

arm-fsl-linux-gnueabi-g++

arm-fsl-linux-gnueabi-nm

arm-fsl-linux-gnueabi-strip

这种方法设置环境变量，只能对当前终端有效，关闭终端再次打开将会失效，需要重新设置。

2. 修改全局配置文件

在终端中添加环境变量，需要每次打开终端都设置，也很麻烦。可以考虑将设置的过程添加到系统配置文件中。`/etc/profile` 是系统全局的配置文件，在该文件中设置交叉编译器的路径，能够让登录本机的全部用户都可以使用这个编译器。

打开终端，输入“`sudo vi /etc/profile`”命令，打开`/etc/profile`文件，在文件末尾添加：

```
export PATH=$PATH:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin
```

保存文件并退出，然后在终端输入“`./etc/profile`”（点+空格+文件名），执行`profile`文件，使刚才的改动生效。如果没有书写错误，此时打开终端，输入`arm-fsl-linux-gnueabi-`，然后按键盘 TAB 键，同样可以看到很多`arm-fsl-linux-gnueabi-`开头的命令。

3. 修改用户配置文件（推荐）

“`/etc/profile`”是全局配置文件，会影响登录本机的全部用户。如果不希望影响其他用户，也可以只修改当前用户的配置文件，通常是“`~/.bashrc`”或者“`~/.bash_profile`”。

修改方法与修改“`/etc/profile`”类似，这是无需`sudo`，直接`vi`打开即可，在文件末尾增加：

```
export PATH=$PATH:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin
```

与执行“`/etc/profile`”的方式一样，输入“`~/.bashrc`”或者“`~/.bash_profile`”，执行修改过的文件，使修改生效。如果无误，打开终端，输入`arm-fsl-linux-gnueabi-`，然后按键盘 TAB 键，同样可以看到很多`arm-fsl-linux-gnueabi-`开头的命令。

4. 测试工具链

简单测试。打开终端，输入交叉编译器命令，如`arm-fsl-linux-gnueabi-gcc`，然后回车，能够得到下列类似信息，说明交叉编译器已经能够正常工作了。

```
vmuser@Linux-host:~$ arm-fsl-linux-gnueabi-gcc
arm-fsl-linux-gnueabi-gcc: no input files
```

进一步测试，可以编写一个简单的 c 文件，用交叉编译器交叉编译，并查看编译结果。在“`~`”目录下创建`hello.c`文件，然后编写如程序清单 6.1 所示内容。

程序清单 6.1 Hello 程序清单

```
#include <stdio.h>
int main(void)
{
    int i;
    for (i=0; i<5; i++) {
        printf("Hello %d!\n", i);
    }
    return 0;
}
```

输入完成后，保存并关闭`hello.c`文件，输入以下命令对`hello.c`进行编译并查看编译后



生成文件的属性：

```
vmuser@Linux-host: ~$ cd ~  
vmuser@Linux-host: ~$ arm-fsl-linux-gnueabi-gcc hello.c -o hello  
vmuser@Linux-host: ~$ file hello  
hello:ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for  
GNU/Linux 2.6.26, not stripped
```

可以看到 hello 程序是 32 位 ARM 指令架构的程序。

5. 安装 32 位的兼容库

如果前面的操作都能得到正确现象，那么就该为自己庆祝一下了，因为交叉编译器已经安装完毕，并且能够正确工作了。但是事情总有例外，很有可能在终端输入 arm-fsl-linux-gnueabi-gcc 命令后，得到的却是下面的结果：

```
-bash: ./arm-fsl-linux-gnueabi-gcc: 没有那个文件或目录
```

此时请确认：

在某个目录下确实存在 arm-fsl-linux-gnueabi-gcc 文件；

在终端输入 arm-fsl-linux-gnueabi-，按 TAB 键，能找到 arm-fsl-linux-gnueabi-* 系列命令。

如果这两个条件都确认无误，那么问题就好解决了。这种问题主要发生在 64 位操作系统上，原因在于大多数交叉编译器为了适应性，通常以 32 位发布，而实际系统是 64 位的，存在架构差异，所以不能执行。

解决办法很简单，安装 32 位兼容库就好了。在 Ubuntu 12.04 上的安装命令：

```
vmuser@Linux-host: ~$ sudo apt-get install ia32-libs
```

32 兼容库需要从 Ubuntu 的源下载，所以此时主机系统应当能访问互联网。

在 Ubuntu 12.04 64 位下安装 32 位库的名字为 ia32-libs，在其它版本的 Ubuntu 名称可能有变。如果不能正确安装兼容库，请读者自行到互联网上寻求解决方案。

安装 32 位兼容库后，再次执行 arm-fsl-linux-gnueabi-gcc 命令，应该得到如下信息：

```
arm-fsl-linux-gnueabi-gcc: no input files
```

6.3 SSH 服务器

6.3.1 SSH 能做什么？

SSH 是 Secure Shell 的缩写，是建立在应用层和传输层基础上的安全协议，能够有效防止远程管理过程中的信息泄露问题。

SSH 实际上是一个 Shell，可以通过网络登录远程系统，当然，前提是远程系统已经开启了 SSH 服务。经常会遇到下列情形：

- Linux 主机不在本地，但又要使用或者维护这台计算机；
- 一个嵌入式 Linux 产品不方便接调试串口，需要进行维护；
- 在远程机器和本地机器之间进行文件传输。

如果远程目标系统已经开启了 SSH 服务，通过 SSH 可以轻松解决以上问题。

使用 SSH 服务，一方面需要在远程系统上安装 SSH 服务，另一方面要在本地系统上安装 SSH 客户端，常见的 SSH 客户端有 putty、SSH Secure Shell Client 等。下面分别介绍。



注意，在本机安装了虚拟机，也可以将虚拟机的 Linux 认为是远程系统。若使用 SSH 客户端软件登录虚拟机中的 Linux 系统，必须配置虚拟机的以太网连接方式为 Bridged（桥接）模式，同时电脑的物理网卡必须接到网络，否则客户端将无法连接 SSH 服务器。

6.3.2 安装 SSH 服务器

在 Linux 主机输入下面命令安装 ssh 服务器：

```
vmuser@Linux-host:~$ sudo apt-get install openssh-server
```

SSH 服务器安装成功后，终端显示如图 6.2 所示。

```
vmuser@Linux-host:~$ sudo apt-get install openssh-server
[sudo] password for vmuser:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包:
  openssh-client
建议安装的软件包:
  libpam-ssh keychain monkeysphere openssh-blacklist openssh-blacklist-extra
  rssh molly-guard
下列软件包将被升级:
  openssh-client openssh-server
升级了 2 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 318 个软件包未被升级。
需要下载 1,281 kB 的软件包。
解压缩后会消耗掉 0 B 的额外空间。
您希望继续执行吗? [Y/n]
获取: 1 http://cn.archive.ubuntu.com/ubuntu/ precise-updates/main openssh-server
  amd64 1:5.9p1-5ubuntu1.3 [338 kB]
获取: 2 http://cn.archive.ubuntu.com/ubuntu/ precise-updates/main openssh-client
  amd64 1:5.9p1-5ubuntu1.3 [943 kB]
下载 1,281 kB, 耗时 2 秒 (484 kB/s)
正在预设定软件包 ...
(正在读取数据库 ... 系统当前共安装有 143512 个文件和目录。)
正预备替换 openssh-server 1:5.9p1-5ubuntu1.1 (使用 .../openssh-server_1%3a5.9p1-
5ubuntu1.3_amd64.deb) ...
正在解压缩将用于更替的包文件 openssh-server ...
正预备替换 openssh-client 1:5.9p1-5ubuntu1.1 (使用 .../openssh-client_1%3a5.9p1-
5ubuntu1.3_amd64.deb) ...
正在解压缩将用于更替的包文件 openssh-client ...
正在处理用于 man-db 的触发器...
正在处理用于 ureadahead 的触发器...
正在处理用于 ufw 的触发器...
正在设置 openssh-client (1:5.9p1-5ubuntu1.3) ...
正在设置 openssh-server (1:5.9p1-5ubuntu1.3) ...
ssh stop/waiting
ssh start/running, process 3497
vmuser@Linux-host:~$
```

图 6.2 成功安装 ssh 服务器

6.3.3 测试 SSH 服务

在虚拟机里，VMware 虚拟网卡设置为 NAT 模式的话，Linux 系统网卡设置为动态 IP 即可；如果虚拟网卡设置为桥接模式，则需要为 Linux 设置一个与 Windows 系统同一个网段的静态 IP 地址。

静态 IP 设置方法，可以在图形界面进入系统设置，选择网卡设置，IPV4 设置为“手动”，并在地址栏填写 IP 地址、掩码等信息，参考图 6.3。



图 6.3 设置静态 IP 地址

当然，也可以在终端使用 ifconfig 命令进行设置：

```
vmuser@Linux-host: ~$ sudo ifconfig eth0 192.168.1.137
```

只有知道了 Linux 主机的 IP 地址后，才能进行 SSH 连接。如果不能确定 IP 地址，可以打开终端，用 ifconfig 命令进行查看和确认：

```
vmuser@Linux-host: ~$ ifconfig
```

进行 SSH 连接之前，最好先用 ping 命令测试 Windows 和 Linux 之间能否正常通信。可以在 Windows，打开 cmd 命令行，输入 ping 命令进行测试，例如测试 IP 为 192.168.1.137 的 Linux 主机，能收到回应帧表示通信正常，如图 6.4 所示。

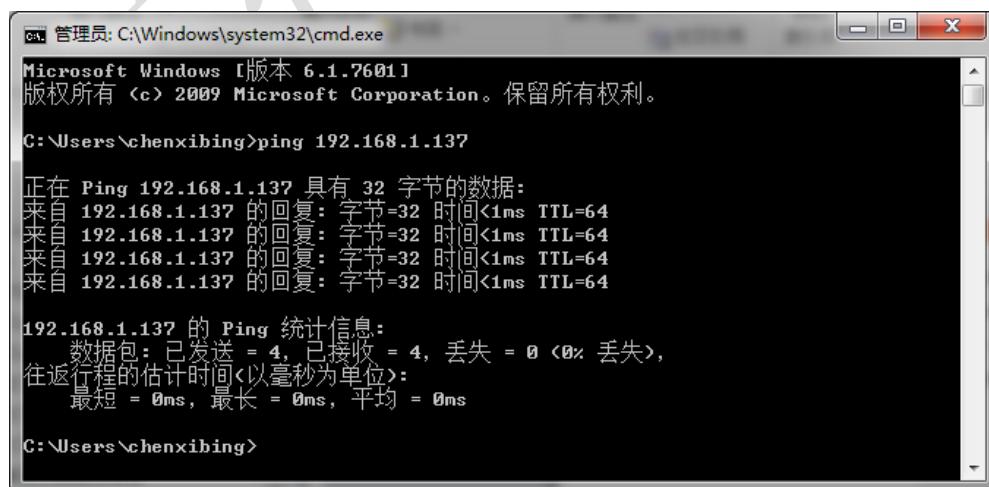


图 6.4 Windows ping 命令测试

也可以在 Linux 下打开终端，用 ping 命令 ping Windows 主机，收到回应帧表示测试正常，如图 6.5 所示。



```
vmuser@Linux-host:~$ ping 192.168.1.100
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
64 bytes from 192.168.1.100: icmp_req=1 ttl=64 time=0.575 ms
64 bytes from 192.168.1.100: icmp_req=2 ttl=64 time=0.269 ms
64 bytes from 192.168.1.100: icmp_req=3 ttl=64 time=0.426 ms
64 bytes from 192.168.1.100: icmp_req=4 ttl=64 time=0.335 ms
64 bytes from 192.168.1.100: icmp_req=5 ttl=64 time=0.425 ms
```

图 6.5 Linux ping 命令测试

注意：Windows 7 默认打开了系统防火墙，会过滤掉 ping 请求。如果在 Linux 下 ping Windows 7，需要先关闭 Windows 7 的防火墙。另外，Windows 也需要设置静态的 IP 地址。

6.3.4 用 Putty 测试

Putty（下载地址：www.putty.org）是一个小巧的多功能绿色工具，可以实现 SSH、telnet、串口等多种协议登录，程序界面如图 6.6 所示。

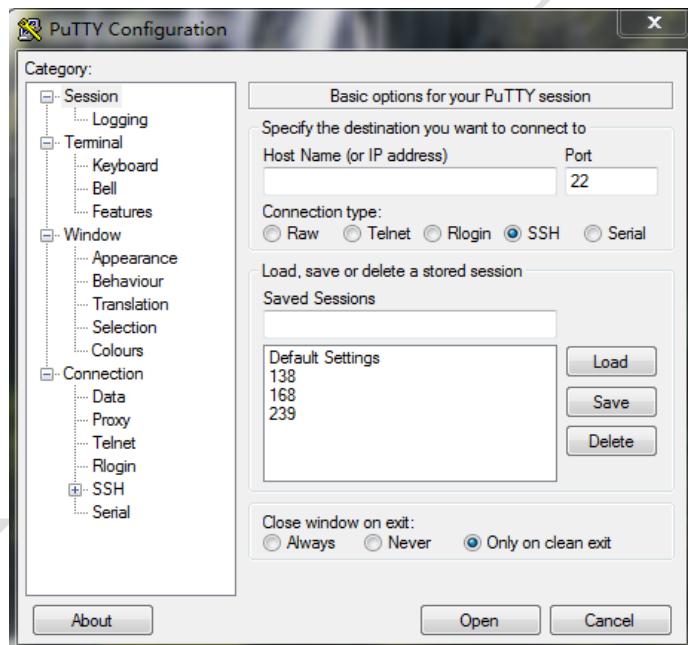


图 6.6 Putty 软件界面

选择 SSH 登录需要进行一些设置。在“Host Name”一栏填写远程系统的 IP 地址，在“Connection type”一栏中选择 SSH，如图 6.7 所示。

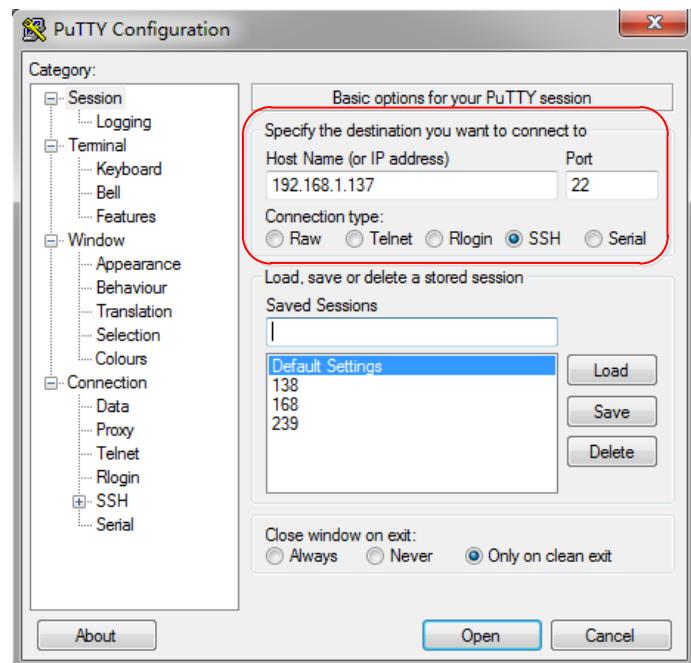


图 6.7 Putty 登录设置

然后点击“Open”将会出现如所示的登录界面，输入用户名和密码即可，如图 6.8 所示（密码不可见）。

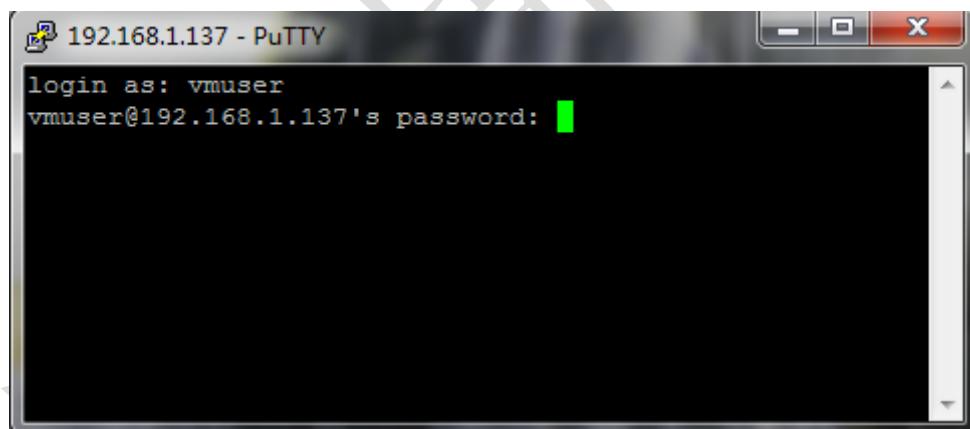


图 6.8 输入用户名和密码

第一次登录会出现如图 6.9 所示的警告框，点击“是”即可。

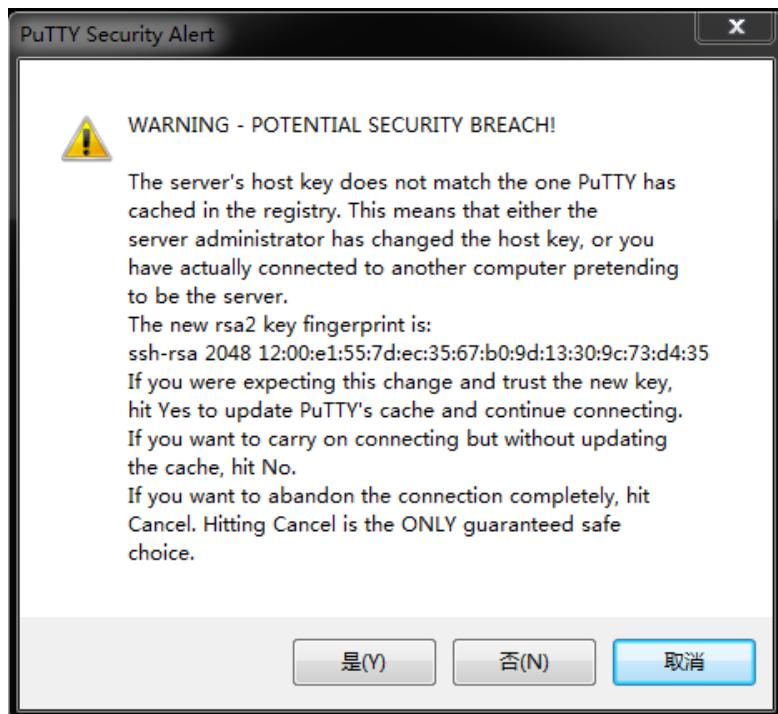


图 6.9 警告提示和确认

通过 Putty 登录成功的界面如图 6.10 所示。然后在这个终端可以输入 Linux 的命令进行操作。

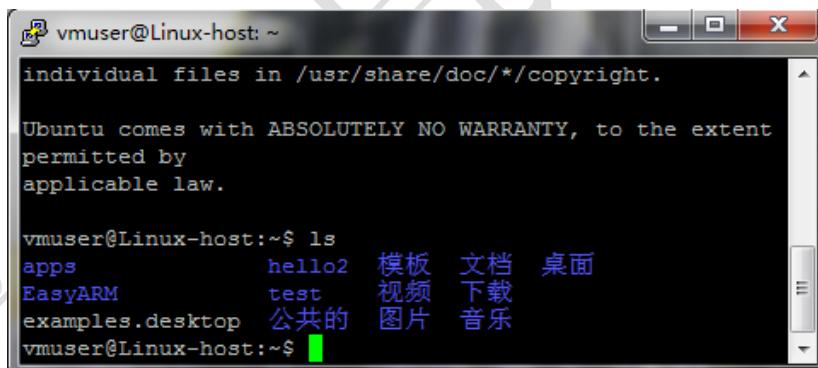
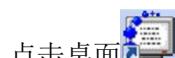


图 6.10 Putty 成功登录远程 Linux

6.3.5 用 SSHSecure Shell 测试

首先请下载并安装 SSH Secure Shell Client 软件（[可通过网络搜索文件名称得到下载地址](#)），该软件安装很简单，不再介绍。SSH Secure Client 除了能进行远程 Shell 登录之外，还能通过 SSH Secure File Transfer Client 进行文件传输。



点击桌面图标，打开 SSH Secure Shell Client 软件，如图 6.11 所示。

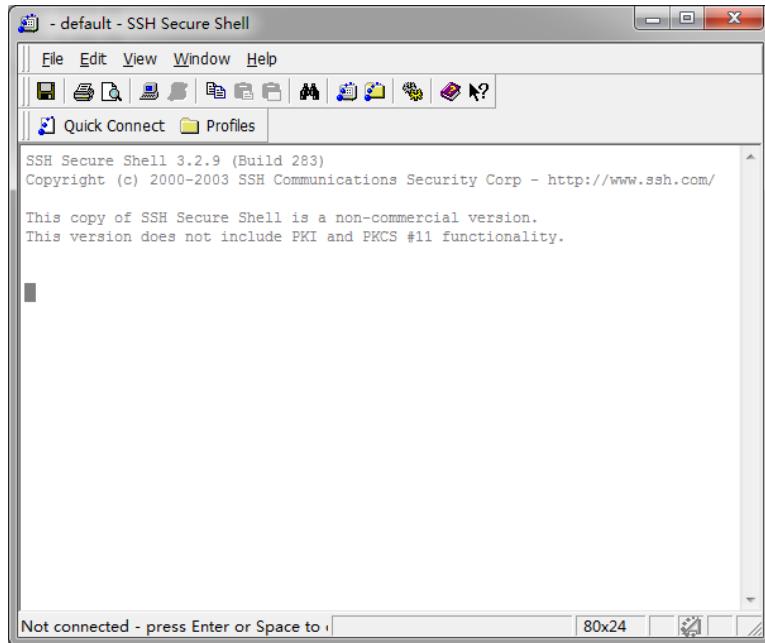


图 6.11 打开 SSH 客户端软件

点击窗口的“Quick Connect”按钮，将弹出一个连接对话框，如图 6.12 所示。在 Host Name 栏 Linux 的主机 IP 地址，如 192.168.1.137，在 User Name 输入用户名，如 vmuser。

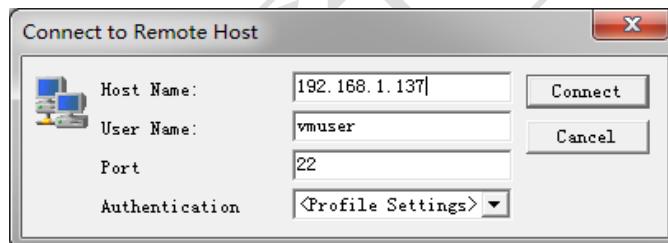


图 6.12 输入主机 IP 和登录用户名

点击“Connect”按钮，如果是第一次连接，会出现如图 6.13 所示的确认提示，点击“Yes”确认即可。

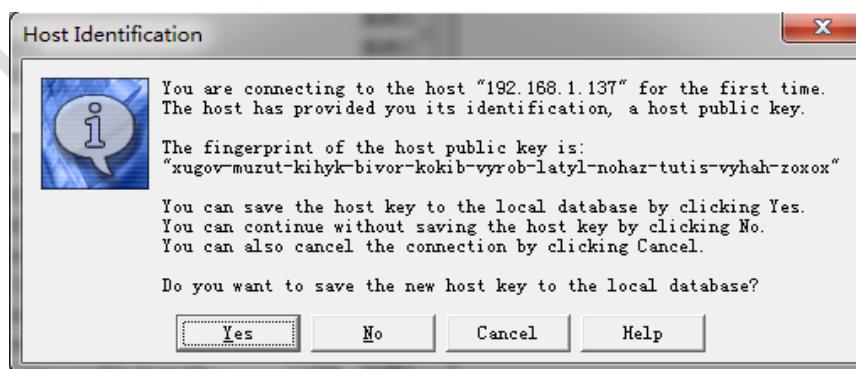


图 6.13 信息确认

然后将弹出一个对话框要求输入登录用户的密码，如图 6.14 所示。



图 6.14 输入密码

输入密码后并无误后，将成功连接到主机，如图 6.15 所示。

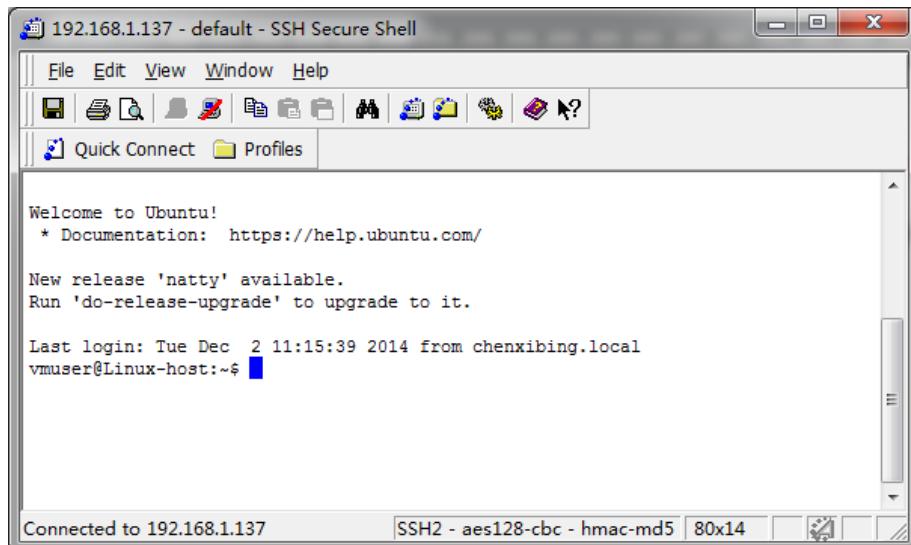


图 6.15 SSH Secure Shell 连接成功

点击界面 图标，或者打开“Window→New File Transfer”可以打开一个远程文件传输界面，如图 6.16 所示。

左边是本地 Windows 主机目录，浏览到任意目录，选中文件或者文件夹后，鼠标右键 →upload 即可将本地文件或者文件夹上传到远程 Linux 主机；右边是远程 Linux 主机文件目录，选中文件或者目录后，鼠标右键 →Download 即可将远程 Linux 主机的文件下载到本地 Windows 主机。

说明：SSH 对中文支持不好，最好不要传输中文文件。

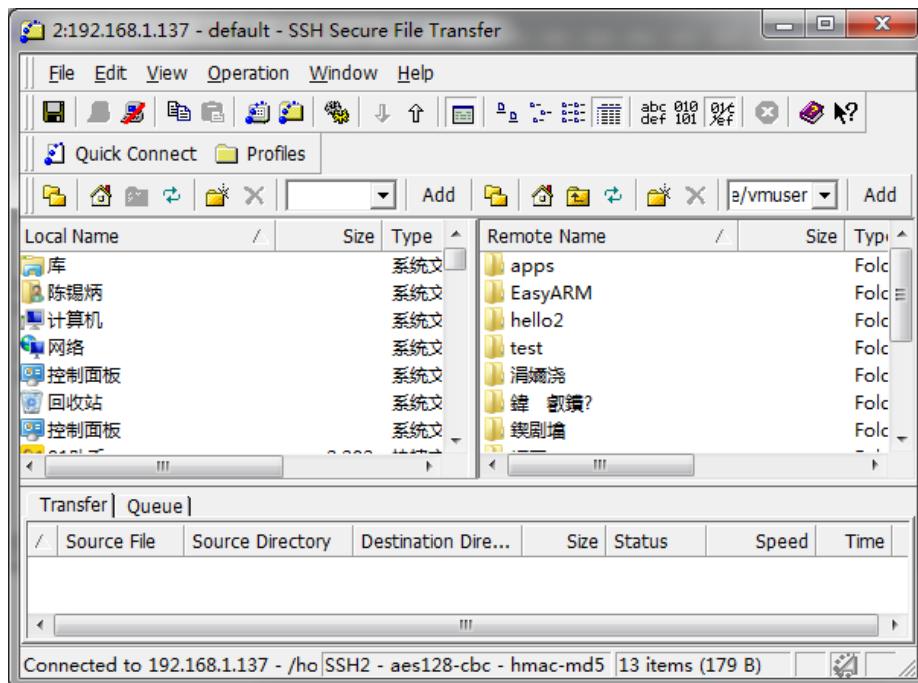


图 6.16 SSH 远程终端

在远程 Linux 主机，如果已经浏览到了一个目的目录，希望打开 Shell 操作该目录，可以选择“Window→New Terminal in Current Directory”，将会默认进入当前目录，如图 6.17 所示。

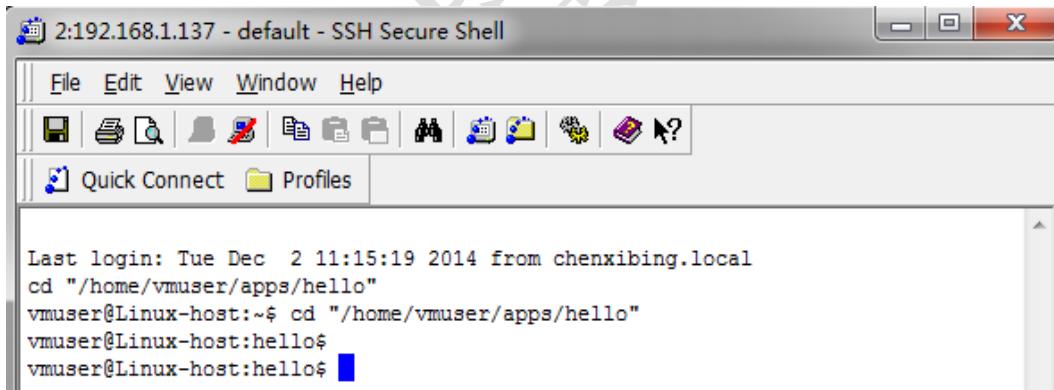


图 6.17 当前路径打开终端

6.4 NFS 服务器

6.4.1 NFS 能做什么？

在嵌入式 Linux 开发中，需要在 Linux 主机为目标机编写程序代码，然后编译程序，生成的程序是要传输到目标机上才能调试、运行。那么如何更快、更便捷地传输文件，将影响到开发工作的效率。NFS 无疑是最好的选择。通过 NFS 服务，主机可以将自己系统中某个指定目录通过网络共享给目标机（和 Windows 的文件网络共享类似）。目标机可以直接运行存放于 Linux 主机共享目录下的程序。这样调试程序时十分方便。

NFS 即网络文件系统（Network File-System），可以通过网络让不同机器、不同系统之间可以实现文件共享。通过 NFS，可以访问远程共享目录，就像访问本地磁盘一样。NFS



只是一种文件系统，本身并没有传输功能，是基于 RPC（远程过程调用）协议实现的，采用 C/S 架构。接下来将介绍如何在 ubuntu 系统中开启 NFS 服务器功能，使得开发套件能共享 ubuntu 系统的指定目录。

6.4.2 安装 NFS 软件包

在 ubuntu 终端输入下面命令安装 NFS 服务器：

```
vmuser@Linux-host: ~$ sudo apt-get install nfs-kernel-server      #安装 NFS 服务器端  
vmuser@Linux-host: ~$ sudo apt-get install nfs-common           #安装 NFS 客户端
```

6.4.3 添加 NFS 共享目录

安装完 NFS 服务器等相关软件后，需要指定用于共享的 NFS 目录，其方法是在 “/etc/exports” 文件里面设置对应的目录及相应的访问权限，每一行对应一个设置。下面介绍如何添加 NFS 共享目录。

在终端输入 “sudo vi /etc/exports” 指令，如下所示：

```
vmuser@Linux-host:~$ sudo vi /etc/exports  
[sudo] password for vmuser:
```

“/etc/exports” 文件打开后，文件内容如程序清单 6.2 所示。

程序清单 6.2 /etc/exports 文件内容

```
# to NFS clients. See exports(5).  
  
# Example for NFSv2 and NFSv3:  
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)  
  
# Example for NFSv4:  
# /srv/nfs4      gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)  
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
```

若需要把 “/nfsroot” 目录设置为 NFS 共享目录，请在该文件末尾添加下面的一行：

```
/nfsroot    *(rw,sync,no_root_squash)
```

其中 “*” 表示允许任何网段 IP 的系统访问该 NFS 目录。添加完成后，文件内容如程序清单 6.3 所示。

程序清单 6.3 添加了 NFS 目录

```
#  
  
# Example for NFSv2 and NFSv3:  
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)  
  
# Example for NFSv4:  
# /srv/nfs4      gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)  
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)  
  
#  
/nfsroot    *(rw,sync,no_root_squash)
```

修改完成后，保存并退出 “/etc/exports” 文件。然后新建 “/nfsroot” 目录，并为该目录



设置最宽松的权限：

```
vmuser@Linux-host:~$ sudo mkdir /nfsroot  
vmuser@Linux-host:~$ sudo chmod -R 777 /nfsroot  
vmuser@Linux-host:~$ sudo chown -R nobody /nfsroot
```

为了方便测试 NFS 是否挂载成功，可以在“/nfsroot”目录下创建 NFS_Test 目录用于测试。

6.4.4 启动 NFS 服务

在终端中执行如下命令，可以启动 NFS 服务：

```
vmuser@Linux-host: ~$ sudo /etc/init.d/nfs-kernel-server start
```

执行如下命令则可以重新启动 NFS 服务，也可以通过重启 ubuntu 来实现：

```
vmuser@Linux-host: ~$ sudo /etc/init.d/nfs-kernel-server restart
```

执行启动命令后，其操作结果如图 6.18 所示，表示 NFS 服务已正常启动。

```
vmuser@Linux-host: ~$ sudo /etc/init.d/nfs-kernel-server start  
[sudo] password for vmuser:  
* Exporting directories for NFS kernel daemon...  
exportfs: /etc/exports [1]: Neither 'subtree_check' or 'no_subtree_check' specified for export "*:/nfsroot".  
Assuming default behaviour ('no_subtree_check').  
NOTE: this default has changed since nfs-utils version 1.0.x  
  
* Starting NFS kernel daemon [ OK ]  
[ OK ]  
vmuser@Linux-host:~$
```

图 6.18 启动 NFS 服务

在 NFS 服务已经启动的情况下，如果修改了“/etc/exports”文件，需要重启 NFS 服务，以刷新 NFS 的共享目录。

当然在下一次启动系统时，NFS 服务是自动启动的。

6.4.5 测试 NFS 服务器

NFS 服务启动后，可以在 Linux 主机上进行自测。自测的基本方法为：将已经设定好的 NFS 共享目录 mount（挂载）到另外一个目录下，看能否成功。

假定 Linux 主机 IP 为 192.168.12.123，其 NFS 共享目录为/nfsroot，可使用如下命令进行测试：

```
vmuser@Linux-host:~$ sudo mount -t nfs 192.168.12.123:/nfsroot /mnt -o noblock
```

如果指令运行没有出错，则 NFS 挂载成功，在主机的/mnt 目录下应该可以看到/nfsroot 目录下的内容（即之前创建的 NFS_Test 目录）。

此外，也可以使用开发套件进行挂载测试，此时需要在开发套件上输入如下指令：

```
root@EasyARM-iMX28x ~# mount -t nfs 192.168.12.123:/nfsroot /mnt -o noblock
```

若挂载成功，在开发套件的/mnt 目录下也可以看见 NFS_Test 目录。之后，开发套件就可以像操作本地目录一样去操作主机的/nfsroot 目录。

注意，要想成功地挂载 NFS 目录，开发套件**必须要先确保与主机之间的网路是畅通的**，可以使用 ping 命令进行测试：

```
root@EasyARM-iMX28x ~# ping 192.168.12.123
```



如果无法 ping 通主机，则需要先仔细检查网络连接与设定。此外，在 mount 与 umount（解除挂载）操作时，用户的当前路径不能是操作的目标路径。例如下面两条指令就是错误的，用户当前所处的路径与要 mount（或 umount）的目标路径相同：

```
root@EasyARM-iMX28x:/mnt# mount -t nfs 192.168.12.123:/nfsroot /mnt -o noblock      # 错误  
root@EasyARM-iMX28x:/mnt# umount /mnt                                         # 错误
```

6.5 TFTP 服务器

6.5.1 TFTP 能做什么？

TFTP (Trivial File Transfer Protocol, 简单文件传输协议)，是 TCP/IP 协议族中用来在客户机和服务器之间进行简单文件传输的协议，开销很小。

这时候有人可能会纳闷，既然前面已经介绍了功能强大的 SSH 和 NFS 服务，还有必要介绍 TFTP 吗？TFTP 尽管简单，但在很多地方还是不可替代的，正如俗话说的“尺有所短，寸有所长”。

TFTP 通常用于内核调试。在嵌入式 Linux 开发过程中，内核调试是其中一个基础、重要的环节。调试内核通常是与 Bootloader 配合使用，只需在嵌入式系统的 Bootloader 中实现网卡驱动和 TFTP 客户端，就可以使用 TFTP 服务从主机上下载内核。

主机要开启 TFTP 服务，必须要先安装 TFTP 服务器软件，可以在 Linux 下实现，也可以在 Windows 下实现。

6.5.2 安装配置 TFTP 软件

用户可以在主机系统联网的情况下，在终端输入下面命令进行安装：

```
vmuser@Linux-host:~$ sudo apt-get install tftpd-hpa tftp-hpa
```

软件安装成功后，终端显示如图 6.19 所示。

The screenshot shows a terminal window with the following output:

```
vmuser@Linux-host:~$ sudo apt-get install tftpd-hpa tftp-hpa
[sudo] password for vmuser:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
tftpd-hpa 已经是最新的版本了。
下列【新】软件包将被安装：
  tftp-hpa
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 318 个软件包未被升级。
需要下载 19.8 kB 的软件包。
解压缩后会消耗掉 77.8 kB 的额外空间。
您希望继续执行吗？[Y/n]y
0% [执行中]
获取: 1 http://cn.archive.ubuntu.com/ubuntu/ precise/main tftp-hpa amd64 5.2-1ubuntu1 [19.8 kB]
下载 19.8 kB, 耗时 0 秒 (26.1 kB/s)
Selecting previously unselected package tftp-hpa.
(正在读取数据库 ... 系统当前共安装有 143512 个文件和目录。)
正在解压缩 tftp-hpa (从 .../tftp-hpa_5.2-1ubuntu1_amd64.deb) ...
正在处理用于 man-db 的触发器...
正在设置 tftp-hpa (5.2-1ubuntu1) ...
vmuser@Linux-host:~$
```

图 6.19 安装 tftp 软件

6.5.3 配置 TFTP 服务器

TFTP 软件安装后，默认是关闭 TFTP 服务的，需要更改 TFTP 配置文件“/etc/default/tftp-hpa”，可通过终端输入如下命令进行修改：



```
vmuser@Linux-host: ~$ sudo vi /etc/default/tftpd-hpa
```

用户需要指定一个目录为 TFTP 根目录。若用户需要把/tftpboot 目录设置为 TFTP 根目录，请在/etc/default/tftp-hpa 文件中的“TFTP_DIRECTORY”变量指定，如下所示：

```
TFTP_USERNAME="tftp"  
TFTP_DIRECTORY="/tftpboot"  
TFTP_ADDRESS="0.0.0.0:69"  
TFTP_OPTIONS="-l -c -s"
```

如果用户的 Linux 系统下尚未创建/tftpboot 目录，需要创建该目录，并需要使用 chmod 命令为该目录设置最宽松的权限。目录创建及权限设置命令如下所示：

```
vmuser@Linux-host: ~$ sudo mkdir /tftpboot
```

```
[sudo] password for vmuser:
```

```
vmuser@Linux-host: ~$ sudo chmod -R 777 /tftpboot
```

```
vmuser@Linux-host: ~$ sudo chown -R nobody /tftpboot
```

说明：在 Windows 下，通过 tftpd32.exe（下载地址：<http://tftpd32.jounin.net>）可以很便捷的实现一个 TFTP 服务器，只需将 tftpd32.exe 放在某个文件夹下并运行即可。

6.5.4 启动 TFTP 服务

TFTP 服务器安装配置完成后，启动 TFTP 服务的终端命令如下：

```
vmuser@Linux-host:~$ sudo service tftpd-hpa start
```

```
tftpd-hpa start/running, process 2389
```

当然直接重启系统也可以启动 TFTP 服务。

6.5.5 测试 TFTP 服务器

在 TFTP 服务器目录/tftpboot 下创建一个测试文件 tftpTestFile：

```
vmuser@Linux-host: ~$ echo "Hello,can you see me?" > /tftpboot/tftpTestFile
```

测试文件准备好了之后，打开终端，输入以下测试命令（**在 Linux 系统中 localhost 表示本地主机**）：

```
vmuser@Linux-host: ~$ tftp localhost  
tftp> get tftpTestFile          # 如果测试失败会打印出错信息  
tftp> q  
vmuser@Linux-host: ~$ cat /tftpboot/tftpTestFile  
Hello,can you see me?          # 文件内容正确，表示 TFTP 服务器配置成功
```

至此，TFTP 服务器已经配置并测试成功，若用户操作结果与上述现象不同，则需要检查相关操作步骤是否按照文档步骤操作。



第二篇 EasyARM-i.MX28xA 开发平台

这一篇内容围绕 EasyARM-i.MX28xA 开发套件展开，首先介绍了开发套件的软硬件资源，然后通过一章的篇幅对该平台的基本操作进行描述，包括硬件连接、上电开机，以及通过串口输入基本命令进行系统操作和设置，也对一些进阶操作进行了展开；最后讲述系统固件恢复的烧写以及固件升级操作方法。

一共分为 3 章，各章的标题和大概内容如下：

第 7 章：开发套件介绍，介绍软硬件资源；

第 8 章：入门实操，介绍上电开机后的基本操作和系统设置，以及一些进阶操作；

第 9 章：系统固件烧写，讲述如何恢复系统以及升级固件。

注：EasyARM-i.MX28xA 泛指 EasyARM-i.MX280A、EasyARM-i.MX283A 及 EasyARM-i.MX287A。



第7章 开发套件介绍

本章主要对 EasyARM-i.MX28xA 开发套件进行介绍，包括硬件和软件资源。对这章内容，只需进行简单了解即可。

说明：EasyARM-i.MX28xA 泛指 EasyARM-i.MX280A、EasyARM-i.MX283A 及 EasyARM-i.MX287A。

7.1 开发套件简介

EasyARM-i.MX280A、EasyARM-i.MX283A 与 EasyARM-i.MX287A 是广州致远电子股份有限公司推出的“0 利润”开源免费硬件的 AWorks 系列产品中的其中 3 款，它们是集教学、竞赛与产品功能评估于一身的开发套件。

开发套件采用 Freescale 的 i.MX28x 系列处理器（**基于 ARM926EJ-S 内核**），具有丰富的硬件资源，提供了完善的 Linux 软件支持包、开发工具和丰富的实用范例，大大降低了 Linux 学习门槛和开发难度，可以帮助用户在短期内实现产品功能验证和开发。

如图 7.1 所示为 EasyARM-i.MX283(7)A 开发套件的主板外观及基本接口分布（图片仅供参考，具体以实物为准）。

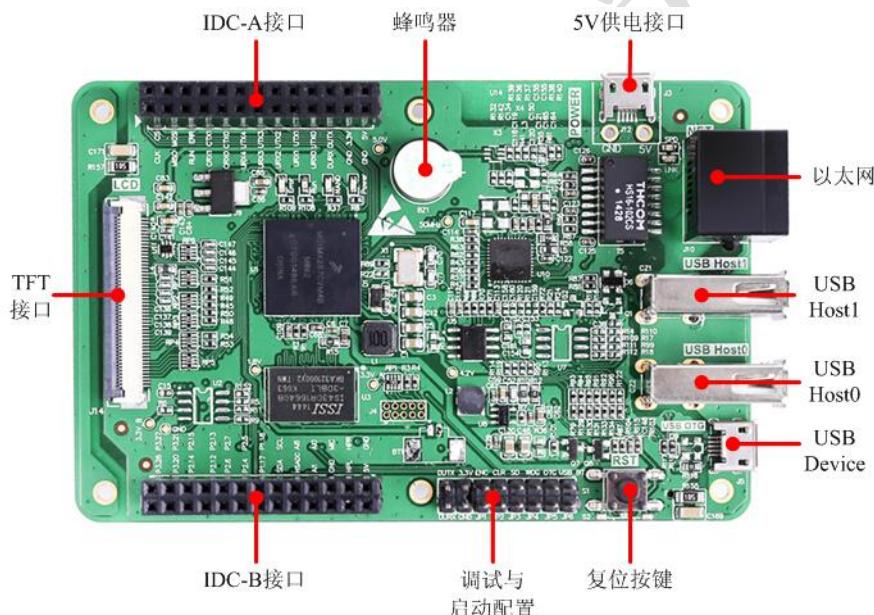


图 7.1 EasyARM-i.MX283(7)A 开发套件的基本接口分布

如图 7.2 所示为 EasyARM-i.MX280A 开发套件的主板外观及基本接口分布（图片仅供参考，具体以实物为准）。

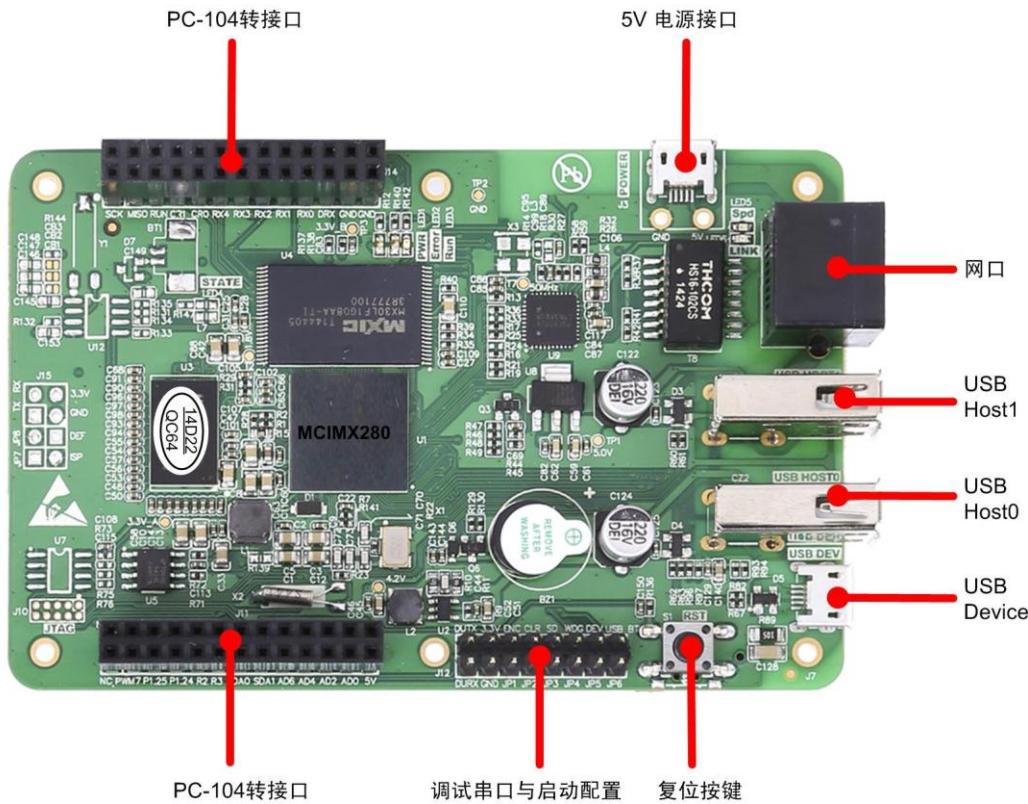


图 7.2 EasyARM-i.MX280A 开发套件的基本接口分布

7.2 硬件资源

EasyARM-i.MX280A、EasyARM-i.MX283A 与 EasyARM-i.MX287A 开发套件具体的资源如表 7.1 所示。

表 7.1 开发套件硬件资源

型号	EasyARM-i.MX280A	EasyARM-i.MX283A	EasyARM-i.MX287A
供电电源	5V	5V	5V
CPU	MCIMX280	MCIMX283	MCIMX287
内存 (DDR2)	64M	64M	128M
TFT	不支持	16bit TFT 液晶接口	16bit TFT 液晶接口
NAND FLASH	128M	128M	128M
USB Host	2 路	2 路	2 路
USB Device	1 路(与 USB Host0 复用)	1 路(与 USB Host0 复用)	1 路(与 USB Host0 复用)
TF 卡接口	1 路	1 路	1 路
CAN	不支持	不支持	2 路
百兆以太网	1 路	1 路	1 路
SPI	支持 SPI2 (复用功能, 默认不开启)	支持 SPI2 (复用功能, 默认不开启)	支持 SPI2/SPI3 (SPI2 为复用功能, 默认不开启)
串口	6 路 (1 路调试串口)	6 路 (1 路调试串口)	6 路 (1 路调试串口)
I ² C	2 路	1 路	1 路



续上表

型号	EasyARM-i.MX280A	EasyARM-i.MX283A	EasyARM-i.MX287A
ADC	8 路(含 1 路高速 ADC)	4 路(含 1 路高速 ADC)	4 路(含 1 路高速 ADC)
GPIO	3~24 路 (21 路默认为其他功能)	9~24 路 (15 路默认为其他功能)	14~37 路 (23 路默认为其他功能)
PCF8563	未焊接 (使用 I2C1 接口)	未焊接 (使用 I2C1 接口)	
WIFI	未焊接 (与 TF 卡二选一)	未焊接 (与 TF 卡、eMMC 三选一)	
eMMC	不支持	未焊接 (与 TF 卡、WIFI 三选一)	

7.3 软件资源

开发套件提供完善的 Linux BSP (EasyARM-i.MX280A、EasyARM-i.MX283A 与 EasyARM-i.MX287A 共用)，包括 Bootloader、Linux 内核源码、Qt 图形界面、开发示例及工具等，具体软件资源如

表 7.2 所列。

表 7.2 开发套件提供的 Linux 资源列表

资源	说明	
Bootloader	版本 U-Boot-2009.08	
Linux 内核	版本 Linux-2.6.35.3	
根文件系统	支持 sysfs、rootfs、bdev、ext3、ext2、ramfs、nfs、jffs2、ubifs、tmpfs 文件系统	
外设驱动	NAND Flash	驱动源码: drivers/mtd/nand/
	SD/MMC	驱动源码: drivers/mmc/host/mxs-mmc.c
	TFT LCD	驱动源码: drivers/video/mxs/lcd_43wvf1g.c、 drivers/video/mxs/mxsfb.c
	触摸屏	驱动源码: drivers/input/touchscreen/mxs-ts.c
	SPI	驱动源码: drivers/spi/spi_mxs.c
	I2C	驱动源码: drivers/i2c/busses/i2c-mxs.c
外设驱动	DUART	驱动源码: drivers/serial/mxs-duart.c
	AUART	驱动源码: drivers/serial/mxs-auart.c
	WDT	驱动源码: drivers/watchdog/mxs-wdt.c
	LRADC	驱动源码: drivers/misc lradc.c
	PWM	驱动源码: drivers/video/backlight/eays283-pwm.c
	GPIO	驱动源码: drivers/gpio/gpiolib.c
	RTC	驱动源码: drivers/rtc/rtc-mxs.c
	蜂鸣器驱动	驱动源码: drivers/leds/leds-mxs.c
图形界面	基于 Qt-4.7.3 的 zy launcher	
外设范例程序	基本的开发范例如以太网，串口、Qt 编程等	
开发所需的基本工具	USB, TF 卡烧写工具，交叉编译工具链 (gcc-4.4.4-glibc-2.11.1-multilib-1.0)，串口工具 (TeraTerm.)，文件传输工具 (SSHSecureShellClient-3.2.9) 等	

7.4 开发所需配件

使用开发套件进行学习和开发，还需要一些配件（非标配，可自行购买），配件和说明如表 7.3 所列。

表 7.3 开发配件

配件名	用途	样例	备注
MicroUSB 线	采用 USB 方式供电 (默认供电方式)	A black MicroUSB cable with two ends, one with a standard MicroUSB connector and the other with a smaller variant.	USB 接口的电源必须有 5V, 1A 的供电能力。
网线	用于应用软件调试和 进行网络通讯	A grey Ethernet cable with RJ45 connectors at both ends.	开发套件和主机直连用交叉 线, 通过交换机用平行线。
RS232-TTL 模块	RS-232 电平转换	A green printed circuit board (PCB) with various components and a DB9 serial port connector.	推荐使用广州致远电子研发 的 TTL 电平转 RS-232 电平串 口模块。
PL2303HX USB 转串口 TTL 模块	USB 串口通信	A black USB-to-serial adapter with a USB port and four data pins.	若电脑没有串口, 需要采用 USB 转串口模块。
TF 卡	用于系统启动、恢复, 数据存储等实验	An 8GB microSD card with a black plastic case.	i.MX28x 系列芯片的 TF 卡驱 动能力较弱, 建议使用 Class4 的 TF 卡

7.5 产品组装

参考光盘资料中的《EasyARM-i.MX283(7)A 硬件使用手册》文档进行产品组装, 组装完后整体效果如图 7.3 所示。对于 EasyARM-i.MX280A, 组装完后整体效果则如图 7.4 所示。



图 7.3 EasyARM-i.MX283(7)A 整体安装效果图



图 7.4 EasyARM-i.MX280A 整体安装效果图



第8章 入门实操

本章讲述 EasyARM-i.MX28xA 开发套件的基本操作，包括硬件连接和开机、基本操作和系统设置，以及一些进阶操作。这一章都是一些操作描述，没有什么难点，但这些都是嵌入式 Linux 开发过程中常用操作，需要熟练掌握。由于设备出厂时的固件与光盘资料自带的固件存在一定的差异，为得到与本章相同的操作结果，请使用光盘资料的最新固件。

说明：EasyARM-i.MX28xA 泛指 EasyARM-i.MX280A、EasyARM-i.MX283A 及 EasyARM-i.MX287A。本章以 EasyARM-i.MX283(7)A 为例进行描述，其相关操作同样适用于 EasyARM-i.MX280A 开发套件。

8.1 开机和登录

登录用户名和密码都是 root。

8.1.1 启动方式设置

开发套件支持 NAND Flash、TF 卡、USB 三种启动方式，可通过跳线进行设置，跳线位置如图 8.1 所示。

注意：EasyARM-i.MX280A 开发套件对应为 JP5 (DEV)。

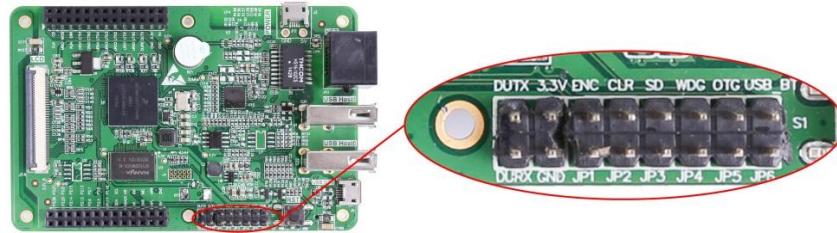


图 8.1 启动方式跳线

使用短路器对跳线的设置有：断开和短接，如图 8.2 所示。



图 8.2 短路器使用示意图

开发套件具体启动方式设置详见表 8.1。

表 8.1 启动方式设置

启动方式	JP3 (SD) 跳线	JP4 (WDG) 跳线	JP6 (USB) 跳线
NAND Flash	断开	短接	断开
TF 卡	短接	短接	断开
USB	断开	短接	短接



注意：JP4 (WDG) 为片外硬件看门狗禁能控制跳线，出厂演示的 Linux 系统未实现片外看门狗控制，故需短接 JP4，以禁止硬件看门狗功能，否则系统将不断重启，此时系统的蜂鸣器会重复鸣叫，默认 NAND 启动时，仅短接 JP4 (WDG)，其他全部断开。

8.1.2 供电连接

开发套件采用 MicroUSB 接口方式供电，用户需自备 MicroUSB 线缆，如图 8.3 所示。



图 8.3 MicroUSB 线

将 MicroUSB 线缆两端分别接到评估套件的 POWER 端口和 USB 接口的电源适配器上（推荐使用 5V@1A 或更大电流的电源）。

注意：电脑 USB 供电仅能提供 500mA 以内的电流，如果开发套件接了大的显示屏或者 USB 接口挂载了大功率的 USB 外设，请不要使用电脑的 USB 接口供电，而要采用独立的电源供电。

设置开发套件为 NAND Flash 启动方式，然后上电。

8.1.3 串口硬件连接

1. 目标机的调试串口

开发套件的调试串口是 TTL 电平，通过针脚引出，如图 8.4 所示。

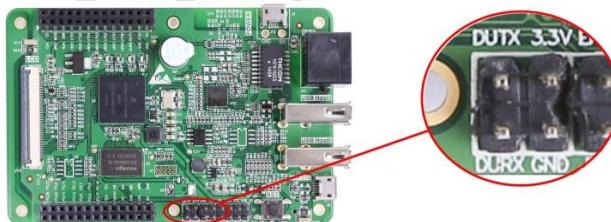


图 8.4 TTL 调试串口

计算机的标准串口通常是 RS-232 电平，因此在用串口连接开发套件的调试串口时，需要将其转换成 RS-232 电平，可用 RS-232 转 TTL 模块实现。图 8.5 所示是一个 RS-232 转 TTL 电平的模块。



图 8.5 RS232-TTL 模块

使用导线（**如杜邦线**），分别连接开发套件和 RS232-TTL 的针脚：3.3V—VCC、GND—GND、DUTX—TXD、DURX—RXD，如图 8.6 所示。

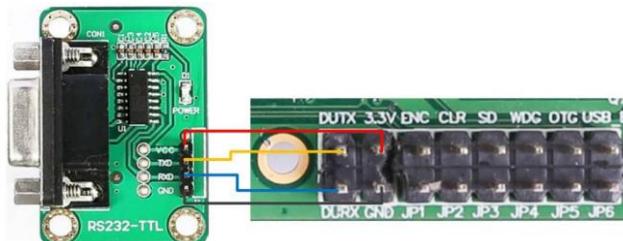


图 8.6 调试串口和 RS232-TTL 模块的连接方法

2. 目标机的 RS-232 调试串口和主机连接

若主机配备有标准串口，可以使用串口延长线连接目标机，如图 8.7 所示。

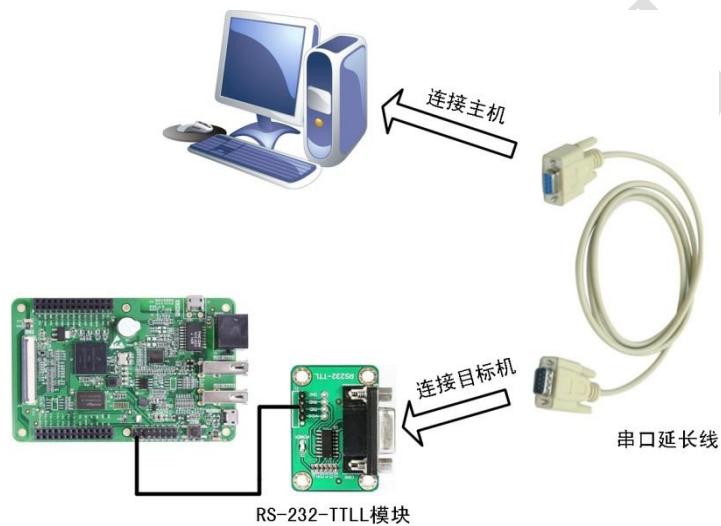


图 8.7 使用串口延长线连接

若主机没有配备标准串口，可以使用 USB 转 RS-232 串口模块来扩展串口，连接方式如图 8.8 所示。

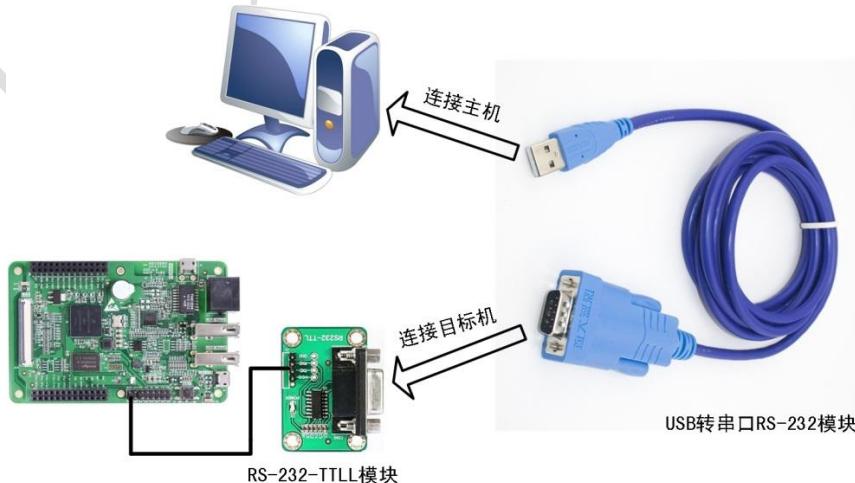


图 8.8 使用 USB 转 RS-232 串口模块连接



在 Windows 主机使用 USB 转 RS-232 串口模块需要安装厂商提供的驱动程序。在 Linux 主机使用 USB 转 RS-232 串口模块基本不需要安装驱动程序（系统已经自带驱动）。

3. 目标机的 TTL 调试串口和主机连接

开发套件的调试串口接口是 TTL 电平的，如果不想使用前面介绍的方式连接串口，可以使用 USB 转 TTL 串口模块进行连接，连接方法如图 8.9 所示。

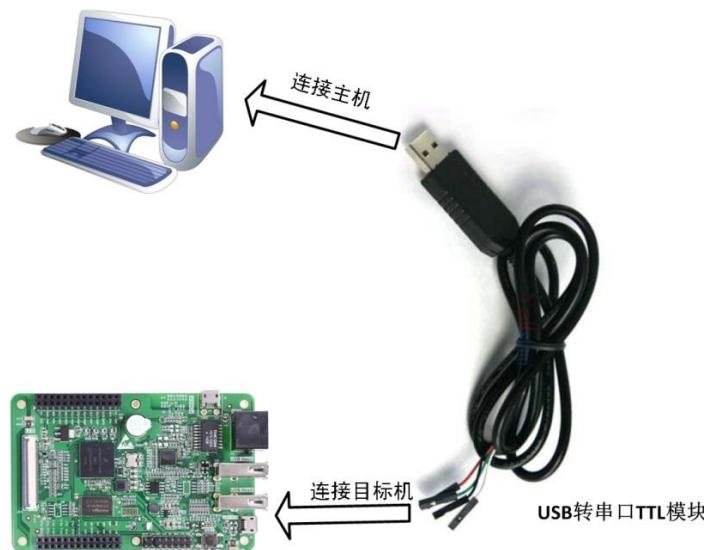


图 8.9 使用 USB 转 TTL 串口模块连接

USB 转串口 TTL 模块的产品很多，这里仅以 PL2303HX USB 转串口 TTL 模块为例进行介绍，PL2303HX USB 转串口 TTL 模块的实物如图 8.10 所示。



图 8.10 PL2303HX USB 转串口 TTL 模块

PL2303HX USB 转串口 TTL 模块的杜邦线接口定义：红色为+5V、黑色为 GND、白色为 RXD、绿色为 TXD，与开发套件调试串口的连接方法如图 8.11。

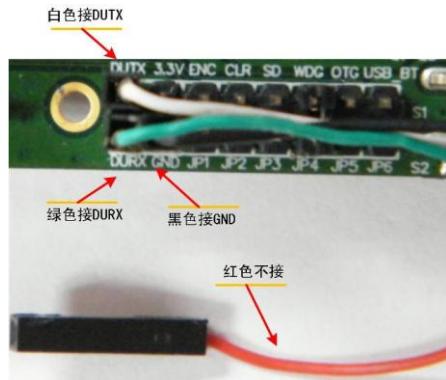


图 8.11 PL2303HX USB 转串口 TTL 模块与目标机的连接方法

在 Windows 主机使用 USB 转 TTL 串口模块需要安装厂商提供的驱动程序。在 Linux 主机使用 USB 转 TTL 串口模块基本不需要安装驱动程序。

8.1.4 Windows 环境串口登录

Windows 下的串口终端软件比较多，如：Tera Term、putty、Xshell 等。这里仅介绍 Tera Term（V4.67 版本）的用法。

1. 安装 Tera Term

产品光盘上附有 Tera Term 的安装程序文件 teraterm-4.67.exe。双击 teraterm-4.67.exe 文件开始安装，安装操作比较简单，这里就不需多述。

说明：Tera Term 安装完成后，安装程序会提示继续安装 LogMeTT 软件和 TTLEditor 软件，可以选择安装或不安装。

2. 打开 Tera Term 软件

Tera Term 安装完成后，需要先打开 Windows 系统的设备管理器对串口进行设置。如图 8.12 示，在设备管理器中找到串行接口（一般为 COM1），然后点击鼠标右键进入该接口的

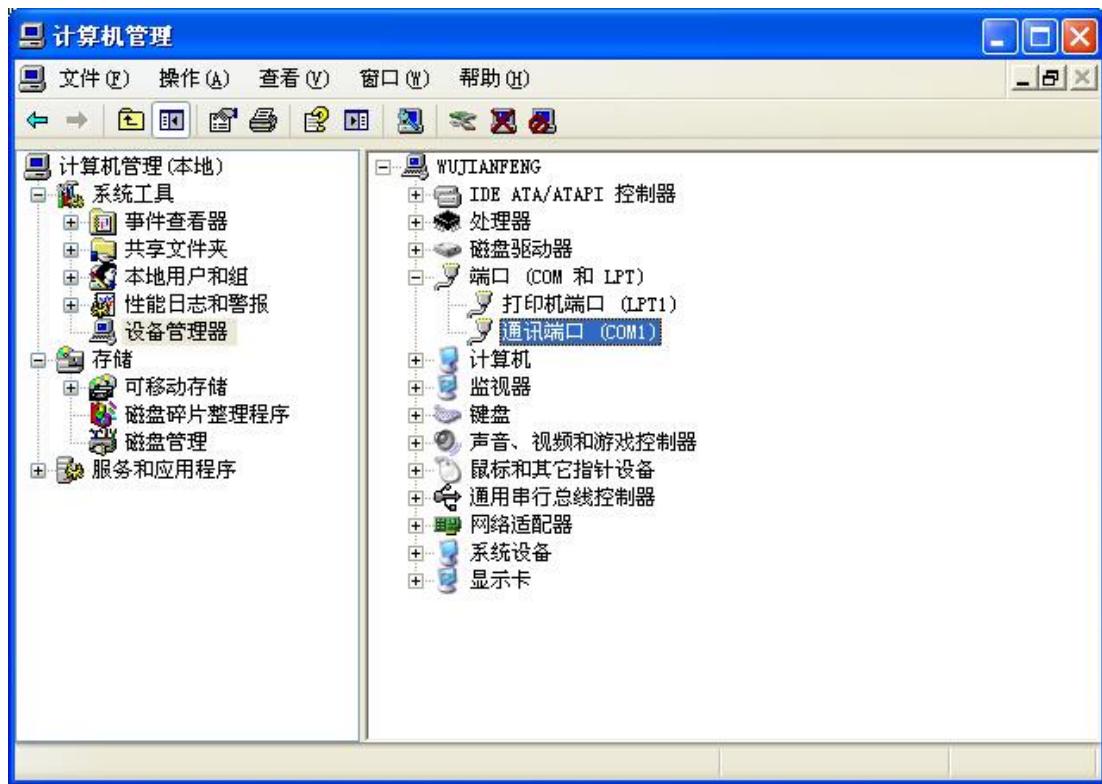


图 8.12 选择通讯端口

属性设置界面，将“端口设置”选项卡设定为如图 8.13 所示，确定后关闭设备管理器。



图 8.13 端口属性设置



然后在桌面双击  图标启动 Tera Term 软件，在弹出的新建连接窗口，选择 Serial 单选框，并在 Port 选择正确的串口号，如图 8.14 所示。



图 8.14 New connection 对话框

在 Windows 下，串口（包括用 USB 扩展的串口）是以端口的形式出现，端口的名称为 COM1、COM2、COM3 等，每个串口对应一个端口。

设置完成后，点击 OK 按钮关闭对话框，返回 Tera Term 如图 8.15 所示的主界面。



图 8.15 Tera Term 主界面

3. 串口设置

在如图 8.15 所示的主界面，点击 Setup->Serial port 打开串口配置界面，并进行串口设置：115200-8n1，无流控，如图 8.16 所示。

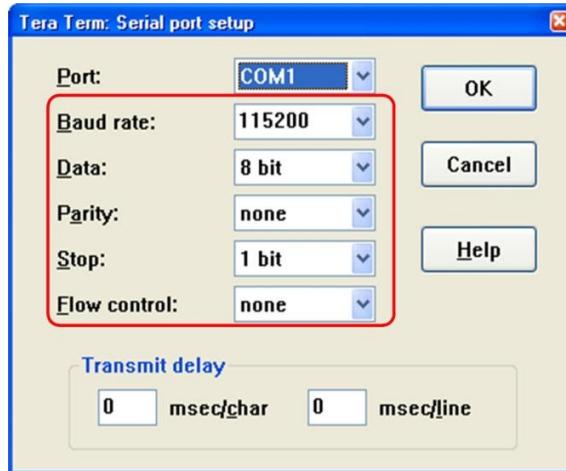


图 8.16 串口配置界面

确认无误后点击 OK 按钮完成设置，返回 Tera Term 主界面。

注意：在主菜单的 Setup->Save setup 可以保存设置。

4. 使用 Tera Term 登录

设置正确的串口软硬件连接，给开发套件上电。在上电的一瞬间，蜂鸣器会“哔”一声响，表示开发套件上已经进入启动程序，同时 Tera Term 会打印系统的启动信息，启动完成后，需要先通过终端发送“回车”键，然后串口终端将出现如图 8.17 所示的登录提示信息。

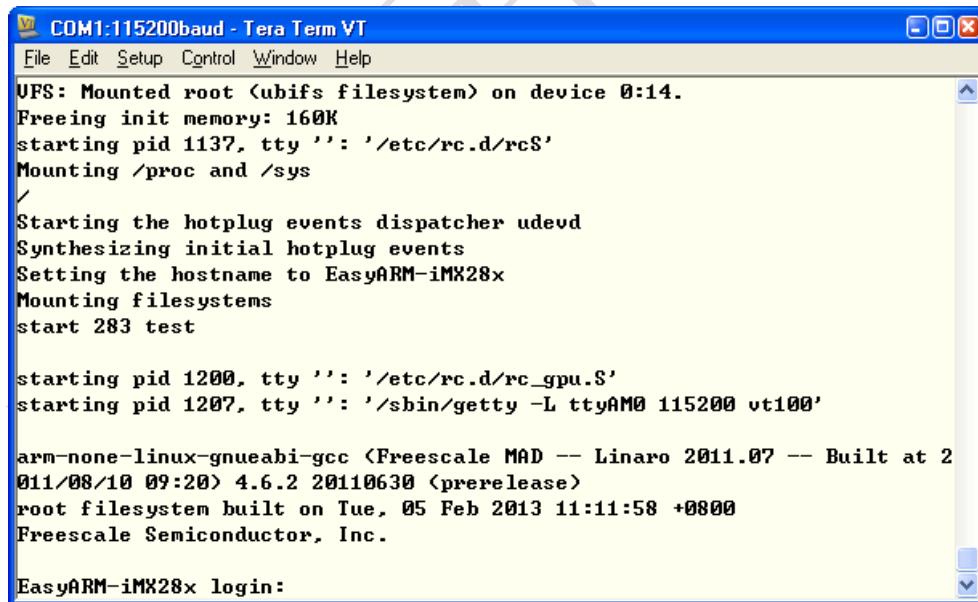


图 8.17 准备登录

用户名和密码都是 root，输入用户名和密码完成登录，如图 8.18 所示。



The screenshot shows a terminal window titled "COM1:115200baud - Tera Term VT". The window has a menu bar with File, Edit, Setup, Control, Window, Help. The main area displays the following text:

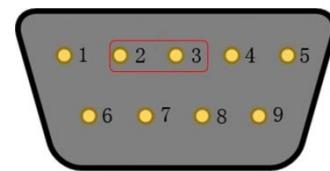
```
arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at  
2011/08/10 09:20) 4.6.2 20110630 (prerelease)  
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800  
Freescale Semiconductor, Inc.  
  
EasyARM-iMX28x login: root  
Password:  
login[1210]: root login on 'ttyAM0'  
  
BusyBox v1.20.2 () built-in shell (ash)  
Enter 'help' for a list of built-in commands.  
  
this board is EasyARM-iMX283  
root@EasyARM-iMX28x ~#
```

图 8.18 登录完成

注意：命令行交互必须以“回车”字符（0x0D）结尾，若用户输入完命令行字符后未按下“Enter”键（即未发送“回车”字符），开发套件是不会响应该命令行指令的。所以在命令行交互时不能使用“串口调试助手软件”来替代“串口终端软件”，因为大部分串口调试助手软件在字符输入或显示时都无法处理“回车”、“退格”等不可打印字符。

出于密码安全的考虑，用户在输入密码时，串口终端是不会回显输入字符的。

若串口终端没有启动信息输出，请检查 Tera Term 选择的端口是否正确和串口是否可用：把串口的 2-3 引脚短接（见右图），然后在 Tera Term 输入任意字符，看输入的字符能否正确回显，如果能正确回显示表示选择的端口正确并且串口正常。



如果输出字符是乱码，则请确认波特率等设置是否正确，确认设置无误后可以清除显示内容后重新操作。

如果液晶硬件组装连接无误，当 EasyARM-i.MX283(7)A 启动完成后，在液晶上将显示图 8.19 所示的 zylauncher 界面。



图 8.19 zylauncher 界面



8.1.5 Linux 环境串口登录

若 Linux 是安装在虚拟机上，建议使用 Windows 的串口终端。只有 Linux 是安装在实体电脑上，才必须使用 Linux 的串口终端。

在 Linux 常用的串口终端软件有 minicom 和 kermit，这里仅介绍 minicom 的使用方法。

1. Linux 的串口接口

在 Linux 串口是以设备文件的方式出现，设备文件在 /dev/ 目录下。若串口是主机配备的，那么串口设备文件为 ttyS0、ttyS1、ttyS2……；若串口是用 USB 扩展出来的，那么串口设备文件为 ttyUSB0、ttyUSB1、ttyUSB2……

使用下面命令可以查看主机配备串口的设备文件名：

```
vmuser@Linux-host:~$ dmesg | grep ttyS
[    0.770047] 00:08: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
[    0.790417] 00:09: ttyS1 at I/O 0x2f8 (irq = 3) is a 16550A
```

使用下面命令可以查看主机的 USB 扩展串口的设备文件名：

```
vmuser@Linux-host:~$ dmesg | grep ttyUSB
[   10.912236] usb 4-2: pl2303 converter now attached to ttyUSB0
```

2. 安装和配置 minicom

输入下面命令安装 minicom：

```
vmuser@Linux-host:~$ sudo apt-get install minicom
```

minicom 安装完成后，需要经过配置才能使用。在终端输入下面命令进入 minicom 的配置界面：

```
vmuser@Linux-host:~$ sudo minicom -s
```

注：加上 “-s” 选项表示进入 minicom 的配置界面。

进入 minicom 的配置界面

命令输入完成后，进入 minicom 的配置界面。这是基于字符界面的菜单，如图 8.20 所示。

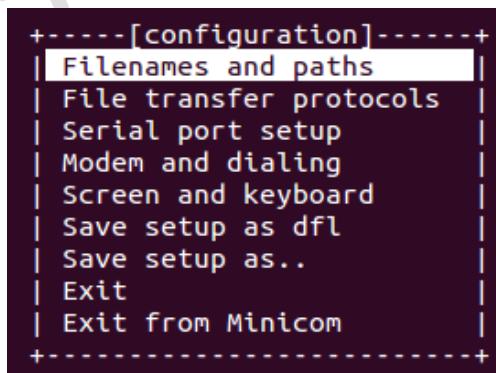


图 8.20 minicom 配置的主菜单

在菜单上，使用键盘的“↑”、“↓”方向键移动亮条到某个菜单项，表示选中该菜单项。选中指定的菜单项后，按“Enter”键进入子菜单或执行菜单项的操作。

进入串口端口配置菜单

选中“Serial prot setup”菜单项，如图 8.21 所示。

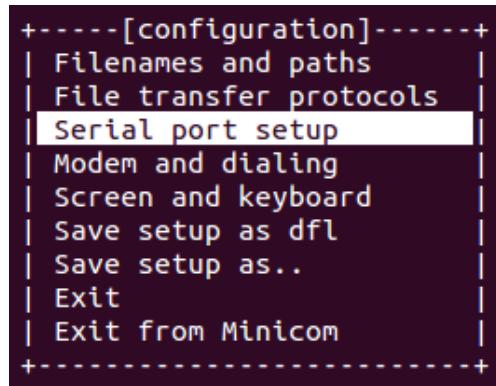


图 8.21 选中 Serial port setup 菜单项

然后按“Enter”键进入串口端口的配置菜单，如图 8.22 所示。

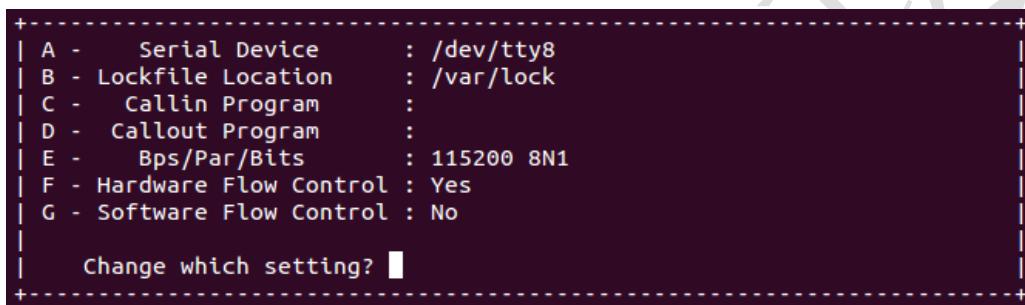


图 8.22 串口端口配置菜单

在该菜单可以看到有 A~G 的菜单项，通过键入字母（不分大小写）进入相应的菜单项。

设置串口设备文件名

键入“A”（或“a”）进入 Serial Device 菜单项，设置串口的设备文件名，如图 8.23 所示。

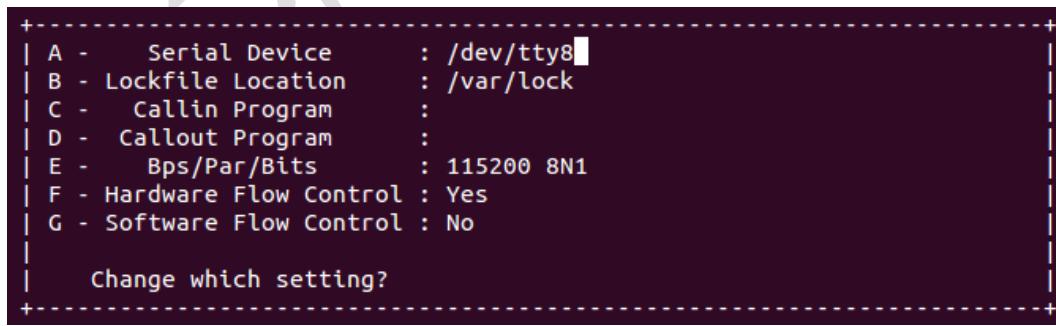


图 8.23 设置串口的设备文件名

把默认的串口设备文件名，改成与目标机连接的串口设备文件名，然后按“Enter”键确定并返回串口端口配置菜单，如图 8.24 所示。具体的串口设备文件名根据自己的实际情况而定。



```
+-----[Serial Port Configuration]-----+
| A - Serial Device      : /dev/ttyS1
| 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? ■
+-----
```

图 8.24 更改串口设备文件名

设置串口属性

串口端口配置菜单中的 E 菜单项(Bps/Par/Bits)是串口属性设置项，需要设置为“115200 8N1”(115200 波特率、8 位数据位、无奇偶校验、1 位停止位)。若该项默认不是“115200 8N1”，则需要设置。键入“E”(或“e”)进入串口属性设置界面，如图 8.25 所示。

```
+-----[Comm Parameters]-----+
|
| Current: 115200 8N1
| Speed          Parity       Data
| A: <next>      L: None      S: 5
| B: <prev>       M: Even      T: 6
| C: 9600         N: Odd       U: 7
| D: 38400        O: Mark      V: 8
| E: 115200       P: Space
|
| Stopbits
| W: 1           Q: 8-N-1
| X: 2           R: 7-E-1
|
| choice, or <Enter> to exit? ■
+-----
```

图 8.25 串口属性设置界面

在该界面，键入“C”~“E”(大小写不分)是选择串口波特率。这时请键入“E”(或“e”)选择 115200。然后再键入“Q”或“R”(大小写不分)设置串口的其它属性。这时请键入“Q”(或“q”)选择 8-N-1。设置完成后，按“Enter”键确定并返回串口端口配置菜单，如图 8.26 所示。

```
+-----[Serial Port Configuration]-----+
| A - Serial Device      : /dev/ttyS1
| 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? ■
+-----
```

图 8.26 返回串口设置主菜单

设置硬/软件流控制

串口端口配置菜单中的 F 菜单项是硬件流控制，该菜单项只有 Yes 或 No 选项，键入“F”

(或“f”)切换 Yes 或 No 的设置。硬件流控制请选择 No。G 菜单项是软件流控制，用同样的方法设置为 No。设置完成后如图 8.27 所示。

```
+-----+  
| A - Serial Device : /dev/ttyS1  
| B - Lockfile Location : /var/lock  
| C - Callin Program :  
| D - Callout Program :  
| E - Bps/Par/Bits : 115200 8N1  
| F - Hardware Flow Control : No  
| G - Software Flow Control : No  
  
| Change which setting?  
+-----+
```

图 8.27 完成串口端口配置

保存设置

这时按“Enter”键返回 minicom 的配置主菜单。然后选中 Save setup as dfl 菜单项，图 8.28 所示。

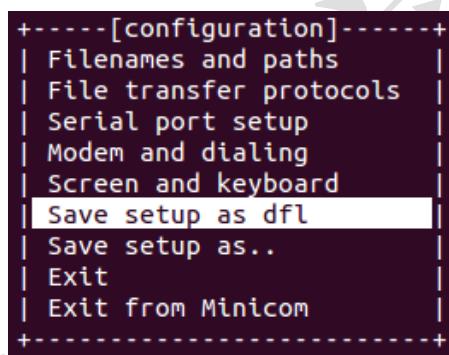


图 8.28 选择 Save setup as dfl

这时按“Enter”键保存刚才的设置。

退出 minicom 的配置

在 minicom 的主菜单，选中 Exit from Minicom 菜单项，如图 8.29 所示。

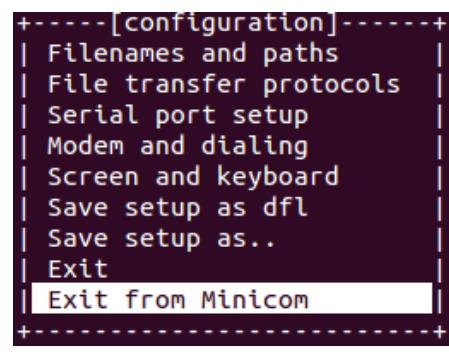


图 8.29 选中 Exit from Minicom 菜单项

这时按“Enter”键退出 minicom 的配置主菜单，返回 shell 终端。

串口连接完成后，输入下面命令进入 minicom 的串口终端界面：

```
vmuser@Linux-host:~$ sudo minicom
```



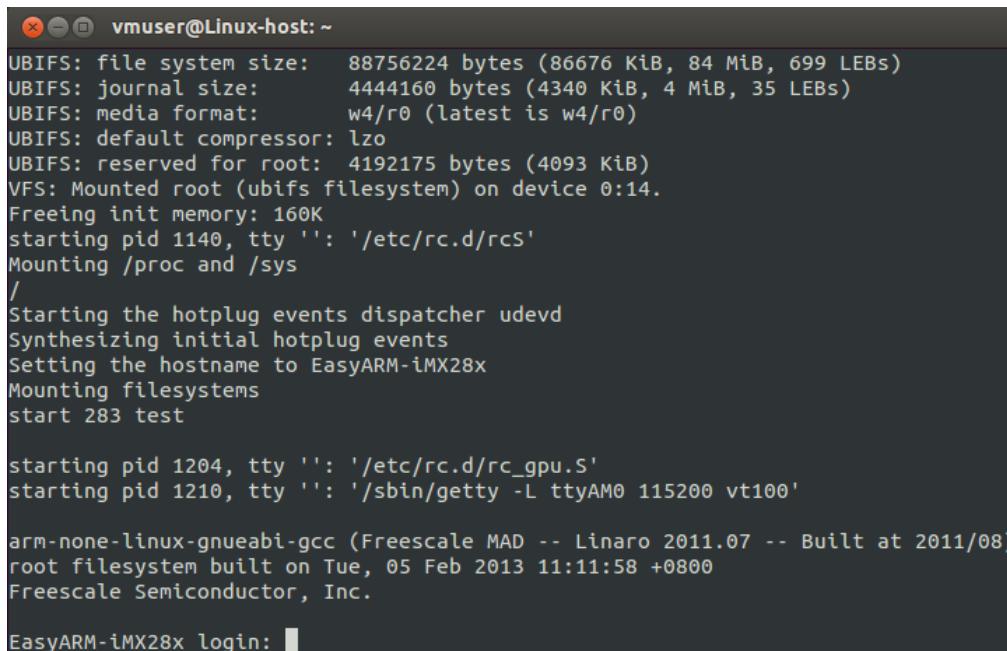
3. 使用 minicom 登录

输入下面命令进入 minicom 的串口终端界面：

```
vmuser@Linux-host:~$ sudo minicom -c on
```

注：加上“-c on”选项表示支持彩色字符显示。

给开发套件上电，minicom 将打印出系统的启动信息，启动完成后，同样需要先通过终端发送“回车”键，然后串口终端将出现如图 8.30 所示的登录提示信息。



```
vmuser@Linux-host: ~
UBIFS: file system size: 88756224 bytes (86676 KiB, 84 MiB, 699 LEBs)
UBIFS: journal size: 4444160 bytes (4340 KiB, 4 MiB, 35 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 4192175 bytes (4093 KiB)
VFS: Mounted root (ubifs filesystem) on device 0:14.
Freeing init memory: 160K
starting pid 1140, tty ''': '/etc/rc.d/rcS'
Mounting /proc and /sys
/
Starting the hotplug events dispatcher udevd
Synthesizing initial hotplug events
Setting the hostname to EasyARM-iMX28x
Mounting filesystems
start 283 test

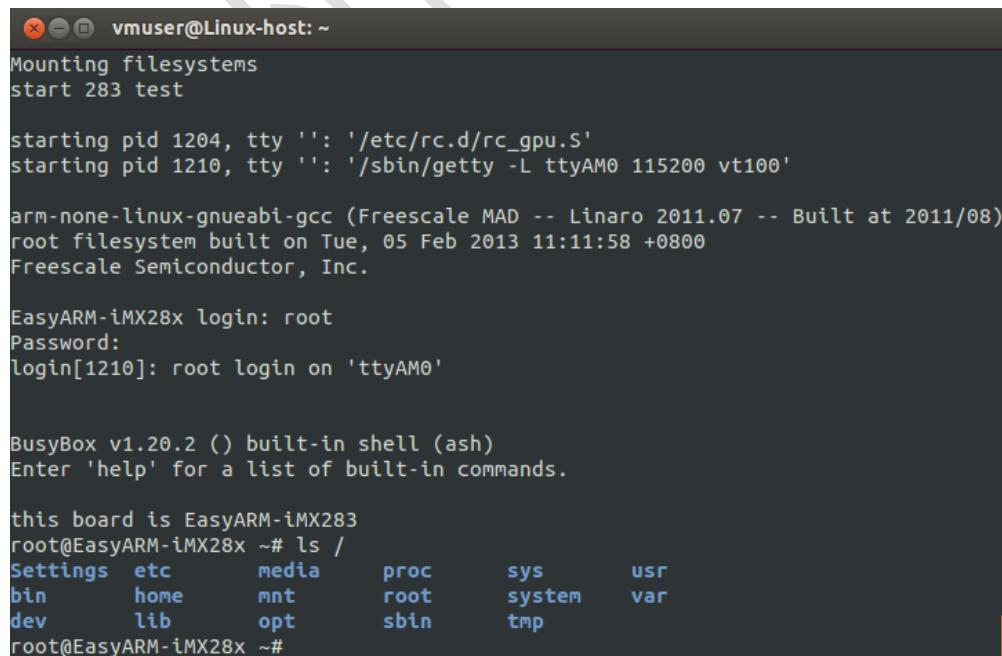
starting pid 1204, tty ''': '/etc/rc.d/rc_gpu.S'
starting pid 1210, tty ''': '/sbin/getty -L ttyAM0 115200 vt100'

arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at 2011/08)
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800
Freescale Semiconductor, Inc.

EasyARM-iMX28x login: ■
```

图 8.30 准备登录

用户名和密码都是 root，输入用户名和密码完成登录，如图 8.31 所示。



```
vmuser@Linux-host: ~
Mounting filesystems
start 283 test

starting pid 1204, tty ''': '/etc/rc.d/rc_gpu.S'
starting pid 1210, tty ''': '/sbin/getty -L ttyAM0 115200 vt100'

arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at 2011/08)
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800
Freescale Semiconductor, Inc.

EasyARM-iMX28x login: root
Password:
login[1210]: root login on 'ttyAM0'

BusyBox v1.20.2 () built-in shell (ash)
Enter 'help' for a list of built-in commands.

this board is EasyARM-iMX283
root@EasyARM-iMX28x ~# ls /
Settings  etc      media     proc      sys      usr
bin        home     mnt       root      system   var
dev        lib       opt       sbin      tmp
root@EasyARM-iMX28x ~#
```

图 8.31 登录完成



8.2 关机和重启

当用户需要关机或重启时，如果有数据存储操作，为了确保数据完全写入，可输入 sync 命令，完成数据同步后关闭电源关机或按 RST 键重启。

用户也可输入 reboot 命令重启：

```
root@EasyARM-iMX28x ~# reboot
```

该命令会自动完成数据同步后重启系统。

8.3 查看系统信息

8.3.1 查看系统内核版本

通过查看/proc/version 文件，可以获得系统内核版本信息：

```
root@EasyARM-iMX28x ~# cat /proc/version
```

```
Linux version 2.6.35.3-571-gcca29a0 (vmuser@Linux-host:) (gcc version 4.4.4 (4.4.4_09.06.2010) ) #235  
PREEMPT Mon Sep 22 14:47:57 CST 2014
```

注：具体信息请以实际返回的为准。

8.3.2 查看内存使用情况

使用 free 命令可以查看内存的使用情况：

```
root@EasyARM-iMX28x ~# free  
total        used         free      shared  
Mem:    124588       34644      89944          0          0  
-/+ buffers:           34644      89944  
Swap:        0          0          0
```

范例信息仅供参考，请以实际产品为准，下同。

8.3.3 查看磁盘使用情况

使用 df -m 指令可以查看系统磁盘的使用情况：

```
root@EasyARM-iMX28x ~# df -m  
Filesystem      1M-blocks   Used   Available  Use%  Mounted on  
ubi0:rootfs        94     35       59  37%   /  
tmpfs             61     0       61  0%   /dev  
shm               61     0       61  0%   /dev/shm  
rwfs              1     0       0  73%  /mnt/rwfs  
rwfs              1     0       0  73%  /var  
tmpfs             16     0      16  0%   /tmp
```

8.3.4 查看CPU等信息

通过查看/proc/cpuinfo 文件，可以获得 CPU 等信息：

```
root@EasyARM-iMX28x ~# cat /proc/cpuinfo  
Processor : ARM926EJ-S rev 5 (v5l)  
BogoMIPS : 226.09  
Features : swp half thumb fastmult edsp java  
CPU implementer : 0x41  
CPU architecture : 5TEJ
```



```
CPU variant      : 0x0
CPU part        : 0x926
CPU revision    : 5

Hardware       : Freescale MX28EVK board
Revision        : 0000
Serial          : 0000000000000000
```

注：BogoMIPS 是处理器的运算能力，表示每秒处理指令的百万数。

8.4 设置开机自动启动

8.4.1 开机启动脚本

开发套件 Linux 系统中的 /etc/rc.d/init.d/start_userapp 文件为开机时自动执行的脚本，该脚本内容如程序清单 8.1 所示。需要开机自动执行的命令或应用程序，都可以在这个文件里添加。

程序清单 8.1 start_userapp 文件内容

```
#!/bin/sh
ifconfig eth0 hw ether 02:00:92:B3:C4:A8
ifconfig eth0 192.168.0.100
# ifconfig eth0 down

# something need to be done before sshd start,
# make sure you know what you are doing before
# delete them.
date -s "2015-09-01 00:00:00" &> /dev/null
echo -e "root\nroot\n" | passwd root &> /dev/null
/etc/rc.d/init.d/sshd start &

# you can add your app start_command here

# start qt command,you can delete it
export TSLIB_PLUGINDIR=/usr/lib/ts/
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_TSDEVICE=/dev/input/ts0
export TSLIB_CALIBFILE=/etc/pointercal
export QT_QWS_FONTDIR=/usr/lib/fonts
export QWS_MOUSE_PROTO=Tslib:/dev/input/ts0
```

8.4.2 添加开机自动执行命令

假定希望开机时系统能自动执行 /root/hello 程序，那么只需要在 start_userapp 文件中增加 hello 程序的绝对路径即可：

```
# you can add your app start_command here
/root/hello
```

注意，如果程序是一个阻塞程序（**运行后不会退出或返回的程序**），则可能会导致位于



其后的指令或程序无法执行。再者，若该程序始终占用串口终端，将会造成其他程序（比如 Shell）无法通过串口终端与用户交互。对于此类应用程序，可以在其后面添加“&”（注意：是“空格”+“&”符号）让其在后台运行，如下所示：

```
# you can add your app start_command here  
/root/hello &
```

注意：若因自启动的阻塞程序造成串口终端无法交互（如启动设置时漏了添加“&”），用户必须重新烧写文件系统（uboot 下通过 tftp 服务器进行更新，更新命令为 run uproofs）或重新安装目标板上的 Linux 系统（已包含文件系统）来解决。

8.4.3 禁止开机启动图形界面

开发套件在开机启动系统时，默认启动 zylauncher 演示程序。如果不希望开机启动 zylauncher 演示程序，可修改/etc/rc.d/init.d/start_userapp 文件，把启动 zylauncher 的命令注释掉（#表示注释）：

```
# /usr/share/zylauncher/start_zylauncher > /dev/null &  
注：不适用于 EasyARM-i.MX280A 开发套件。
```

8.5 加载驱动模块

8.5.1 在 shell 终端加载和使用驱动模块

加载驱动模块时需要使用 insmod 命令。以开发套件的 Linux 系统为例，加载 lradc.ko 驱动模块的命令为：

```
root@EasyARM-iMX28x ~# insmod lradc.ko
```

lradc.ko 驱动模块加载完成后，会在/dev/目录下生成相应的设备文件节点：

```
root@EasyARM-iMX28x ~# ls /dev/magic-adc  
/dev/magic-adc
```

此时可以使用 lradc_test 应用程序（可在“3、Linux\4、开发示例\2、功能部件\2、ADC”目录下找到）测试驱动模块是否正常工作。

```
root@EasyARM-iMX28x ~# ./lradc_test
```

测试程序运行结果是开发套件的终端会打印出当前的 ADC 值。

```
CH0:3.04 CH1:1.85 CH6:2.84 Vbat:4.17  
CH0:3.04 CH1:1.85 CH6:2.84 Vbat:4.15  
CH0:3.04 CH1:1.85 CH6:2.84 Vbat:4.16  
CH0:3.04 CH1:1.85 CH6:2.84 Vbat:4.17
```

8.5.2 在脚本文件加载和使用驱动模块

当驱动模块被加载后，udev 会在比较短的时间内为驱动模块生成设备文件节点。所以在 shell 终端下加载驱动模块后，再运行依赖该驱动模块的应用程序时，在绝大多数情况下是不会出问题。毕竟手工输入命令的速度，是远远比不上 udev 为驱动模块生成设备文件节点的速度。但是若在脚本文件加载驱动模块后，马上执行依赖该驱动模块的应用程序，就有可能出现应用程序运行时，由于 udev 还没完成生成设备文件节点，从而导致应用程序无法打开设备文件节点的错误。

解决办法是，在脚本文件加载驱动的命令执行后，立即运行 udevtrigger 命令生成设备文件，再执行使用该设备文件的应用程序。在/etc/rc.d/init.d/start_userapp 文件设置开机自动



执行 lradc_test 程序的方法如下：

```
# you can add your app start_command here  
insmod /root/lradc.ko  
udevtrigger  
/root/lradc_test
```

注意：实际操作过程中/root/lradc_test 程序会循环打印 ADC 值 50 次后才退出程序，在此期间终端输入功能不可用。

8.6 网络设置

1. ifconfig 命令

开发套件的以太网接口为 eth0。在 Linux 系统下，使用 ifconfig 命令可以显示或配置网络设备，其常用的组合命令格式如下：

```
#ifconfig 网络端口 IP 地址 hw<HW> ether MAC 地址 netmask 掩码地址 broadcast 广播地址 [up|down]
```

2. 设置 IP 地址

通过 ifconfig 命令可以查看或者设置网卡的 IP 地址。操作示例：

```
root@EasyARM-iMX28x ~# ifconfig eth0 192.168.28.236  
eth0: Freescale FEC PHY driver [Generic PHY] (mii_bus:phy_addr=0:05, irq=-1)      # 表示网卡启动  
PHY: 0:05 - Link is Up - 100/Full                                              # 表示网线连接
```

该命令把 eth0 网卡的 IP 地址设置为 192.168.28.236，同时启动网卡。

输入 ifconfig eth0 命令可以查看 eth0 网卡当前的状态：

```
root@EasyARM-iMX28x ~# ifconfig eth0  
eth0      Link encap:Ethernet  HWaddr 02:00:92:B3:C4:A8  
          inet addr:192.168.28.236  Bcast:192.168.28.255  Mask:255.255.255.0  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
          RX packets:1936 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:138010 (134.7 KiB)  TX bytes:0 (0.0 B)
```

3. 动态获得 IP 地址

如果希望使用动态 IP 地址，可用 udhcpc 命令。操作示例：

```
root@EasyARM-iMX28x ~# udhcpc  
udhcpc (v1.20.2) started  
eth0: Freescale FEC PHY driver [Generic PHY] (mii_bus:phy_addr=0:05, irq=-1)  
Sending discover...  
PHY: 0:05 - Link is Up - 100/Full  
Sending select for 192.168.138.103...  
Lease of 192.168.138.103 obtained, lease time 7200  
Deleting routers  
adding dns 192.168.138.1  
root@EasyARM-iMX28x ~# ifconfig eth0                                         # 查看设置效果  
eth0      Link encap:Ethernet  HWaddr 02:00:92:B3:C4:A8  
          inet addr:192.168.138.103  Bcast:255.255.255.255  Mask:255.255.255.0
```



```
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:25 errors:0 dropped:0 overruns:0 frame:0
TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:3957 (3.8 KiB) TX bytes:656 (656.0 B)
```

4. 修改 MAC 地址

开发套件支持修改网卡 MAC 地址，可用 ifconfig 命令修改。操作示例：

```
root@EasyARM-iMX28x ~# ifconfig eth0 hw ether 00:11:22:33:44:55
root@EasyARM-iMX28x ~# ifconfig eth0          # 查看设置效果
eth0      Link encap:Ethernet HWaddr 00:11:22:33:44:55
          inet addr:192.168.28.236 Bcast:192.168.28.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:3355 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:237724 (232.1 KiB) TX bytes:0 (0.0 B)
```

5. 设置子网掩码

设置子网掩码示例如下：

```
root@EasyARM-iMX28x ~# ifconfig eth0 netmask 255.255.255.0
root@EasyARM-iMX28x ~# ifconfig eth0          # 查看设置效果
eth0      Link encap:Ethernet HWaddr 00:11:22:33:44:55
          inet addr:192.168.28.236 Bcast:192.168.28.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:3920 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:279347 (272.7 KiB) TX bytes:0 (0.0 B)
```

6. 设置广播地址

设置广播地址的示例如下：

```
root@EasyARM-iMX28x ~# ifconfig eth0 broadcast 192.168.28.225
root@EasyARM-iMX28x ~# ifconfig eth0          # 查看设置效果
eth0      Link encap:Ethernet HWaddr 00:11:22:33:44:55
          inet addr:192.168.28.236 Bcast:192.168.28.225 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:5052 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:358016 (349.6 KiB) TX bytes:0 (0.0 B)
```

7. 关闭/启动网卡

使用下面命令关闭 eth0 网卡：

```
root@EasyARM-iMX28x ~# ifconfig eth0 down
```

在网卡关闭后，可以使用下面命令启动 eth0 网卡：



```
root@EasyARM-iMX28x ~# ifconfig eth0 up
eth0: Freescale FEC PHY driver [Generic PHY] (mii_bus:phy_addr=0:05, irq=-1)      # 表示网上启动
PHY: 0:05 - Link is Up - 100/Full                                         # 表示网线连接
```

8. 设置默认网关

如果开发套件连接到路由器或者交换机上，并且上层网络已经可以连接到互联网，那么设置好网关后开发套件应该可以连接到互联网上。若开发套件与计算机连接在同一网络环境下，网关参数可以参考计算机的设置。添加、删除或查看网关参数使用“route”命令，如需要将默认网关设置为 192.168.28.1，其示例指令如下：

```
root@EasyARM-iMX28x ~# route add default gw 192.168.28.1
```

若需要删除该网关设置，其示例指令如下：

```
root@EasyARM-iMX28x ~# route del default gw 192.168.28.1
```

若需要查看当前网关设置，其示例指令如下：

```
root@EasyARM-iMX28x ~# route -n
```

9. 设置 DNS

若开发套件需要使用域名访问互联网，则需要先设定 DNS，否则访问可能不正常。DNS 记录在/etc/resolv.conf 配置文件中。

打开“/etc/resolv.conf”文件后在其中添加 DNS 配置，可以添加多行，若首选 DNS 及备用 DNS 分别为 192.168.0.1 和 192.168.0.2，则其示例配置如下所示：

```
# nameserver ip address
nameserver 192.168.0.1
nameserver 192.168.0.2
```

文件修改后保存，DNS 的修改即时生效。

10. 开机自动设置网络参数

使用上述的 ifconfig、udhcpc、route 命令设置的网络参数，在断电或复位后丢失。若需要开机自动设置网络参数，则需要开机时自动执行设置网络参数的命令。

在/etc/rc.d/init.d/start_userapp 文件增加网络配置命令：

```
# you can add your app start_command here
ifconfig eth0 192.168.28.236
route add default gw 192.168.28.1

# start qt command,you can delete it
```

8.7 通过 SSH 登录系统

若开发套件设置了“开机自动设置 IP”并和主机建立了网络连接，开发套件的 Linux 系统启动完成后，用户可以通过 SSH 登录系统。

注意：

(1) 目标机和主机的 IP 地址必须在同一网段。在通过网络登录目标机的 Linux 系统前，最好先在主机 ping 一下目标机，看网络是否畅通。

(2) 通过网络登录系统后，请勿改变目标机的网络参数，否则可能会造成登录中断。



密码更新后即可通过 SSH 登录系统。账号：root，密码：root。

在 Windows 下可以使用 putty/xShell 登录开发板。在 Linux 主机可以通过 ssh 命令登录，命令如下：

```
vmuser@Linux-host:~$ ssh root@192.168.28.236 # 假设 192.168.28.236 为目标机的 IP 地址
```

在第一次试图登录目标机时，该命令会用询问用户：

```
Are you sure you want to continue connecting (yes/no)?
```

这时输入 yes 即可，然后输入登录密码：

```
vmuser@Linux-host:~$ ssh root@192.168.28.236
```

```
The authenticity of host '192.168.28.236 (192.168.28.236)' can't be established.
```

```
RSA key fingerprint is dc:37:5e:c2:32:ec:51:b0:cb:c3:81:7d:8d:2c:8e:a0.
```

```
Are you sure you want to continue connecting (yes/no)? yes
```

```
Warning: Permanently added '192.168.28.236' (RSA) to the list of known hosts.
```

```
root@192.168.28.236's password:
```

```
BusyBox v1.20.2 () built-in shell (ash)
```

```
Enter 'help' for a list of built-in commands.
```

```
this board is EasyARM-iMX283
```

```
root@EasyARM-iMX28x ~#
```

这时已经登录到目标机的 Linux 系统。

8.8 TF 卡使用

把 TF 卡插入开发套件的 TF 插槽后，Linux 系统将会自动检测到 TF 卡，并打印 LOG 信息：

```
mmc0: new high speed SD card at address b368
```

```
mmcblk0: mmc0:b368 MSD 1.85 GiB
```

```
mmcblk0: p1 p2 p3
```

开发套件的 Linux 系统会为 TF 卡的每个分区都在/media 目录生成一个目录，目录的名字为 sd-mmcbblk_n（n 表示不同的分区，n=0、1、2、3……），TF 卡的每个分区分别挂载在这些目录下。用户在这些目录下保存的文件，就是储存在 TF 卡的分区里。

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，例如：

```
root@EasyARM-iMX28x ~# df -m
```

Filesystem	1M-blocks	Used	Available	Use%	Mounted on
ubi0:rootfs	94	35	59	37%	/
tmpfs	61	0	61	0%	/dev
shm	61	0	61	0%	/dev/shm
rwf	1	0	0	73%	/mnt/rwf
rwf	1	0	0	73%	/var
tmpfs	16	0	16	0%	/tmp
/dev/mmcbblk0p1	1871	35	1741	2%	/media/sd-mmcbblk0p1

TF 卡在使用前需要格式化，开发套件支持 TF 卡使用 FAT、EXT2、EXT3 文件系统。



TF 卡在使用完成后，在弹出 TF 卡前，需要卸载 TF 所有分区的挂载：

```
vmuser@Linux-host: ~$ umount /media/sd-mmcbblk (n=0、1、2……)
```

注意在卸载 TF 卡分区挂载时，当前工作目录不得在分区挂载的目录下。

8.9 U 盘使用

开发套件有两个 USB Host 接口：USB Host1 和 USB Host0。USB Host0 接口需要使能时，要保持板上的 JP5（OTG）跳线断开。

把 U 盘插入 USB Host 接口后，Linux 会检测到 U 盘插入，并打印设备信息：

```
usb 2-1: new high speed USB device using fsl-ehci and address 2
scsi0 : usb-storage 2-1:1.0
scsi 0:0:0:0: Direct-Access      SanDisk   Cruzer           1.01 PQ: 0 ANSI: 2
sd 0:0:0:0: [sda] 7821312 512-byte logical blocks: (4.00 GB/3.72 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Assuming drive cache: write through
  sda: sda1
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Attached SCSI removable disk
EXT2-fs (sda1): warning: mounting unchecked fs, running e2fsck is recommended
```

开发套件的 Linux 系统会为 U 盘的每个分区都在/media 目录生成一个目录，目录的名字为 `usb-sdsn`（`s` 用于区分不同的 U 盘，`n` 用于区分不同的分区，`s=a、b、c…… n=0、1、2、3……`），U 盘的每个分区分别挂载在这些目录下。在这些目录下保存的文件，实际上就是储存在 U 盘的分区里。

输入 `df -m` 命令可以查看 U 盘分区的挂载的情况和分区的使用，例如：

```
root@EasyARM-iMX28x /media# df -m
Filesystem      1M-blocks    Used   Available  Use%  Mounted on
ubi0:rootfs        94       35       59   37%    /
tmpfs             61        0       61   0%     /dev
shm               61        0       61   0%     /dev/shm
rwfs              1         0        0   73%    /mnt/rwfs
rwfs              1         0        0   73%    /var
tmpfs             16        0       16   0%     /tmp
/dev/sda1        858        0      815   0%    /media/usb-sda1
/dev/sdb1        1871      35     1741   2%    /media/usb-sdb1
```

U 盘在使用前需要格式化，开发套件支持 U 盘使用 FAT、EXT2、EXT3 文件系统。

U 盘在使用完成后，在拔出 U 盘前，需要卸载 U 盘的所有分区的挂载：

```
vmuser@Linux-host: ~$ umount /media/usb-sdasn (n=0、1、2……)
```

注意在卸载 U 盘分区挂载时，当前工作目录不得在分区挂载的目录下。

8.10 USB Device 使用

开发套件的 JP5（OTG），跳线用短路器短接后，USB OTG 接口使能。USB OTG 接口支持 USB Device 功能，可以把开发套件虚拟成一个 U 盘。



注意：EasyARM-i.MX280A 开发套件对应为 JP5 (DEV)。

把开发套件虚拟成 U 盘需要加载板上的/root/g_file_storage.ko 驱动，加载该命令的格式为：

```
# insmod /root/g_file_storage.ko stall=0 file=块设备 removable=1
```

注意：块设备是指生成的块设备名字，比如：TF 卡的块设备名一般为/dev/mmcblk0p1。

在加载 g_file_storage.ko 驱动时，需要传入几个参数，这里只需关心 file 参数。file 参数表示把开发套件虚拟成 U 盘后，使用哪一个块设备储存这个虚拟 U 盘的数据。当这个虚拟 U 盘连接到电脑后，在电脑看到这个虚拟 U 盘的文件系统就是这个块设备的文件系统。若块设备还没有格式化，可以在电脑上格式化。

8.10.1 把 TF 卡作为虚拟 U 盘的储存空间

把 TF 卡格式化(假设格式化成 FAT 文件系统)，然后插入到开发套件卡槽。板载的 Linux 系统会检测到 TF 卡的插入，然后为 TF 卡的分区生成块设备，并自动把块设备挂载到指定的目录。

输入 df -m 命令查看 TF 卡的块设备和挂载的目录，如下示例：

```
root@EasyARM-iMX28x ~# df -m
Filesystem      1M-blocks   Used   Available   Use%   Mounted on
ubi0:rootfs        94       35       59    37%   /
tmpfs             61        0       61     0%   /dev
shm               61        0       61     0%   /dev/shm
rwfs              1         0        0    73%   /mnt/rwfs
rwfs              1         0        0    73%   /var
tmpfs             16        0       16     0%   /tmp
/dev/mmcblk0p1     1871      35     1741     2%   /media/sd-mmcblk0p1
```

在该例子中，TF 卡的块设备为/dev/mmcblk0p1，挂载到/media/sd-mmcblk0p1 目录下。

输入下面命令加载 g_file_storage.ko 驱动：

```
root@EasyARM-iMX28x ~# insmod /root/g_file_storage.ko stall=0 file=/dev/mmcblk0p1 removable=1
g_file_storage gadget: File-backed Storage Gadget, version: 20 November 2008
g_file_storage gadget: Number of LUNs=1
g_file_storage gadget-lun0: ro=0, file: /dev/mmcblk0p1
fsl-usb2-udc: bind to driver g_file_storage
```

命令执行完成后，使用 MicroUSB 线连接电脑（假设为 Windows 系统）和开发套件的 USB OTG 接口。这时 Windows 电脑的“我的电脑”下，将看到多了一个 U 盘驱动器，这就是开发套件虚拟出来的 U 盘。进入该 U 盘，新建一个 new.txt 的文件，然后在电脑卸载这个 U 盘。

这时在开发套件可以查看刚才新建的 new 目录：

```
root@EasyARM-iMX28x ~# ls /media/sd-mmcblk0p1/new.txt
/media/sd-mmcblk0p1/new.txt
```

使用类似的方法也可以把 U 盘作为虚拟 U 盘的储存空间。

8.10.2 使用普通文件作为虚拟 U 盘的储存空间

普通文件可以作为虚拟块设备使用，因此也可以用作虚拟 U 盘的储存空间。普通文件



可以储存在文件系统的任何位置。生成特定大小的普通文件可以用 dd 命令，其命令格式为：

```
dd if=文件 of=loop_file bs=size count=num
```

dd 命令的执行需要几个参数：

if 参数表示生成文件的数据是从哪个文件输入；

of 参数表示要生成的 loop 文件路径；

bs 参数表示生成文件每块大小；

count 参数表示生成文件有多少个块。

使用下面命令生成一个 10M 大小的普通文件：

```
root@EasyARM-iMX28x ~# dd if=/dev/zero of=/dev/shm/disk bs=1024 count=10240
10240+0 records in
10240+0 records out
10485760 bytes (10.0MB) copied, 0.329593 seconds, 30.3MB/s
```

生成的普通文件为 /dev/shm/disk，大小为 $1024 \times 10240 = 10\text{MB}$ 。

输入下面命令加载 g_file_storage.ko 驱动：

```
root@EasyARM-iMX28x ~# insmod /root/g_file_storage.ko stall=0 file=/dev/shm/disk removable=1
```

命令执行完成后，使用 MicroUSB 线连接电脑（假设为 Windows 系统）和开发套件的 USB OTG 接口。这时 Windows 电脑的“我的电脑”下，将看到多了一个 U 盘驱动器，这就是开发套件虚拟出来的 U 盘。由于普通文件还没有格式化，所以得到的虚拟 U 盘需要格式化，可以在 Windows 直接对虚拟 U 盘进行格式化。格式化完成后，进入该 U 盘，新建一个 new.txt 文件，然后卸载这个 U 盘。

这时在开发套件把普通文件挂载到 /mnt/ 目录：

```
root@EasyARM-iMX28x ~# mount /dev/shm/disk /mnt/
```

挂载完成后，进入 /mnt/ 目录即可看到刚才新建的 new.txt 文件：

```
root@EasyARM-iMX28x ~# cd /mnt/
root@EasyARM-iMX28x /mnt# ls
new.txt
```

8.11 LED 使用

在开发套件上有 NAND、RUN、ERR 三个 LED：

- NAND LED 是 NAND Flash 读/写指示灯，由硬件直接控制，用户不可控制，当程序访问 NAND Flash 时，该 LED 闪烁；
- RUN LED 是系统心跳灯（默认），不断按固定节奏闪烁表示系统正在运行；
- ERR LED 留给用户自由控制使用；
- RUN LED 和 ERR LED 的功能可以由用户设置。

8.11.1 LED 的操作接口

在开发套件的 /sys/class/leds 目录下有 led-err 和 led-run 两个目录，如下所示：

```
root@EasyARM-iMX28x ~# cd /sys/class/leds/
root@EasyARM-iMX28x /sys/class/leds# ls
beep      led-err    led-run
```

其中 led-err 目录是 ERR LED 的操作接口，led-run 目录是 RUN LED 操作接口。以 RUN



LED 为例，进入 led-run 目录，该目录的内容为：

```
root@EasyARM-iMX28x /sys/class/leds# cd /sys/class/leds/led-run/
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# ls
brightness      max_brightness    subsystem      uevent
device          power           trigger
```

其中 brightness 用于控制 LED 亮灭，trigger 用于设置 LED 的触发条件。

8.11.2 触发条件设置

trigger 文件用于查看和设置 LED 的触发条件。查看触发条件示例：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# cat trigger
none nand-disk mmc0 [heartbeat]
```

可以看到当前 LED 支持的触发条件有：none、nand-disk、mmc0、heartbeat，其中[heartbeat] 表示当前 LED 的触发条件为 heartbeat。

往 trigger 写入特定字符串可以设置 LED 触发条件，例如将 LED 触发条件设置为用户控制，可写入“none”，操作示例如下所示，请注意“>”号左右两个各有一个空格。

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo none > trigger
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# cat trigger
[none] nand-disk mmc0 heartbeat
```

1. 设置为用户控制

当 LED 的触发条件设置为 none 时，可以自由控制 LED 的点亮和熄灭。设置 LED 的触发条件为 none 的方法为：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo none > trigger
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# cat trigger
[none] nand-disk mmc0 heartbeat
```

这时可使用 brightness 文件控制 LED 的点亮和熄灭：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo 1 > brightness #控制 LED 点亮
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo 0 > brightness #控制 LED 熄灭
```

2. 设置为心跳指示

若用户需要把 LED 的触发条件设置为系统心跳指示，设置方法为：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo heartbeat > trigger
```

这时 LED 由系统时钟所控制。LED 按固定的节奏点亮和熄灭，表示系统正在运行。

3. 设置为 TF 卡检测

若需要把 LED 的触发条件设置为 TF 卡检测，设置方法为：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo mmc0 > trigger
```

这时把 TF 卡插入到 TF 卡槽时，LED 会闪烁一下。

4. 设置为 NAND Flash 读/写指示

若需要把 LED 的触发条件设置为 NAND Flash 读/写指示，设置方法为：

```
root@EasyARM-iMX28x /sys/devices/platform/mxs-leds.0/leds/led-run# echo nand-disk > trigger
```

这时对 NAND Flash 发生读/写操作时，LED 会发生闪烁。例如，在/home 目录保存一个文件：



```
root@EasyARM-iMX28x /# dd if=/dev/zero of=/home/disk bs=1024 count=10240
10240+0 records in
10240+0 records out
10485760 bytes (10.0MB) copied, 0.345239 seconds, 29.0MB/s
root@EasyARM-iMX28x /# sync
```

当上述命令运行时，可以发现 LED 在闪烁。

8.12 蜂鸣器使用

为方便使用蜂鸣器，开发套件的蜂鸣器提供了类似于 LED 的操作接口，对应的操作文件是/sys/class/leds/beep/brightness。写入 1 使蜂鸣器鸣叫，写入 0 停止鸣叫。

操作示例：

```
root@EasyARM-iMX28x ~# echo 1 > /sys/class/leds/beep/brightness      # 控制蜂鸣器鸣叫
root@EasyARM-iMX28x ~# echo 0 > /sys/class/leds/beep/brightness      # 控制蜂鸣器停止鸣叫
```

说明：EasyARM-i.MX28xA 开发套件的蜂鸣器在使用时不需要短接跳线。

8.13 LCD 背光控制

开发套件的 LCD 背光控制接口文件为/sys/class/backlight/mxs-bl/brightness。该文件可以设置的值为 0~100 之间：当设置为 0 时，背光最暗；当设置为 100 时，背光最亮，其设置命令如下：

```
root@EasyARM-iMX28x ~# echo 100 > /sys/class/backlight/mxs-bl/brightness
```

LCD 亮度默认值为 80：

```
root@EasyARM-iMX28x ~# cat /sys/class/backlight/mxs-bl/brightness
80
```

注意：不适用于 EasyARM-i.MX280A 开发套件。

8.14 触摸屏校准

触摸屏校准命令为 ts_calibrate，在终端输入 ts_calibrate 命令，LCD 上出现如图 8.32 所示的 5 点校准界面。

```
root@EasyARM-iMX28x ~# ts_calibrate
```

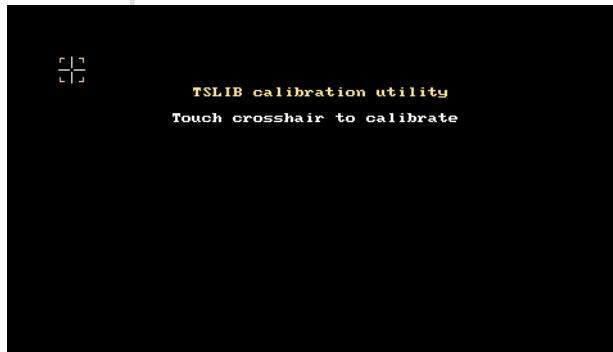


图 8.32 触摸屏校准界面

使用触笔点击“+”指针的中心，直到校准完成。输入 reboot 命令重启系统，或者先输入 sync 命令，然后按复位键重启系统。

注意：不适用于 EasyARM-i.MX280A 开发套件。



8.15 GPIO 操作

EasyARM-i.MX283A 可用作 GPIO 功能的接口如图 8.33 所示。本小节所描述的 GPIO 操作同样适用于 EasyARM-i.MX280A 和 EasyARM-i.MX287A，只是不同的开发套件在 /dev 目录下生成的设备文件不一样而已。

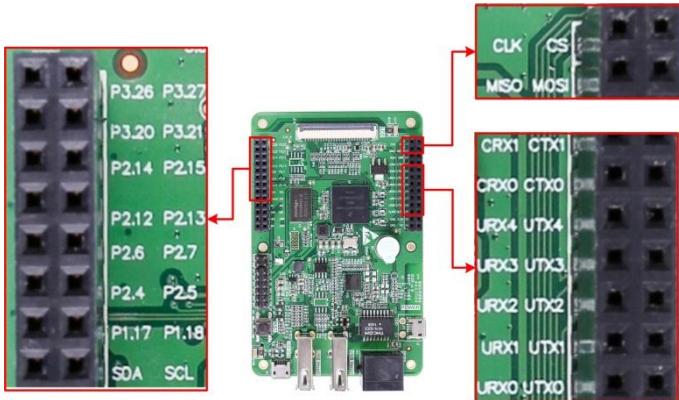


图 8.33 EasyARM-i.MX283A 的可用 GPIO

在这些接口中，以 Px.xx (x 为数字) 命名的接口是 GPIO 专用的接口，而其它接口则在有需要的情况下可以复用为 GPIO 功能（但一旦用作 GPIO 功能，除非重启，否则不能恢复为原来的功能）。

光盘资料“3、Linux\4、开发示例\6、驱动示例\2、gpio_driver”目录下有 gpio_driver.ko 模块驱动文件，将其拷贝至开发套件的“/root/”目录下，然后输入下面命令加载驱动模块：

```
root@EasyARM-iMX28x ~# insmod /root/gpio_driver.ko
```

对于 EasyARM-i.MX283A 开发套件，驱动加载完成后，会为每个 gpio 端口都生成一个设备文件节点：

```
root@EasyARM-iMX28x ~# ls /dev/gpio*
/dev/gpio-CLK      /dev/gpio-MOSI      /dev/gpio-P2.7      /dev/gpio-URX2
/dev/gpio-CRX0      /dev/gpio-P1.17      /dev/gpio-P3.20      /dev/gpio-URX3
/dev/gpio-CRX1      /dev/gpio-P1.18      /dev/gpio-P3.21      /dev/gpio-URX4
/dev/gpio-CS        /dev/gpio-P2.12      /dev/gpio-P3.26      /dev/gpio-UTX0
/dev/gpio-CTX0      /dev/gpio-P2.13      /dev/gpio-P3.27      /dev/gpio-UTX1
/dev/gpio-CTX1      /dev/gpio-P2.14      /dev/gpio-RUN       /dev/gpio-UTX2
/dev/gpio-DURX       /dev/gpio-P2.15      /dev/gpio-SCL       /dev/gpio-UTX3
/dev/gpio-DUTX       /dev/gpio-P2.4       /dev/gpio-SDA       /dev/gpio-UTX4
/dev/gpio-ERR        /dev/gpio-P2.5       /dev/gpio-URX0
/dev/gpio-MISO       /dev/gpio-P2.6       /dev/gpio-URX1
```

注意：请以用户实际开发套件所生成的设备文件为准。

这些设备文件节点和 GPIO 接口的丝印一一对应，例如可以控制 P3.27 接口的设备文件节点是 /dev/gpio-P3.27。通过这些设备文件节点，用户可以在 shell 直接操作指定的 GPIO。

以 P3.27 为例，控制 P3.27 输出高电平的方法为：

```
root@EasyARM-iMX28x ~# echo 1 > /dev/gpio-P3.27
```

控制 P3.27 输出低电平的方法为：

```
root@EasyARM-iMX28x ~# echo 0 > /dev/gpio-P3.27
```



在 P3.27 读取输入电平状态的方法为：

```
root@EasyARM-iMX28x ~# cat /dev/gpio-P3.27  
0 或 1
```

该命令会返回 0 或 1：0 表示输入的是低电平；1 表示输入的是高电平。

至于其它可以用作 GPIO 的接口操作方法也是一样。

8.16 进阶操作

本小节所介绍的实用操作需要先搭建好必要的开发环境，开发环境的搭建可以参考第 6 章内容。

8.16.1 挂载 NFS 目录

若主机配置好了 NFS 服务，则可以把主机的 NFS 目录挂载到开发套件本地的目录上。在挂载主机的 NFS 目录前，建议使用 ping 命令测试目标机和主机之间网络是否畅通。

假设主机的 NFS 目录为/nfsroot，IP 地址为 192.168.28.235，通过串口终端登录开发套件，并输入下面命令挂载主机的 NFS 目录：

```
root@EasyARM-iMX28x ~# mount -t nfs 192.168.28.235:/nfsroot /mnt -o noblock  
root@EasyARM-iMX28x ~#
```

若命令没有出错，表示挂载成功。用户进入开发套件的/mnt/目录就可以访问主机的 /nfsroot 目录。

8.16.2 使用 NFS 根文件系统

Linux 内核支持从网络加载根文件系统，这对嵌入式 Linux 开发非常有用，特别是在系统开发初期。将根文件系统放在主机上，方便文件系统的调整，也不用考虑文件系统的体积，等系统开发完毕后再进行裁剪即可。

1. 设置 NFS 根文件系统

假设主机的 IP 地址为 192.168.28.235，NFS 目录为/nfsroot/。

把光盘的 rootfs_imx28x.tar.bz2 文件复制到主机的 NFS 目录/nfsroot/，然后解压：

```
vmuser@Linux-host:~/nfsroot$ sudo tar -jxvf rootfs_imx28x.tar.bz2
```

这里必须用 root 权限解压，否则会出错。

解压完成后，会在/nfsroot 目录下生成 rootfs 目录。那么 NFS 根文件系统就在主机的 /nfsroot/rootfs/ 目录下：

```
vmuser@Linux-host:~/nfsroot$ ls rootfs  
bin dev etc home lib media mnt opt proc root sbin sys system tmp usr var
```

注意：

- (1) 在挂载 NFS 文件系统之前，应确保主机的 NFS 服务已经正确开启；
- (2) 请务必检查/nfsroot/rootfs/etc/rc.d/init.d/start_userapp 启动脚本是否有设置网卡 IP 地址的操作。如有，请先用“#”屏蔽掉此行代码，否则当该脚本运行时将会导致开发套件与 NFS 服务器的网络连接断开。

2. 在目标机设置内核 NFS 启动参数

启动开发套件，然后立即在串口终端不断键入空格键，直到进入 u-boot 的命令行模式：

```
In: serial
```



```
Out: serial
Err: serial
Net: fec_get_mac_addr
got MAC address from IIM: 00:04:9f:c7:82:19
FEC0
Warning: FEC0 MAC addresses don't match:
Address in SROM is      00:04:9f:c7:82:19
Address in environment is 02:00:92:b3:c4:a8

Hit any key to stop autoboot: 0
MX28 U-Boot >
```

这时出现了“MX28 U-Boot >”符号，这是 u-boot 的命令提示符，表示准备接受用户的命令输入。

这时需要设置 U-boot 的 bootargs 环境变量。bootargs 环境变量保存了内核的启动参数。这里需要设置内核启动参数来指示内核启动时使用 NFS 根文件系统。设置内核 NFS 启动的参数一般格式为：

```
setenv bootargs root=/dev/nfs rw console=$(consolecfg) nfsroot=$(serverip):$(rootpath)
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname):$(device):off
```

其中各个参数的意义如下：

consolecfg	——	调试串口配置；
serverip	——	提供 NFS 服务的主机 IP；
ipaddr	——	本机 IP (目标系统 IP)；
gateway	——	网关；
netmask	——	子网掩码；
hostname	——	目标板的主机名；
device	——	网络设备
rootpath	——	主机 NFS 根文件系统路径。

若主机的 IP 为 192.168.28.235，NFS 根文件系统路径为/nfsroot/rootfs；开发套件的 IP 为 192.168.28.236，则设置内核启动参数的命令为（注意，命令中的非关键域，例 hostname，可以留空但是不能没有；另外，请不要直接复制跨行命令）：

```
MX28 U-Boot >setenv bootargs 'root=/dev/nfs rw console=ttyAM0,115200n8 nfsroot=192.168.28.235:/nfsroot/
rootfs ip=192.168.28.236:192.168.28.235:192.168.28.254:255.255.255.0:epc.zlgmcu.com:eth0:off mem=64M'
```

设置完成后，输入 saveenv 命令保存设置：

```
MX28 U-Boot > saveenv
Saving Environment to NAND...
Erasing Nand...
NAND Erasing at 0x0000000000100000 -- 100% complete.
Writing to Nand... done
```

在 U-boot 终端输入 reset 命令重启开发套件，当然重新上电或复位重启也可以。

3. 进入 NFS 根文件系统

开发套件启动时，会在终端打印启动信息。在启动信息中可以看到设置的内核启动参数



是否传给了内核，并且是否正确：

```
Starting kernel ...
```

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

```
Linux version 2.6.35.3-571-gcca29a0-g2300c2d-dirty (zhuguojun@zlgmcu) (gcc version 4.4.4 (4.4.4_09.06.2010) )
```

```
#284 PREEMPT Tue Nov 11 19:28:55 CST 2014
```

```
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
```

```
CPU: VIVT data cache, VIVT instruction cache
```

```
Machine: Freescale MX28EVK board
```

```
Memory policy: ECC disabled, Data cache writeback
```

```
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 32512
```

```
Kernel command line: root=/dev/nfs rw console=ttyAM0,115200n8 nfsroot=192.168.28.235:/nfsroot/rootfs
```

```
ip=192.168.28.236:192.168.28.235:192.168.28.254:255.255.255.0:epc.zlgmcu.com:eth0:off mem=64M
```

```
PID hash table entries: 512 (order: -1, 2048 bytes)
```

开发套件启动完成后，将打印下面信息：

```
mxs-rtc mxs-rtc.0: setting system clock to 1970-01-01 00:01:51 UTC (111)
```

```
eth0: Freescale FEC PHY driver [Generic PHY] (mii_bus:phy_addr=0:05, irq=-1)
```

```
IP-Config: Complete:
```

```
    device=eth0, addr=192.168.28.236, mask=255.255.255.0, gw=192.168.28.254,
```

```
    host=epc, domain=, nis-domain=zlgmcu.com,
```

```
    bootserver=192.168.28.235, rootserver=192.168.28.235, rootpath=
```

```
Looking up port of RPC 100003/2 on 192.168.28.235
```

```
PHY: 0:05 - Link is Up - 100/Full
```

```
Looking up port of RPC 100005/1 on 192.168.28.235
```

```
VFS: Mounted root (nfs filesystem) on device 0:14.
```

```
Freeing init memory: 160K
```

```
starting pid 1120, tty ": '/etc/rc.d/rcS'
```

```
Mounting /proc and /sys
```

```
/
```

```
Starting the hotplug events dispatcher udevd
```

```
Synthesizing initial hotplug events
```

```
Setting the hostname to EasyARM-iMX28x
```

```
Mounting filesystems
```

```
Booted NFS, not relocating: /tmp /var
```

```
umount: can't umount /tmp: Invalid argument
```

```
start 283 test
```

```
starting pid 1180, tty ": '/etc/rc.d/rc_gpu.S'
```

```
starting pid 1186, tty ": '/sbin/getty -L ttyAM0 115200 vt100'
```

```
arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at 2011/08/10 09:20) 4.6.2 20110630  
(prerelease)
```

```
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800
```

```
Freescale Semiconductor, Inc.
```

EasyARM-iMX28x login:

用户名和密码都是 root，输入后完成登录，之后可进行其它操作。

```
root@EasyARM-iMX28x ~# ls /  
Settings  etc      media    proc    sys    usr  
bin       home     mnt      root    system  var  
dev       lib      opt      sbin    tmp
```

4. 恢复设置

若需要恢复开发套件从 NAND Flash 启动根文件系统，可以在 U-boot 的终端重新设置内核启动参数：

```
MX28 U-Boot > setenv bootargs gpmi=g console=ttyAM0,115200n8 ubi.mtd=1 root=ubi0:rootfs rootfstype=ubifs fec mac= ethact mem=64M
```

然后输入 saveenv 保存设置，接着重启开发套件即可。

8.16.3 使用 TFTP 启动内核

U-boot 支持通过网络从 TFTP 服务器下载内核到本地内存，然后启动内核。这种方法通常用于内核调试阶段。

假设主机的 IP 为 192.168.28.235，TFTP 服务器的根目录为/tftpboot/。把光盘的内核固件文件 uImage 上传到主机的 TFTP 服务器的根目录。

然后启动开发套件并进入 U-boot 命令行，输入命令设置主机的 IP 和目标机的 IP：

```
MX28 U-Boot > setenv serverip 192.168.28.235      # 设置主机 IP
MX28 U-Boot > setenv ipaddr 192.168.28.236      # 设置目标机 IP
MX28 U-Boot > saveenv                          # 保存设置

Saving Environment to NAND...
Erasing Nand...
NAND Erasing at 0x0000000000100000 -- 100% complete.
Writing to Nand.
```

设置完成后，运行组合命令 `setftpboot`，设置开发套件使用 TFTP 启动内核：

```
MX28 U-Boot > run settftpboot
Saving Environment to NAND...
Erasing Nand...
NAND Erasing at 0x0000000000100000 -- 100% complete.
Writing to Nand... done
MX28 U-Boot >
```

重启开发套件，U-boot 将会从 TFTP 服务器下载内核到本地内存：



其中“Loading:”后的“#”表示下载正在执行；“T”表示下载出现停顿，可能是网络不畅造成的。

如果 TFTP 下载无误，内核下载完成后将会自行启动。

若需要恢复从 NAND Flash 启动内核，可以在 U-boot 终端输入下面命令：

```
MX28 U-Boot > run setnandboot  
Saving Environment to NAND...  
Erasing Nand...  
NAND Erasing at 0x0000000000100000 -- 100% complete.  
Writing to Nand... done
```

然后重启动开发套件即可。

8.16.4 内存文件系统

开发套件的/dev/shm 目录挂载了内存文件系统，在该目录操作的文件都是储存在内存中，系统断电时储存的内容丢失。

注意，在内存文件系统储存文件会占用系统的内存空间。



第9章 系统固件烧写

本章主要讲述开发套件 Linux 固件的烧写方法，以通过 TF 卡、USB 两种方式进行整体固件烧写，也可以通过网络进行局部固件升级。本章所引用的固件目录位于光盘资料中的“3、Linux\5、Linux 系统恢复”目录下。

9.1 NAND Flash 存储器分区

开发套件板载 128MB 的 NAND Flash，其扇区大小为 128KB，Linux 内核以及文件系统都安装在其中，NAND Flash 的分区情况如表 9.1 所列。

表 9.1 NandFlash 分区信息

分区	地址范围	大小	用途
Bootloader、kernel	0x00000000-0x01400000	20M	U-Boot 及其环境变量参数、内核
rootfs	0x01400000-0x08000000	108MB	根文件系统

9.2 烧写流程图

开发套件从 NAND Flash 启动时，有两种不同启动方式：

- “uboot-kernel-rootfs” 模式：Linux 内核（uImage）通过 U-boot 导引启动，这是出厂默认方式；
- “kernel-rootfs”：Linux 内核（imx_ivt_linux.sbf）通过引导代码启动。

由于启动方式的差异，所需要的固件也是有差别的，分别提供了两套不同的固件，详见产品光盘出厂固件目录。

无论选择那种启动方式，烧写方法都是类似的，可以通过 TF 卡或者 USB 方式进行烧写，大致流程如图 9.1 所示。

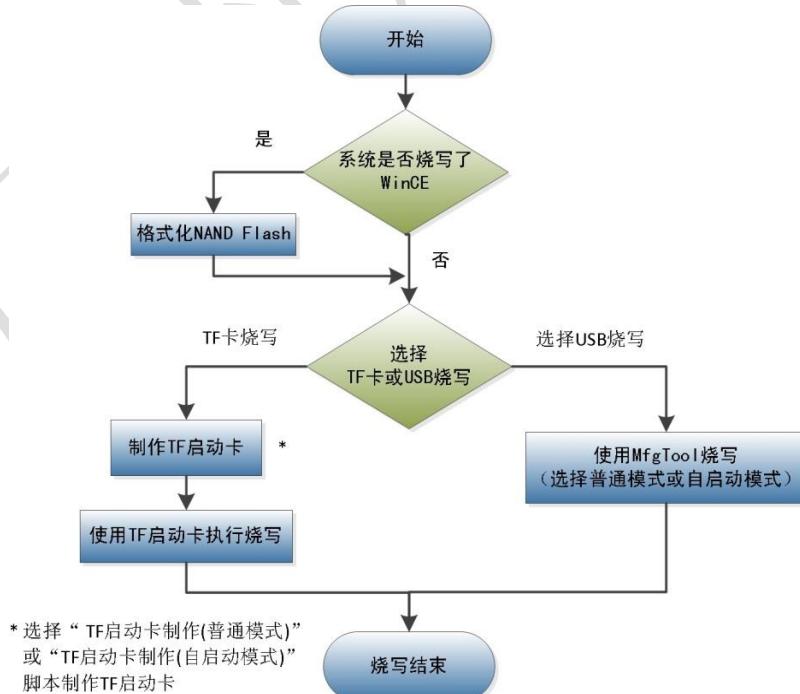


图 9.1 烧写流程图



9.3 格式化 NAND Flash

如果板子原本就是 Linux 操作系统, 仅仅进行固件恢复或者升级, 则无需格式化 NAND, 除非在使用过程中安装了 WinCE, 要重新安装 Linux, 才需进行 NAND 格式化操作。

可通过 USB Boot 或者 SD Boot 两种方式完成 NAND 格式化。

9.3.1 通过 USB Boot 引导格式化 NAND Flash

1. 格式化操作

使用 USB Boot 方式进行 NAND 格式化的步骤如下:

- 1) 把开发套件设置为 USB 启动方式(使用短路器短接 JP4 和 JP6 跳线, 保持 JP1、JP2、JP3 和 JP5 跳线的断开);
- 2) 使用 MicroUSB 通信电缆接开发套件的 USB OTG 接口和主机;
- 3) 建立主机和开发套件的调试串口连接;
- 4) 在主机打开串口终端监听串口数据;
- 5) 给开发套件接通电源;
- 6) 进入光盘文件中的“3、Linux\5、Linux 系统恢复\MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录, 双击“OTG 格式化 NAND Flash.bat”脚本程序, 将弹出如图 9.2 所示的界面, 但很快将自动关闭。

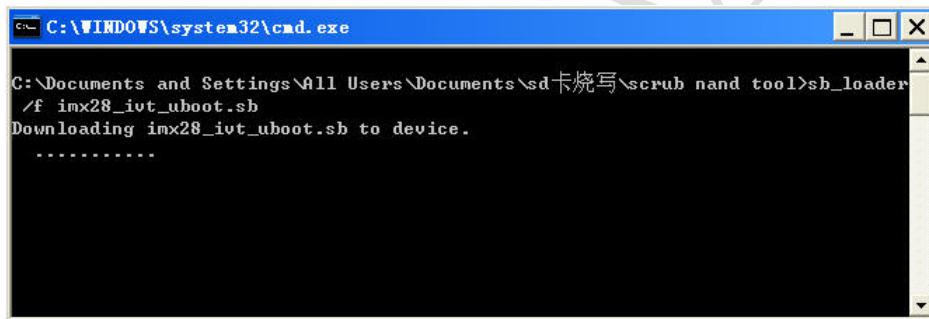


图 9.2 脚本界面

2. 格式化结果判断

这时串口终端将打印格式化输出信息。当看到“nand scrub done”的提示信息时, 表示格式化成功, 如下所示:

```
NAND scrub: device 0 whole chip
Warning: scrub option will erase all factory set bad blocks!
        There is no reliable way to recover them.
        Use this command only for testing purposes.
NAND Erasing at 0x0000000007fe0000 -- 100% complete.
OK!
nand scrub done.
MX28 U-Boot >
```

如果看到串口终端的输出信息在“nand scrub done”上一行打印了“ERROR!”字样, 则表示格式化失败。

如果串口出现少量“MTD Erase failure: -%d at:0xXXXXXXXXXXXXXXX” 的提示信息, 如:



```
NAND scrub: device 0 whole chip
Warning: scrub option will erase all factory set bad blocks!
        There is no reliable way to recover them.
        Use this command only for testing purposes.
NAND Erasing at 0x0000000001700000 -- 9% complete.
nand0: MTD Erase failure: -5 at:0x000000000018c0000
NAND Erasing at 0x000000000047a0000 -- 28% complete.
nand0: MTD Erase failure: -5 at:0x00000000004900000
NAND Erasing at 0x000000000070a0000 -- 44% complete.
nand0: MTD Erase failure: -5 at:0x000000000070c0000
NAND Erasing at 0x00000000008cc0000 -- 55% complete.
nand0: MTD Erase failure: -5 at:0x00000000008d40000
NAND Erasing at 0x0000000000ffe0000 -- 100% complete.
OK!
nand scrub done.
MX28 U-Boot >
```

这是由于 NAND Flash 存在坏块所致，这属于正常情况，NAND Flash 允许存在一定数量的坏块。

如果出现的信息全是这个错误，则有可能是 NAND 损坏。

若启动“OTG 格式化 NAND Flash.bat”批处理时，串口终端没有反应，请检查是否有下列情形：

串口终端通信参数是否设置好；

MicroUSB 通信电缆是否连接正常；

“OTG 格式化 NAND Flash.bat”在启动一次后，开发套件必须重新上电或按 RST 复位后，才能再一次进行格式化；

设置为 USB 启动方式的开发套件在接入电脑后，电脑的设备管理器中会多一个 HID 设备出来，如图 9.3 所示；若电脑中未发现这个 HID 设备，请先检查启动模式配置及与电脑的连接是否正常，然后重新复位开发套件并插拔 USB 连接线；



图 9.3 正常连接的情况

“OTG 格式化 NAND Flash.bat”是调用了 imx28_ivt_uboot_erase.sb 文件及飞思卡尔原厂提供的 sb_loader.exe 程序，所以运行该脚本前需要保证同一目录下的 imx28_ivt_uboot_erase.sb 文件及 sb_loader.exe 文件正常且未被占用；

在 Windows 7 系统必须以管理员身份运行 OTG 格式化 NAND Flash.bat 脚本。

9.3.2 通过 SD Boot 方式格式化 NAND Flash

1. 制作格式化启动卡

通过 SD Boot 方式格式化 NAND Flash 需要先制作一张格式化 NAND Flash 专用的 TF 启动卡，其制作步骤如下：

将一张空白的 TF 通过读卡器插入电脑（操作系统必须为 Windows XP 专业版或 Win7 旗舰版），并记下电脑分配给它的盘符（推荐使用 Class 4 的 TF 卡）；

双击运行光盘资料中“3、Linux\5、Linux 系统恢复\MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下的“TF 格式化 NAND Flash.bat”脚本文件（Win7 系统必须以管理员身份运行该脚本），该脚本运行后显示的界面如图 9.4 所示；

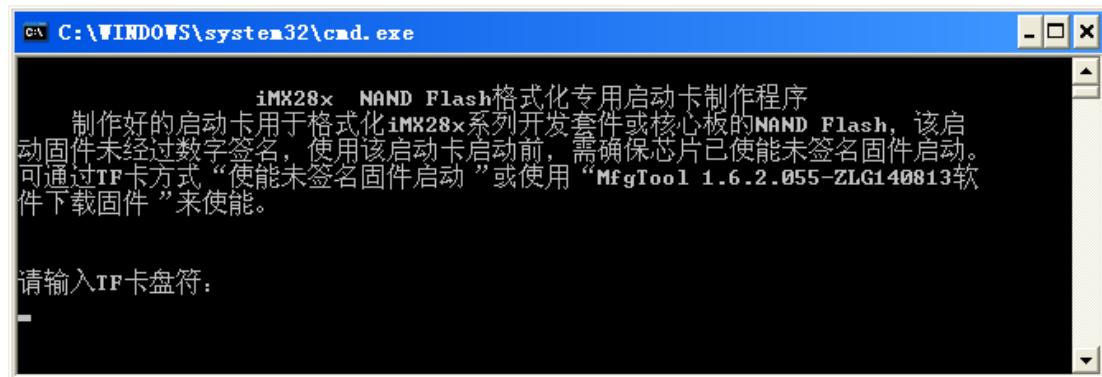


图 9.4 TF 格式化 NAND Flash.bat 脚本启动界面

然后输入系统分配给 TF 卡的盘符（这里假设为 g 盘）并按回车键，如图 9.5 所示；

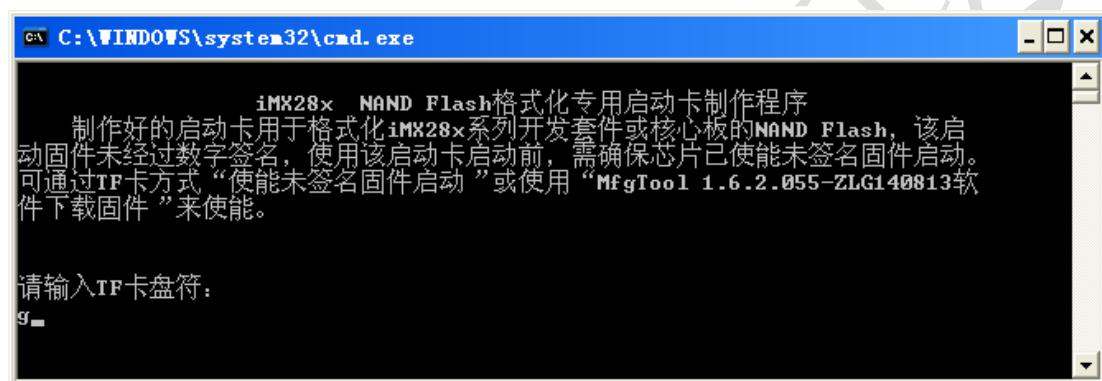


图 9.5 输入分配给 TF 卡的盘符

启动卡制作完后如图 9.6 所示，此时按照移除 U 盘的方式移除该 TF 卡即可。

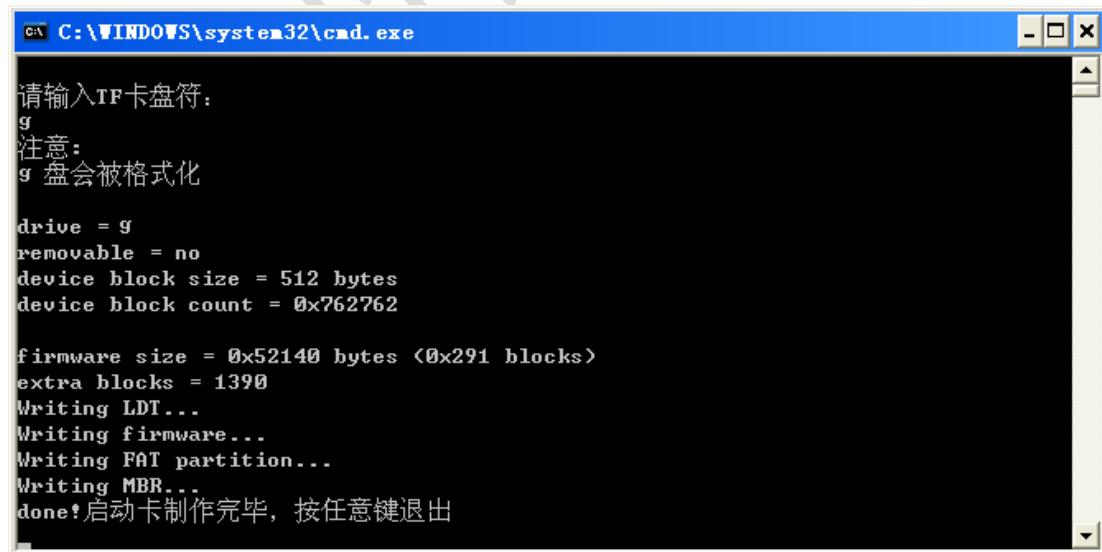


图 9.6 NAND 格式化专用启动卡制作完成

2. 执行格式化

格式化专用启动卡制作好了之后按如下步骤进行 NAND Flash 的格式化：



把开发套件设置为 SD 启动方式（使用短路器短接 JP3 和 JP4 跳线，保持 JP1、JP2、JP5 和 JP6 跳线的断开）；

建立主机和开发套件的调试串口连接；

打开串口终端软件，并进行正确设置（115200,8n1）；

将格式化专用启动卡接入开发套件的 TF 卡座；

接通开发套件电源，等待格式化程序运行完毕。

格式过程中串口终端打印信息与“通过 USB Boot 引导格式化”时串口打印的信息完全相同。

9.4 TF 卡烧写方案

TF 卡烧写方案是：在 Windows 系统制作固件烧写用的 TF 启动卡；然后使用 TF 启动卡烧写固件。

烧写过程分两步：制作 TF 启动卡和进行固件烧写操作。

9.4.1 TF 卡烧写用的固件

TF 卡烧写所需的固件在光盘的“Linux 系统恢复\ MfgTool 1.6.2.055-ZLG140813\Profiles\ MX28 Linux Update\OS Firmware\files”目录。该目录下的内容如图 9.7 所示。

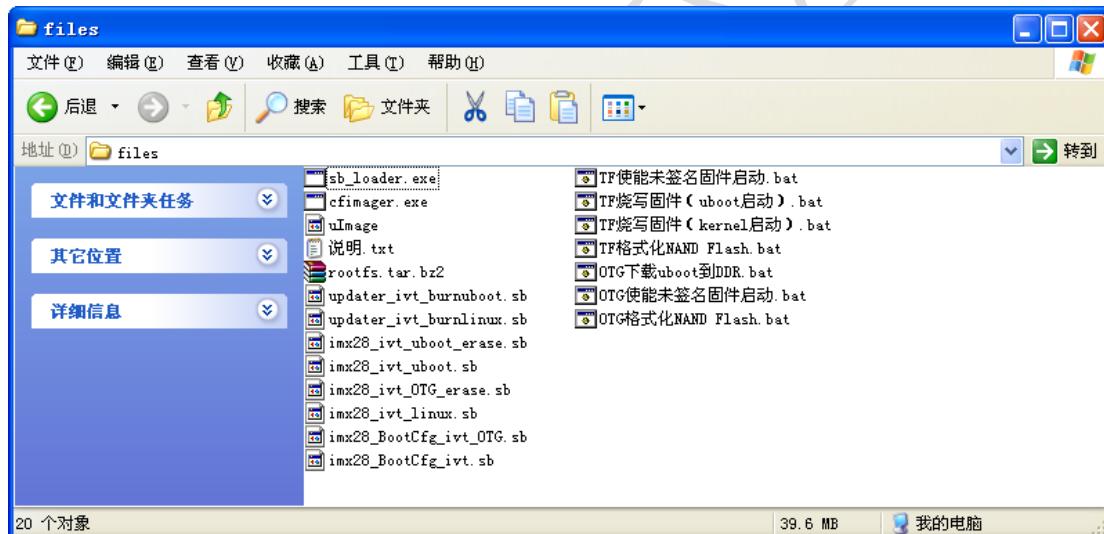


图 9.7 “TF 卡烧写方案”的目录内容

该目录提供了“TF 烧写固件（uboot 启动）.bat”和“TF 烧写固件（kernel 启动）.bat”两个脚本文件，它们均可用于制作烧写固件的 TF 启动卡。

在制作 TF 启动卡之前，请用户根据所使用的开发套件型号来使用对应的烧写固件，具体操作步骤如下：

- 对于 EasyARM-i.MX280A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX280A 内核固件”目录下的所有文件替换到“Linux 系统恢复\ MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下的对应文件。
- 对于 EasyARM-i.MX283A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX283A 内核固件”目录下的所有文件替换到“Linux 系统恢复\



MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files” 目录下的对应文件。

- 对于 EasyARM-i.MX287A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX287A 内核固件”目录下的所有文件替换到“Linux 系统恢复*MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files*”目录下的对应文件。

9.4.2 制作 TF 启动卡

准备一张 TF 卡（经验证，Class2 和 Class10 不能使用，推荐使用 Class4）和一个读卡器。请确保该 TF 卡只有一个分区，并且是 FAT32 格式。若有一个分区请先使用 Windows 的磁盘管理工具删除所有分区后再重建一个主分区。

把 TF 卡安装入读卡器，再把读卡器插入 PC 机的 USB 端口。这时 Windows 将在“我的电脑”中增加了一个驱动器，如图 9.8 所示。



图 9.8 添加的驱动器

进入 TF 卡烧写固件所在的目录，双击“TF 烧写固件（uboot 启动）.bat”或“TF 烧写固件（kernel 启动）.bat”脚本文件，将弹出如图 9.9 所示的界面。

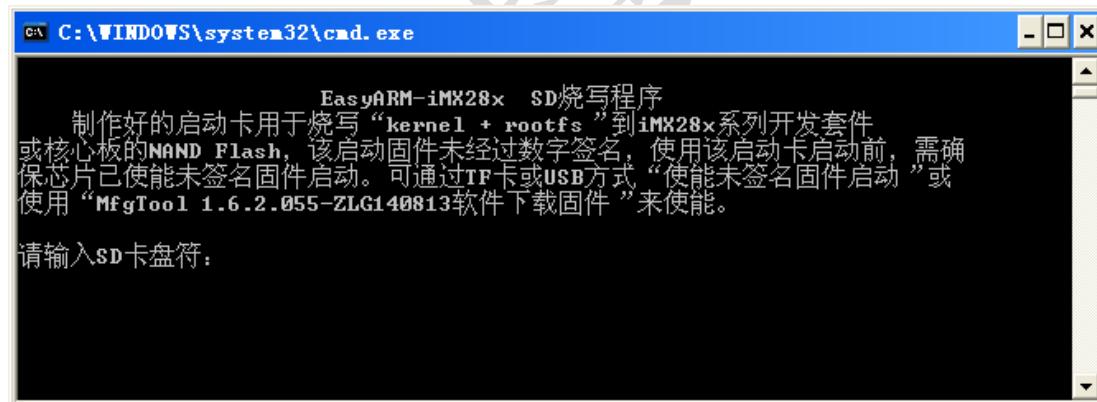


图 9.9 提示用户输入读卡器的盘符

这是提示用户输入刚插入的读卡器的盘符（假设为 G 盘），然后键入“Enter”，执行启动卡的制作。制作过程如图 9.10 所示。这里需要花几分钟的时间。



```
C:\WINDOWS\system32\cmd.exe
EasyARM-i.MX28x SD烧写程序
制作好的启动卡用于烧写“kernel + rootfs”到iMX28x系列开发套件或核心板的NAND Flash，该启动固件未经过数字签名，使用该启动卡启动前，需确保芯片已使能未签名固件启动。可通过TF卡或USB方式“使能未签名固件启动”或使用“MfgTool 1.6.2.055-ZLG140813软件下载固件”来使能。
请输入SD卡盘符:
g
注意:
    文件会被烧写在 g 盘

drive = g
removable = no
device block size = 512 bytes
device block count = 0x762762

firmware size = 0x516050 bytes (0x28b1 blocks)
extra blocks = 1870
Writing LDT...
Writing firmware...
Writing FAT partition...
```

图 9.10 输入盘符

制作完成后，将显示如图 9.11 所示的信息。

```
C:\WINDOWS\system32\cmd.exe
请输入SD卡盘符:
g
注意:
    文件会被烧写在 g 盘

drive = g
removable = no
device block size = 512 bytes
device block count = 0x762762

firmware size = 0x516050 bytes (0x28b1 blocks)
extra blocks = 1870
Writing LDT...
Writing firmware...
Writing FAT partition...
Writing MBR...
done! 已复制      1 个文件。
已复制      1 个文件。
已复制      1 个文件。
烧写完毕，按键退出
```

图 9.11 制作完成

这时键入“Enter”键退出。至此，可用于烧写固件的 TF 启动卡已经制作好。

9.4.3 固件烧写步骤

使用 TF 启动卡烧写固件的步骤如下：

把制作好的 TF 启动卡插入到开发套件的 TF 卡卡槽；

把开发套件设置为 SD 启动方式（使用短路器短接 JP3 和 JP4 跳线，保持 JP1、JP2、JP5 和 JP6 跳线的断开）；

建立主机和开发套件的调试串口连接；



打开串口终端软件，并进行正确设置（115200,8n1）；

给开发套件重新上电或按 RST 键复位。

这时开发套件自动进入固件烧写程序，同时在串口终端打印烧写过程信息，整个过程需要几分钟时间。

当开发套件的蜂鸣器发出“哔，哔，哔……”的声音时，表示烧写完成，并在串口终端打印如下信息：

注意：由于 updater 不能统一的原因，在烧写完 EasyARM-i.MX280A 时，蜂鸣器不会鸣叫。

```
UBIFS: un-mount UBI device 0, volume 0
write rootfs done
writing uboot.....
writing uboot done
writing kernel...
Erase Total 32 Units
Performing Flash Erase of length 131072 at offset 0x5e0000 done
2+1 records in
2+1 records out
writing kernel done
system install is done,you can reboot system
```

然后把开发套件设置为 NAND Flash 启动方式（拔出 JP3 的短路器，仅短接 JP4，其他全部断开），按“RST”键复位系统。开发套件将从 NAND Flash 启动 Linux 系统。

9.5 USB 烧写方案

通过 USB 烧写固件需要使用飞思卡尔提供的 MfgTool 软件。MfgTool 软件在光盘的 *Linux 系统恢复\ MfgTool 1.6.2.055-ZLG140813* 目录。该目录的内容如图 9.12 所示，其中 MfgTool.exe 程序是 USB 固件烧写的程序。



图 9.12 MfgTool 软件

说明：MfgTool 软件不支持 Win8 系统。

在进行 USB 烧写之前，请用户根据所使用的开发套件型号来使用对应的烧写固件，具体操作步骤如下：

- 对于 EasyARM-i.MX280A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX280A 内核固件”目录下的所有文件替换到“Linux 系统恢复\ MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下的对应文件。
- 对于 EasyARM-i.MX283A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX283A 内核固件”目录下的所有文件替换到“Linux 系统恢复\ MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下的对应文件。



- 对于 EasyARM-i.MX287A 开发套件，需要把“5、Linux 系统恢复\基本固件\EasyARM-i.MX287A 内核固件”目录下的所有文件替换到“Linux 系统恢复\MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下的对应文件。

9.5.1 执行 USB 烧写

1. 硬件连接

硬件连接方法如下：

把开发套件设置为 USB 启动方式（使用短路器短接 JP4 和 JP6 跳线，保持 JP1、JP2、JP3 和 JP5 跳线的断开）；

准备两根 MicroUSB 线缆，用其中一根连接开发套件的 USB OTG 接口和主机，另外一根则给开发套件供电。

2. 设置 MfgTool 软件

双击运行 MfgTool.exe 软件，软件界面如图 9.13 所示。



图 9.13 MfgTool 的主界面

点击主菜单中的 Options→Configuration...菜单项，打开 MfgTool 的配置界面，如图 9.14 所示。

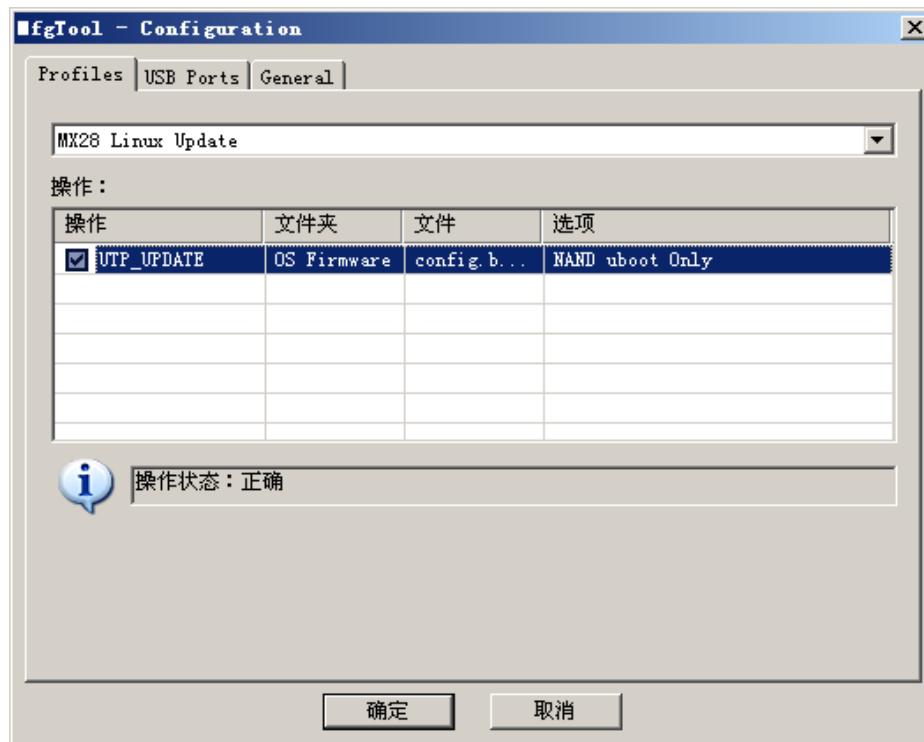


图 9.14 MfgTool 的配置界面

在 MfgTool 的配置界面，进入“Profiles”标签，在 UTP_UPDATE 项的“选项”列中选择“NAND kernel-rootfs (128MB)”或“NAND uboot-kernel-rootfs (128MB)”，如图 9.15 所示，然后点击“确定”。

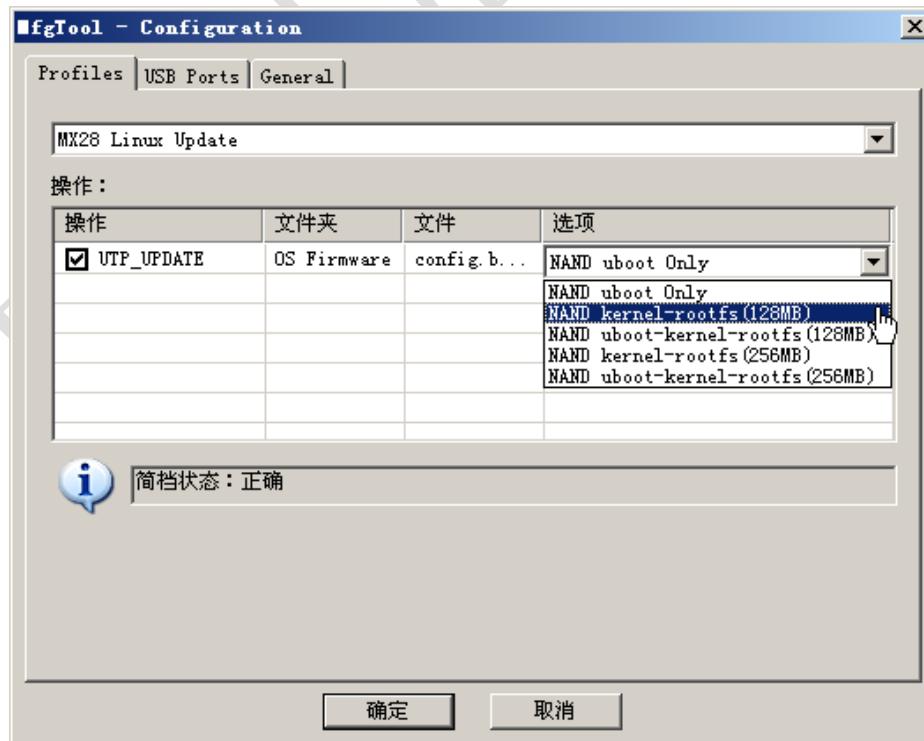


图 9.15 选择“NAND kernel-rootfs (128MB)”

切换到“USB Ports”标签，勾选已经连接上的“HID-compliantdevice”（即开发套件对



应的设备)如图 9.16 所示。然后点击“确定”。



图 9.16 勾选连接的 HID-compliantdevice

3. 执行烧写

返回到 MfgTool 主界面后显示软件正在监视 HID-compliantdevice，如图 9.17 所示。



图 9.17 MfgTool 监视 HID-compliantdevice

此时点击“开始”执行烧写操作，如图 9.18 所示。



图 9.18 执行烧写

直到烧写完成后点击“停止”，如图 9.19 所示。



图 9.19 烧写完成

注意：在烧写过程中需要保持 MiniUSB 线缆的连接正常和开发套件的正常供电，否则在烧写过程容易出错。

烧写完成后，把开发套件设置为 NAND Flash 启动方式（取出 JP6 的短路器，仅短接 JP4，其他全部断开），按“RST”键复位系统。开发套件将从 NAND Flash 启动 Linux 系统。

若使用 MfgTool 软件时出错，请检查是否有下列情形：

MicroUSB 线缆是否连接正常；

MfgTool 在点击“开始”烧写后，开发套件必须再重新上电或按 RST 复位后，才能再次进行烧写；

设置为 USB 启动方式的 EasyARM-i.MX283A 在接入电脑后，在电脑的设备管理器会多一个 HID 设备出来，如图 9.3 所示，若电脑中未发现这个 HID 设备，请先检查启动模式配置及与电脑的连接是否正常，然后重新复位 EasyARM-i.MX283A 并插拔 USB 连接线；

在 Windows 7 系统，建议以管理员身份运行 MfgTool 软件。

在使用 MfgTool 烧写固件的过程中，目标板将被虚拟成大容量存储设备（U 盘），如果用户的电脑系统受文件监控软件的保护，将可能无法正常进行烧写。所以，在使用 MfgTool 烧写固件前需要先关闭会对 U 盘读写进行监控的软件，如杀毒软件及防火墙等。

9.6 使用网络升级内核或文件系统

若开发套件是以“uboot-kernel-rootfs”模式启动，系统通过 U-boot 引导内核。而通过 U-Boot，可以通过网络升级内核和文件系统。

9.6.1 网络升级用的固件



网络升级所需的固件在光盘的“3、Linux\5、Linux 系统恢复\基本固件”目录，如图 9.20 所示。

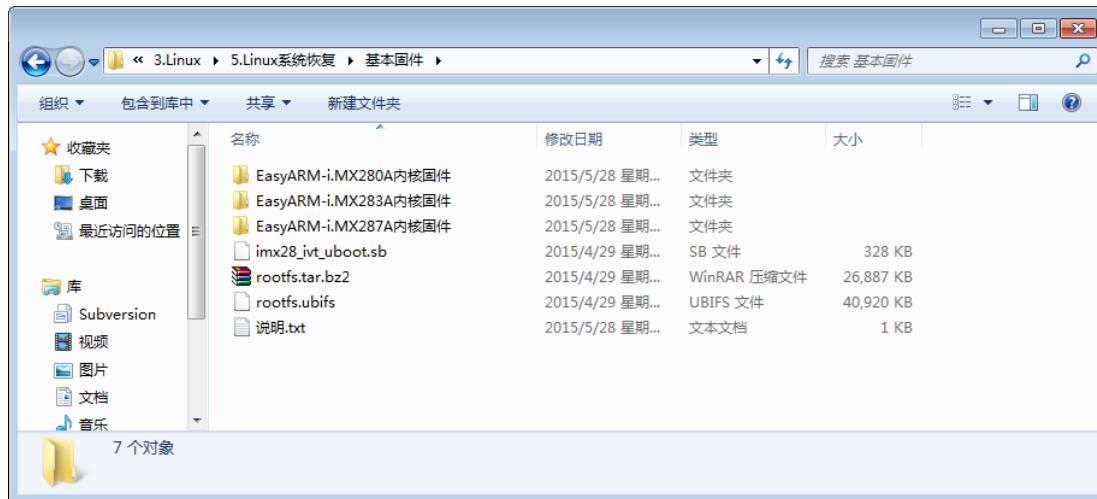


图 9.20 固件内容

网络烧写需要两个文件：uImage（内核固件）和 rootfs.ubifs（文件系统固件）。把这两个文件复制到主机 TFTP 服务器目录。

注意：

对于 EasyARM-i.MX280A 开发套件，内核固件为：EasyARM-i.MX280A 内核固件\uImage。

对于 EasyARM-i.MX283A 开发套件，内核固件为：EasyARM-i.MX283A 内核固件\uImage。

对于 EasyARM-i.MX287A 开发套件，内核固件为：EasyARM-i.MX287A 内核固件\uImage。

9.6.2 升级步骤

使用网络执行内核或文件系统固件升级的步骤如下：

1. 硬件连接

硬件连接方法如下：

把开发套件设置为 NAND Flash 启动方式（使用短路器短接 JP4 跳线，JP1、JP2、JP3、JP5 及 JP6 跳线保持断开）；

建立开发套件的调试串口和主机之间的串口连接；

在主机打开串口终端；

使用网线连接开发套件和主机。

2. 进入 U-boot 终端

上电启动开发套件并进入 U-Boot 命令行：

```
In:    serial
Out:   serial
Err:   serial
Net:   fec_get_mac_addr
got MAC address from IIM: 00:04:9f:c7:82:19
FEC0
Warning: FEC0 MAC addresses don't match:
Address in SROM is      00:04:9f:c7:82:19
```



```
Address in environment is 02:00:92:b3:c4:a8
```

```
Hit any key to stop autoboot: 0  
MX28 U-Boot >
```

3. 在 U-boot 配置目标机和主机 IP

开发套件和主机的 IP 必须在同一个网段。本示例中主机的 IP 为 192.168.28.235，EasyARM-iMX28A 的 IP 设置为 192.168.28.236。

在串口终端输入如下命令：

```
MX28 U-Boot > setenv ipaddr 192.168.28.236  
MX28 U-Boot > setenv serverip 192.168.28.235  
MX28 U-Boot > saveenv
```

4. 测试目标机和主机之间网络是否畅通

在执行升级之前需要先确保开发套件与主机的网络连接畅通，可以使用 ping 命令进行测试。如果出现“host x.x.x.x is alive”这样的提示，则表示网络连接正常：

```
MX28 U-Boot > ping 192.168.28.235  
Using FEC0 device  
host 192.168.28.235 is alive #表示网络连接畅通
```

如果出现“host x.x.x.x is not alive”的提示，则表示网路有故障，请检查硬件连接或者网络配置：

```
MX28 U-Boot > ping 192.168.28.235  
Using FEC0 device  
ping failed; host 192.168.28.235 is not alive #表示网络不通
```

5. 执行烧写

在开发套件与主机的网络连接畅通的条件下，输入 run upkernel 命令升级内核固件：

```
MX28 U-Boot > run upkernel  
Using FEC0 device  
TFTP from server 192.168.28.234; our IP address is 192.168.28.236  
Filename 'uImage'.  
Load address: 0x41600000  
Loading: T #####  
#####  
#####  
done  
Bytes transferred = 2519700 (267294 hex)
```

```
NAND erase: device 0 offset 0x200000, size 0x300000  
NAND Erasing at 0x00000000004e0000 -- 100% complete.  
OK!  
nand scrub done.
```

```
NAND write: device 0 offset 0x200000, size 0x300000  
3145728 bytes written: OK
```



输入 run uprootfs 命令升级文件系统:

```
MX28 U-Boot > run uprootfs
```

```
NAND erase: device 0 offset 0x1400000, size 0x6c00000
NAND Erasing at 0x0000000007fe0000 -- 100% complete.
OK!
nand scrub done.
..... 省略 .....
#####
#####
#####
done
Bytes transferred = 40124416 (2644000 hex)
Volume "rootfs" found at volume id 0
```

9.6.3 故障排除

但若主机 TFTP 服务器不可访问（**即使网络可以 ping 通**），将会导致升级内核或文件系统命令执行失败，此时串口终端显示信息如下：

```
MX28 U-Boot > run upkernel
Using FEC0 device
TFTP from server 192.168.28.235; our IP address is 192.168.28.236
Filename 'uImage'.
Load address: 0x41600000
Loading: T T T T T T T T T T
```

或

```
MX28 U-Boot > run uprootfs

NAND erase: device 0 offset 0x1400000, size 0x6c00000
NAND Erasing at 0x0000000007fe0000 -- 100% complete.
OK!
.....省略.....
TFTP from server 192.168.28.234; our IP address is 192.168.28.236
Filename 'rootfs.ubifs'.
Load address: 0x41600000
Loading: T T T T T T T T T T
```

此时需要检查是否忘记打开 TFTP 服务器或者 TFTP 服务器软件被相关防火墙拦截。

若 TFTP 服务器根目录下未放置 uImage 及 rootfs.ubifs 文件，在执行升级内核或文件系统时将会出现“File not found”的错误，如下所示：

```
MX28 U-Boot > run upkernel
Using FEC0 device
TFTP from server 192.168.28.235; our IP address is 192.168.28.236
Filename 'uImage'.
Load address: 0x41600000
```



```
Loading: T
```

```
TFTP error: 'File not found' (1)
```

```
Starting again
```

这时只需要将光盘文件中的 uImage 及 rootfs.ubifs 文件放到 TFTP 服务器目录即可。

注意：开发套件的 U-Boot 仅支持 100Mb 的全双工、半双工的网络，不支持 10Mb 的全双工、半双工的网络。



第三篇 基本固件编译及其简单应用

这一篇内容围绕 u-boot、内核及文件系统的编译和打包以及它们的简单应用展开。首先介绍 u-boot，通过一章的篇幅对该平台 Linux 系统的 Boot Loader 进行描述，包括源码目录结构、编译方法，基本命令和工具介绍等；然后再通过一章篇幅介绍该平台使用的 Linux 内核及内核驱动要点；最后则介绍该平台文件系统的打包及简单应用。

本篇一共分为 3 章，各章的标题和大概内容如下：

第 10 章：u-boot 编译及其简单应用，介绍开发套件所使用的 u-boot 的编译及其简单应用；

第 11 章：内核编译及其驱动设计要点，介绍开发套件所使用的 kernel 编译及内核驱动设计的注意事项；

第 12 章：根文件系统的打包及其简单应用。



第10章 u-boot 编译及其简单应用

开发套件中的 Linux 可以使用 U-Boot 作为 Boot Loader，本章首先介绍如何编译开发套件光盘资料自带的 U-Boot，然后讲解 U-Boot 的一些简单应用。

本章所引用的源码位于光盘目录“3、Linux\6、源代码\”下，具体源码包为“bootloader.tar.bz2”。

10.1 U-Boot 简介

U-Boot，全称 Universal Boot Loader，是遵循 GPL 条款的开源项目。从 FADSROM、8xxROM、PPCBoot、ARMBoot 逐步发展演化而来。U-Boot 不仅支持嵌入式 Linux 系统，它还可用于 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等嵌入式操作系统的引导。U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、x86、ARM、NIOS、XScale 等多种常见架构的处理器。

10.2 U-Boot 源代码目录

U-Boot 源代码目录按照 CPU 或者开发板进行组织。进入 U-Boot 源码目录，可以看到如下的一些目录（省略了其它文件），常见的主要目录的说明如表 10.1 所示。

api	api_examples	board	common	cpu	disk	doc	drivers
examples	fs	include	lib_arm	lib_avr32	lib_blackfin	libfdt	
lib_generic	lib_i386	lib_m68k	lib_microblaze	lib_mips	lib_nios	lib_nios2	lib_ppc
lib_sh	lib_sparc	nand_spl	net	onenand_ipl	post	tools	

表 10.1 U-Boot 重要目录说明

目 录	说 明
board	存放了 U-Boot 所支持开发板的相关代码，目前支持几十个不同体系架构的开发板，如 evb4510、pxa255_idp、omap2420h4 等
common	U-Boot 命令实现代码目录
cpu	包含了不同处理器相关的代码，按照不同的处理器进行分类，如 arm920t、arm926ejs、i386、nios2 等
drivers	U-Boot 所支持外设的驱动程序。按照不同类型驱动进行分类如 spi、mtd、net 等等
fs	U-Boot 所支持的文件系统的代码。目前 U-Boot 支持 cramfs、ext2、fat、fdos、jffs2、reiserfs
include	U-Boot 头文件目录，里面还包含各种不同处理器的相关头文件，以 asm-体系架构这样的目录出现。另外，不同开发板的配置文件也在这个目录下：include/configs/开发板.h
lib_xxx	不同体系架构的一些库文件目录
net	U-Boot 所支持网络协议相关代码，如 bootp、nfs 等
tools	U-Boot 工具源代码目录。其中的一些小工具如 mkimage 就非常实用

10.3 编译 U-Boot

请把光盘资料中的 bootloader.tar.bz2 文件复制到 Linux 主机的工作目录，然后解压该压缩包：

```
vmuser@Linux-host: ~$ tar -jxvf bootloader.tar.bz2
```

然后得到一个 bootloader 目录。进入该目录可以看到其下有 elftosb、u-boot-2009.08 和



imx-bootlets-src-10.12.01 等子目录。u-boot-2009.08 目录内是 U-Boot 的源代码文件，把这些源文件编译后可得到 u-boot 文件，该 u-boot 文件需要通过 imx-bootlets-src-10.12.01^[1]目录下的工程进一步编译成 imx28_ivt_uboot.sb 文件（用于烧写到 NAND Flash 的文件）。elftosb 目录则提供了 32bit 和 64bit Linux 系统适用的 elftosb^[2]转换工具。

注[1]: imx-bootlets-src-10.12.01 工程简称 bootlets 工程，是针对 i.MX28x 芯片设计的引导加载程序，专门负责用户程序执行前的初始化工作，如初始化内存、加载用户程序、初始化 PMU 等。该程序的初始化工作执行完后将执行跳转指令，将 CPU 的控制权交给用户程序。该工程主要包含三大功能模块，分别是：

power_prep — 这部分功能代码主要负责初始化片上 PMU；

boot_prep — 这部分功能代码主要负责初始化系统时钟及内存；

linux_prep — 这部分功能代码主要负责 Linux 内核引导预处理。

针对 i.MX28x 系列芯片的 Linux 发行包提供两种镜像引导启动方式，分别是“Linux 内核引导启动”和“U-boot 引导启动”。经过 bootlets 工程编译后的“启动流”分别如图 10.1 和图 10.2 所示。

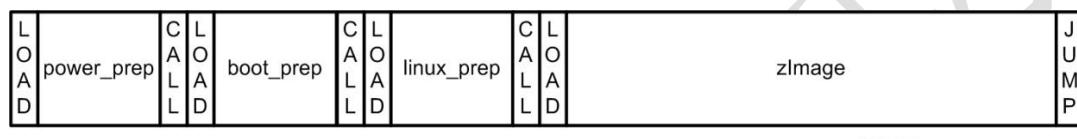


图 10.1 Linux 内核引导启动方式

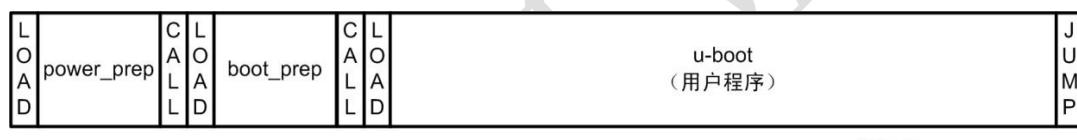


图 10.2 U-boot 引导启动

imx28_ivt_linux.sb 和 imx28_ivt_uboot.sb 文件即是符合图 10.1 和图 10.2 所示的“启动流文件”，即可直接启动的固件文件。

注[2]: elftosb

Freescale 提供的一种文件转换工具，可以将 elf 格式文件转换为 i.MX28x 系列芯片适用的 sb (Secure Boot，更多详细介绍请参考《MCIMX28RM.pdf》文档) 格式文件。

生成适用于开发套件的 U-Boot 文件需要按如下步骤进行操作：

首先，进入 u-boot-2009.08 目录，清除原有的编译文件，其对应的终端命令如下：

```
vmuser@Linux-host: ~$ cd bootloader/u-boot-2009.08  
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ make ARCH=arm  
CROSS_COMPILE=arm-fsl-linux-gnueabi- distclean
```

注意：distclean 前面是有一个空格的；此外，为了方便读者阅读，在容易出错的命令后面添加了一个显式的回车符，表示一句完整命令的结束。

其次，需要配置 U-Boot 的平台为 mx28_evk_config，对应的终端命令如下：

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ make ARCH=arm  
CROSS_COMPILE=arm-fsl-linux-gnueabi- mx28_evk_config  
Configuring for mx28_evk board...
```



然后，执行编译，对应的终端命令如下：

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ make ARCH=arm  
CROSS_COMPILE=arm-fsl-linux-gnueabi-
```

编译完成后将在 u-boot-2009.08 目录的根目录下得到 u-boot 文件。但是 u-boot 文件并不能作为固件在开发套件的 NAND Flash 中直接烧写后启动。u-boot 文件需要使用 imx-bootlets-src-10.12.01 目录下的工具进一步编译成带电源配置的 imx28_ivt_uboot.sb 固件文件。

把 u-boot 复制到 imx-bootlets-src-10.12.01 目录下：

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ cp u-boot .. imx-bootlets-src-10.12.01
```

进行 u-boot 转换前需要先将 elftosb 目录下的“elftosb_32bit 或 elftosb_64bit”文件改名为“elftosb”并复制到“/usr/bin/”目录下（请以用户搭建的 Linux 上位机系统位宽为准）。复制完后需要给 elftosb 赋予可执行的权限（如使用我们官网提供的 ubuntu 则不需要这步操作）。

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ cd ..elftosb/  
vmuser@Linux-host: ~/bootloader/elftosb$ mv elftosb_64bit elftosb      # 请根据实际情况进行选择  
vmuser@Linux-host: ~/bootloader/elftosb$ sudo cp elftosb /usr/bin/  
vmuser@Linux-host: ~/bootloader/elftosb$ sudo chmod 777 /usr/bin/elftosb
```

进入 imx-bootlets-src-10.12.01 目录，然后执行编译命令：

```
vmuser@Linux-host: ~/bootloader/elftosb$ cd .. imx-bootlets-src-10.12.01  
vmuser@Linux-host: ~/bootloader/imx-bootlets-src-10.12.01$ ./build
```

编译完成后 imx-bootlets-src-10.12.01 目录下的 imx28_ivt_uboot.sb 文件就是可以烧写到 NAND Flash 的固件文件。具体的烧写方法请参考 9.5 小节“烧写 U-Boot”的内容。

整个编译过程需要输入较多的命令，为了方便用户操作，在“u-boot-2009.08”目录下已经写好了专用的编译脚本文件 build-uboot，在目录下执行该脚本即可自动完成想要的操作（不含 elftosb 文件的拷贝，该操作在 Linux 主机环境中仅需执行一次），具体命令如下所示：

```
$ ./build-uboot  
U-Boot build menu, please select your choice:  
1  make distclean  
2  config for mx28  
3  build U-Boot  
q  exit
```

红色部分为脚本执行时的选择菜单。当出现该菜单时，输入对应的数字并按回车键即可。其中“build U-Boot”选项已经包含了将 u-boot 复制到 imx-bootlets-src-10.12.01 工程进行编译的操作，最终的编译结果位于 imx-bootlets-src-10.12.01 工程目录下。

注意：为确保该脚本命令执行正确，请保持 imx-bootlets-src-10.12.01 目录与 u-boot-2009.08 目录在同一个文件夹下，并且不要随意修改 imx-bootlets-src-10.12.01 工程下的 build 脚本文件。

10.4 U-Boot 基本命令

在 U-Boot 启动阶段，在串口终端按任意键（**如空格键**）即可进入 U-Boot 的命令行模式（如果 U-Boot 未设置等待延时，则需要在 U-Boot 启动阶段持续按下任意键才能进入 U-Boot



的命令行模式), 可以输入已支持的命令对 U-Boot 进行配置。

```
U-Boot 1.3.3 (Feb 10 2009 - 10:09:52)
```

```
DRAM: 64 MB
```

```
NAND: 256 MB
```

```
In: serial
```

```
Out: serial
```

```
Err: serial
```

```
Hit any key to stop autoboot: 0
```

```
MX28 U-Boot >
```

在 U-Boot >提示符下, 输入?或者 help 可以查看 U-Boot 所支持的全部命令以及对应介绍。

```
MX28 U-Boot > ?
```

```
?           - alias for 'help'  
autoscr     - DEPRECATED - use "source" command instead  
base         - print or set address offset  
bdinfo       - print Board Info structure  
boot         - boot default, i.e., run 'bootcmd'  
bootd        - boot default, i.e., run 'bootcmd'  
bootm        - boot application image from memory  
bootp        - boot image via network using BOOTP/TFTP protocol  
chpart       - change active partition  
cmp          - memory compare  
coninfo      - print console devices and information  
cp           - memory copy  
crc32        - checksum calculation  
dhcp         - boot image via network using DHCP/TFTP protocol  
echo         - echo args to console  
fatinfo      - print information about filesystem  
fatload      - load binary file from a dos filesystem  
fatls        - list files in a directory (default /)  
go           - start application at address 'addr'  
help         - print online help  
iminfo       - print header information for application image  
imxtract     - extract a part of a multi-image  
itest        - return true/false on integer compare  
loadb        - load binary file over serial line (kermit mode)  
loads        - load S -Record file over serial line  
loady        - load binary file over serial line (ymodem mode)  
loop         - infinite loop on address range  
md           - memory display  
mii          - MII utility commands  
mm           - memory modify (auto-incrementing address)  
mmc          - MMC sub system  
mmcinfo      - mmcinfo <dev num>-- display MMC info
```



mtdparts	- define flash/nand partitions
mtest	- simple RAM read/write test
mw	- memory write (fill)
mxs_mmc	- MXS specific MMC sub system
nand	- NAND sub-system
nboot	- boot from NAND device
nfs	- boot image via network using NFS protocol
nm	- memory modify (constant address)
ping	- send ICMP ECHO_REQUEST to network host
printenv	- print environment variables
arpboot	- boot image via network using RARP/TFTP protocol
reset	- Perform RESET of the CPU
run	- run commands in an environment variable
saveenv	- save environment variables to persistent storage
setenv	- set environment variables
sleep	- delay execution for some time
source	- run script from memory
tftpboot	- boot image via network using TFTP protocol
ubi	- ubi commands
ubifsload	- load file from an UBIFS filesystem
ubifsls	- list files in a directory
ubifsmount	- mount UBIFS volume
version	- print monitor version

其中 NAND Flash 操作另有一系列命令，可使用 help nand 查看。

MX28 U-Boot > help nand

nand - NAND sub-system

Usage:

nand info	- show available NAND devices
nand device [dev]	- show or set current device
nand read	- addr off[partition size]
nand write	- addr off[partition size] read/write 'size' bytes starting at offset 'off' to/from memory address 'addr', skipping bad blocks.
nand erase [clean] [off size]	- erase 'size' bytes from offset 'off' (entire device if not specified)
nand bad	- show bad blocks
nand dump[.oob] off	- dump page
nand scrub	- really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...]	- mark bad block(s) at offset (UNSAFE)
nand biterr off	- make a bit error at offset (UNSAFE)

10.4.1 预设的组合命令

使用 U-Boot 提供的基本命令，可以完成对系统的操作，但是如果每次操作都输入一系



列命令，既繁琐又容易出错。为此，光盘中的 U-Boot 预设了一些组合命令，可以方便快速地实现系统固化、升级更新等操作。

这些组合命令预存于 U-Boot 的环境变量中，进入 U-Boot 命令行后输入 printenv 可以查看已经预设的组合命令。输入“run 组合命令”即可运行这些组合命令。

更新内核 - 系统预设了一条 upkernel 命令，能够完成从 tftp 服务器加载 uImage 内核文件、擦除 NAND Flash、烧写内核以及设置内核参数等一系列工作，命令如下：

```
upkernel=tftp $(loadaddr) $(serverip):$(kernel);nand erase clean $(kerneladdr) $(kernelsize);nand write.jffs2  
$(loadaddr) $(kerneladdr) $(kernelsize);
```

更新文件系统 - 系统预设了一条 uproootfs 命令，能够完成从 tftp 服务器加载 rootfs.ubifs 文件、擦除 NAND Flash 和烧写文件系统等一系列工作，命令如下：

```
uproootfs=mtdparts default;nand erase rootfs;ubi part rootfs;ubi create rootfs;tftp $(loadaddr) $(rootfs);ubi write  
$(loadaddr) rootfs $(filesize)
```

从 NAND Flash 加载内核并启动的预设命令是 nand_boot：

```
nand_boot=nand read.jffs2 $(loadaddr) $(kerneladdr) $(kernelsize);bootm $(loadaddr)
```

注意：当前发布的 U-Boot 不支持 run upuboot 命令，强制执行该命令将会对存储的固件数据造成破坏。

10.5 U-Boot Tools

U-Boot 提供了一些有用的小工具，存放在 U-Boot 源码目录下的 tools 文件夹中。这些工具都是在主机上使用的。编译完毕，可以将这些小工具复制到系统目录（如/usr/bin）中，以方便使用。

其中的 mkimage 工具，在编译内核的时候需要用到，务必复制到系统/usr/bin 目录下（**如使用 ZLG 官网提供的 ubuntu，不需这一步**），或者将 U-Boot 的 tools 目录添加到 PATH 环境变量当中。该工具可以生成 U-Boot 格式的文件，以配合 U-Boot 使用。

先进入 tools 目录，复制 mkimage 到/usr/bin 目录

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08$ cd tools/  
vmuser@Linux-host: ~/bootloader/u-boot-2009.08/tools$ cp mkimage /usr/bin/
```



第11章 内核编译及其驱动要点

本章主要介绍了开发套件 Linux 内核源码的编译、剪裁，以及开发套件系统固件的制作。同时还介绍了一些驱动的使用和编写过程。

本章所引用的内核源码位于光盘“3、Linux\6、源代码”目录下，源码包的具体名称为“linux-2.6.35.3.tar.bz2”；而驱动源码则位于光盘“3、Linux\4、开发示例\6、驱动示例”目录下。

11.1 编译内核

参考 10.5 节内容准备好 mkimage 文件，将其复制到/usr/bin/目录下（使用 ZLG 官网提供的 ubuntu 则不需要操作这一步）。

```
vmuser@Linux-host: ~/bootloader/u-boot-2009.08/tools$ sudo cp mkimage /usr/bin/
```

11.1.1 解压内核文件

请把光盘中的“linux-2.6.35.3.tar.bz2”复制到 Linux 主机硬盘的工作目录，然后解压该压缩包：

```
vmuser@Linux-host: ~$ tar -jxvf linux-2.6.35.3.tar.bz2
```

解压完成之后得到“linux-2.6.35.3”目录，运行以下命令，进入该目录：

```
vmuser@Linux-host: ~$ cd linux-2.6.35.3
```

11.1.2 设置内核对应的型号

由于 EasyARM-i.MX280A、EasyARM-i.MX283A、EasyARM-i.MX287A 使用同一份内核代码，所以我们在配置、编译内核代码之前，需要先选择我们的设备型号。进入内核源码的根目录后，输入命令：

```
vmuser@Linux-host: ~/linux-2.6.35.3$ ./config-kernel
```

该命令将打印如图 11.1 所示的菜单。

```
Linux kernel build menu, please select your choice:  
1 make distclean  
2 make menuconfig  
3 config as EasyARM-i.MX280A  
4 config as EasyARM-i.MX283A or EasyARM-i.MX287A  
5 config as EasyARM-i.MX287B  
8 copy zImage to tftp dir  
q exit
```

图 11.1 config-kernel 菜单

这里用户可以根据开发套件的型号输入对应的数字，其中：

3——把内核配置成 EasyARM-i.MX280A 使用；

4——把内核配置成 EasyARM-i.MX283A 或 EasyARM-i.MX287A 使用（这两个型号的内核固件使用相同的配置）。

5——把内核配置成 EasyARM-i.MX287B 使用。

然后再按“Enter”键确认，如图 11.2 所示。



```
./config-kernel
Linux kernel build menu, please select your choice:
1 make distclean
2 make menuconfig
3 config as EasyARM-i.MX280A
4 config as EasyARM-i.MX283A or EasyARM-i.MX287A
5 config as EasyARM-i.MX287B
6 copy zImage to tftp dir
q exit
4
select 4
arch/arm/mach-mx28/Kconfig:18:warning: defaults for choice values not supported
drivers/net/wireless/bcmhd/Kconfig:45:warning: defaults for choice values not supported
#
# configuration written to .config
#
# config as EasyARM-i.MX283A or EasyARM-i.MX287A, you can use 'make menuconfig' to config your kernel, or use 'make uImage/make zImage' to build your kernel
```

图 11.2 选择设备型号

11.1.3 备份内核配置文件

注意：默认的内核配置文件为.config，如需修改内核配置，请提前备份该文件。具体方法为在“linux-2.6.35.3”目录执行以下命令（假如您的设备是EasyARM-i.MX283A）：

```
vmuser@Linux-host: ~/linux-2.6.35.3$ cp .config EasyARM-iMX283A_backup_defconfig
```

欲恢复默认内核配置时，只需拷贝回原来的.config文件即可：

```
vmuser@Linux-host: ~/linux-2.6.35.3$ cp EasyARM-iMX283A_bakcup_defconfig .config
```

“EasyARM-iMX283A_bakcup_defconfig”只是示例名字，用户可以自行定义。此外在“arch/arm/configs”目录下也有备份的配置文件。

11.1.4 编译内核

在“linux-2.6.35.3”目录下执行“make uImage”命令即可编译。编译完成后将在“arch/arm/boot”目录下生成内核固件uImage。

11.2 生成 imx28_ivt_linux.sb 内核固件

imx28_ivt_linux.sb 内核固件可以让系统直接从 Linux 内核启动，因此不需要 U-Boot 的引导，从而可减少系统的开机时间。

制作 imx28_ivt_linux.sb 固件首先需要 zImage 文件。在内核代码目录输入“make zImage”命令进行编译：

```
vmuser@Linux-host: $ cd linux-2.6.35.3
vmuser@Linux-host: ~/linux-2.6.35.3$ make zImage
```

编译完成后将在 arch/arm/boot/目录下生成 zImage 文件。把这个 zImage 文件复制到 imx-bootlets-src-10.12.01 目录下（见 10.3 节）。进入 imx-bootlets-src-10.12.01 目录，然后执行“./build”命令：

```
vmuser@Linux-host: ~/linux-2.6.35.3$ cp arch/arm/boot/zImage ..../bootloader/imx-bootlets-src-10.12.01
vmuser@Linux-host: ~/linux-2.6.35.3$ cd ..../bootloader/imx-bootlets-src-10.12.01
vmuser@Linux-host: ~/bootloader/imx-bootlets-src-10.12.01$ ./build
```

命令执行完成后，将在 imx-bootlets-src-10.12.01 目录下生成 imx28_ivt_linux.sb 固件。

Linux 内核启动时，是需要传入启动参数的；当 Linux 内核通过 U-Boot 来引导启动时，启动参数由 U-Boot 来传递。但是若系统直接从 Linux 内核启动，其启动参数则由 bootlets 中的 linux_prep 传递。

在 imx-bootlets-src-10.12.01 目录下的 linux_prep/cmdlines/iMX28_EVK.txt 文件内容如下：



```
gpmi=g console=ttyAM0,115200n8 console=tty0 ubi.mtd=1 root=ubi0:rootfs rootfstype=ubifs fec_mac=ethact
```

linux_prep/board/iMX28_EVK.c 文件的 cmdline_def 变量(在该文件的最后一行)的值为:

```
char cmdline_def[] = "gpmi=g console=ttyAM0,115200n8 ubi.mtd=1 root=ubi0:rootfs rootfstype=ubifs  
fec_mac=ethact";
```

当用户需要修改 imx28_ivt_linux.sb 的启动参数时,这两个地方均要修改,而且内容要保持一致。

imx28_ivt_linux.sb 固件生成后,即可进行固件烧写。请把该固件文件替换到“3、Linux\5、Linux 系统恢复\MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录,然后通过 TF 卡或者 USB 方式烧写。

11.3 配置内核

Linux 内核源码具有高可配置性。用户按 11.1.2 章节所示方法配置了相应的型号后,就可以根据自己的需要对内核进行裁减,或者添加自己所需要的驱动。

输入 make menuconfig 命令即可打开内核的配置界面,如图 11.3 所示。

```
$ make menuconfig
```

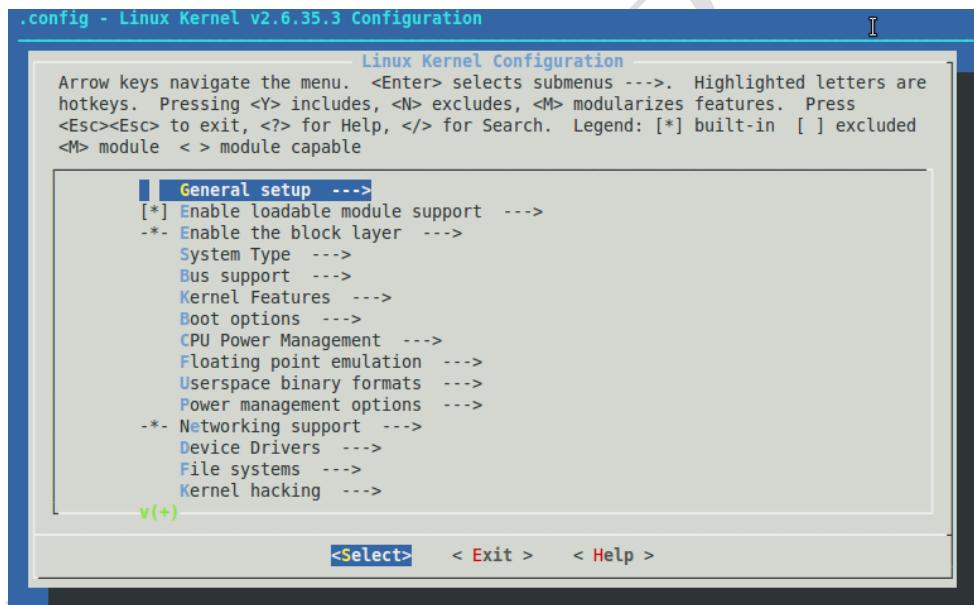


图 11.3 内核配置主菜单

在该界面中用户可以随意配置内核。键盘的“上”和“下”方向键控制选择光标在菜单的移动,当光标移动到某一个菜单项时,按下 Enter 键即可进入该菜单项的子菜单;键盘的“左”和“右”方向键选择 Select、Exit、Help。选择 Exit 后,按下 Enter 键将返回上一层菜单;当选择 Help 时,按下 Enter 键可查看该菜单的帮助内容。

如需启用某项功能,请将光标移动到该条目后按 y 键。这个操作会把这项功能静态编译进内核,如下所示:

```
[*] Network device support    --->
```

如果在光标选中该条目后按 M 键时,则会把此项功能动态编译成内核模块,在相应的目录中生成*.ko 文件,如下图所示。

```
<M> Serial ATA and Parallel ATA drivers    --->
```

1. 启用 TF 卡驱动

进入 Device Drivers 子菜单，如图 11.4 所示：

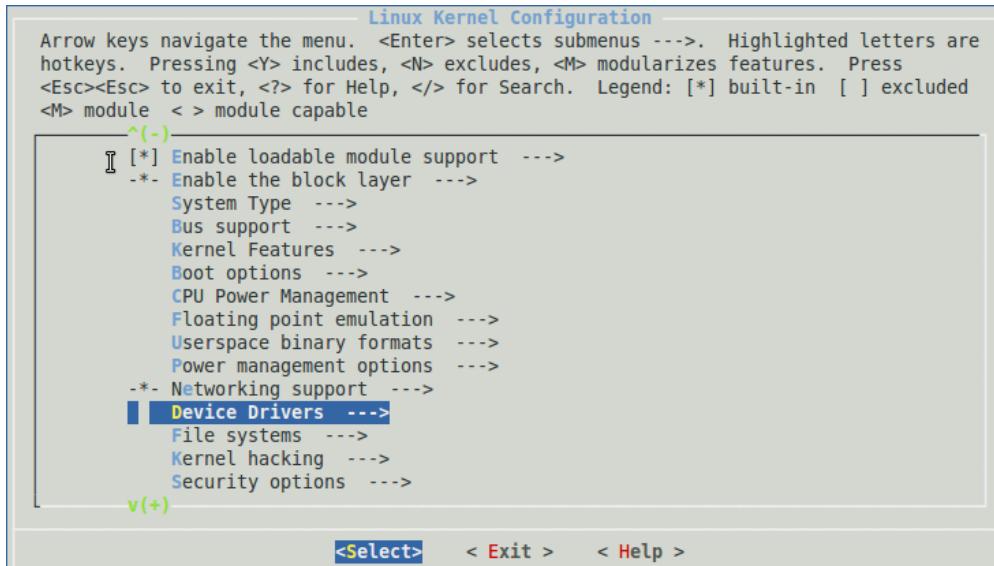


图 11.4 进入 Device Drivers 子菜单

选中 Block devices，如图 11.5 所示：

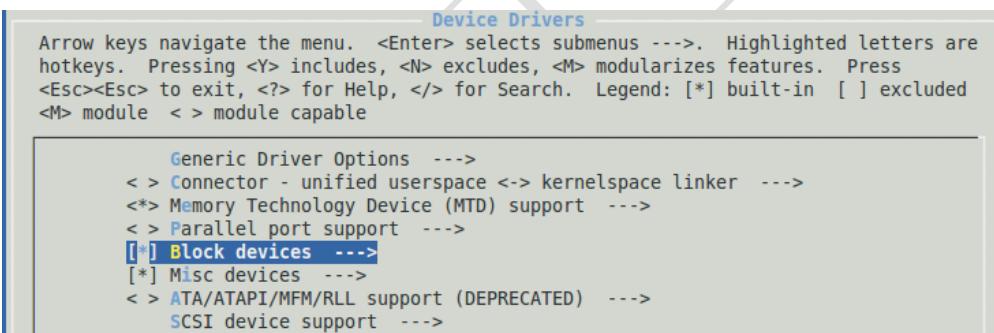


图 11.5 选中 Block devices 支持

进入 MMC/SD/SDIO card support 子菜单，如图 11.6 所示：

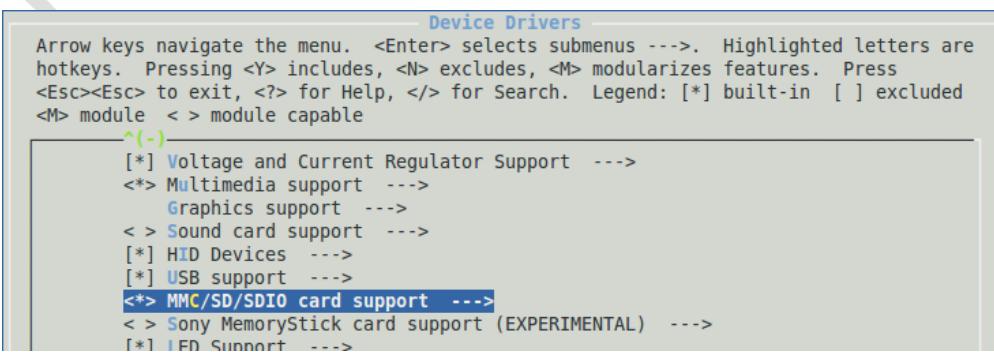


图 11.6 进入 MMC/SD 配置子菜单

将 MMC/SD/SDIO card support 子菜单内各模块选中情况设置为图 11.7 所示：



```
MMC/SD/SDIO card support
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

-- MMC/SD/SDIO card support
[ ] MMC debugging
[*] Assume MMC/SD cards are non-removable (DANGEROUS)
*** MMC/SD/SDIO Card Drivers ***
<*> MMC block device driver
[*] Use bounce buffer for simple hosts
< > SDIO UART/GPS class support
< > MMC host test driver
*** MMC/SD/SDIO Host Controller Drivers ***
< > Secure Digital Host Controller Interface support
< > MMC/SD/SDIO over SPI
[ ] Freescale i.MX Secure Digital Host Controller Interface PIO mode
<*> MXS MMC support
```

图 11.7 TF 卡模块配置

2. 启用 USB host

进入 Device Drivers 子菜单，如图 11.8 所示：

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

^(-)
[*] Enable loadable module support --->
-- Enable the block layer --->
  System Type --->
  Bus support --->
  Kernel Features --->
  Boot options --->
  CPU Power Management --->
  Floating point emulation --->
  Userspace binary formats --->
  Power management options --->
-- Networking support --->
  Device Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
v(+)

<Select> < Exit > < Help >
```

图 11.8 进入 Device Drivers 子菜单

进入 USB support 子菜单，如图 11.9 所示：

```
Device Drivers
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

^(-)
[*] Voltage and Current Regulator Support --->
<*> Multimedia support --->
  Graphics support --->
< > Sound card support --->
[*] HID Devices --->
[*] USB support --->
<*> MMC/SD/SDIO card support --->
< > Sony MemoryStick card support (EXPERIMENTAL) --->
[*] LED Support --->
[ ] Accessibility support --->
```

图 11.9 进入 USB support 子菜单



选中如图 11.10 所示的两个模块:

```
USB support
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

--- USB support
<*> Support for Host-side USB
[ ] USB verbose debug messages
[ ] USB announce new devices
    *** Miscellaneous USB options ***
[ ] USB device filesystem (DEPRECATED)
[ ] USB device class-devices (DEPRECATED)
[ ] Dynamic USB minor allocation
[*] USB runtime power management (suspend/resume and wakeup)
    Rely on OTG Targeted Peripherals List
    Disable external hubs
< > USB Monitor
< > Enable Wireless USB extensions (EXPERIMENTAL)
< > Support WUSB Cable Based Association (CBA)
    *** USB Host Controller Drivers ***
```

图 11.10 选中 Host 和 USB 电源管理两个模块

按图 11.11 和图 11.12 所示, 选中相应的模块:

```
USB support
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

^(-)
< > Cypress C67x00 HCD support
<*> EHCI HCD (USB 2.0) support
[*] Support for Freescale controller
[*] Support for Host1 port on Freescale controller
[*] Support for DR host port on Freescale controller
[ ] Use IRAM for USB
    Select transceiver for DR port (Internal UTMI) --->
[*] Root Hub Transaction Translators
[ ] Improved Transaction Translator scheduling (EXPERIMENTAL)
< > OXU210HP HCD support
< > ISP116X HCD support
< > ISP 1760 HCD support
```

图 11.11 按图选中各个 USB 子模块

```
USB support
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

^(-)
< > USB Wireless Device Management support
< > USB Test and Measurement Class support
    *** NOTE: USB STORAGE depends on SCSI but BLK_DEV_SD may ***
    *** also be needed; see USB_STORAGE Help for more info ***
<*> USB Mass Storage support
[ ] USB Mass Storage verbose debug
< > Datafab Compact Flash Reader support
< > Freecom USB/ATAPI Bridge support
< > ISD-200 USB/ATA Bridge support
< > USBAT/USBAT02-based storage support
```

图 11.12 选中 U 盘存储模块

3. 启用 I²C

在 Device Drivers 菜单中, 进入 I2C support 子菜单, 如图 11.13 所示:

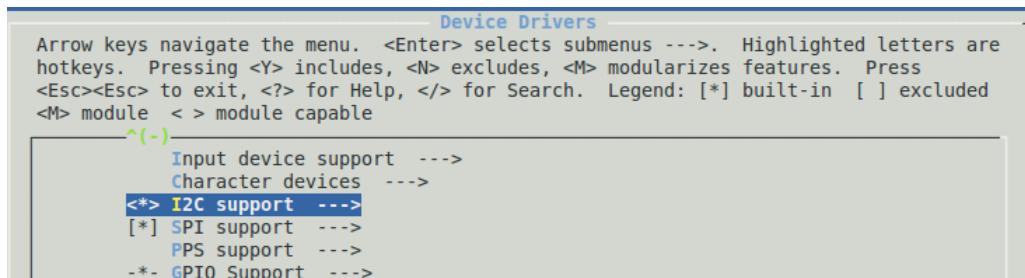


图 11.13 进入 I2C support 子菜单

按图 11.14 所示选中各个模块:

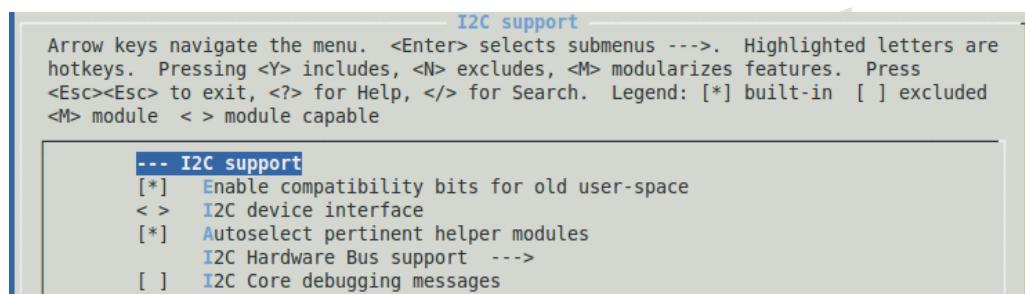


图 11.14 选中图中两个 I2C 子模块

进入 I2C Hardware Bus support 子菜单, 如图 11.15 所示:

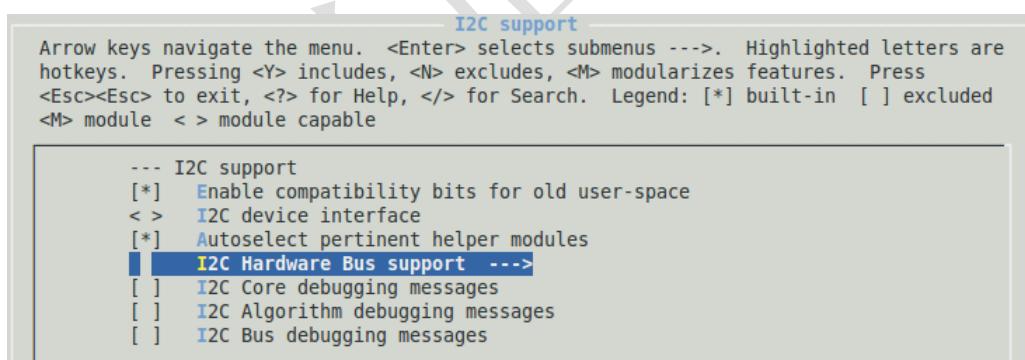


图 11.15 进入 I2C Hardware Bus support 子菜单

按图 11.16 所示选中各个子模块:

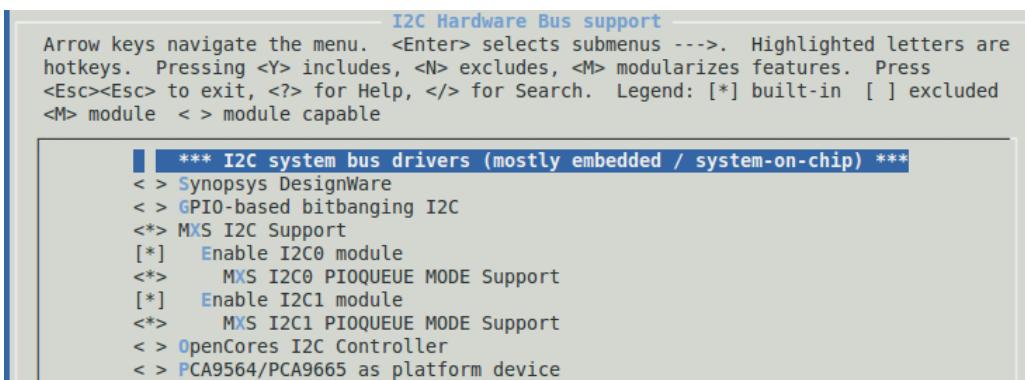




图 11.16 选中 I2C 各子模块

4. 启用 PCF8563

PCF8563 是使用 I²C 总线与处理器连接的 RTC 时钟芯片。由于 PCF8563 性能稳定，很受用户的欢迎。开发套件内核代码支持 PCF8563 连接到 I2C1 的应用。请进入“device drivers”的“Real Time Clock”如图 11.17 所示。

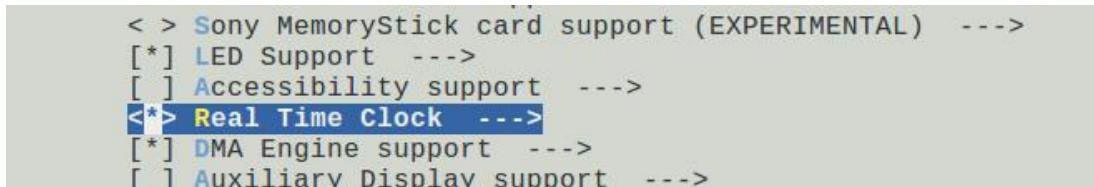


图 11.17 进入 RTC 菜单项

进入了“Real Time Clock”菜单项后，请关闭如图 11.18 所示的项。

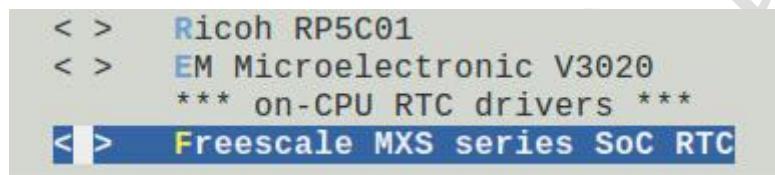


图 11.18 关闭处理器内部 RTC

然后打开 PCF8563 项，如图 11.19 所示。

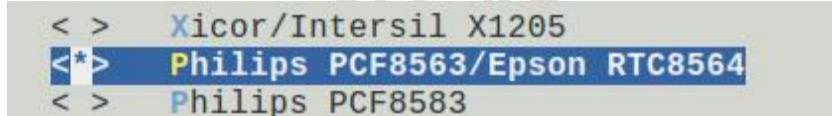


图 11.19 打开 PCF8563 项

重新编译并烧写内核。新内核启动后，会自动把 RTC 时间设置为系统的时钟。

说明：PCF8563 时钟芯片默认未焊接。

11.4 内核 GPIO 使用方法

本小节主要介绍如何在内核中使用 GPIO，其应用方法与第 13.1 章节介绍的应用层 GPIO 编程是完全不同的，请用户注意区分。

当需要在内核中把某一引脚用作 GPIO 时，需要先将其配置成 GPIO 功能模式，然后才能操作该 GPIO。而在内核中占用的 GPIO 将不能在应用层导出为 GPIO 使用。

下面以 GPIO1_17 为例，介绍 GPIO 的配置和操作。从 IMX28x 芯片手册中可以得知 GPIO1_17 的引脚名字是 LCD_D17，在内核代码的 arch/arm/mach-mx28/mx28_pins.h 文件中查阅到对 PINID_LCD_D17 引脚的定义如下：

```
#define PINID_LCD_D17          MXS_PIN_ENCODE(1, 17)
```

修改 arch/arm/mach-mx28/mx28evk_pins.c 文件中的 static struct pin_desc mx28evk_fixed_pins[] 数组中对引脚的描述就能设置引脚的功能。

如需要把 LCD_D17 引脚配置成 GPIO 功能，并设置其属性（如输出电压，输出电流等，具体可设置的属性需要参考《IMX28RM.pdf》手册），则需要对 PINID_LCD_D17 的描述做



如下的修改：

```
{  
    .name      = "RS485_DIR",           //自定义的名称  
    .id        = PINID_LCD_D17,         //引脚 ID  
    .fun       = PIN_GPIO,              //fun 选择 PIN_GPIO 功能  
    .strength  = PAD_8MA,  
    .voltage   = PAD_3_3V,  
    .drive     = 1,  
},
```

PIN_GPIO 定义在 arch/arm/plat-mxs/include/mach/pinctrl.h 文件中，如程序清单 11.1 所示。PIN_GPIO 的值为 3，对应 LCD_D17 引脚的 GPIO 功能模式（关于引脚模式配置的更多介绍，可查阅 i.MX283 芯片用户手册《IMX28RM.pdf》第 9 章 Pin Control and GPIO (PinCtrl)）。

程序清单 11.1 引脚的功能定义

```
enum pin_fun {  
    PIN_FUN1 = 0,  
    PIN_FUN2,  
    PIN_FUN3,  
    PIN_GPIO,  
};
```

将引脚修改为 GPIO 工作模式并设置好相应属性后，就可以调用 GPIOLIB 的通用函数来控制这个 GPIO 了。首先调用 gpio_request 获得这个 GPIO 的控制权：

```
gpio_request(MXS_PIN_TO_GPIO(PINID_LCD_D17), "RS485DIR");
```

当需要 LCD_D17 作输出功能使用时，调用 gpio_direction_output 设置该 GPIO 为输出模式：

```
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D17), 0); //设置为输出模式，初始化输出低电平
```

或

```
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D17), 1); //设置为输出模式，初始化输出高电平
```

然后调用 gpio_set_value 来设置输出高电平或低电平：

```
gpio_set_value(MXS_PIN_TO_GPIO(PINID_LCD_D17), 0);           // 控制输出低电平
```

或

```
gpio_set_value(MXS_PIN_TO_GPIO(PINID_LCD_D17), 1);           // 控制输出高电平
```

当需要 LCD_D17 作为输入功能使用时，调用 gpio_direction_input 设置该 GPIO 为输入模式：

```
gpio_direction_input(MXS_PIN_TO_GPIO(PINID_LCD_D17));
```

然后调用 gpio_get_value 获得 GPIO 的电平输入状态：

```
gpio_get_value(MXS_PIN_TO_GPIO(PINID_LCD_D17));
```

返回值为“1”表示输入为高电平；返回值为“0”表示输入为低电平。

若需要将 LCD_D17 这个引脚用作应用层 GPIO，即释放内核所占的 GPIO 资源，则需要将该引脚的描述信息从 pin_desc mx28evk_fixed_pins[] 数组中删除或注释掉，具体操作方法详见程序清单 11.2。



程序清单 11.2 释放内核所占 GPIO 资源示例

```
/*
{
    .name      = "RS485_DIR",                      //自定义的名称
    .id        = PINID_LCD_D17,                     //引脚 ID
    .fun       = PIN_GPIO,                          //fun 选择 PIN_GPIO 功能
    .strength  = PAD_8MA,
    .voltage   = PAD_3_3V,
    .drive     = 1,
},
*/
```

注意：在 `pin_desc mx28evk_fixed_pins[]` 数组中注册的引脚将无法在应用层导出其对应的 GPIO，若用户需要将某些未在内核中使用的功能复用引脚用作应用层的 GPIO，则可以通过内核配置关闭复用的模块驱动，或修改 `pin_desc mx28evk_fixed_pins[]` 数组。

11.5 LED 驱动

开发套件板载有一个 Error LED，其原理图如图 11.20 所示。当 ERR 为低电平时，LED 点亮；当 ERR 为高电平时，LED 熄灭。

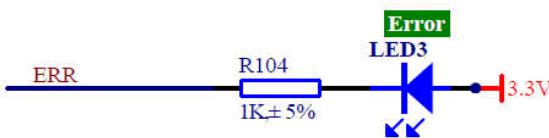


图 11.20 LED 原理图

根据 EasyARM-i.MX283A/287A 开发套件的原理图，ERR 是连接到处理器的 LCD_D23，该引脚可以复用为 GPIO1_23（对于 EasyARM-i.MX280A，ERR 连接到处理器的 GPIO3_20 上）。这个引脚的定义在内核源码的 `arch/arm/mach-mx28/mx28_pins.h` 文件，如下所示：

```
#define PINID_LCD_D23 MXS_PIN_ENCODE(1, 23)
```

这里仅以此为例，说明在 Linux 驱动代码中如何操作一个 GPIO 引脚。我们先对这个 LED 控制引脚作一个宏定义，以方便驱动的升级：

```
#define LED_GPIO MXS_PIN_TO_GPIO(PINID_LCD_D23)
```

根据这个定义，并结合上一节的（[内核 GPIO 使用方法](#)）内容，使用 `gpio_direction_output()` 函数便可控制 LED_GPIO 引脚输出高/低电平，以实现对 Error LED 的控制，如下所示：

```
gpio_direction_output(LED_GPIO, 1);           //输出高电平，LED 熄灭  
gpio_direction_output(LED_GPIO, 0);           //输出低电平，LED 点亮
```

LED 驱动代码在光盘资料“3、Linux\4、开发示例\6、驱动示例\4、led”目录下的 `led.c` 文件中。该文件引用了内核源码 `arch/arm/mach-mx28/mx28_pins.h` 文件，如下所示：

```
#include <../arch/arm/mach-mx28/mx28_pins.h>
```

下面详细介绍这个 `led.c` 驱动程序代码文件。

11.5.1 驱动加载

这个 LED 驱动模块文件是 `led.ko`。可使用下面的指令把驱动模块加载到内核：



```
root@EasyARM-iMX28x ~# insmod led.ko
```

在运行该命令时，led.c 文件中的 module_init 宏指向的 gpio_init 函数将被调用，以实现对 LED 驱动的初始化工作。gpio_init 函数的实现如程序清单 11.3 所示。

程序清单 11.3 驱动的初始化操作

```
static int __init gpio_init(void)
{
    misc_register(&gpio_misctdev);
    printk(DEVICE_NAME" up.\n");
    return 0;
}
module_init(gpio_init);
```

在该函数中，调用了 misc_register 函数注册一个 miscdevice 类型的 gpio_misctdev 对象。该对象的实现如程序清单 11.4 所示。

程序清单 11.4 gpio_misctdev 的实现

```
static struct miscdevice gpio_misctdev = {
    .minor  = MISC_DYNAMIC_MINOR,
    .name   = DEVICE_NAME,
    .fops   = &gpio_fops,
};
```

miscdevice 类型对象是描述 misc 类型设备的驱动。Linux 对大多数的驱动做了分类：块设备驱动、网络驱动、I²C 驱动、USB 驱动、SPI 驱动、音频驱动等，而一些不好分类的则都归纳为 misc 类型设备驱动。这个 Error LED 是自定义的，所以它的驱动应该归类于 misc 设备驱动。

misc 设备的驱动程序主设备号都是 10，其子设备号是由 minor 成员指定。若 minor 成员赋值为 MISC_DYNAMIC_MINOR，那么系统对其次设备号进行动态分配。为避免驱动注册时与已注册的 misc 类驱动的次设备号相冲突，一般采用次设备号动态分配方式。

gpio_misctdev 对象的 name 成员赋值为 DEVICE_NAME。DEVICE_NAME 的定义为：

```
#define DEVICE_NAME "imx28x_led"
```

当 gpio_misctdev 对象被注册后，将在文件系统中生成设备文件节点：

```
/dev/imx28x_led
```

gpio_misctdev 对象的 fops 成员指向的 gpio_fops 对象，是一个 file_operations 类型的文件操作列表。其实现如程序清单 11.5 所示。

程序清单 11.5 gpio_fops 的实现

```
static struct file_operations gpio_fops={
    .owner        = THIS_MODULE,
    .open         = gpio_open,
    .write        = gpio_write,
    .release      = gpio_release,
    .ioctl        = gpio_ioctl,
};
```



文件操作列表的实现是驱动程序实现的关键部分，它主要用于提供对设备文件进行相关操作的方法。例如，当应用程序对设备文件执行如下操作时，其对应的成员指向的函数就会被调用：

- 执行 open 操作时，文件操作列表的 open 成员指向的函数将被调用；
- 执行 write 操作时，文件操作列表的 write 成员指向的函数将被调用；
- 执行 ioctl 操作时，文件操作列表的 ioctl 成员指向的函数将被调用；
- 执行 close 操作时，文件操作列表的 release 成员指向的函数将被调用。

11.5.2 卸载驱动

在 Shell 终端执行下面指令时，将卸载 LED 驱动模块：

```
root@EasyARM-iMX28x ~# rmmod led.ko
```

当执行该指令时，led.c 文件中 module_exit 宏指向的 gpio_exit 函数将被调用，以实现驱动模块的退出操作。gpio_exit 函数的实现如程序清单 11.6 所示。

程序清单 11.6 gpio_exit 函数的实现

```
static void __exit gpio_exit(void)
{
    misc_deregister(&gpio_miscdev);
    printk(DEVICE_NAME " down.\n");
}
module_exit(gpio_exit);
```

gpio_exit 函数的主要目的是把初始化时注册的 gpio_miscdev 对象注销。当 gpio_miscdev 对象注销后，文件系统中的 /dev/imx28x_led 设备文件也随之消失。

11.5.3 open 调用的实现

当应用程序对 /dev/imx28x_led 设备文件执行 open 操作时，文件操作列表的 open 成员指向的 gpio_open 函数将被执行。gpio_open 函数的实现如程序清单 11.7 所示。

程序清单 11.7 gpio_open 函数的实现

```
static int gpio_open(struct inode *inode, struct file *filp)
{
    gpio_request(LED_GPIO, "led");
    return 0;
}
```

该函数主要实现了向内核申请这个 GPIO 端口，同时把该引脚配置成 GPIO 工作模式。

当应用程序对 /dev/imx28x_led 设备文件执行的 open 调用完成后，如果没有出错，将获得 fd (int 类型的) 文件描述符。后面的操作都是对这个 fd 文件描述符进行操作。

11.5.4 write 调用的实现

当应用程序对 fd 文件描述符执行 write 调用的时候，文件操作列表的 write 成员指向的 gpio_write 函数将会被调用。gpio_write 函数的实现如程序清单 11.8 所示。

程序清单 11.8 gpio_write 函数的实现

```
ssize_t gpio_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
```



```
{  
    char data[2];  
    copy_from_user(data, buf, count);  
    gpio_direction_output(LED_GPIO, data[0])  return count;  
}
```

在该函数的传入参数中有 buf 和 count 参数。buf 表示应用程序调用 write 时，要写入数据的缓冲区；count 表示缓冲区中有效数据的长度。

注意，buf 缓冲区是在用户空间的，而 LED 驱动程序是运行在内核空间的，并且在内核空间的代码是不能直接访问用户空间的缓冲区的，所以需要使用 copy_from_user 宏把用户空间 buf 中要写入的数据复制到 data 缓冲区中。

至于应用程序要控制的 GPIO 输出电平值（高电平值为 1，低电平值为 0），则保存在 data[0]，然后调用 gpio_direction_output 函数来实施操作。

11.5.5 ioctl 函数的实现

当应用程序对 fd 文件描述符执行 ioctl 调用时，文件操作列表的 ioctl 成员指向的 gpio_ioctl 函数将会被调用。gpio_ioctl 函数的实现如程序清单 11.9 所示。

程序清单 11.9 gpio_ioctl 的调用

```
static int gpio_ioctl(struct inode *inode, struct file *filp, unsigned int command, unsigned long arg)  
{  
    int data;  
  
    switch (command) {  
    case 0:  
        gpio_direction_output(LED_GPIO, 0);  
        break;  
  
    case 1:  
        gpio_direction_output(LED_GPIO, 1);  
        break;  
    }  
    return 0;  
}
```

gpio_ioctl 的输入参数中，有 command 和 arg 参数。command 参数表示应用程序传入的控制命令；arg 参数表示控制命令的命令参数。在这里应用程序只会传入 LED 点亮命令（命令值为 0）或 LED 熄灭命令（命令值为 1），而没有命令参数。

gpio_ioctl 函数调用 gpio_direction_output 函数后，再根据命令控制 GPIO 引脚输出高/低电平。

11.5.6 close 调用的实现

当应用程序对 fd 文件描述符执行 close 调用时，文件操作列表中的 release 成员指向的 gpio_release 函数将会被调用。gpio_release 函数的实现如程序清单 11.10 所示。

程序清单 11.10 gpio_release 函数的实现



```
static int gpio_release(struct inode *inode, struct file *filp)
{
    gpio_free(LED_GPIO);
    return 0;
}
```

该函数的主要工作是释放 GPIO 引脚。

11.5.7 编译驱动代码

在光盘的“3、Linux\4、开发示例\6、驱动示例\4、led”目录下提供了 Makefile 文件，可进行本节 LED 驱动代码的编译工作。该 Makefile 的内容如程序清单 11.11 所示。

程序清单 11.11 Makefile 文件的实现

```
# Makefile2.6
ifeq ($(KERNELRELEASE),)
#kbuild syntax. dependency relationships of files and target modules are listed here.
#gpiodrv-objs := gpiodrv.c
obj-m := led.o
else
PWD := $(shell pwd)
KVER = 2.6.35.3
KDIR:=$(KERNEL_PATH)
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions
endif
```

当需要对这个驱动代码进行编译时，必须先编译光盘中提供的 Linux 内核代码（见前面的 11.1 小节）。内核编译完成后，把该驱动代码 Makefile 中的 *KDIR* 变量指向刚编译好的内核源码路径，也可以在命令行里面执行“*export KERNEL_PATH=内核代码目录*”来实现这一点。当然，在`~/.bashrc`文件里面添加环境变量 *KERNEL_PATH*，可以达到一次修改，多处使用的效果。操作如下：

```
vi /home/vmuser/.bashrc
export KERNEL_PATH=/home/vmuser/zlgmcu/EasyARM-iMX28x/AWorks/linux-2.6.35.3
```

然后在驱动代码的目录下，执行 make 命令即可完成驱动的编译，如图 11.21 所示。

```
zhuguojun@zlgmcu:~/EasyARM-iM283/drivers/led$ make
make -C /home/zhangguojun/EasyARM-iM283/kernel/linux-2.6.35.3 M=/home/zhangguojun/EasyARM-iM283/drivers/led modules
make[1]: 正在进入目录 '/home/zhangguojun/EasyARM-iM283/kernel/linux-2.6.35.3'
  CC [M] /home/zhangguojun/EasyARM-iM283/drivers/led/led.o
/home/zhangguojun/EasyARM-iM283/drivers/led/led.c: In function 'gpio_ioctl':
/home/zhangguojun/EasyARM-iM283/drivers/led/led.c:66: warning: unused variable 'data'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/zhangguojun/EasyARM-iM283/drivers/led/led.mod.o
  LD [M] /home/zhangguojun/EasyARM-iM283/drivers/led/led.ko
make[1]:正在离开目录 '/home/zhangguojun/EasyARM-iM283/kernel/linux-2.6.35.3'
zhuguojun@zlgmcu:~/EasyARM-iM283/drivers/led$
```

图 11.21 编译驱动

编译完成后，将得到 led.ko 驱动模块文件。



11.5.8 测试程序

led 目录下有一个 test 目录，内有本节 LED 驱动的测试程序代码。其中 led_test.c 文件的内容如程序清单 11.12 所示。

程序清单 11.12 LED 测试程序代码

```
#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 <limits.h>
#include <asm/ioctl.h>
#include <time.h>
#include <pthread.h>

int main(void)
{
    int fd;
    char buf[1] = {0};

    fd = open("/dev/imx28x_led", O_RDWR);
    if (fd < 0) {
        perror("open /dev/imx283_led");
    }

    printf("test write....\n");
    buf[0] = 0;
    write(fd, buf, 1);           // 控制 LED 点亮
    sleep(2);

    buf[0] = 1;
    write(fd, buf, 1);           // 控制 LED 熄灭
    sleep(1);

    printf("test ioctl..... \n");
    ioctl(fd, 0);                //控制 LED 点亮
    sleep(2);
    ioctl(fd, 1);                //控制 LED 熄灭
}
```

输入 make 命令即可完成编译。编译完成后，将生成 led_test 文件。

LED 驱动和测试程序编译完成后，请把 led.ko 和 led_test 复制到 EasyARM-i.MX28x 开产品用户手册

©2013 Guangzhou ZLG MCU Technology Co., Ltd.



发套件的工作目录上进行测试。请输入下面命令：

```
root@EasyARM-iMX28x ~# insmod led.ko
imx283 up.
root@EasyARM-iMX28x ~# ./led_test
test write....
test ioctl.....
```

11.6 GPIO 中断示例程序

当把开发套件某一个具有 GPIO 功能的引脚，配置成 GPIO 功能模式并设置为输入工作状态时，该引脚就可以检测外部输入的中断信号。这里把开发套件上的“URX0”设置为中断信号输入检测引脚。URX0 是连接到 i.MX28x 处理器的 AUART0_RX 引脚，该引脚可以复用为 GPIO3_0。根据内核源码中“arch/arm/mach-mx28/mx28_pins.h”文件可知，该引脚的宏定义为：

```
#define PINID_AUART0_RX MXS_PIN_ENCODE(3, 0)
```

因此该引脚中断检测驱动的示例代码如程序清单 11.13 所示。

程序清单 11.13 中断检测示例代码

```
#include<linux/init.h>
#include<linux/module.h>
#include<mach/gpio.h>
#include<asm/io.h>
#include"mach/../../mx28_pins.h"
#include <mach/pinctrl.h>
#include "mach/mx28.h"
#include<linux/fs.h>
#include <linux/io.h>
#include<asm/uaccess.h>
#include<linux/miscdevice.h>
#include<linux/irq.h>
#include<linux/sched.h>
#include<linux/interrupt.h>
#include<linux/timer.h>

#include <linux/kernel.h>
#include <linux/delay.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define GPIO_BUTTON_PIN      MXS_PIN_TO_GPIO(PINID_AUART0_RX)

//中断处理函数
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
```



```
    printk("irq test \n");
    return IRQ_RETVAL(IRQ_HANDLED);
}

static int __init gpio_drv_init(void)                                //这是驱动加载时就会执行的函数
{
    int irq_no;
    int iRet;

    gpio_free(GPIO_BUTTON_PIN);
    iRet = gpio_request(GPIO_BUTTON_PIN, "irqtest");           //先申请 GPIO 口，同时
    if (iRet != 0) {                                         //把这个引脚配置成 GPIO
        printk("request gpio failed \n");
        return -EBUSY;
    }

    gpio_direction_input(GPIO_BUTTON_PIN);                      //把该 GPIO 口设置为输入模式
    irq_no = gpio_to_irq(GPIO_BUTTON_PIN);          //根据该 GPIO 获得其对应 IRQ 中断号，有了中断号
                                                    //才可以设置中断触发方式和注册中断处理函数
    set_irq_type(irq_no, IRQF_TRIGGER_FALLING); //设置中断触发方式为下降沿触发

    //申请中断并设置中断处理函数
    iRet = request_irq(irq_no, buttons_irq, IRQF_DISABLED, "gpio_int", NULL);
    if (iRet != 0){
        printk("request irq failed!! ret: %d  irq:%d gpio:%d  \n", iRet, irq_no, GPIO_BUTTON_PIN);
        return -EBUSY;
    }

    return 0;
}

static void __exit gpio_drv_exit(void)
{
    int irq_no;

    irq_no = gpio_to_irq(GPIO_BUTTON_PIN);
    free_irq(irq_no, NULL);
    gpio_free(GPIO_BUTTON_PIN);
}

module_init(gpio_drv_init);
module_exit(gpio_drv_exit);

MODULE_AUTHOR("EasyARM28xx By zhugojun");
```



```
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("gpio button interrupt module");
```

在上述代码中，当该驱动被加载进内核时，`gpio_drv_init` 函数就会被调用。该函数首先调用 `gpio_request` 函数申请 `AUART0_RX` 引脚对应的 GPIO 资源 (`GPIO3_0`)，同时把 `AUART0_RX` 配置成 GPIO 工作模式，然后调用 `gpio_direction_input` 函数把 `GPIO3_0` 设置为输入模式。这样 GPIO 就配置完成了，接下来就是设置中断触发模式和注册中断处理函数。

`buttons_irq` 就是我们注册的中断处理函数。当 `GPIO3_0` 有中断信号输入时，该函数就会被调用。同时，该函数会打印“irq test”。

把该驱动编译后，就会得到“`gpioInt.ko`”。执行下面命令加载驱动：

```
root@EasyARM-iMX28x ~# insmod gpioInt.ko
```

测试方法是：将杜邦线的一头接到“URX0”引脚上保持固定；而另一头首先接到“3.3V”引脚，然后立马转接到“GND”以模拟一次高电平到低电平跳变，即可实现“URX0”引脚的下降沿信号输入。这时，串口终端会打印相关的测试信息，如图 11.22 所示。

```
root@EasyARM-iMX283 /mnt# insmod gpioInt.ko
root@EasyARM-iMX283 /mnt# irq test
irq test
irq test
```

图 11.22 中断测试信息

11.7 设置 LCD 的时序

当用户想更换LCD屏时，也需要在内核中修改LCD的时序设置。在内核中，针对IMX28x平台的LCD的时序定义在 `drivers/video/mxs/lcd_43wvf1g.c` 文件，如程序清单 11.14 所示。

程序清单 11.14 LCD 时序列表

```
#define DOTCLK_H_ACTIVE 480 //屏幕 x 轴像素长度
#define DOTCLK_H_PULSE_WIDTH 41 //脉冲宽度
#define DOTCLK_HF PORCH 5 //水平前沿
#define DOTCLK_HB PORCH 5 //水平后沿
#define DOTCLK_H_WAIT_CNT (DOTCLK_H_PULSE_WIDTH + DOTCLK_HB PORCH)
#define DOTCLK_H_PERIOD (DOTCLK_H_WAIT_CNT + DOTCLK_HF PORCH + DOTCLK_H_ACTIVE)

#define DOTCLK_V_ACTIVE 272 //屏幕 y 轴像素长度
#define DOTCLK_V_PULSE_WIDTH 20 //脉冲宽度
#define DOTCLK_VF PORCH 5 //垂直前沿
#define DOTCLK_VB PORCH 5 //垂直后沿
#define DOTCLK_V_WAIT_CNT (DOTCLK_V_PULSE_WIDTH + DOTCLK_VB PORCH)
#define DOTCLK_V_PERIOD (DOTCLK_VF PORCH + DOTCLK_V_ACTIVE + DOTCLK_V_WAIT_CNT)
```

上述宏定义了 LCD 控制器输出的时序，LCD 控制器时序的具体设置流程可参考该文件中的 `init_panel` 函数。

该文件中的 `static struct mxs_platform_fb_entry fb_entry` 变量初始化如程序清单 11.15 所示，用于设置屏幕的分辨率、像素时钟。

程序清单 11.15 `fb_entry` 的定义与初始化



```
static struct mxs_platform_fb_entry fb_entry = {  
    .name = "HW480272F",                      // LCD 名字  
    .x_res = 272,                             // 屏幕 y 轴像素长度,  
    .y_res = 480,                             // 屏幕 x 轴像素长度  
    // 注意这里的 x、y 是与实际的屏的 x、y 对调的  
    .bpp = 16,                                // 显示颜色位数  
    .dclk_f = 8000000,                         // 像素时钟频率  
    .lcd_type = MXS_LCD_PANEL_DOTCLK,  
    .init_panel = init_panel,  
    .release_panel = release_panel,  
    .blank_panel = blank_panel,  
    .run_panel = mxs_lcdif_run,  
    .stop_panel = mxs_lcdif_stop,  
    .pan_display = mxs_lcdif_pan_display,  
    .bl_data = &bl_data,  
};
```

用户可以修改程序清单 11.14 的宏和 fb_entry 结构体中变量的值来修改 LCD 控制器的输出时序，实现所需要的 LCD 的驱动。程序清单 11.14 的宏定义了 LCD 的时序，这些值都可以在 LCD 的数据手册中查阅。像素时钟 fb_entry.dclk_f 可以参考 LCD 数据手册所推荐的值来进行设置，也可以通过以下计算公式计算：

```
pixclock=1012/(( DOTCLK_H_ACTIVE + DOTCLK_HF_PORCH + DOTCLK_HB_PORCH  
+DOTCLK_H_PULSE_WIDTH)  
*( DOTCLK_V_ACTIVE + DOTCLK_VF_PORCH + DOTCLK_VB_PORCH + DOTCLK_V_PULSE_WIDTH)  
* refresh)  
//refresh 一般为 60
```



第12章 根文件系统的打包及其简单应用

本章主要介绍开发套件根文件系统的打包、ubifs 镜像的制作以及 NFS 根文件系统的实现。

12.1 Linux 根文件系统

通常情况下，Linux 内核启动后期，会寻找并挂载根文件系统。根文件系统可以存在于磁盘上，也可以是存在于内存中的映像，它包含了 Linux 系统正常运行所必须的程序和库等等，并且这些文件都按照一定的目录结构存放。Linux 根文件系统大概包括如下内容：

- 基本的目录结构：/bin、/sbin、/dev、/etc、/lib、/var、/proc、/sys、/tmp 等；
- 基本程序运行所需的库文件，如 glibc 等；
- 基本的系统配置文件，如 inittab、rc 等；
- 必要的设备文件，如 /dev/ttyS0、/dev/console 等；
- 基本应用程序，如 sh、ls、cd、mv 等。

12.2 FHS 标准

理论上，Linux 根文件系统的目录结构是可以随意安排的，事实上很多 Linux 系统开发人员也这么做，但这就带来了不同开发人员之间不统一的问题，很容易出现混乱。后来这样的问题得到了重视，文件层次标准（FHS，Filesystem Hierarchy Standard）就在这种情况下出台的。FHS 经历了几个版本，目前最新版本是 2004 年 01 月 29 日发布的 V2.3 版本，详见 www.pathname.com/fhs。

FHS 对 Linux 根文件系统的基本目录结构做了比较详细的规定，尽管 FHS 不是强制标准，但事实上大部分 Linux 发行版都遵循这个标准。下面对 FHS V2.3 进行一些简要说明。

12.2.1 顶层目录

整个根文件系统都是挂在根目录（/）下，FHS 对顶层目录的要求和说明如表 12.1 所列。

表 12.1 FHS 顶层目录

目 录	说 明
bin	基本命令的二进制文件（所有用户可用），里面不能再包含目录
boot	Boot Loader 静态文件
dev	设备文件
etc	系统配置文件，这些文件必须是静态的；不能存放二进制文件
home	用户 home 目录
lib	基本的共享库和内核模块
media	可移动介质的挂载点
mnt	临时的文件系统挂载点
opt	附加的应用程序软件包
root	root 用户目录（可选）
sbin	基本的系统管理命令的二进制文件（仅 root 用户可用）
srv	系统服务的一些数据
tmp	临时文件



续上表

目 录	说 明
usr	该目录有二级标准
var	可变数据

12.2.2 “/usr” 目录

“/usr” 目录包含了系统很大一部分的内容，对其中的目录结构 FHS 也有相应地规定，如表 12.2 所列。

表 12.2 /usr 目录

目 录	说 明
bin	大部分的用户命令
include	C 程序所需要包含的头文件
lib	库文件
locale	本地层次（ 安装完毕后为空 ）
sbin	不是至关重要的系统命令（ 仅 root 用户可用 ）
share	独立数据
X11R6	X-Window 系统（ 可选 ）
games	游戏和教育相关的二进制文件
src	源代码（ 可选 ）

FHS 标准规定得相当详细，对一些目录以及深层子目录都做了详细的规定，对目录里面的文件也做了规定，更多的细节请参考 FHS 文档。事实上，很多 Linux 发行版也只是大体上遵循这个规范，不同发行商之间往往都会有一些差别。特别是在嵌入式领域，很多可选目录都被裁剪掉了，同时还会增加一些新的目录。

12.3 BusyBox

构建根文件系统就是根据系统的需要，将必要的命令或者文件集合起来，根据 FHS 的要求进行存放。根文件系统需要的命令、库文件等，可以考虑逐个单独编译得到，这是传统的 LFS (Linux From Scratch) 方式。这种方式操作起来比较复杂。后来 busybox 的出现改变了这一局面。

BusyBox 项目是一个遵循 GPLv2 的开源项目（[项目主页 www.busybox.net](http://www.busybox.net)），它将 100 多种 UNIX 命令和工具集成到一个可执行文件中，而这个可执行文件只有 1M 左右，被称为“嵌入式 Linux 的瑞士军刀”。BusyBox 的出现极大地简化了 Linux 根文件系统的构建，只需对 BusyBox 进行配置和编译/交叉编译，即可得到 Linux 系统运行所需的基本命令和库文件等。

12.4 NFS 根文件系统

Linux 内核支持从网络加载根文件系统，这对嵌入式 Linux 的开发非常有用，特别是在系统开发的初期阶段。将根文件系统放在主机上，方便文件系统的调整，也无需考虑文件系统的体积，等系统开发完毕后再进行裁剪即可。

使用 NFS 根文件系统，需要内核的网卡驱动能够正常工作，并且在内核中支持网络连接及 NFS 根文件系统功能，同时将内核参数设置为从 NFS 启动。设置内核 NFS 启动的参数一般格式为：

```
root=/dev/nfs rw console=$(consolectf) nfsroot=$(serverip):$(rootpath)
```



```
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname)::off
```

其中个参数的意义如下：

- consolecfg —— 调试串口配置；
- serverip —— NFS 服务器 IP；
- ipaddr —— 本机 IP（**目标系统 IP**）；
- gateway —— 网关；
- netmask —— 子网掩码；
- hostname —— 目标板的主机名；
- rootpath —— 主机 NFS 根文件系统路径。

如“使用调试串口屏 ttyAM0，NFS 服务器 IP 为 192.168.12.123，NFS 根文件系统路径为/nfsroot，目标板 IP 为 192.168.12.124”的启动参数配置为：

```
root=/dev/nfs rw console=ttyAM0,115200 nfsroot=192.168.12.123:/nfsroot      # 注意此处并未换行  
ip=192.168.12.124:192.168.12.123:192.168.12.1:255.255.255.0:epc.zlgmcu.com:eth0:off
```

把光盘附带的 rootfs_imx28x.tar.bz2（可在“3、Linux\4、开发示例\4、文件系统”目录下找到）复制 Linux 主机上的/nfsroot 目录，然后解压该包：

```
vmuser@Linux-host: /nfsroot$ tar -jxvf rootfs_imx28x.tar.bz2
```

解压完成后，将得到/nfsroot/rootfs 目录，它将用作 NFS 根文件系统的目录。重新启动开发套件并进入 U-Boot 的命令行（参考第 10.4 章节“U-Boot 基本命令”描述进入 U-Boot 命令的方法），设置 bootargs 环境变量：

```
MX28 U-Boot > setenv bootargs 'root=/dev/nfs rw console=ttyAM0,115200n8      # 注意此处并未换行  
nfsroot=192.168.12.123:/nfsroot/rootfs                          # 注意此处并未换行  
ip=192.168.12.124:192.168.12.123:192.168.12.1:255.255.255.0:epc.zlgmcu.com:eth0:off'  
MX28 U-Boot > saveenv                                     #保存环境变量  
MX28 U-Boot > reset                                      #重启 EasyARM-iMX283
```

系统重启完成后，就进入 Linux 主机上的/nfsroot/rootfs 所在的文件系统。

12.5 生成文件系统映像

12.5.1 生成 rootfs.ubifs 固件

系统开发后期，对根文件系统进行裁剪后，最终需要进行固化。根文件系统映像用什么样的文件系统，需要根据实际情况进行选择。目前内核可支持的文件系统为 UBIFS。在 Linux 内核源码中配备有 UBIFS 文件系统的实现代码。

可以按下面的方法制作针对开发套件的 UBIFS 根文件系统映像。

注意开发套件根文件所在分区的参数：分区大小为 240MB；页大小为 2048 字节(2KB)；擦除块大小为 128KB。

(1) 准备 UBIFS 文件系统映像制作工具

制作 UBIFS 文件系统映像，需要使用 mkfs.ubifs 和 ubinize 工具。在光盘的“3、Linux\4、开发示例\4、文件系统\ubi 工具”目录下有 mkfs.ubifs、ubinize 程序文件，分别提供了 32 位和 64 位的版本，请读者根据自己的实际情况选择适合的版本，然后将这两个文件复制到 Linux 主机下的/usr/sbin/目录下，并为它们添加可执行的权限：

```
vmuser@Linux-host: ~$ sudo chmod 777 /usr/sbin/mkfs.ubifs
```



```
vmuser@Linux-host: ~$ sudo chmod 777 /usr/sbin/ubinize
```

如果发现这两个工具不适用于自己的 ubuntu，也可以在有互联网络连接的情况下，使用 apt-get 命令在线安装：

```
vmuser@Linux-host: ~$ sudo apt-get install mtd-utils
```

(2) 准备根文件系统和配置文件

根文件系统可以由用户自己制作，也可以使用光盘所提供的已经制作好的。如果想要使用光盘提供的根文件系统，请把光盘“3、Linux\4、开发示例\4、文件系统”目录下的 rootfs_imx28x.tar.bz2、build_rootfs 以及 ubinize.cfg 文件，复制到 Linux 主机上的~/filesystem 目录中。然后执行下面命令：

```
vmuser@Linux-host: ~/filesystem$ tar -jxvf rootfs_imx28x.tar.bz2 # 解压 rootfs.tar.bz2 文件  
vmuser@Linux-host: ~/filesystem$ chmod 777 build_rootfs # 给 build_rootfs 文件添加可执行权限
```

(3) 生成 ubi 根文件系统

build_rootfs 文件是脚本程序，其内容是：

```
mkfs.ubifs -r rootfs -m 2048 -e 126976 -c 1900 -o ubifs.img  
ubinize -o ubi.img -m 2048 -p 128KiB -s 512 ubinize.cfg  
mv ubifs.img rootfs.ubifs
```

执行该脚本程序就能生成根文件系统镜像：

```
vmuser@Linux-host: ~/filesystem$ ./ build_rootfs
```

此时生成的 rootfs.ubifs 文件就是所需的 ubi 根文件系统映像，把该文件复制 tftp 服务器的根目录，就可以供开发套件更新文件系统。

12.5.2 生成 rootfs.tar.bz2 固件

把光盘中“3、Linux\4、开发示例\4、文件系统”目录下的 rootfs_imx28x.tar.bz2 文件复制到 Linux 主机的~/filesystem 目录，然后执行下面命令：

```
vmuser@Linux-host: ~/filesystem$ tar -jxvf rootfs_imx28x.tar.bz2 # 解压 rootfs.tar.bz2 文件  
vmuser@Linux-host: ~/filesystem$ cd rootfs # 进入 rootfs  
vmuser@Linux-host: ~/filesystem/rootfs$ tar -cjvf rootfs.tar.bz2 * # 生成 rootfs.tar.bz2 固件
```

rootfs.tar.bz2 固件生成后，可以将其复制到光盘的“3、Linux\5、Linux 系统恢复\MfgTool 1.6.2.055-ZLG140813\Profiles\MX28 Linux Update\OS Firmware\files”目录下并覆盖原有文件，然后参考第 9 章内容通过 USB 或者 TF 卡进行烧写。

12.6 修改预置的开机启动设置

在 8.4.1 章节中讲到开发套件 Linux 系统中的/etc/rc.d/init.d/start_userapp 文件为开机时自动执行的脚本。当用户需要修改预置的开机启动设置时，可以通过在生成文件系统映像前修改该文件来实现。



第四篇 Linux 应用开发

这一篇内容围绕开发套件 Linux 系统的应用开发来展开。首先介绍开发套件功能部件的应用编程，然后介绍基于嵌入式 Linux Qt 库的图形应用程序开发。

一共分为 2 章，各章的标题和大概内容如下：

第 13 章：功能部件应用编程，主要介绍开发套件的 ADC、GPIO、UART、I2C、SPI 及 PWM 等接口的编程；

第 14 章：嵌入式 Linux Qt 编程，主要介绍如何使用开发套件进行基于嵌入式 Linux Qt 库的图形应用程序开发。

广州周立功



第13章 功能部件应用编程

本章主要介绍 EasyARM-i.MX28xA 开发套件的 GPIO、ADC、UART、I2C、PWM、SPI 及 CAN 接口的编程。开始本章的学习之前，需要设置好交叉编译环境，并能交叉编译普通的应用程序。

所有的例程源码均可在“3、Linux\4、开发示例\2、功能部件”目录下找到。

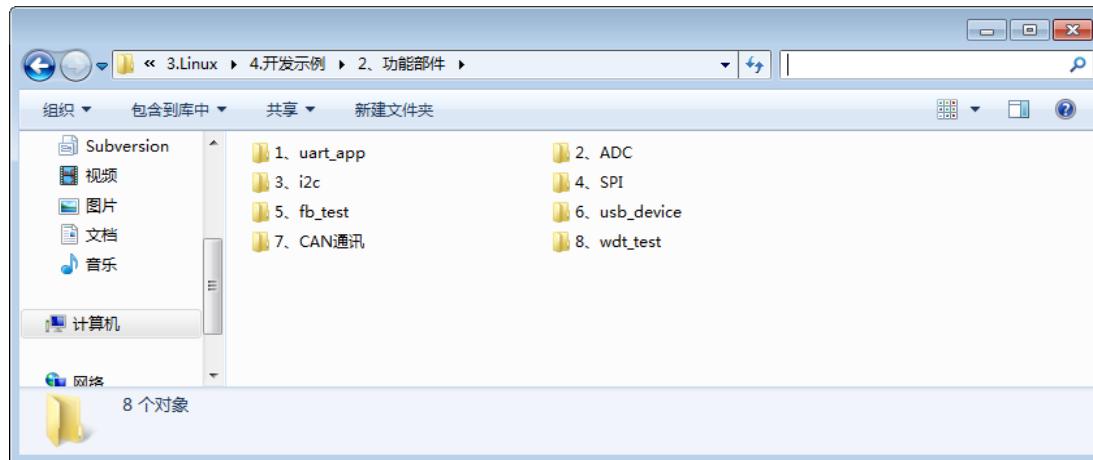


图 13.1 所有功能部件例程

编程规则约定：

涉及到设备操作的代码，均要包含以下头文件：

```
#include<fcntl.h>
#include<sys/ioctl.h>
```

使用了 printf 或 sleep 函数的代码，则需要包含以下头文件：

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

注：下文将不再赘述代码中头文件的使用问题。

EasyARM-i.MX28xA 泛指 EasyARM-i.MX280A、EasyARM-i.MX283A 及 EasyARM-i.MX287A。

13.1 GPIO 应用编程

开发套件有两排通过排针引出的管脚，EasyARM-i.MX283A 和 EasyARM-i.MX287A 开发套件上排针所对应的管脚是一样的，如图 13.2 所示。而对于 EasyARM-i.MX280A，其管脚功能如图 13.3 所示。图中标记了可以用作 GPIO 功能的所有引脚（RUN 引脚除外）。

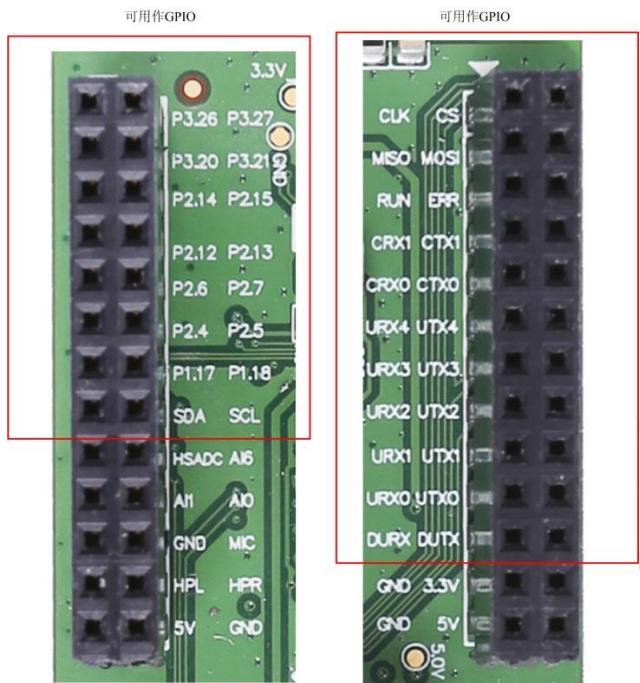


图 13.2 EasyARM-i.MX283(7)A 的 GPIO 功能排针

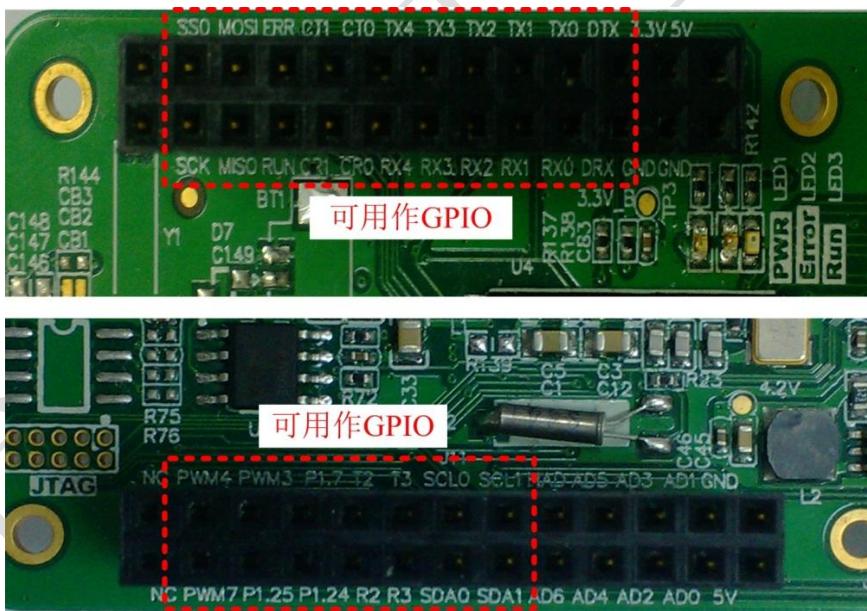


图 13.3 EasyARM-i.MX280A 的 GPIO 功能排针

在这些引脚中，有的在内核中已经复用为 UART、SPI 或 I^C 等，但为了方便 GPIO 的应用，开发套件示例工程中提供了通用 GPIO 驱动接口，对应的驱动文件名称为 `gpio_driver.ko`（源码位于“3、Linux\4、开发示例\6、驱动示例\2、gpio_driver”目录下）。通过 `gpio_driver.ko` 驱动接口，无需修改内核就可以操作相关 GPIO。

注意：

- 通过 `gpio_driver.ko` 驱动接口进行 GPIO 操作时，可以抢占内核驱动的 GPIO 资源，若所操作的 GPIO 在内核中已被复用为其他功能，则内核中的驱动可能失效，用户需自行规避这种应用冲突。



- gpio_driver.ko 驱动依赖开发套件使用的内核，使用非匹配的内核可能导致无法安装该驱动。
- 使用不同的内核编译配置（对应不同的开发套件），驱动加载后得到的设备文件可能不相同，请以实际情况为准。

13.1.1 驱动加载

通过 gpio_driver.ko 驱动接口操作 GPIO 前需要先加载 gpio_driver.ko 驱动，加载 gpio_driver.ko 驱动后才能得到具体的 GPIO 设备文件。

使用如下命令加载驱动：

```
root@EasyARM-iMX28x ~# insmod gpio_driver.ko
```

驱动加载时会根据不同的内核编译配置（对应不同的开发套件，具体请参考 11.1.2 章节）在 /dev/ 目录生成对应的 GPIO 设备文件节点，如图 13.4 所示是将内核配置为“EasyARM-i.MX283A or EasyARM-i.MX287A”时生成的设备文件节点。请以用户实际开发套件所生成的设备文件节点为准。对于 EasyARM-i.MX280A，其生成的设备文件节点是不同的。

```
root@EasyARM-iMX283 ~# ls /dev/gpio-*
/dev/gpio-CLK      /dev/gpio-MOSI     /dev/gpio-P2.7    /dev/gpio-URX2
/dev/gpio-CRX0     /dev/gpio-P1.17   /dev/gpio-P3.20   /dev/gpio-URX3
/dev/gpio-CRX1     /dev/gpio-P1.18   /dev/gpio-P3.21   /dev/gpio-URX4
/dev/gpio-CS       /dev/gpio-P2.12   /dev/gpio-P3.26   /dev/gpio-UTX0
/dev/gpio-CTX0     /dev/gpio-P2.13   /dev/gpio-P3.27   /dev/gpio-UTX1
/dev/gpio-CTX1     /dev/gpio-P2.14   /dev/gpio-RUN    /dev/gpio-UTX2
/dev/gpio-DURX     /dev/gpio-P2.15   /dev/gpio-SCL    /dev/gpio-UTX3
/dev/gpio-DUTX     /dev/gpio-P2.4    /dev/gpio-SDA    /dev/gpio-UTX4
/dev/gpio-ERR      /dev/gpio-P2.5    /dev/gpio-URX0
/dev/gpio-MISO     /dev/gpio-P2.6    /dev/gpio-URX1
```

图 13.4 GPIO 设备文件节点

引脚和设备文件节点的关系如表 13.1 所示。

表 13.1 引脚和设备节点的关系

引脚	对应的设备文件节点	引脚	对应的设备文件节点
DURX	/dev/gpio-DURX	DUTX	/dev/gpio-DUTX
URX0	/dev/gpio-URX0	UTX0	/dev/gpio-UTX0
URX1	/dev/gpio-URX1	UTX1	/dev/gpio-UTX1
URX2	/dev/gpio-URX2	UTX2	/dev/gpio-UTX2
URX3	/dev/gpio-URX3	UTX3	/dev/gpio-UTX3
URX4	/dev/gpio-URX4	UTX4	/dev/gpio-UTX4
CRX0	/dev/gpio-CRX0	CTX0	/dev/gpio-CTX0
CRX1	/dev/gpio-CRX1	CTX1	/dev/gpio-CTX1
MISO	/dev/gpio-MISO	MOSI	/dev/gpio-MOSI
CLK	/dev/gpio-CLK	CS	/dev/gpio-CS
P3.26	/dev/gpio-P3.26	P3.27	/dev/gpio-P3.27
P3.20	/dev/gpio-P3.20	P3.21	/dev/gpio-P3.21



P2.14	/dev/gpio-P2.14	P2.15	/dev/gpio-P2.15
P2.12	/dev/gpio-P2.12	P2.13	/dev/gpio-P2.13
P2.6	/dev/gpio-P2.6	P2.7	/dev/gpio-P2.7
P2.4	/dev/gpio-P2.4	P2.5	/dev/gpio-P2.5
P1.17	/dev/gpio-P1.17	P1.18	/dev/gpio-P1.18
SDA	/dev/gpio-SDA	SCL	/dev/gpio-SCL

13.1.2 使用驱动接口

当 `gpio_driver.ko` 驱动加载后，开发套件上所有可以用作 GPIO 的引脚都有一个对应的设备文件节点。这时，`gpio_driver.ko` 驱动还没有对这些引脚产生任何的影响，只有当用户在应用程序中打开了某个引脚对应的 GPIO 设备文件节点时，`gpio_driver.ko` 驱动才会将这个引脚设置为 GPIO 功能（注意：一旦打开了这个设备文件节点后，除非重启开发套件，否则对应引脚的功能配置将无法再恢复到原来的功能模式，因此用户在应用此驱动时要避免功能应用上的冲突）。

1. 打开操作

在用户应用程序操作某个引脚的 GPIO 功能前，需要对这个引脚对应的设备文件节点执行打开操作。如“MISO”引脚对应的设备文件节点是`/dev/gpio-MISO`，那么其打开操作如程序清单 13.1 所示。

程序清单 13.1 打开 GPIO 设备文件节点

```
int fd = 0;
fd = open("/dev/gpio-MISO", O_RDWR);
if (fd < 0) {
    perror("open /dev/gpio-MISO error \n");
}
```

当打开操作完成后，就获得文件描述符。后面的进一步操作，都要使用这个文件描述符。

2. 电平信号的输入/输出操作

使用 `ioctl` 调用，我们可以很方便地在指定引脚上实现输出高/低电平、读入电平信号等操作。我们在 `gpio.h` 定义了相关命令，如程序清单 13.2 所示。

程序清单 13.2 GPIO 操作命令

```
enum {
    SET_GPIO_HIIGHT = 9,           //输出高电平
    SET_GPIO_LOW,                 //输出低电平
    GET_GPIO_VALUE,               //读入输入电平
};
```

输出高电平操作如下所示：

```
ioctl(fd, SET_GPIO_HIIGHT);
```

输出低电平操作如下所示：

```
ioctl(fd, SET_GPIO_LOW);
```

这里需要注意的是，高/低电平的输出命令是不需要任何参数的。

读取输入电平操作如程序清单 13.3 所示。



程序清单 13.3 读入电平操作示例

```
int data;
ioctl(fd, GET_GPIO_VALUE, (long)(&data));
printf("value:%d \n", data);
```

若 data 为“1”，表示输入的是高电平；若 data 为“0”，表示输入的是低电平。

3. 示例代码

光盘提供了通用 GPIO 驱动的测试代码，如程序清单 13.4 所示。

程序清单 13.4 GPIO 测试程序

```
#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 <limits.h>
#include <asm/ioctls.h>
#include <time.h>
#include <pthread.h>

#include "gpio.h"

static void show_help(void) // 打印本程序的用法
{
    printf("example:\n");
    printf("    gpio_test /dev/gpio-xxx H #set gpio output hight \n");
    printf("    gpio_test /dev/gpio-xxx L #set gpio output low   \n");
    printf("    gpio_test /dev/gpio-xxx R #read gpio input status(H/L) \n");
}

int main(int argc, char *argv[])
{
    int fd;
    char cmd;
    int  data;

    if (argc != 3) //如果用户输入参数不足 2 个，打印本程序的用法
        show_help();
        return -1;
}
```



```
fd = open(argv[1], O_RDWR);           //要操作的设备文件节点取自第 1 个参数
if (fd < 0) {
    printf("faile to open %s \n", argv[1]);
    return -1;
}

cmd = argv[2][0];                   //要操作的命令取自第 2 个参数
switch(cmd) {
case 'H':                         //输出高电平
    printf("set %s output Highth \n", argv[1]);
    ioctl(fd, SET_GPIO_HIGHT);
    break;

case 'L':                          //输出低电平
    printf("set %s gpio output low \n", argv[1]);
    ioctl(fd, SET_GPIO_LOW);
    break;

case 'R':                          //读取输入电平的状态
    ioctl(fd, GET_GPIO_VALUE, (long)(&data));
    if (data) {                     //如果返回值为”1”， 表示输入的是高电平
        printf("get %s %d status H \n", argv[1], data);
    } else {                        //如果返回值为”0”， 表示输入的是低电平
        printf("get %s %d status L \n", argv[1], data);
    }
    break;

default:
    show_help();
    break;
}
return 0;
}
```

假如我们测试的引脚为 MISO，并且需要在该引脚上执行输出电平的操作，使用方法是：

```
root@EasyARM-iMX28x ~# ./gpio_test /dev/gpio-MISO H          # 输出高电平
```

或

```
root@EasyARM-iMX28x ~# ./gpio_test /dev/gpio-MISO L          # 输出低电平
```

若我们需要在该引脚上执行读入电平状态的操作，使用方法是：

```
root@EasyARM-iMX28x ~# ./gpio_test /dev/gpio-MISO R
```

若只需要在引脚输出高/低电平，我们提供了比较简便的方法：

```
root@EasyARM-iMX28x ~# echo 1 > /dev/gpio-MISO          # 输出高电平
```

或



root@EasyARM-i.MX28x ~# echo 0 > /dev/gpio-MISO

输出低电平

13.2 ADC 接口

EasyARM-i.MX283(7)A 开发套件在 IDC-B 排针提供了 ADC0、ADC1、ADC6 和 HSADC 四路 ADC 电压模拟量采集接口，而 EasyARM-i.MX280A 则在上述的基础上增多了 ADC2、ADC3、ADC4、ADC5 四路电压模拟量采集接口（共 8 路）。HSADC 为 2M 采样率的高速 ADC，可用于摄像头数据的采集。默认情况下，从 ADC0 到 ADC6 称为低精度 ADC 通道 (LRADC)，HSADC 就称为高速 ADC 通道。其中，LRADC 通道内部含有一个除 2 模拟电路。在未开启内部除 2 电路时，其量程为 0~1.85V；开启除 2 电路时，量程为 0~3.7V。所有 ADC 通道的参考源均来自于内部的参考电压 1.85V。

EasyARM-i.MX283(7)A 开发套件的 ADC 接口所在位置如图 13.5 所示。

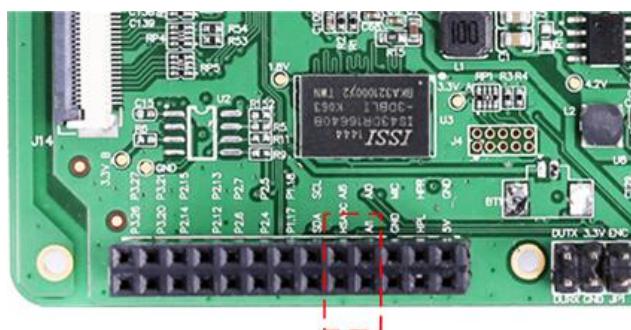


图 13.5 EasyARM-i.MX283(7)A 的 ADC 接口示意图

对于 EasyARM-i.MX280A 开发套件，其 ADC 接口所在位置如图 13.6 所示。

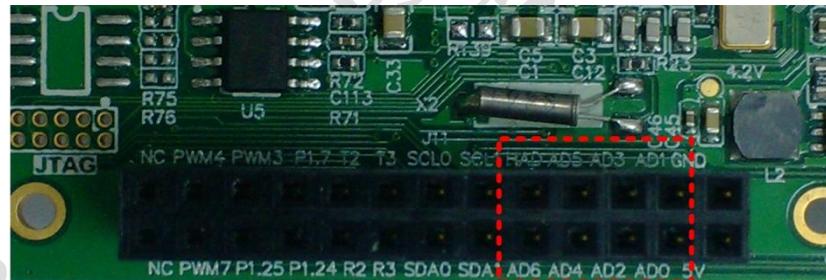


图 13.6 EasyARM-i.MX280A 的 ADC 接口示意图

13.2.1 LRADC

1. LRADC 驱动模块的加载

LRADC 驱动以动态加载模块的形式提供，因此在操作 LRADC 之前要先加载其驱动模块。

root@EasyARM-i.MX28x ~# insmod /root/lradc.ko

zlg EasyARM-imx283 adc driver up.

2. 操作接口

本着简单易用原则，该驱动使用字符设备文件的操作方式，仅需调用 ioctl 函数。其电压读取操作的代码如下：

```
int iRes, fd;  
fd = open("/dev/magic-adc", 0);  
ioctl(fd, cmd, &iRes);
```



其中，iRes 为读取的电压 AD 值，cmd 为读取操作命令。根据是否开启硬件除 2 电路，每个 LRADC 通道可对应两个不同的读取命令，如表 13.2 所示。

表 13.2 ADC 读取命令表

ADC 通道	不开启硬件除 2 电路	开启硬件除 2 电路	开启硬件除 4 电路
ADC0	IMX28_ADC_CH0	IMX28_ADC_CH0_DIV2	—
ADC1	IMX28_ADC_CH1	IMX28_ADC_CH1_DIV2	—
ADC2	IMX28_ADC_CH2	IMX28_ADC_CH2_DIV2	—
ADC3	IMX28_ADC_CH3	IMX28_ADC_CH3_DIV2	—
ADC4	IMX28_ADC_CH4	IMX28_ADC_CH4_DIV2	—
ADC5	IMX28_ADC_CH5	IMX28_ADC_CH5_DIV2	—
ADC6	IMX28_ADC_CH6	IMX28_ADC_CH6_DIV2	—
读取电池电压值	—	—	IMX28_ADC_VBAT_DIV4

EasyARM-i.MX280A 开发套件支持表 13.2 里的所有命令，而对于 EasyARM-i.MX283A 和 EasyARM-i.MX287A 开发套件，ADC 读取命令只有 7 个：

IMX28_ADC_CH0: 读取 ADC0 电压值；

IMX28_ADC_CH1: 读取 ADC1 电压值；

IMX28_ADC_CH6: 读取 ADC6 电压值；

IMX28_ADC_CH0_DIV2: 读取 ADC0 电压值（**开启硬件除 2 电路**）；

IMX28_ADC_CH1_DIV2: 读取 ADC1 电压值（**开启硬件除 2 电路**）；

IMX28_ADC_CH6_DIV2: 读取 ADC6 电压值（**开启硬件除 2 电路**）；

IMX28_ADC_VBAT_DIV4: 读取电池电压值（**开启硬件除 4 电路**）；

3. 计算公式

对于不开启硬件除 2 电路的 ADC 通道，计算公式为： $V = 1.85 \times (Val / 4096)$

对于开启硬件除 2 电路的 ADC 通道，计算公式为： $V = 2 \times 1.85 \times (Val / 4096)$

对于开启硬件除 4 电路的 ADC 通道，计算公式为： $V = 4 \times 1.85 \times (Val / 4096)$

4. 操作示例

如程序清单 13.5 所示的例程，每一行打印多个 LRADC 通道采样数据，打印 50 行后退出程序。

程序清单 13.5 ADC 操作示例

```
#include<stdio.h>      /* using printf()      */
#include<stdlib.h>      /* using sleep()       */
#include<fcntl.h>        /* using file operation */
#include<sys/ioctl.h>    /* using ioctl()       */
#include "lradc.h"

int main(int argc, char *argv[])
{
    int fd;
    int iRes;
    int time = 50;
    double val;
```



```
fd = open("/dev/magic-adc", 0);
if(fd < 0) {
    printf("open error by APP- %d\n", fd);
    close(fd);
    return 0;
}
while(time--) {

    sleep(1);

    ioctl(fd, IMX28_ADC_CH0_DIV2, &iRes);      /* 开启除 2      CH0      */
    val = (iRes * 3.7) / 4096.0;
    printf("CH0:%.2f ", val);

    ioctl(fd, IMX28_ADC_CH1, &iRes);            /* 不开除 2      CH1      */
    val = (iRes * 1.85) / 4096.0;
    printf("CH1:%.2f ", val);
    #if 0                                     /* 对于 EasyARM-iMX280A, 请设置为 1 */
        ioctl(fd, IMX28_ADC_CH2_DIV2, &iRes);      /* 开启除 2      CH2      */
        val = (iRes * 3.7) / 4096.0;
        printf("CH2:%.2f ", val);

        ioctl(fd, IMX28_ADC_CH3, &iRes);          /* 不开除 2      CH3      */
        val = (iRes * 1.85) / 4096.0;
        printf("CH3:%.2f ", val);

        ioctl(fd, IMX28_ADC_CH4_DIV2, &iRes);      /* 开启除 2      CH4      */
        val = (iRes * 3.7) / 4096.0;
        printf("CH4:%.2f ", val);

        ioctl(fd, IMX28_ADC_CH5, &iRes);          /* 不开除 2      CH5      */
        val = (iRes * 1.85) / 4096.0;
        printf("CH5:%.2f ", val);
    #endif
        ioctl(fd, IMX28_ADC_CH6_DIV2, &iRes);      /* 开启除 2      CH6      */
        val = (iRes * 3.7) / 4096.0;
        printf("CH6:%.2f ", val);

        ioctl(fd, IMX28_ADC_VBAT_DIV4, &iRes);     /* 电池电压默认除 4 */
        val = (iRes * 7.4) / 4096.0;
        printf("Vbat:%.2f ", val);

        printf("\n");
    }
    close(fd);
}
```



{}

13.2.2 HSADC

出厂固件已经默认启用了 HSADC 设备，其主设备号为 250，次设备号为 0，设备名“/dev/mxs-hsadc0”。

1. 操作接口

打开 HSADC 的代码如下：

```
struct hsadc_context context;
context.fd = open("/dev/mxs-hsadc0", O_RDWR, S_IRUSR|S_IWUSR);
if(FAILED(context.fd)){
    perror("open hsadc");
    goto fail;
}
```

HSADC 采集的数据有效位为 12 位，最少需要“short”类型变量来存放。读取数据时应用层只需要调用 read()即可。

```
struct hsadc_context *pContext;
ret = read(pContext->fd, pContext->buf, BUF_SIZE);
if(ret <= 0)
continue;
```

2. 计算公式

对于 HSADC 电路，电压范围是：0-1.85V，计算公式为： $V = 1.85 \times (Val / 4096)$

3. 操作示例

程序清单 13.6 所示的例程，操作 HSADC 进行 10 次数据采样，每次采样 100 个数值，取其平均值后再打印出来。

程序清单 13.6 HSADC 操作示例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>

#define tag_log(x, args...) fprintf(stdout, "dtu_hsadc_test: "x, ##args)

#define FAILED(x) ((x)<0)

#define SAMPLE_COUNT (0x100)
#define SAMPLE_WORD_SIZE (2) // sample word size in byte, 8-bit mode is 1 byte, 10,12-bit mode are 2 bytes
#define BUF_SIZE (SAMPLE_COUNT * SAMPLE_WORD_SIZE)

struct hsadc_context{
    int fd;
    int testcount;
```



```
pthread_t thread_id;
char * buf;
};

/*
 * How can we make sure hsadc is ok?
 *
 *      Get value 10 times, if 6 times success continually, we though it ok.
 *
 */

void *hsadc_thread(void *pArgs)
{
    float iVret = 0;
    int j = 0;
    int ret = 0;
    struct hsadc_context *pContext = (struct hsadc_context*)pArgs;
    unsigned short *pbuf = (unsigned short *)pContext->buf;
    while(pContext->testcount){
        memset(pContext->buf, 0x00, BUF_SIZE);
        ret = read(pContext->fd, pContext->buf, BUF_SIZE);
        if(ret <= 0)
            continue;
        // hsadc work on 12 bits depth , so Vin = value / 4096 * Vref
        for(j=0; j<(ret>>1); j++){
            iVret += pbuf[j]&0xffff;
        }
        iVret /= (ret>>1);
        iVret = (iVret * 1.8) / 4096.0;
        tag_log("\r          HSADC:%.2f\n", iVret);
        pContext->testcount--;
        iVret = 0;
        //usleep(300*1000);
    }
    return 0;
}

int hsadc_test(void)
{
    int ret = -1;
    struct hsadc_context context;
    context.fd = open("/dev/mxs-hsadc0", O_RDWR, S_IRUSR|S_IWUSR);
    if(FAILED(context.fd)){
        perror("open hsadc");
        goto fail;
    }

    /* Set up the ADC configuration */
    /* ... (configuration code) ... */

    /* Start the conversion */
    /* ... (conversion control code) ... */

    /* Read the result */
    /* ... (result reading code) ... */

    /* Stop the conversion */
    /* ... (conversion stop code) ... */

    /* Clean up */
    /* ... (cleanup code) ... */

    ret = 0;
fail:
    /* ... (failure handling code) ... */
}
```



```
}

context.buf = (char*)malloc(BUF_SIZE);
if(!context.buf){
    goto fail;
}

context.testcount = 10; /* test 10 times */
ret = pthread_create(&(context.thread_id), NULL, &hsadc_thread, &context);
if(FAILED(ret)){
    goto fail;
}

fail:
if(context.thread_id)
    pthread_join(context.thread_id, 0);
if(context.fd > 0)
    close(context.fd);
if(context.buf)
    free(context.buf);
return 0;
}

int main(int argc, char *argv[])
{
    hsadc_test();
    return 0;
}
```

13.3 串口编程

开发套件引出了 6 路串口，其中有 1 路是调试串口，不可用作数据通信接口。所以开发套件实际为用户提供了 5 路可作数据通信用的串口。对于 EasyARM-i.MX283(7)A 开发套件，可用串口如图 13.7 所示；而 EasyARM-i.MX280A 开发套件，可用串口如图 13.8 所示。

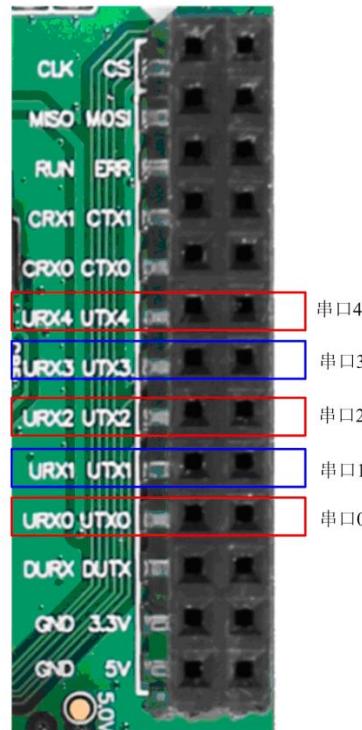


图 13.7 EasyARM-i.MX283(7)A 可用的应用串口

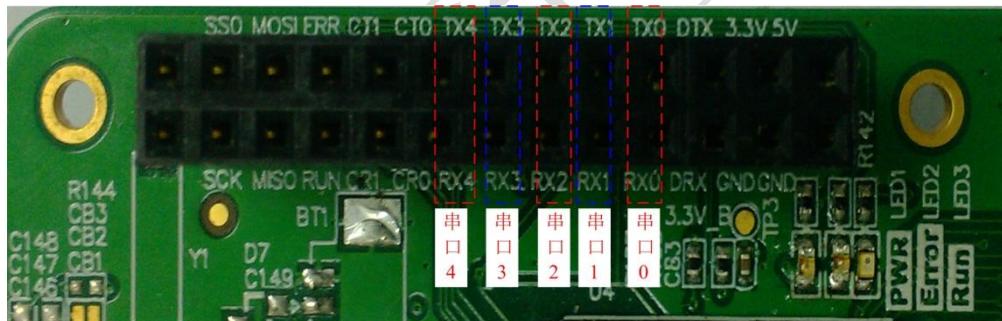


图 13.8 EasyARM-i.MX280A 可用的应用串口

硬件接口和设备文件节点的对应关系如表 13.3 所示。

表 13.3 硬件接口与设备文件对应关系

串口标号	设备文件节点
串口 0	/dev/ttysP0
串口 1	/dev/ttysP1
串口 2	/dev/ttysP2
串口 3	/dev/ttysP3
串口 4	/dev/ttysP4

13.3.1 访问串口设备

1. 打开串口

在开发套件的 Linux 系统中，串口设备文件名是 “/dev/ttysP%d”（其中%d=0、1、2、3、4）。打开串口设备文件的操作如程序清单 13.7 所示。



程序清单 13.7 打开串口设备

```
int fd;
fd = open("/dev/ttySP0", O_RDWR | O_NOCTTY | O_NDELAY);
if (fd < 0) {
    printf("open SPI device error\n");
}
```

相对于普通设备操作，当调用 open() 函数打开串口设备文件时，除了需要用到 O_RDWR 选项标志外，通常还需要使用 O_NOCTTY 和 O_NDELAY 选项标志。

O_NOCTTY 选项是告诉 Linux “本程序不作为串口的‘控制终端’”。如果不使用该选项，一些输入字符可能会影响到进程的运行（如一些产生中断信号的键盘输入字符等）。

O_NDELAY 表示让程序忽略 DCD 信号线。如果不使用该选项，进程可能在 DCD 信号线被拉低时进入休眠状态。

2. 发送数据

使用标准的 write 系统调用向串口写入数据，如程序清单 13.8 所示。

程序清单 13.8 向串口设备写入数据

```
int iNum;
iNum = write(fd, "Hello ZLG! \r", 14);
if (iNum < 0) {
    printf("write data to serial failed! \n");
}
```

3. 读取数据

使用标准的 read 系统调用读入串口数据，如程序清单 13.9 所示。

程序清单 13.9 读入串口数据

```
int len;
unsigned char pBuf[0xff];
len = read(fd, pBuf, 0xff);
```

当串口设备工作在原始数据模式时，read 函数返回串口缓冲区里的数据。如果缓冲区没有数据可读，read 函数会阻塞直到新的数据到来。为了使 read 调用能立即返回，可以采用下面的操作：

```
fcntl(fd, F_SETFL, FNDELAY);
```

FNDELAY 选项会使 read 函数在串口没有数据到来的情况下立即返回（非阻塞方式）。若要恢复正常状态，可以再次调用不带 FNDELAY 选项的 fcntl() 函数：

```
fcntl(fd, F_SETFL, 0);
```

这些操作通常在调用完 open() 函数（带 O_NDELAY 选项）打开串口设备后执行。

4. 关闭串口

```
close(fd);
```

关闭一个串口设备会引起该串口 DTR 信号线的电平置高，使大部分的 modem 设备被挂起。



13.3.2 配置串口属性

本节主要讲解串口的波特率、数据位、校验方式等属性的获取和设置。

1. 属性描述

串口属于终端设备，其接口属性使用 `termios` 结构描述。该结构如程序清单 13.10 所示。

程序清单 13.10 `termios` 结构

```
struct termios {  
    tcflag_t c_cflag /* 控制标志 */;  
    tcflag_t c_iflag /* 输入标志 */;  
    tcflag_t c_oflag /* 输出标志 */;  
    tcflag_t c_lflag /* 本地标志 */;  
    tcflag_t c_cc[NCCS] /* 控制字符 */;  
};
```

粗略而言，控制标志影响着 RS-232 串行线的行为（如：忽略调制解调器的状态线、每个字符需要一个或两个停止位等）；输入标志负责控制驱动程序对输入字符的处理策略（如：剥除输入字节的第 8 位，允许输入奇偶校验等），输出标志决定着驱动程序如何正确输出（如：执行输出处理、将换行符映射为 CR/LF 等），本地标志会改变驱动程序与用户间的交互方式（如：本地回显的开和关等），`c_cc` 数组则包含了所有可以更改的特殊字符。

(1) 控制标志

`c_cflag` 成员控制着串口设备的波特率、数据位、奇偶校验、停止位以及流控机制，表 13.4 列出了 `c_cflag` 的部分可用选项。

表 13.4 `c_cflag` 部分可用选项

标志	说 明	标志	说 明
CBAUD	波特率位屏蔽	CSIZE	数据位屏蔽
B0	0 位/秒 (挂起)	CS5	5 位数据位
B110	100 位/秒	CS6	6 位数据位
B134	134 位/秒	CS7	7 位数据位
B1200	1200 位/秒	CS8	8 位数据位
B2400	2400 位/秒	CSTOPB	2 位停止位，否则为 1 位
B4800	4800 位/秒	CREAD	启动接收
B9600	9600 位/秒	PARENB	进行奇偶校验
B19200	19200 位/秒	PARODD	奇校验，否则为偶校验
B57600	57600 位/秒	HUPCL	最后关闭时断开
B115200	115200 位/秒	CLOCAL	忽略调制解调器状态行

`c_cflag` 成员的 `CREAD` 和 `CLOCAL` 选项通常是要启用的，这两个选项将使驱动程序启动接收字符装置，同时忽略掉串口信号线的状态。

(2) 输入标志

`c_iflag` 成员负责控制串口设备对输入数据的处理，表 13.5 所示是 `c_iflag` 的部分可用标志。



表 13.5 c_iflag 标志

标 志	说 明
INPCK	打开输入奇偶校验
IGNPAR	忽略奇偶错字符
PARMRK	标记奇偶错
ISTRIP	剥除字符第 8 位
IXON	启用/停止输出控制流起作用
IXOFF	启用/停止输入控制流起作用
IGNBRK	忽略 BREAK 条件
INLCR	将输入的 NL 转换为 CR
IGNCR	忽略 CR
ICRNL	将输入的 CR 转换为 NL

- 设置输入校验

当 c_cflag 成员的 PARENB (奇偶校验) 选项启用时, c_iflag 也应启用奇偶校验选项。操作方法是启用 INPCK 和 ISTRIP 选项:

```
options.c_iflag |= (INPCK | ISTRIP);
```

注意: IGNPAR 选项在某些应用场合带有一定的危险性, 它指示串口驱动程序忽略奇偶校验错误, 也就是说, IGNPAR 使奇偶校验出错的字符也能通过输入。这在测试通信链路的质量时也许是有用的, 但在通常的数据通信应用场合却不应该被使用。

- 设置软件流控制

使用软件流控制的方法是启用 IXON、IXOFF 和 IXANY 选项:

```
options.c_iflag |= (IXON | IXOFF | IXANY);
```

相反, 要禁用软件流控制则可对上面的选项取反:

```
options.c_iflag &= ~(IXON | IXOFF | IXANY);
```

(3) 输出标志

c_oflag 成员定义输出过滤规则, 如表 13.6 所示是 c_oflag 成员的部分选项标志。

表 13.6 c_oflag 标志

标 志	说 明
BSDLY	退格延迟屏蔽
CMSPAR	标志或空奇偶性
CRDLY	CR 延迟屏蔽
FFDLY	换页延迟屏蔽
OCRNL	将输出的 CR 转换为 NL
OFDEL	填充符为 DEL, 否则为 NULL
OFILL	对于延迟使用填充符
OLCUC	将输出的小写字符转换为大写字符
ONLCR	将 NL 转换为 CR-NL
ONLRET	NL 执行 CR 功能
ONOCR	在 0 列不输出 CR
OPOST	执行输出处理
OXTABS	将制表符扩充为空格



- 启用输出处理

启用输出处理需要在 `c_oflag` 成员中开启 OPOST 选项，其操作方法如下：

```
options.c_oflag |= OPOST;
```

- 使用原始输出

使用原始输出，就是禁用输出处理，使数据能不经处理、过滤地输出到串行设备接口。当 OPOST 被禁止时，`c_oflag` 的其它选项也会被忽略，其操作方法如下：

```
options.c_oflag &= ~OPOST;
```

(4) 本地标志

本地标志 `c_lflag` 控制着串口驱动程序如何管理输入的字符，如表 13.7 所示是 `c_lflag` 的部分可用标志。

表 13.7 `c_lflag` 标志

标志	说明
ISIG	启用终端产生的信号
ICANON	启用规范输入
XCASE	规范大/小写表示
ECHO	进行回送
ECHOE	可见擦除字符
ECHOK	回送 kill 符
ECHONL	回送 NL
NOFLSH	在中断或退出键后禁用刷清
IEXTEN	启用扩充的输入字符处理
ECHOCTL	回送控制字符为^(char)
ECHOPRT	硬拷贝的可见擦除方式
ECHOKE	Kill 的可见擦除
PENDIN	重新打印未决输入
TOSTOP	对于后台输出发送 SIGTTOU

- 选择规范模式

规范模式是行处理的。调用 `read` 读取串口数据时，每次返回一行数据。当选择规范模式时，需要启用 ICANON、ECHO 和 ECHOE 选项：

```
options.c_lflag |= (ICANON | ECHO | ECHOE);
```

串口设备用作用户终端时，通常要把串口设备配置成规范模式。

- 选择原始模式

在原始模式下，串口输入的数据是不经过处理的，接收到的数据将被完整保留。要使串口设备工作在原始模式，需要关闭 ICANON、ECHO、ECHOE 和 ISIG 选项，其操作方法如下所示：

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
```

(4) 控制字符组

`c_cc` 数组的长度是 NCCS，一般介于 15-20 之间。`c_cc` 数组每个成员的下标都用一个宏来表示。表 13.8 列出了 `c_cc` 部分下标的标志名及其说明。



表 13.8 c_cc 标志

标 志	说 明
VINTR	中断
VQUIT	退出
VERASE	擦除
VEOF	行结束
VEOL	行结束
VMIN	需读取的最小字节数
VTIME	与“VMIN”配合使用，限定允许传输或等待的最长时间

在规范模式下，调用 `read` 读取串口数据时，通常是返回一行数据；而原始模式下串口输入的数据是不分行的。在原始模式下读取时，返回的数据量需要考虑两个因素：MIN 和 TIME。MIN 和 TIME 在 `c_cc` 数组中的下标名分别是 VMIN 和 VTIME。

MIN 是指一次 `read` 调用所期望返回的最小字节数。TIME 与 MIN 组合使用，其具体含义分以下四种情形：

- 当 $MIN > 0, TIME > 0$ 时

TIME 为接收到第一个字节后，所允许的数据传输或等待的最长分秒数（1 分秒 = 0.1 秒）。定时器在收到第一个字节后启动，若在计时器超时之前已经接收到 MIN 个字节，则 `read` 调用会返回 MIN 个字节；否则，在计时器超时后将返回实际接收到的字节。

注意：因为定时器只有在接收到第一个字节时才会启动，所以至少可以返回 1 个字节。但是在接收到第一个字节之前，调用者都会处于阻塞状态。此外，如果在调用 `read` 之前缓冲区已经有数据可用，那么在 `read` 被调用后定时器会立即启动。

- 当 $MIN > 0, TIME = 0$ 时

MIN 个字节被完整接收后，`read` 才返回，这可能会造成 `read` 无限期地阻塞。

- 当 $MIN = 0, TIME > 0$ 时

TIME 为允许等待的最大时间，计时器在调用 `read` 后立即启动。`read` 在串口接收到 1 字节数据或者计时器超时后返回。如果计时器超时，则返回 0。

- 当 $MIN = 0, TIME = 0$ 时

如果有数据可用，则 `read` 最多返回所要求的字节数；如果没有数据可用，则 `read` 立即返回 0。

2. 属性设置

使用函数 `tcgetattr` 和 `tcsetattr` 可以获取和设置串口的 `termios` 结构属性，如程序清单 13.11 所示。

程序清单 13.11 设置和获取 `termios` 结构属性

```
#include <termios.h> /* 使用终端接口函数需要包含此头文件*/  
int tcgetattr(int fd, struct termios *termpptr);  
int tcsetattr(int fd, int opt, const struct termios *termpptr);
```

其中：`fd` 为串口设备的文件描述符；`termpptr` 参数在 `tcgetattr` 函数中被用于存放串口设置的 `termios` 结构体；`opt` 是整型变量，它的使用方法如下：

- `TCSANOW`: 更改立即发生；
- `TCSADRAIN`: 发送了所有输出后更改才发生，若更改输出参数应启用此选项；



- TCSAFLUSH：发送了所有输出后更改才发生，更进一步，在更改发生时删除（Flush）所有未读的输入数据。

在串口的驱动程序中，有输入缓冲区和输出缓冲区。改变串口属性时，缓冲区中的数据可能还存在，此时需要考虑更改后的属性在什么时候生效。tcsetattr 的参数 opt 可以指定在什么时候更新串口的属性。

上述两个函数执行时，若成功则返回 0，若出错则返回 -1。

掌握了如何获取和设置串口的属性结构后，下面将介绍串口主要属性的修改，即修改 termios 结构体的成员。

termios 结构体各成员的各选项，除了需要用屏蔽标志的选项外（如波特率选项、数据位选项等），都是按位表示的，对这些选项的设置或清除可以直接用“^”或“&”逻辑运算来完成。

对于那些需要用屏蔽标志的选项，则应该先用“&”运算清除原有设置，再用“^”运算启用新设置。例如，为了设置字符长度，需先用字符长度屏蔽标志 CSIZE 将表示字符长度的位清 0，然后再将对应的位设置为 CS5、CS6、CS7 或 CS8。

（1）设置波特率

串口的输入和输出波特率可分别用 cfsetispeed() 和 cfsetospeed() 函数来设置，如程序清单 13.12 所示。

程序清单 13.12 设置串口输入/输出波特率函数

```
#include <termios.h>
int cfsetispeed(struct termios *termptr, speed_t speed);
int cfsetospeed(struct termios *termptr, speed_t speed);
```

这两个函数若执行成功，返回 0；若出错则返回 -1。

使用这两个函数时，应当理解输入、输出波特率是存在于串口设备的 termios 结构中的。在调用任一 cfset 函数之前，先要用 tcgetattr 获得设备的 termios 结构。与此类似，在调用任一 cfset 函数后，波特率都要被设置到 termios 结构中去，也就是说，为使这种更改生效，应当调用 tcsetattr 函数。操作方法如程序清单 13.13 所示。

程序清单 13.13 设置波特率示例

```
if (tcgetattr(fd, &opt) < 0) {
    return ERROR;
}
cfsetispeed(&opt, B9600);
cfsetospeed(&opt, B9600);
if (tcsetattr(fd, TCSANOW, &opt) < 0) {
    return ERROR;
}
```

（2）设置数据位

设置数据位不需要专用的函数，只需要在设置数据位之前用数据位屏蔽标志（CSIZE）把对应数据位清零，然后再设置新的数据位即可，如下所示：

```
options.c_cflag &= ~CSIZE; /* 先把数据位清零 */
```



```
options.c_cflag |= CS8; /* 把数据位设置为 8 位 */
```

(3) 设置奇偶校验

正如设置数据位一样，奇偶校验也是直接在 cflag 成员上设置的。下面是各种类型的校验设置方法。

- 无奇偶校验 (8N1):

```
options.c_cflag &= ~PARENB;  
options.c_cflag &= ~CSTOPB;  
options.c_cflag &= ~CSIZE;  
options.c_cflag |= CS8;
```

- 7 位数据位奇偶校验 (7E1):

```
options.c_cflag |= PARENB;  
options.c_cflag &= ~PARODD;  
options.c_cflag &= ~CSTOPB;  
options.c_cflag &= ~CSIZE;  
options.c_cflag |= CS7;
```

- 奇校验 (7O1):

```
options.c_cflag |= PARENB;  
options.c_cflag |= PARODD;  
options.c_cflag &= ~CSTOPB;  
options.c_cflag &= ~CSIZE;  
options.c_cflag |= CS7;
```

13.3.3 操作示例

程序清单 13.14 所示的是原始模式下的串口操作示例。

程序清单 13.14 串口操作示例

```
#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 <limits.h>  
  
#define DEV_NAME "/dev/ttySP0"  
  
int main(void)  
{  
    int iFd, i;  
    int len;  
    unsigned char ucBuf[1000];  
    struct termios opt;
```



```
iFd = open(DEV_NAME, O_RDWR | O_NOCTTY);
if(iFd < 0) {
    perror(DEV_NAME);
    return -1;
}
tcgetattr(iFd, &opt);
cfsetispeed(&opt, B115200);
cfsetspeed(&opt, B115200);
if (tcgetattr(iFd, &opt)<0) {
    return -1;
}
opt.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
opt.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
opt.c_oflag &= ~(OPOST);
opt.c_cflag &= ~(CSIZE | PARENB);
opt.c_cflag |= CS8;
opt.c_cc[VMIN] = 255;
opt.c_cc[VTIME] = 150;
if (tcsetattr(iFd, TCSANOW, &opt)<0) {
    return -1;
}
tcflush(iFd,TCIOFLUSH);
for (i = 0; i < 1000; i++){
    ucBuf[i] = 0xff - i;
}
write(iFd, ucBuf, 0xff);

len = read(iFd, ucBuf, 0xff);
printf("get date: %d \n", len);
for (i = 0; i < len; i++){
    printf(" %x", ucBuf[i]);
}
printf("\n");
close(iFd);
return 0;
}
```

注：测试上述代码时，需要把/dev/ttyS0 接口的 RXD 和 TXD 用杜邦线短接起来。当程序执行时，程序会在串口把发出去的数据读回来并打印到终端屏幕上。

13.3.4 串口数据监控

在项目应用中，常常需要使用串行接口来采集传感器的数据，如图 13.9 所示。

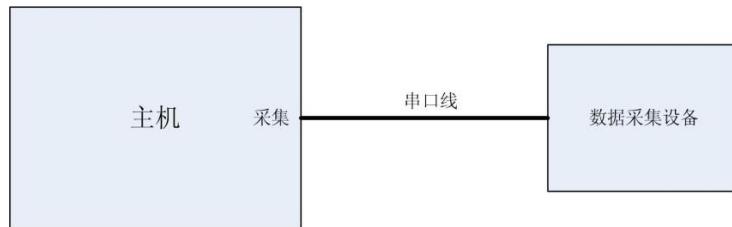


图 13.9 采集串口数据

正如前面的描述，我们的应用程序调用 `read` 来读取串口的数据。可能会有用户产生疑问：如果是异步数据，那么我怎样知道数据什么时候要来，需要调用 `read` 来读取数据呢？其实当 `read` 被调用时，会马上读取串口驱动的软件缓冲区里面的数据。但是如果此时软件缓冲区中没有数据，或者数据长度还没有达到用户的要求，`read` 就会进入休眠状态等待数据的到来（具体是根据“TIME”和“MIN”的设置，见前面的描述）。当有数据到达串行接口时，数据就会先进入该接口的硬件缓冲区，同时产生一个串口中断，而中断处理函数会把硬件缓冲区的数据复制到软件缓冲区中去。如果软件缓冲区的数据长度达到了用户 `read` 调用的要求，串口中断处理函数会唤醒由于调用 `read` 而处在休眠状态的进程/线程，这样 `read` 调用就可以返回了。

为了保证串口的数据不会丢失，应用程序必须在任何时候都要有 `read` 调用处于待命状态。这就要求应用程序在 `read` 调用返回后马上进入另一次 `read` 调用，如此反复。因此在程序设计中，我们需要有一个线程反复地执行 `read` 调用以监控串口数据；而另外一个线程处理数据，如图 13.10 所示。

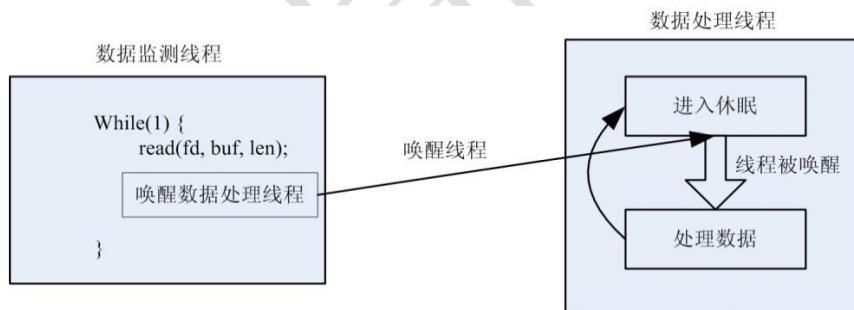
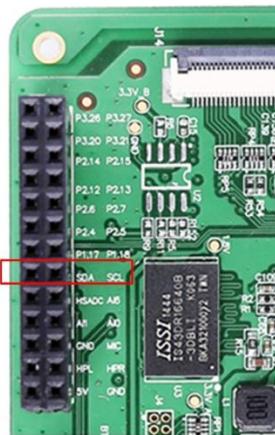
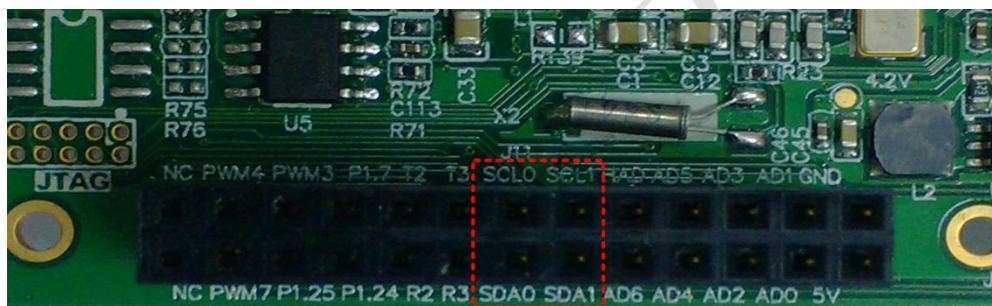


图 13.10 串口数据处理一般方法

数据监测线程只负责串口数据的接收。当串口中有数据到达时，就唤醒数据处理线程，进入数据处理操作。这里不需要担心数据监测线程“一直处于无限循环中而造成 CPU 不必要的负担”。当串口有数据到达时数据监测线程才会处于执行状态，其它时候处于休眠状态。

13.4 I²C 接口

Linux 系统实现了 I²C 总线的驱动，用户通过应用程序即可进行 I²C 总线的通信。对于 EasyARM-i.MX283(7)A 开发套件，底板上只引出了 I²C1 接口，如图 13.11 所示；而对于 EasyARM-i.MX2830A，底板上引出了 I²C0 和 I²C1 接口，如图 13.12 所示。

图 13.11 EasyARM-i.MX283(7)A 的 I²C1 接口位置图图 13.12 EasyARM-i.MX280A 的 I²C0 和 I²C1 接口位置图

13.4.1 open 调用

本小节以 EasyARM-i.MX283(7)A 开发套件为例，来说明在 Linux 操作系统环境中 I²C 接口的使用方法，其原理和操作同样适用于 EasyARM-i.MX280A 开发套件。

在使用 I²C 驱动操作接口时，先调用 open 函数打开 I²C 的设备文件节点，以获得文件描述符，如程序清单 13.15 所示。

程序清单 13.15 打开 I²C 设备文件

```
int fd;
fd = open("/dev/i2c-1", O_RDWR);
if (fd < 0) {
    perror("open i2c-1 \n");
}
```

13.4.2 ioctl 调用

当使用 I²C 驱动来操作 I²C 从机器件时，需要先把 I²C 从机器件的地址和地址长度设置到 I²C 的驱动设备文件接口。

1. 设置从机地址

使用 I2C_SLAVE 命令调用 ioctl() 函数，可以设置 I²C 从机地址，命令参数就是 I²C 从机



地址右移一位后的值。当需要把 I²C 从机地址设置为 0xA0 时，示例代码如下：

```
ioctl(fd, I2C_SLAVE, 0xA0 >> 1);
```

注意：地址需要右移一位，因为地址的 Bit0 是读写控制位，稍后驱动程序会把从机地址的命令参数左移一位，并补上读写控制位。

2. 设置地址长度

使用 I2C_TENBIT 命令调用 ioctl() 函数，可以设置 I²C 从机地址的长度。当命令参数为 1 时，表示 I²C 从机地址长度为 10 位；当命令参数为 0 时，表示 I²C 从机地址长度为 8 位。

当需要把 I²C 从机地址长度设置为 8 位时，可参考如下代码：

```
ioctl(fd, I2C_TENBIT, 0);
```

注：I²C 的 ioctl 调用所支持的命令在 linux-2.6.35.3/include/linux/i2c-dev.h 头文件中有定义，因此必须在程序中使用 #include<i2c-dev.h>。i2c-dev.h 如程序清单 13.16 所示。

程序清单 13.16 ioctl 命令定义

```
/* /dev/i2c-X ioctl commands. The ioctl's parameter is always an
 * unsigned long, except for:
 *
 * - I2C_FUNCS, takes pointer to an unsigned long
 * - I2C_RDWR, takes pointer to struct i2c_rdwr_ioctl_data
 * - I2C_SMBUS, takes pointer to struct i2c_smbus_ioctl_data
 */

#define I2C_RETRIES          0x0701           /* number of times a device address should */
                                              /* be polled when not acknowledging */
#define I2C_TIMEOUT           0x0702           /* set timeout in units of 10 ms */
#define I2C_SLAVE              0x0703           /* Use this slave address
                                              /* NOTE: Slave address is 7 or 10 bits, but
                                              /* 10-bit addresses are NOT supported! (due
                                              /* to code brokenness)
                                              */
#define I2C_SLAVE_FORCE        0x0706           /* Use this slave address, even if it is already
                                              /* in use by a driver!
                                              */
#define I2C_TENBIT             0x0704           /* 0 for 7 bit addrs, != 0 for 10 bit */
#define I2C_FUNCS              0x0705           /* Get the adapter functionality mask */
#define I2C_RDWR                0x0707           /* Combined R/W transfer (one STOP only) */
#define I2C_PEC                 0x0708           /* != 0 to use PEC with SMBus */
#define I2C_SMBUS               0x0720           /* SMBus transfer */
```

13.4.3 write 调用

当设置好 I²C 从机的地址后，就可以调用 write() 函数向 I²C 从机器件写入数据。示例代码如下：

```
write(fd, buf, len); // len 为 buf 缓冲区的长度
```

当 write() 函数调用后，I²C 主机会向 I²C 从机器件发出 I²C 总线起始信号，然后发送器件的地址，地址发送完并接收到应答后再将 buf 缓冲区中的数据发出，数据发送完后 I²C 主机发出总线结束信号。

13.4.4 read 调用



当设置好 I²C 从机的地址后，就可以调用 read() 函数从 I²C 从机器件读入数据。示例代码如下：

```
read(fd, buf, len); //len 表示要读数据的长度
```

当 read() 函数调用后，I²C 主机向 I²C 从机器件发出总线起始信号，然后发送器件地址，地址发送完并接收到从机应答后，接着向 I²C 从机发出读数据时钟，驱动程序将接收到的数据存入 buf 中，数据读取完后 I²C 主机发出总线结束信号。

对于类似于 EEPROM 之类具有子地址的 I²C 接口的器件，在发送或读取数据之前需要先发送 I²C 子地址。

```
write(fd, addr, 1); //发送要读取的数据的子地址  
read(fd, rx_buf, 16); //读取数据
```

EEPROM 的读写操作例程请参考开发示例中的 I²C 接口部分的代码。

13.4.5 close 调用

当 I²C 驱动操作完成后，请调用 close() 函数关闭之前打开的 I²C 驱动设备文件：

```
close(fd);
```

13.4.6 应用程序读写 DS2460 例程

例程中使用的 I²C 设备是 DS2460 加密芯片，该芯片与开发套件的 I²C0 总线相连，如图 13.13 所示。用户也可使用其他 I²C 设备进行测试，可参考程序清单 13.17 来编写对应 I²C 设备的测试程序。

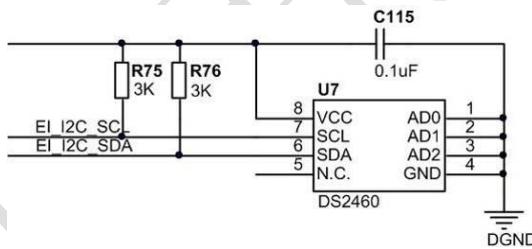


图 13.13 DS2460 连接原理图

DS2460 的 SCL 和 SDA 引脚分别和 i.MX28xx 处理器的 I²C0_SCL 和 I²C0_SDA 引脚相连，所以使用 “/dev/i2c-0” 文件设备节点可以控制该芯片。如图 13.13 所示 DS2460 芯片的从机地址是 0x80。DS2460 的内部地址从 0x00 ~ 0x3F 是内部 SRAM 地址空间。

如程序清单 13.17 所示的测试程序从 DS2460 的内部地址空间 0x00 ~ 0x0F 分别写入 1、2、3……16，然后再在这片地址空间读取出数据并打印出来显示。

程序清单 13.17 DS2460 测试程序

```
#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 "iic.h"

#define I2C_ADDR 0x80
#define DATA_LEN 17

int main(void)
{
    unsigned int uiRet;
    int i;

    unsigned char tx_buf[DATA_LEN];
    unsigned char rx_buf[DATA_LEN];
    unsigned char addr[2] ;

    addr[0] = 0x00;
    GiFd = open("/dev/i2c-0", O_RDWR);
    if (GiFd == -1)
        perror("open serial 0\n");
    uiRet = ioctl(GiFd, I2C_SLAVE, I2C_ADDR >> 1);
    if (uiRet < 0) {
        printf("setenv address fail: %x \n", uiRet);
        return -1;
    }
    tx_buf[0] = addr[0];
    for (i = 1; i < DATA_LEN; i++) {
        tx_buf[i] = i;
    }
    write(GiFd, tx_buf, DATA_LEN);
    write(GiFd, addr, 1);
    read(GiFd, rx_buf, DATA_LEN - 1);
    printf("read from ds2460's eeprom:");
    for(i = 0; i < DATA_LEN - 1; i++) {
        printf(" %x", rx_buf[i]);
    }
    printf("\n");
    return 0;
}
```

在光盘资料中有该程序代码的实现文件，如果用户已经连接 DS2460 加密芯片到开发套件的 I²C0 上，可以直接编译测试，在开发套件的/root 目录下有该测试程序，用户可以直接进行测试。比如登录 EasyARM-i.MX283A 的 Linux 系统，进入命令行终端，输入如下指令可以测试 I²C 的读写：

```
root@EasyARM-iMX28x ~# ./i2c_ds2460_test
```

如图 13.14 所示表示测试成功，打印出了 16 进制的数字 1~16。



```
UBIFS: mounted UBI device 0, volume 0, name "rootfs"
UBIFS: file system size: 218652672 bytes (213528 KIB, 208 MiB, 1722 LEBs)
UBIFS: journal size: 10919936 bytes (10664 KIB, 10 MiB, 86 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 4952688 bytes (4836 KIB)
VFS: Mounted root (ubifs filesystem) on device 0:15.
Freeing init memory: 160K

arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at 2011/08/
10 09:20) 4.6.2 20110630 (prerelease)
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800
Freescale Semiconductor, Inc.

EasyARM-iMX283 login: root
Password:

BusyBox v1.20.2 () built-in shell (ash)
Enter 'help' for a list of built-in commands.

root@EasyARM-iMX283: ~# ./i2c_ds2460_test
read from ds2460's eeprom: 1 2 3 4 5 6 7 8 9 a b c d e f 10
root@EasyARM-iMX283: ~#
```

图 13.14 测试 DS2460

13.5 PWM 接口

EasyARM-i.MX283(7)A 开发套件上只引出了 PWM_4、PWM_7 两路 PWM 输出引脚，PWM_3 用于液晶屏背光控制（故未引出），PWM_4 和 PWM_7 在排针上引出，如图 13.15 所示。而 EasyARM-i.MX280A 则引出了 PWM_3、PWM_4、PWM_7 三路 PWM 输出引脚，如图 13.16 所示。

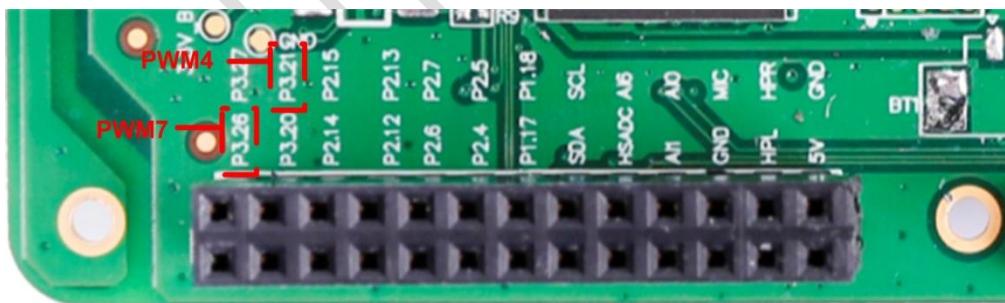


图 13.15 EasyARM-i.MX283(7)A 2 路 PWM 输出位置



图 13.16 EasyARM-i.MX280A 上的 3 路 PWM 输出位置

13.5.1 PWM 周期计算

PWM 周期的计算方法如下：

$$T_{pwm} = 1/(pwm_clk/div) \times period$$

演示的 PWM 驱动使用 24MHz 的时钟源，PWM 默认使用了 64 分频及 400 的 period，所以其 PWM 周期为：

$$1/(24000000/64) \times 400 = 0.001066667s$$

可通过修改 period、div 的值来修改 PWM 的周期， $period \leq max_brightness$ 的范围 1~65536（对应寄存器值 0~65535）。

表 13.1 div 的取值和对应的分频

div	分频
0	1
1	2
2	4
3	8
4	16
5	64
6	256
7	1024

注：PWM3 与 PWM4、PWM7 默认采用相同的时钟源、分频，其 period (EasyARM-i.MX283(7)A 开发套件的 PWM3 为 max_brightness) 默认为 100，其周期的计算方法与 PWM4 和 PWM7 相同；需要说明的是开发套件中 PWM_3 用于液晶屏背光控制，可修改占空比，不可修改周期。

13.5.2 PWM 占空比设置与输出

PWM_4 和 PWM_7 两个通道 PWM 以 backlight 类型的驱动的形式存在，输出的频率为 937.5Hz，可以通过系统命令进行查看/设置 PWM 的周期和占空比，也可以通过应用程序进行操作。

13.5.3 系统命令操作 PWM3 示例

PWM_3、PWM_4 和 PWM_7 对应的驱动设备文件分别为位于 /sys/class/backlight/ 目录下的 mxs-bl、easy283-pwm.4 及 easy283-pwm.7 文件。通过命令查看及设置液晶背光 PWM_3 占空比的示例指令如下：



```
root@EasyARM-iMX28x ~# cd /sys/class/backlight/
root@EasyARM-iMX28x /sys/class/backlight# ls
easy283-pwm.4  easy283-pwm.7  mxs-bl
root@EasyARM-iMX28x /sys/class/backlight# cd mxs-bl
root@EasyARM-iMX28x /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat max_brightness
100
root@EasyARM-iMX28x /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat brightness
100
root@EasyARM-iMX28x /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# echo 10 > brightness
root@EasyARM-iMX28x /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat brightness
10
```

操作相关指令解释如下：

- cat max_brightness，用于查看可设置的占空比的最大值；
- cat brightness，查看当前占空比的值；
- echo 10 > brightness，设置 PWM 的占空比；实际输出的 PWM 波形的占空比为 brightness / max_brightness，本例中为 10/100。

指令执行完后可以观察到开发套件液晶背光亮度被调低了（系统上电后默认的背光 PWM 占空比为 100%）。

注意：操作适用于 EasyARM-i.MX283(7)A 开发套件。

13.5.4 系统命令操作 PWM4 示例

通过命令查看及设置 PWM_4 周期、占空比的示例指令如下：

```
root@EasyARM-iMX28x ~# cd /sys/class/backlight/
root@EasyARM-iMX28x /sys/class/backlight# ls
easy283-pwm.4  easy283-pwm.7  mxs-bl
root@EasyARM-iMX28x /sys/class/backlight# cd easy283-pwm.4
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# ls
actual_brightness  device  period  uevent
bl_power  div  power
brightness  max_brightness  subsystem
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# cat max_brightness
65535
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# cat period
400
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# cat brightness
400
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# cat div
5
root@EasyARM-iMX28x /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4# echo 200 > brightness
```

操作相关指令解释如下：

- cat max_brightness，查看可设置的占空比的最大周期装载值，该值不可修改；
- cat period，周期装载值，period≤max_brightness；
- cat div，分频因子（0-7），详细见表 13.1 所示；



- cat brightness, 查看当前占空比的值, brightness≤period;
- echo 200 > brightness, 设置 PWM 的占空比; 实际输出的 PWM 波形的占空比为 brightness/ period, 本例中为 200/400。

注: PWM7 操作与 PWM4 操作相同。

13.6 SPI 接口

EasyARM-iMX287A 开发套件引出了 SPI3 接口, 其物理位置如图 13.17 所示。下面以该接口的使用为例, 介绍如何在 Linux 下操作 SPI 接口。(注意: 只有 EasyARM-iMX287A 开发套件有 SPI3 接口, 其它两款开发套件只支持 SPI2 接口并且默认没有开启。如果需要使用 SPI2 接口, 请参考其它文档进行功能复用。)。

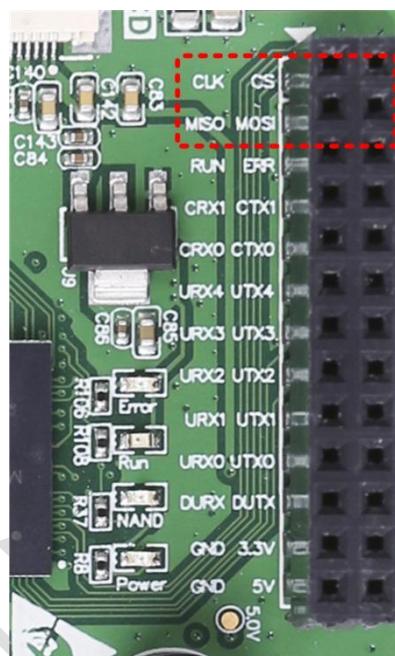


图 13.17 EasyARM-iMX287A 的 SPI3 所在位置

SPI 驱动设备文件名: /dev/spidev1.0。

需要注意的是 i.MX28x 系列处理器的 SPI 控制只支持半双工的通信方式, 在发送数据时不能接收数据, 在接收数据时不能发送数据。

另外, 其 ioctl 函数使用的参数都来自内核目录下 linux-2.6.35.3/include/linux/spi/spidev.h 文件中, 所以对 SPI 的编程需要加入 “#include <spidev.h>” 这行头文件包含语句。

13.6.1 open 调用

在使用 SPI 设备驱动之前, 请使用 open 调用打开驱动设备文件, 获得文件描述符, 如程序清单 13.18 所示。

程序清单 13.18 打开 SPI 设备文件

```
fd = open("/dev/spidev1.0", O_RDWR);
if (fd < 0) {
    printf("can not open SPI device\n");
}
```



13.6.2 ioctl 调用

Linux 的 SPI 驱动为用户提供了相当全面的命令，通过这些命令用户可以配置 SPI 总线的时序、设置总线速率和实现全双工通信。

1. 设置极性和相位

设置 SPI 极性及相位可以通过调用 ioctl() 函数时传递 SPI_IOC_WR_MODE 命令参数实现，如表 13.9 所示。

表 13.9 SPI_IOC_WR_MODE 命令

命 令	SPI_IOC_WR_MODE
调用方式	ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
功能描述	设置 SPI 总线的极性和相位
参数说明	mode 类型: U32 可选值: SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3; 关于 SPI 总线极性和相位的 4 种模式请参考 SPI 协议
返回值说明	0: 设置成功 1: 设置不成功

2. 读取极性和相位

读取 SPI 极性及相位设置模式可以通过调用 ioctl() 函数时传递 SPI_IOC_RD_MODE 命令参数实现，如表 13.10 所示。

表 13.10 SPI_IOC_RD_MODE 命令

命 令	SPI_IOC_RD_MODE
调用方式	ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
功能描述	读取 SPI 总线的极性和相位设置模式
参数说明	mode 类型: U32 参数返回值为: SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、 SPI_MODE_3; 关于 SPI 总线极性和相位的 4 种模式请参考 SPI 协议。
返回值说明	恒为 0: 读取成功

3. 设置每字的数据位长度

设置 SPI 总线上每字的数据位长度可以通过调用 ioctl() 函数时传递 SPI_IOC_WR_BITS_PER_WORD 命令参数实现，如表 13.11 所示。

表 13.11 SPI_IOC_WR_BITS_PER_WORD 命令

命 令	SPI_IOC_WR_BITS_PER_WORD
调用方式	ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
功能描述	设置 SPI 总线上每字的数据位长度
命令参数说明	bits 类型: U32 取值可在 1~N 之间
返回值说明	恒为 0: 设置成功

4. 设置最大总线速率

设置 SPI 总线的最大速率可以通过调用 ioctl() 函数时传递



SPI_IOC_WR_MAX_SPEED_HZ 命令参数实现，如表 13.12 所示。

表 13.12 SPI_IOC_WR_MAX_SPEED_HZ

命 令	SPI_IOC_WR_MAX_SPEED_HZ
调用方式	ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
功能描述	设置 SPI 总线的最大速率
命令参数说明	speed 类型: U32 单位为 Hz, 取值可在 1~N 之间
返回值说明	恒为 0: 设置成功

5. 数据接收/发送操作命令

在调用 ioctl() 函数时传递 SPI_IOC_MESSAGE(1) 命令（**SPI_IOC_MESSAGE(1)** 是一个带参数的宏，其在 spidev.h 文件定义）参数则可以实现 SPI 总线数据的收发，该命令介绍如表 13.13 所示。

表 13.13 SPI_IOC_MESSAGE(1) 命令

命 令	SPI_IOC_MESSAGE(1)
调用方式	ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
功能描述	实现在 SPI 总线接收/发送数据操作
命令参数说明	tr 类型: struct spi_ioc_transfer 参数 tr 是 struct spi_ioc_transfer 结构的类型，用于封装要接收/发送的数据，详细请阅读下文。
返回值说明	0: 操作成功 1: 操作失败

使用 SPI_IOC_MESSAGE(1) 命令进行接收/发送的数据都需要使用 struct spi_ioc_transfer 结构体来封装，该结构体的定义如程序清单 13.19 所示。

程序清单 13.19 struct spi_ioc_transfer 结构体的定义

```
struct spi_ioc_transfer {
    __u64      tx_buf;          //指向要发送数据的缓冲区
    __u64      rx_buf;          //指向要接收数据的缓冲区

    __u32      len;             //发送数据和接收数据缓冲区中数据的长度
    __u32      speed_hz;        //发送/接收这些数据需要的总线速率

    __u16      delay_usecs;
    __u8       bits_per_word;   //发送/接收这些数据在 SPI 总线上，每字是多少位
    __u8       cs_change;
    __u32      pad;
}
```

len 是指 tx_buf 和 rx_buf 所指向的缓冲区长度。

speed_hz 不能大于 SPI_IOC_WR_MAX_SPEED_HZ 的总线速率。

由于 iMX28xx 处理器的 SPI 控制器只支持半双工，因此 struct spi_ioc_transfer 结构体中的 tx_buf 和 rx_buf 只能设置一个有效，另一个必须设置为 0，否则调用 ioctl 时会返回 1 提



示操作错误。

13.6.3 示例代码

程序清单 13.20 是 Linux 源码中自带的 SPI 测试代码，做了一些修改后可以读取 MX25L1635E 型号的 SPI Flash 的 ID。读取 MX25L1635E ID 的命令码为 0x9F，其 ID 为 0XC22515，因此在 SPI 接口上接上 MX25L1635E 器件运行此测试程序将会看到读取回来的数据为 C2 25 15。

MX25L1635E 是一款支持四线通讯的 SPI Flash，通讯时钟高达 75MHz，64Mb 存储容量被划分为 4K 与 64K 两组扇区，且支持扇区及块除擦。其电路如图 13.18 所示：

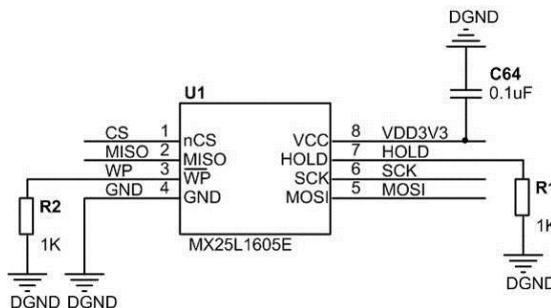


图 13.18 SPI Flash 信号连接图

测试时需要将上图中四个悬空的引脚 CS、MISO、MOSI、SCK 分别连接开发板的 CS、MISO、MOSI、CLK 四个管脚，VDD3V3 及 DGND 分别连接至开发套件的 3.3V 及 GND 上。访问 MX25L1635E 的测试代码如程序清单 13.20 所示。

程序清单 13.20 SPI 测试代码

```
#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 "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/spidev1.0";
static uint8_t mode = 0;
```



```
static uint8_t bits = 8;
static uint32_t speed = 50000;
static uint16_t delay;
#define WRITE 0
static void transfer(int fd)
{
    int ret;
    int i = 10000;
    uint8_t tx[] = {
        0x9f, 0x00, 0x00, 0x00, 0x01, 0x02,
    };
    uint8_t rx[ARRAY_SIZE(tx)] = {0, };
    struct spi_ioc_transfer tr_txrx[] = {
        {
            .tx_buf = (unsigned long)tx, /* 发送数据缓存区 */
            .rx_buf = 0, /* 半双工通信，接收缓存区置为 0 */
            .len = 1, /* 发送命令码，1 个字节 */
            .delay_usecs = delay,
            .speed_hz = speed,
            .bits_per_word = bits,
        },
        {
            .rx_buf = (unsigned long)rx, /* 接收数据缓存区 */
            .len = 3, /* 接收数据长度为 3 */
            .delay_usecs = delay,
            .speed_hz = speed,
            .bits_per_word = bits,
        }
    };
    ret = ioctl(fd, SPI_IOC_MESSAGE(2), &tr_txrx[0]); /* 发送 2 条消息 */
    if (ret == 1) {
        pabort("can't receive spi message");
    }
    for (ret = 0; ret < tr_txrx[1].len; ret++) {
        if (!(ret % 6))
            puts("");
        printf("% .2X ", rx[ret]);
    }
    puts("");
}

void print_usage(const char *prog)
{
    printf("Usage: %s [-DsbdlHOLC3]\n", prog);
}
```



```
puts(" -D --device    device to use (default /dev/spidev1.0)\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);
}

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 'T':
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;
}
}
}

/*
 * 示例程序为读 MX25L1635E spiflash 的 id 功能,接上 MX25L1635E 器件并运行此测试程序,将会读取      *
器件的 ID 为 0XC22515
*/
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 wr 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);
sleep (1);
transfer(fd);
```



```
close(fd);
return ret;
}
```

程序源码也可以参见光盘中的“开发示例”对应目录下的“spidev_test.c”文件，将程序编译成可执行文件（光盘中的示例文件为“spidev_test”）并将其拷贝至开发套件的“/root/”目录下，然后通过串口终端输入以下命令即可运行程序，程序运行后返回结果显示如图 13.19 所示。

```
root@EasyARM-iMX28x ~# ./spidev_test
```

```
spi mode: 0
bits per word: 8
max speed: 50000 Hz (50 KHz)

C2 25 15
```

图 13.19 SPI 示例执行结果

13.7 CAN 接口

在 EasyARM-i.MX287A 开发套件上才有 CAN 接口，其位置如图 13.20 所示。



图 13.20 EasyARM-i.MX287A

13.7.1 使用 CAN 设备

开发套件的 Linux BSP 中提供了 socket CAN 接口。在配置 CAN 设备前需要先关闭 CAN 设备，关闭 CAN 设备可使用如下指令：

```
root@EasyARM-iMX28x ~# ifconfig can0 down
root@EasyARM-iMX28x ~# ifconfig can1 down
```

然后设置波特率，使用如下命令：

```
root@EasyARM-iMX28x ~# echo 1000000 > /sys/devices/platform/FlexCAN.0/bitrate
root@EasyARM-iMX28x ~# echo 1000000 > /sys/devices/platform/FlexCAN.1/bitrate
```

可查看波特率：

```
root@EasyARM-iMX28x ~# cat /sys/devices/platform/FlexCAN.0/bitrate
root@EasyARM-iMX28x ~# cat /sys/devices/platform/FlexCAN.1/bitrate
```

开启两个 CAN 设备：

```
root@EasyARM-iMX28x ~# ifconfig can0 up
root@EasyARM-iMX28x ~# ifconfig can1 up
```

使用开发板上的 CAN0 接口进行测试，CTX0、CRX0 通过 CAN 收发器（请参考光盘资料中的《EasyARM-i.MX283(7)A 硬件使用手册 V1.00.pdf》）连接到 ZLG 的 CANScope 或 CANalyst 工具上。编译运行下面的程序：



程序清单 13.21 CAN 测试程序

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <linux/socket.h>
#include <linux/can.h>
#include <linux/can/error.h>
#include <linux/can/raw.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>

#ifndef AF_CAN
#define AF_CAN 29
#endif
#ifndef PF_CAN
#define PF_CAN AF_CAN
#endif

static void print_frame(struct can_frame *fr)
{
    int i;
    printf("recv: can.id=0x%08x ", fr->can_id & CAN_EFF_MASK);
    //printf("%08x\n", fr->can_id);
    printf("dlc = %d ", fr->can_dlc);
    printf("data = ");
    for (i = 0; i < fr->can_dlc; i++)
        printf("%02x ", fr->data[i]);
    printf("\n");
}

#define errout(_s) fprintf(stderr, "error class: %s\n", (_s))
#define errcode(_d) fprintf(stderr, "error code: %02x\n", (_d))

static void handle_err_frame(const struct can_frame *fr)
{
    if (fr->can_id & CAN_ERR_TX_TIMEOUT) {
        errout("CAN_ERR_TX_TIMEOUT");
    }
    if (fr->can_id & CAN_ERR_LOSTARB) {
```



```
errout("CAN_ERR_LOSTARB");
errcode(fr->data[0]);
}

if (fr->can_id & CAN_ERR_CRTL) {
    errout("CAN_ERR_CRTL");
    errcode(fr->data[1]);
}

if (fr->can_id & CAN_ERR_PROT) {
    errout("CAN_ERR_PROT");
    errcode(fr->data[2]);
    errcode(fr->data[3]);
}

if (fr->can_id & CAN_ERR_TRX) {
    errout("CAN_ERR_TRX");
    errcode(fr->data[4]);
}

if (fr->can_id & CAN_ERR_ACK) {
    errout("CAN_ERR_ACK");
}

if (fr->can_id & CAN_ERR_BUSOFF) {
    errout("CAN_ERR_BUSOFF");
}

if (fr->can_id & CAN_ERR_BUSERROR) {
    errout("CAN_ERR_BUSERROR");
}

if (fr->can_id & CAN_ERR_RESTARTED) {
    errout("CAN_ERR_RESTARTED");
}

#define myerr(str)      fprintf(stderr, "%s, %s, %d: %s\n", __FILE__, __func__, __LINE__, str)

static int test_can_rw(int fd, int master)
{
    int ret, i;
    struct can_frame fr, frdup;
    struct timeval tv;
    fd_set rset;

    while (1) {
        tv.tv_sec = 1;
        tv.tv_usec = 0;
        FD_ZERO(&rset);
        FD_SET(fd, &rset);
```



```
ret = select(fd+1, &rset, NULL, NULL, NULL);
if (ret == 0) {
    myerr("select time out");
    return -1;
}

/* select 调用无错返回时，表示有符合规则的数据帧到达 */
ret = read(fd, &frdup, sizeof(frdup));
if (ret < sizeof(frdup)) {
    myerr("read failed");
    return -1;
}
if (frdup.can_id & CAN_ERR_FLAG) { /* 检查数据帧是否错误 */
    handle_err_frame(&frdup);
    myerr("CAN device error");
    continue;
}
print_frame(&frdup); /* 打印数据帧信息 */
ret = write(fd, &frdup, sizeof(frdup)); /* 把接收到的数据帧发送出去 */
if (ret < 0) {
    myerr("write failed");
    return -1;
}
}

return 0;
}

int main(int argc, char *argv[])
{
    int s;
    int ret;
    struct sockaddr_can addr;
    struct ifreq ifr;
    int master;

    srand(time(NULL));
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW); /* 创建套接字 */
    if (s < 0) {
        perror("socket PF_CAN failed");
        return 1;
    }
}
```



```
/* 把套接字绑定到 can0 接口 */
strcpy(ifr.ifr_name, "can0");
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if (ret < 0) {
    perror("ioctl failed");
    return 1;
}

addr.can_family = PF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

ret = bind(s, (struct sockaddr *)&addr, sizeof(addr));
if (ret < 0) {
    perror("bind failed");
    return 1;
}

/* 设置过滤规则 */
if (0) {
    struct can_filter filter[2];
    /* 第 1 个规则是可以接收 ID 为 0x200 & 0xFFF 的数据帧 */
    filter[0].can_id = 0x200 | CAN_EFF_FLAG;
    filter[0].can_mask = 0xFFF;
    /* 第 2 个规则是可以接收 ID 为 0x20F& 0xFFF 的数据帧 */
    filter[1].can_id = 0x20F | CAN_EFF_FLAG;
    filter[1].can_mask = 0xFFF;

    /* 启用过滤规则，只要 CAN0 接收到的数据帧满足上面 2 个规则中的任何一个也被接受*/
    ret = setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &filter, sizeof(filter));
    if (ret < 0) {
        perror("setsockopt failed");
        return 1;
    }
}
test_can_rw(s, master);      /*进入测试*/

close(s);
return 0;
}
```

注意，在测试执行之前，开发套件上的 CAN 接口必须通过 CAN 收发器连接到其它 CAN 设备或节点（如 ZLG 的 CANScope 或 CANalyst 工具），以组成 CAN 总线。应用程序要同通过 CAN 发送数据之前，CAN 接口会通过 CAN 收发器试图获得 CAN 总线的仲裁。如果获得仲裁，才会发送数据。所以如果没有完整的硬件连接，EasyARM-i.MX287A/B 上的 CAN 接口是不会有任何信号的。



在 CANTest 软件中，设置帧 ID 为 0x20F 或 0x200，数据为图 13.21 所示：



图 13.21 CANTest 软件发送数据设置

注： CANScope 或 CANalyst 等工具为广州致远电子开发的通用 CAN 分析仪，可通过致远电子销售购买，光盘中不提供对应的测试软件及使用方法。如无此工具，用户可自行使用其他方式测试 CAN 通信，CAN 收发器推荐使用 CTM8251AT 模块，接法如图 13.22 所示：

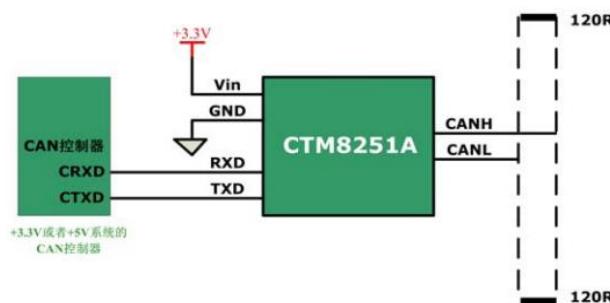


图 13.22 CAN 隔离收发器 CTM8251AT

在 CANTest 软件里点击发送两次，运行结果如所下示：

```
root@EasyARM-iMX28x:/mnt/nfs# ./cantest
=====
-----
0000020f
dlc = 8
data = 00 01 02 03 04 05 06 07
=====
-----
0000020f
dlc = 8
data = 00 01 02 03 04 05 06 07
=====
```

13.7.2 socket CAN 编程指南

1. 创建套接字

就像 TCP/IP 协议一样，在使用 CAN 网络之前需要先打开一个套接字，CAN 的套接字使用到了一个新的协议族，所以在调用 `socket()` 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，另一个是广播管理协议。可以以下所示的方式来打开一个套接字：



```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

或者

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

2. 绑定 CAN 接口

在成功创建一个套接字之后，通常需要使用 bind() 函数将套接字绑定在某个指定的 CAN 接口上（这和 TCP/IP 使用不同的 IP 地址不同）。在绑定 (CAN_RAW) 或连接 (CAN_BCM) 套接字之后，就可以在套接字上使用 read()/write()，也可以使用 send()/sendmsg() 和对应的 recv* 操作。

基本的 CAN 帧结构和套接字地址结构定义在/include/linux/can.h，如程序清单 13.22 所示。

程序清单 13.22 can_frame 的定义

```
/*
 * 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()，返回给用户空间的数据是一个 struct can_frame 的结构体。

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，如程序清单 13.23 所示。

程序清单 13.23 struct sockaddr_can 结构体

```
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(), 如程序清单 13.24 所示。

程序清单 13.24 绑定接口

```
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。这样套接字就可以从所有使用的 CAN 接口接收 CAN 帧。recvfrom()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()可以指定从哪个接口发送。

3. 接收/发送帧

从一个 CAN_RAW 套接字上读取 CAN 帧也就是读取 struct can_frame 结构体，如程序清单 13.25 所示。

程序清单 13.25 接收 CAN 帧

```
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 帧也是类似的用到 write()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_ifindex = 0)，推荐使用 recvfrom() 获取数据源接口信息，程序清单 13.26 所示。

程序清单 13.26 获取数据源接口信息



```
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，如程序清单 13.27。

程序清单 13.27 指定输出接口的详细信息

```
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));
```

4. 使用过滤器

在上面的阶段中，我们从 CAN 接口中接收所有的数据帧，也不管我们是不是感兴趣。如果我们只想要指定 ID 的数据帧，那我们需要使用过滤器。

- 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。过滤规则定义在/include/linux/can.h 中，如程序清单 13.28 所示。

程序清单 13.28 can_filter 的定义

```
struct can_filter {
    canid_t can_id;
    canid_t can_mask;
};
```

过滤规则的匹配：

```
<接收帧 id> & mask == can_id & mask
```

启用过滤器的示例如程序清单 13.29 所示。

程序清单 13.29 启用过滤器示例代码

```
/* valid bits in CAN ID for frame formats */
#define CAN_SFF_MASK 0x0000007FFU /* 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 */\n\nstruct can_filter rfilter[2];\n\nrfilter[0].can_id    = 0x123;\nrfilter[0].can_mask = CAN_SFF_MASK;\nrfilter[1].can_id    = 0x200;\nrfilter[1].can_mask = 0x700;\n\nsetsockopt(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（所有成员为 0），这样原始套接字就会忽略接到的 CAN 帧。

- 原始套接字选项 CAN_RAW_ERR_FILTER

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不同的各类，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 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。错误掩码的使用示例如下：

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );\nsetsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, &err_mask, sizeof(err_mask));
```



第14章 嵌入式 Linux Qt 编程

本章主要介绍如何使用开发套件进行基于嵌入式 Linux Qt 库的图形应用程序开发。本章所引用的范例源码位于光盘“3、Linux\4、开发示例\5、QT”目录下。

注：本章不适用于 EasyARM-i.MX280A 开发套件。

14.1 背景知识

在阅读本章前，希望读者对下面所列举的知识点有一定的了解，这样有助于更好地理解本章内容。

- C++ 基础知识，了解简单的类，继承，重载等面向对象概念；
- Linux 基础知识，了解基本的 Shell 命令，懂得对 Linux 进行简单的配置；
- 嵌入式开发基础知识，了解基本的嵌入式开发流程，了解简单的嵌入式开发工具的使用，如调试串口的使用，NFS 文件系统的挂载方法；
- 交叉编译与动态库的基础知识。

14.2 Qt 介绍

14.2.1 Qt 简介

Qt 是一个跨平台应用程序和 UI 开发框架。使用 Qt 只需一次性开发应用程序，无须重新编写源代码，便可跨不同桌面和嵌入式操作系统部署这些应用程序。Qt 原为奇趣科技公司（Trolltech，www.trolltech.com）开发维护，后来被 nokia 公司收购，在 nokia 的推动下，Qt 的发展非常快速，版本不断更新。目前最新的 Qt 主版本为 4.8.1，所支持的平台如图 14.1 所示：



图 14.1 Qt 支持的平台

14.2.2 Qt/E 简介

嵌入式 Linux 发行版本上的 Qt 属于 Qt 的 Embedded Linux 分支平台。这个分支平台一般被简称为 Qt/E。Qt/E 在原始 Qt 的基础上，做了许多出色的调整以适合嵌入式环境。同 Qt/X11 相比，Qt/E 很节省内存，因为它不需要 X server 或是 Xlib 库，它在底层摒弃了 Xlib，采用 framebuffer 作为底层图形接口。Qt/E 的应用程序可以直接写内核帧缓冲，因此它在嵌入式 Linux 系统上的应用非常广泛。

Qt/E 所面对的硬件平台较多，当开发人员需要在某硬件平台上移植 Qt/E 时，需要下载 Qt 源代码，利用交叉编译器编译出 Qt 库。接着需要将 Qt 库复制两份，一份放置在开发主机上，供编译使用；一份放在目标板上，供运行时动态加载使用。流程如图 14.2 所示：

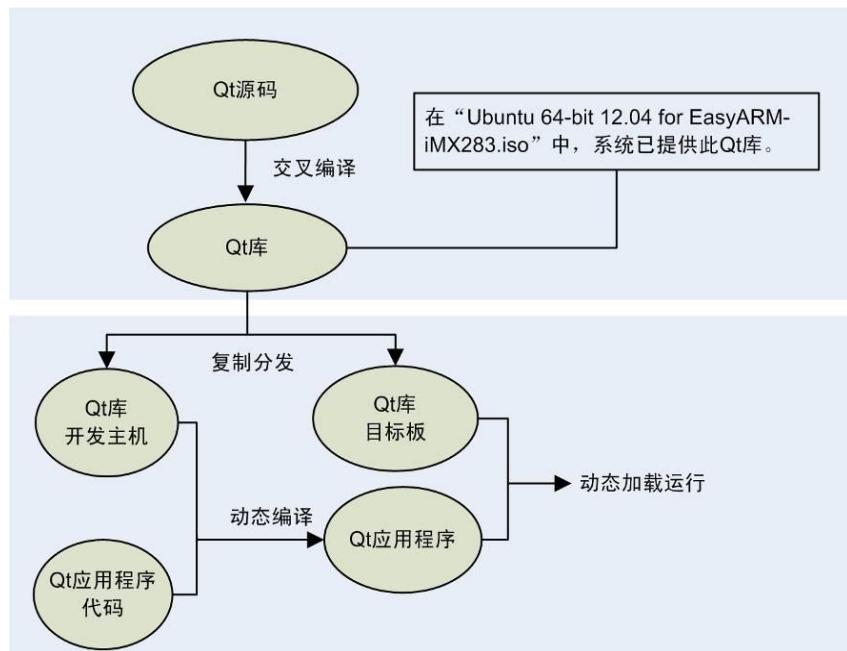


图 14.2 Qt/E 编程图示

开发套件提供已交叉编译好的 Qt-4.7.3 库（注意：EasyARM-i.MX28x 系列开发套件提供的是 Qt-4.7.3 库，而早期产品 EasyARM-iMX283/287 开发套件 V1.05 以及 V1.05 之前光盘资料使用的是 Qt-4.8.0，从 V1.06 开始使用 Qt-4.7.3），用户无需编译，只需要进行简单设置即可把 Qt-4.7.3 作为默认的 Qt 库。用户需得到此库的两份拷贝，一份内嵌在交叉编译工具链中，供编译时链接使用，另一份内嵌在目标板文件系统中，放置在系统库目录下，供 Qt 程序运行时动态加载使用。

14.3 编译环境的搭建 (Qt-4.7.3)

用户按 6.2 章节述的方法安装交叉编译器所得到的 Qt 库默认是 Qt-4.8.0 版本的。该版本的 Qt 库存在 USB 鼠标位置溢出的问题（鼠标移动产生的坐标可以超出视窗范围）。将 Qt 库版本降级为 Qt-4.7.3 即可解决该问题。按下文所介绍方法操作，即可将 Qt 库降级为 Qt-4.7.3。

14.3.1 交叉编译工具链的 Qt 库替换

首先，把光盘资料中的“3、Linux\2、工具软件\Linux 工具软件\qt4.7.3.tar.bz2”压缩文件通过命令拷贝到 Linux 主机的/opt 目录下（非 root 用户需加上 sudo 前缀）。然后执行如下所示指令，将 qt4.7.3.tar.bz2 文件解压到当前目录：

```
vmuser@Linux-host:/opt$ sudo tar -jxvf qt4.7.3.tar.bz2
```

然后需要修改 Linux 主机的 PATH 环境变量。执行如下所示指令即可打开/etc/profile 文件，对 PATH 环境变量进行修改。

```
vmuser@Linux-host:/opt$ sudo vi /etc/profile
```

打开 profile 文件后，在该文件中找到 PATH 环境变量设置的那行文本，在其中添加 Qt 库的路径（/opt/qt4.7.3/bin），如下文所示（注意：路径插入的位置必须是在其他路径前面）：

```
PATH=/opt/qt4.7.3/bin:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin:$PATH
```

修改完后保存并退出 vi 编辑器，然后执行如下命令使环境变量 PATH 生效（注意：命令为“.(点) (空格) /etc/profile”）。



```
vmuser@Linux-host:/opt$ ./etc/profile
```

经过上面的设置之后，即可把交叉编译工具链中原来的 Qt-4.8.0 替换为 Qt-4.7.3，使用 qmake -v 可查看到当前的 Qt 版本为 Qt-4.7.3。

```
vmuser@Linux-host:/opt$ qmake -v  
QMake version 2.01a  
Using Qt version 4.7.3 in /opt/qt4.7.3/lib  
vmuser@Linux-host:/opt$
```

注意：如果用户使用的是 64 位的 Linux 主机，运行 qmake -v 可能会提示无法加载共享库的问题，那么就需要安装共享库：apt-get install lib32stdc++6，操作命令如下（更详细的操作请见 6.2.25）：

```
vmuser@Linux-host:/opt$ sudo apt-get install lib32stdc++6
```

14.3.2 目标板文件系统的 Qt 库替换

注意：该小节操作仅针对原来使用 EasyARM-i.MX283/287 学习套件 V1.05 及早期光盘资料中的目标文件系统的用户。

如果用户直接使用 EasyARM-i.MX28x 系列开发套件资料或使用 EasyARM-i.MX283/287 学习套件 V1.06 光盘资料提供的文件系统（新文件系统），则无需更新文件系统中的 Qt 库，可直接跳过本节。

将原目标文件系统中的 Qt 库更新为 Qt-4.7.3 所需替换的文件或目录详见表 14.1。

表 14.1 更新目标文件系统 Qt 库所需替换的文件或目录

主要替换内容	所在路径	备注
图形桌面应用框架	/usr/share/zylaucher	用新文件系统下对应的同名目录替换整个目录
Qt 库	/usr/lib	用新文件系统下对应的同名目录替换整个目录
基于 Qt 的实用工具	/usr/bin/	替换 ts_calibrate、ts_harvest、ts_print、ts_test 及 ts_print_raw

14.4 Qt 开发体验

14.4.1 编译 helloworld 程序

“helloworld”是一个最简单最基本的 Qt 程序，其开发流程如图 14.3 所示。

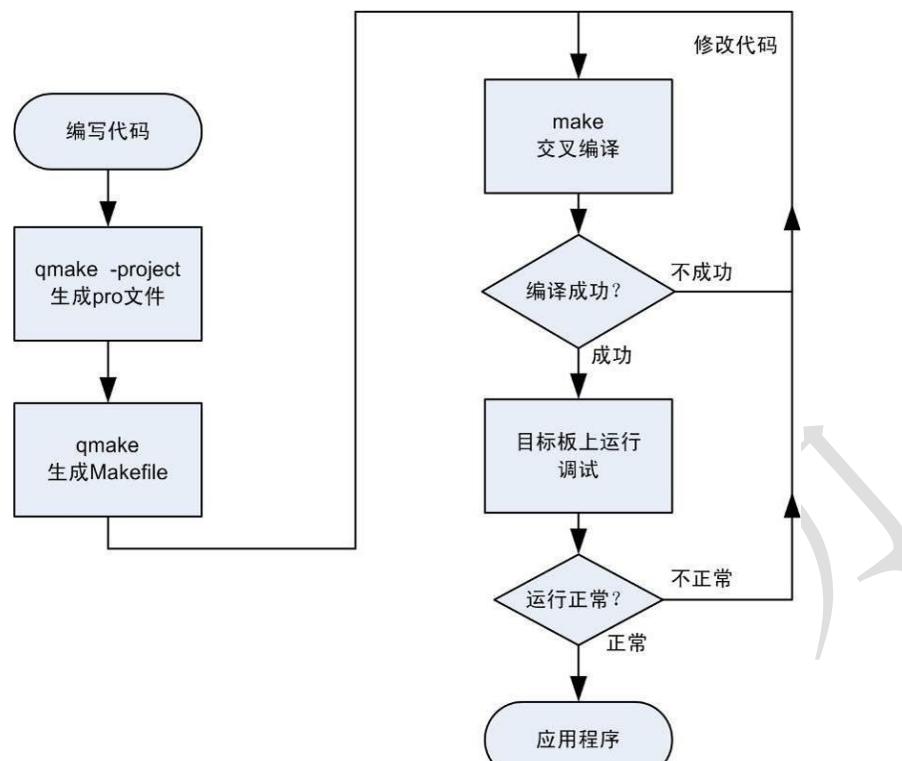


图 14.3 helloworld 开发流程图

helloworld.cpp 程序代码如程序清单 14.1 所示。

程序清单 14.1 helloworld 程序代码

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel label(QString("helloworld qt"));
    label.show();
    app.exec();
}
```

- 第 1 行代码包含了头文件 `<QtGui>`，它包含了 Qt 的 GUI 类定义的头文件。
- 第 2 行代码是 `main` 函数定义，它包含了用于接收命令行参数的两个形参 `argc`、`argv`。
- 第 4 行代码利用 `main` 传递的命令行参数 (`argc`, `argv`) 构造了一个 `QApplication` 类的对象 `app`。这个对象负责管理整个用户资源，是 QtGUI 程序必须包含的一个对象。通过这个对象，来自窗口系统和其它资源的事件可以被调度和分发处理。
- 第 5 行代码创建了一个显示 “helloworld qt” 的 `QLabel` 窗口部件 (widget)。窗口部件是一种可视化元素，如：按钮、菜单、滚动条和框架等，在 Qt 中，几乎所有的窗口部件都可以用作窗口。
- 第 6 行代码使 `QLabel` 标签 (`label`) 可见。窗口部件被创建后，默认是隐藏的，这就允许先对其进行初始化设置然后再显示它们，从而避免窗口部件在初始化过程中的闪烁现象。



- 第 7 行代码将应用程序的控制权传递给 Qt。此时程序会进入事件循环状态，这是一种等待模式，程序将等待并响应用户的动作，例如按键或鼠标单击等操作。

将 hellow.cpp 拷贝至~/EasyARM-iMX283/hellow 目录下，运行以下命令将生成 hellow.pro 文件：

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ qmake -project
```

hellow.pro 文件描述整个工程所包含的源码及相应的资源文件。qmake 是 Qt 中用来管理工程的项目工具，有关 qmake 与 pro 文件的相关信息将在下一节做详细的介绍。

执行 qmake 命令，将根据上一步的 pro 文件，生成 makefile 文件。

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ qmake
```

根据 makefile 文件执行 make 命令则可以编译出可执行程序。以后需要再编译时，也只需执行最后一步，即 make 命令。

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ make
```

经过上述步骤，可以在 hellow 目录下见到 hellow 文件，这个文件就是可执行的 Qt 程序。

14.4.2 在目标板上运行 hellow 程序

1. 作为普通应用程序启动

把 hellow 文件通过 nfs 或其它方式下载到开发套件的/root/目录下，然后通过串口终端登录开发套件的 Linux 系统，并通过如下指令即可启动该程序 (**J7 (DUART)** 需要通过串口线与电脑相连)。

```
root@EasyARM-iMX28x ~# ./hellow
```

hellow 程序运行后，如图 14.4 所示。



图 14.4 运行 hellow 程序

2. 作为窗口服务器启动

如果用户的 Linux 系统上电启动后没有加载任何 qt 程序(开发套件演示的 Linux 系统上电后会自动启动 zylauncher 图形框架程序 framework，该程序是一个被用作窗口服务器的 Qt



程序), 则运行 hellow 程序的命令需要加上-qws 参数, 具体指令如下:

```
root@EasyARM-iMX28x ~# ./hellow -qws
```

-qws 指明这个 Qt 程序同时作为一个窗口服务器运行, 在目标系统上启动的第一个 Qt 程序应使用此参数启动。

此外, 在启动作为窗口服务器的 Qt 程序前, 还需要先设定其鼠标设备。通过修改环境变量 QWS_MOUSE_PROTO 的值可以指定 Qt 的鼠标设备, 示例命令如下:

```
export QWS_MOUSE_PROTO=[Tslib:/dev/input/event%d] [LinuxInput:/dev/input/event%d]
```

上述命令中 Tslib 字段用于指定触摸屏设备文件, LinuxInput 字段用于指定鼠标设备文件, “[]” 符号表示该字段为可选字段, 实际命令中不需要输入该符号。Qt 支持多个鼠标设备, 即允许 QWS_MOUSE_PROTO 环境变量的值由多个可选段组成, 不同字段之间用空格隔开。

正常情况下, 输入设备文件均位于 /dev/input 目录下, 但具体的设备文件名(如 event0)对应的是触摸屏还是鼠标, 则可能因实际应用而各有差异。

如程序清单 14.2 所示, 是一段与“start_zylauncher”文件内容相似的启动脚本示例代码, 该脚本用于自动探测接入开发套件的鼠标设备是触摸屏还是普通鼠标, 并根据实际探测的结果自动设置 Qt 程序执行所需的环境变量, 然后启动首个 Qt 程序。

程序清单 14.2 启动脚本代码

```
#!/bin/sh
SCRIPT_PATH=`cd "$(dirname "$0")"; pwd`
cd "$SCRIPT_PATH"

devs_list=`ls /dev/input/event*`
has_ts=0
QWS_MOUSE_PROTO=""
QWS_KEYBOARD=""
for dev in $devs_list
do
    ./input_type "$dev"
    case $? in
        1) QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""Tslib:$dev"
           has_ts=1
           ;;
        2) QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""LinuxInput:$dev"
           ;;
        3) QWS_KEYBOARD="$QWS_KEYBOARD ""LinuxInput:$dev"
           ;;
        *) ;;
    esac
done

# delete space in head and tail
QWS_MOUSE_PROTO=`echo $QWS_MOUSE_PROTO`
```



```
QWS_KEYBOARD=`echo $QWS_KEYBOARD`  
  
echo "[QWS_MOUSE_PROTO=$QWS_MOUSE_PROTO]"  
echo "[QWS_KEYBOARD=$QWS_KEYBOARD]"  
  
export "QWS_MOUSE_PROTO=$QWS_MOUSE_PROTO"  
export "QWS_KEYBOARD=$QWS_KEYBOARD"  
  
# 判断触摸屏校准文件是否存在  
#if [ "$has_ts" == "1" -a ! -f /etc/pointercal ] ; then  
#     ts_calibrate  
#fi  
  
export TSLIB_CALIBFILE=/etc/pointercal  
export TSLIB_CONFFILE=/etc/ts.conf  
export TSLIB_PLUGINDIR=/usr/lib/ts  
  
export LDPATH=/usr/lib  
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$LDPATH  
export QT_QWS_FONTDIR=$LDPATH/fonts  
export QT_PLUGIN_PATH=$LDPATH/qt/plugins:$SCRIPT_PATH/framework/plugins  
export POINTERCAL_FILE=$SCRIPT_PATH/framework/qt_pointercal  
  
.fb_set /dev/fb0  
  
# 这里可以修改成其他需要启动的首个 Qt 程序  
.hellow -qws
```

将脚本代码保存为 qtLauncher 文件，并拷贝至开发套件 Linux 系统的“/usr/share/zhiyuan/zylauncher/”目录下，同时修改开发套件中 Linux 系统的启动文件由原来的 start_zylauncher 更改为 qtLauncher，即修改根文件系统“/etc/rc.d/init.d/”目录下的 start_userapp 文件，在串口终端中利用 vi 编辑器修改该文件的命令如下：

```
root@EasyARM-iMX28x ~# vi /etc/rc.d/init.d/start_userapp
```

用 vi 编辑器修改完开发套件根文件系统的 start_userapp 文件后如图 14.5 所示。



```
#you can add your app start_command here

# start ssh
/bin/dropbear
#start qt command,you can delete it
export TSLIB_PLUGINDIR=/usr/lib/ts/
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_TSDEVICE=/dev/input/ts0
export TSLIB_CALIBFILE=/etc/pointercal
export QT_QWS_FONTDIR=/usr/lib/fonts
export QWS_MOUSE_PROTO=Tslib:/dev/input/ts0

boardname=`cat /sys/devices/platform/zlg-systemType/board_name`
if [ ! $boardname = "280" ]
then
    /usr/share/zylauncher/qtLauncher >/dev/null &
fi

:wq
```

图 14.5 修改 start_userapp 文件

注意：由于大多数串口终端在处理 vi 编辑器相关指令时容易出错，建议在上位机中修改该文件，然后通过文件拷贝方式覆盖开发套件上的 start_userapp 文件，或通过挂载 NFS 根文件系统的方式进行测试。

在本例中，作为窗口服务器的 hellow 程序启动成功后，界面如图 14.6 所示。

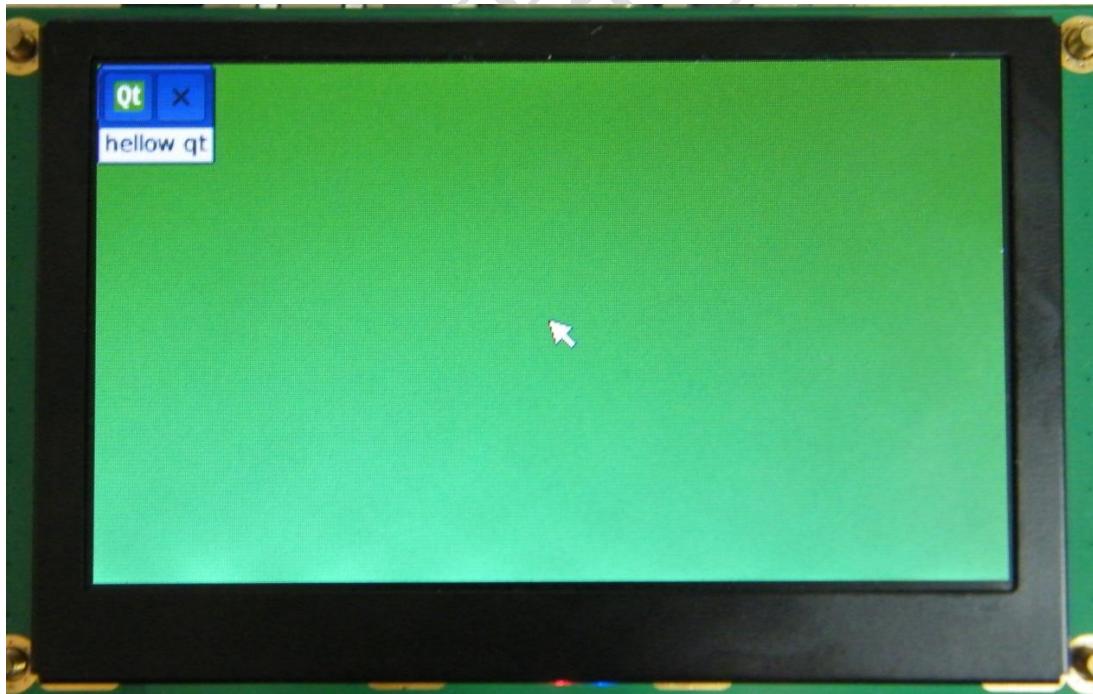


图 14.6 hellow 程序运行界面

14.5 qmake 与 pro 文件

qmake 是一个用来为不同平台和编译器生成 Makefile 的工具。手写 Makefile 是比较困难并且容易出错的，尤其是需要给不同的平台和编译器组合写几个 Makefile。使用 qmake，编程人员只需创建一个简单的“pro 文件”并且运行 qmake 即可生成恰当的 Makefile。



对于某些简单的项目，可以在其项目顶层目录下直接 qmake -project 自动生成“pro 文件”，例如 hellow 程序。但对于一些复杂的 Qt 程序，自动生成的“pro 文件”可能并不符合要求，这时就需要程序员手动改写“pro 文件”。因此，下来就简单介绍“pro 文件”相关内容。

pro 文件主要有三种模板：

- app (**应用程序模板**)
- lib (**库模板**)
- subdirs(**递归编译模板**)

在 pro 文件中可以通过以下代码指定所使用的模板。

```
TEMPLATE = app
```

如果不指定 TEMPLATE，pro 文件默认为 app 模式。项目中使用最多也是 app 模式。app 模式的 pro 文件主要用于构造适用于应用程序的 Makefile。

14.5.1 pro 文件例程

下面通过一个例子简单地介绍 app 模式下 pro 文件(**关于 lib 与 subdirs 模式的 pro 文件，用户可以参看 qmake 的相关文档**)。这个 pro 文件内容将完全手动编写。在实际的项目中，程序员可以使用 qmake -project 生成 pro 文件，再在这个 pro 文件上进行相应修改。

假设工程项目中有如下源代码文件：

```
hello.cpp  
hello.h  
main.cpp
```

首先，需要在 pro 文件中指定 cpp 文件，可以通过 SOURCES 变量指定，代码如下：

```
SOURCES += hello.cpp
```

对于每一个 cpp 文件，都需要如此指定。代码如下：

```
SOURCES += hello.cpp  
SOURCES += main.cpp
```

也可以通过反斜线形式指定：

```
SOURCES = hello.cpp \  
         main.cpp
```

下来需要指定所需的 h 文件，通过 HEADERS 指定。pro 文件中代码如下：

```
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp
```

项目生成的可执行程序文件名会自动设置，程序文件名与 pro 文件名一致，但在不同的平台下，其扩展名是不同的。比如 pro 文件名为 hello.pro，在 Windows 平台下，其会生成 hello.exe；在 Linux 平台下，会生成 hello。可以使用 TARGET 指定可执行程序的基本文件名。代码如下：

```
TARGET = helloworld
```

接下来最后一步便是设置 CONFIG 变量。由于此项目为一个 Qt 项目，因此要将 qt 添加到 CONFIG 变量中，以告知 qmake 将 Qt 相关的库与头文件信息添加到 Makefile 文件中。现在完整的 pro 文件内容如下所示：



```
CONFIG += qt  
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp  
TARGET = helloworld
```

现在就可以利用此 pro 文件生成 Makefile，命令如下：

```
qmake -o Makefile hello.pro
```

如果当前目录下只有一个 pro 文件，可以直接使用命令：

```
qmake
```

在生成 Makefile 文件后，即可使用 make 命令进行编译。

14.5.2 pro 文件常见配置

对于 app 模式的 pro 文件，常用的变量有下面这些：

- HEADERS：指定项目的头文件 (.h);
- SOURCES：指定项目的 C++文件 (.cpp);
- FORMS：指定需要 uic 处理的由 Qt desinger 生成的.ui 文件;
- RESOURCES：指定需要 rcc 处理的 .qrc 文件;
- DEFINES：指定预定义的 C++预处理器符号;
- INCLUDEPATH：指定 C++编译器搜索全局头文件的路径;
- LIBS：指定工程要链接的库;
- CONFIG：指定各种用于工程配置和编译的参数;
- QT：指定工程所要使用的 Qt 模块（默认是 core gui，对应于 QtCore 和 QtGui）;
- TARGET：指定可执行文件的基本文件名;
- DESTDIR：指定可执行文件放置的目录。

其中，CONFIG 变量常用的参数如下：

- debug：编译出具有调试信息的可执行程序;
- release：编译不带调试信息的可执行程序，与 debug 同时存在时，release 失效;
- qt：指应用程序使用 Qt，此选项是默认包括的;
- dll：动态编译库文件;
- staticlib：静态编译库文件;
- console：指应用程序需要写控制台。

14.6 Qt 编程简单入门

14.6.1 例程讲解

在下面的描述中，将说明在一个窗口中如何排列多个控件，并学习如何利用 signal 和 slot (信号与槽) 使控件同步。

程序要求用户通过 slider 输入年龄，并利用 lcd 显示年龄。程序中使用了三个控件：QLCDNumber, QSlider 和 QWidget。QWidget 是这个程序的主窗口。QLCDNumber 和 QSlider 被放在 QWidget 中，所以它们是 QWidget 的 children。反过来，也可以称 QWidget 是 QLCDNumber 和 QSlider 的 parent。QWidget 没有 parent，因为它是程序的顶层窗口。在 QWidget 及其子类的构造函数中，都有一个 QWidget*参数，用来指定它们的父控件。

源代码如程序清单 14.3 所示：



程序清单 14.3 Qt 例程代码

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QLCDNumber>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Enter Your Age");
    QLCDNumber *lcd = new QLCDNumber;
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     lcd, SLOT(display(int)));
    slider->setValue(35);
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(lcd);
    layout->addWidget(slider);
    window->setLayout(layout);
    window->show();
    return app.exec();
}
```

- 第 8, 9 行建立程序的主窗口控件，设置标题。
- 第 10 到 12 行创建主窗口的 children，并设置允许值的范围。
- 第 13 到第 14 行是 lcd 和 slider 的连接，以使 lcd 能同步显示 slider 所指示的值。当 slider 的值发生变化时，会发出 valueChanged(int)信号，lcd 的 display (int)函数就会相应地设置一个新值。
- 第 15 行将 slider 的值设置为 35，这时 slider 发出 valueChanged(int)信号，int 的参数值为 35，这个参数传递给 lcd 的 display(int)函数，将 lcd 的值也设置为 35。
- 在第 17 至 18 行，使用了一个布局管理器排列 lcd 和 slider 控件。布局管理器能够根据需要确定控件的大小和位置。Qt 有三个主要的布局管理器：

QHBoxLayout，水平排列控件；

QVBoxLayout，垂直排列控件；

QGridLayout，按矩阵方式排列控件。

- 第 19 行，QWidget::setLayout()把这个布局管理器放在 window 上。这个语句将 lcd 和 slider 的 parent 设为 window，即布局管理器所在的控件。如果一个控件由布局管理器确定它的大小和位置，那么创建它的时候就不必指定一个明确的 parent 控件。

至此，虽然还没有看见 lcd 和 slider 控件的大小和位置，但它们已经水平排列好了，QHBoxLayout 能合理安排它们。

在 Qt 中建立用户界面就是这样简单灵活。程序员的任务就是实例化所需要的控件，按照需要设置它们的属性，把它们放到布局管理器中。同时利用 Qt 的信号和槽机制来管理用



户的交互行为。

14.6.2 信号和槽机制

信号和槽是 Qt 编程的一个重要部分，这个机制可以在对象之间彼此并不了解的情况下将它们的行为联系起来。在上面的例程中，已经连接了信号和槽，发送了信号，触发槽函数的响应，下面将更深入介绍这个机制。

槽和普通的 c++ 成员函数很像。它们可以是虚函数 (`virtual`)，也可被重载 (`overload`)，可以是公有的 (`public`)，保护的 (`protective`)，也可是私有的 (`private`)，它们可以象任何 c++ 成员函数一样被调用，可以传递任何类型的参数。不同在于一个槽函数能和一个信号相连接，只要信号发出了，这个槽函数就会自动被调用。信号和槽函数间的连接通过 `connect` 实现。`connect` 函数语法如下：

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

sender 和 receiver 是 `QObject` 对象 (`QObject` 是所有 Qt 对象的基类) 指针，signal 和 slot 是不带参数的函数原型。`SIGNAL()` 和 `SLOT()` 宏的作用是把他们转换成字符串。

在前面的例子中，已经连接了信号和槽，而在实际使用中还需要考虑如下一些规则：

- 一个信号可以连接到多个槽；

```
connect(slider, SIGNAL(valueChanged(int)), spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)), this, SLOT(updateStatusBarIndicator(int)));
```

当信号发出后，槽函数都会被调用，但是调用的顺序是随机的，不确定的。

- 多个信号可以连接到一个槽；

```
connect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()), this, SLOT(handleMathError()));
```

任何一个信号发出，槽函数都会执行。

- 一个信号可以和另一个信号相连；

```
connect(lineEdit, SIGNAL(textChanged(const QString &)), this, SIGNAL(updateRecord(const QString &)));
```

第一个信号发出后，第二个信号也同时发送。除此之外，信号与信号连接上和信号和槽连接相同。

- 连接可以被删除。

```
disconnect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));
```

这个函数很少使用，一个对象删除后，Qt 自动删除这个对象的所有连接。

信号和槽函数必须有着相同的参数类型，这样信号和槽函数才能成功连接：

```
connect(ftp, SIGNAL(rawCommandReply(int, QString &)), this, SLOT(processReply(int, QString &)));
```

如果信号里的参数个数多于槽函数的参数，多余的参数被忽略：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)), this, SLOT(checkErrorCode(int)));
```

如果参数类型不匹配，或者信号和槽不存在，在 debug 状态时，Qt 会在运行期间给出警告。如果信号和槽连接时包含了参数的名字，Qt 将会给出警告。

本小节的例子，使用 `qmake -project`，`qmake`，`make` 可直接生成可执行程序，无需修改 pro 文件。其运行界面如图 14.7 所示：

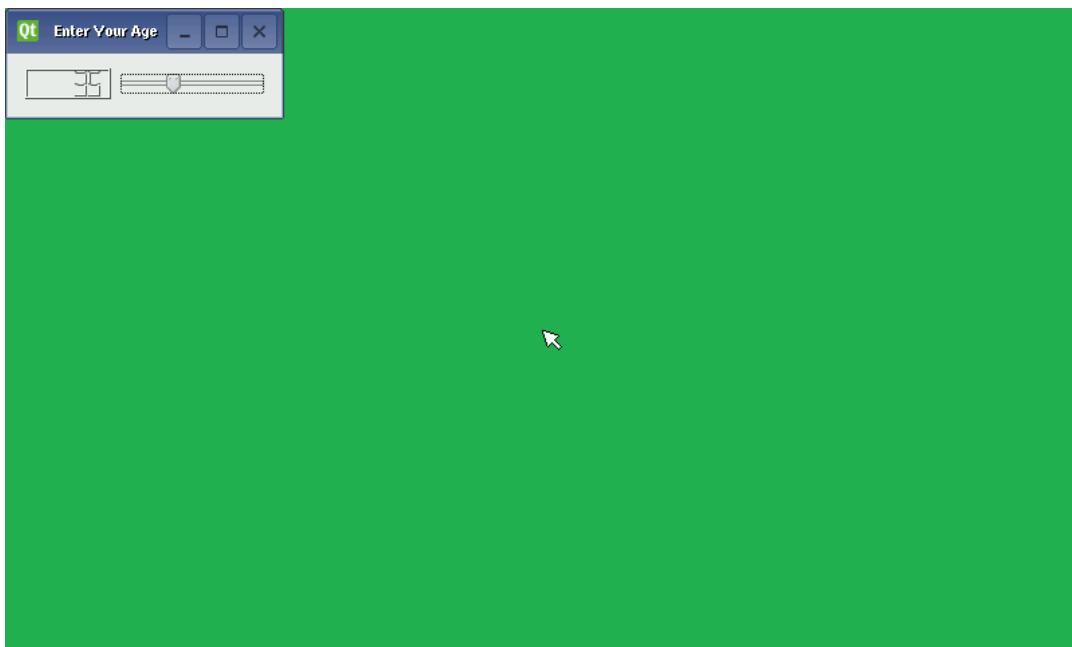


图 14.7 Qt 例程界面

14.7 Qt SDK 的使用

14.7.1 Qt SDK 简介

Qt 是一个跨平台的图形框架，在安装了桌面版本的 Qt SDK 的情况下，用户可以先在 PC 主机上进行 Qt 应用程序的开发调试，待应用程序基本成型后，再将其移植到目标板上。

桌面版本的 Qt SDK 主要包括以下两个部分：

- 用于桌面版本的 Qt 库；
- Qt Creator（集成开发环境）。

Qt Creator 是一个强大的跨平台 IDE，集编辑，编译，运行，调试功能于一体。其代码编辑器支持关键字高亮，上下文信息提示，自动完成，智能重命名等高级功能。IDE 中集成的可视化界面编辑器，可以让用户以所见即所得的方式进行图形程序的设计。其编译，运行无需敲入命令，直接点击按钮或使用快捷键即可完成。同时还支持图形化的调试方式，可以以插入断点，单步运行，追踪变量，查看函数堆栈等方式进行应用程序的调试开发。Qt Creator 主界面如图 14.8 所示。

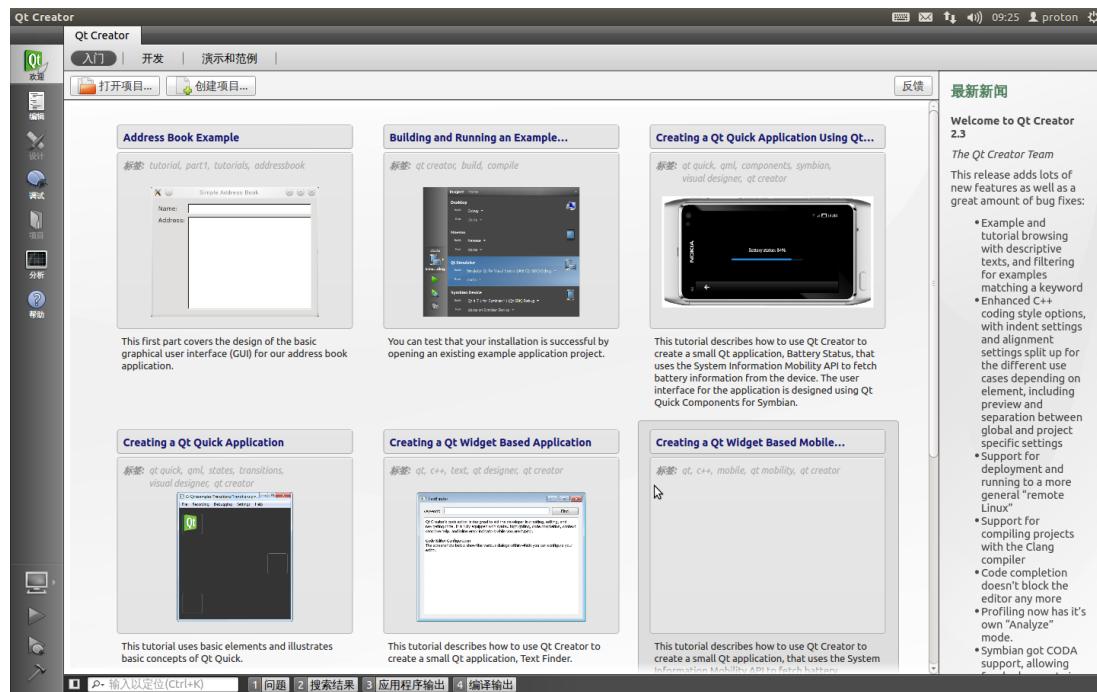


图 14.8 Qt Creator 主界面

14.7.2 Qt SDK 安装

桌面版本的 Qt SDK 支持三个平台：Windows、Linux、Mac。这里只讲述 Linux 桌面版本的 Qt SDK 的安装。其他平台下的安装可参阅官方资料。用户可以在 Qt 官方网站找到三个平台下对应的安装包。推荐通过 ubuntu 下的 apt-get 获取 Linux 版的 Qt SDK。在 Linux 主机可以正常上网的条件下，先进行安装源的更新，否则可能导致 QT-SDK 安装失败。安装源更新命令如下：

```
vmuser@Linux-host:~$ sudo apt-get update
```

其执行过程如图 14.9 所示。

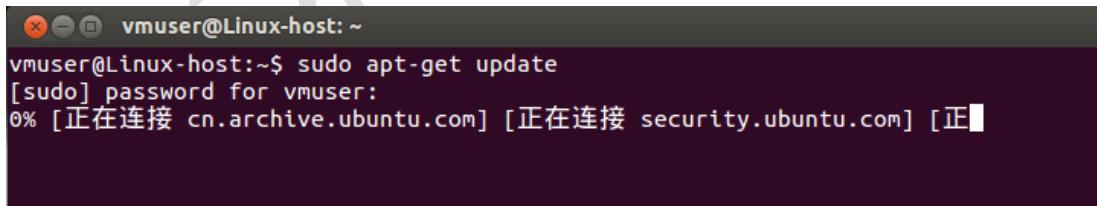


图 14.9 更新 Linux 安装源

安装源更新后，可以使用如下命令获取并安装 QT SDK：

```
vmuser@Linux-host:~$ sudo apt-get install qt-sdk
```

在 QT SDK 的安装过程中可能会出现如图 14.10 所示的警告窗口，这时只需要选中该窗口并按“回车”键即可。

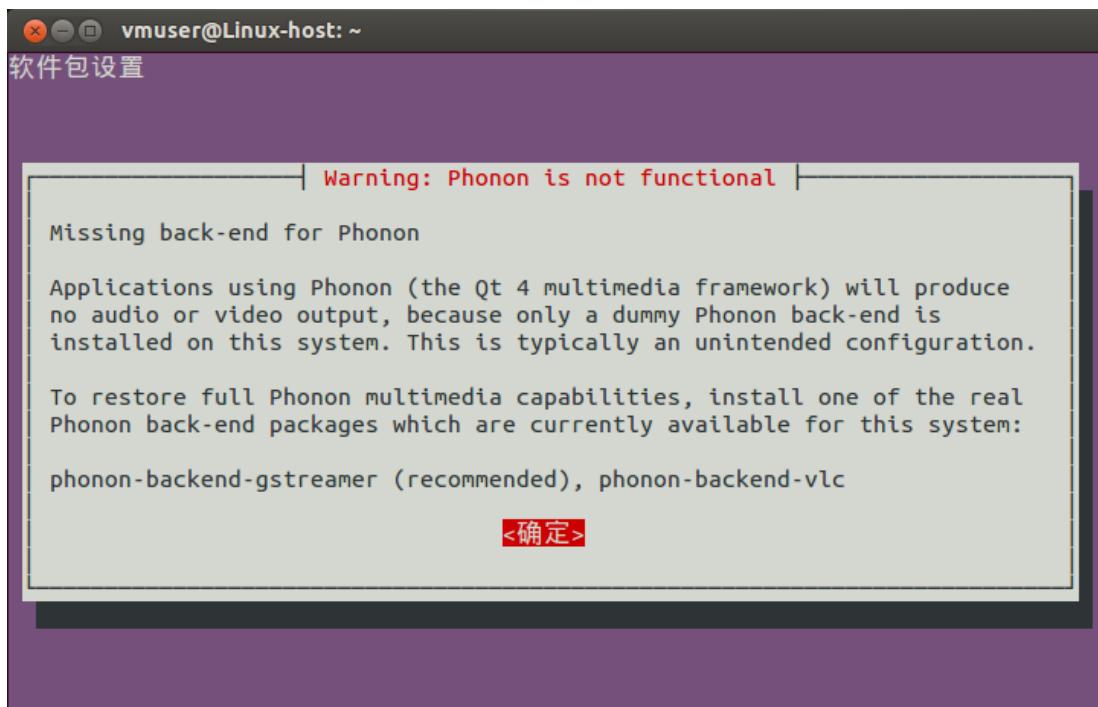


图 14.10 QT SDK 安装过程中可能出现的警告窗口

当 QT SDK 安装完成后，终端显示如图 14.11 所示。

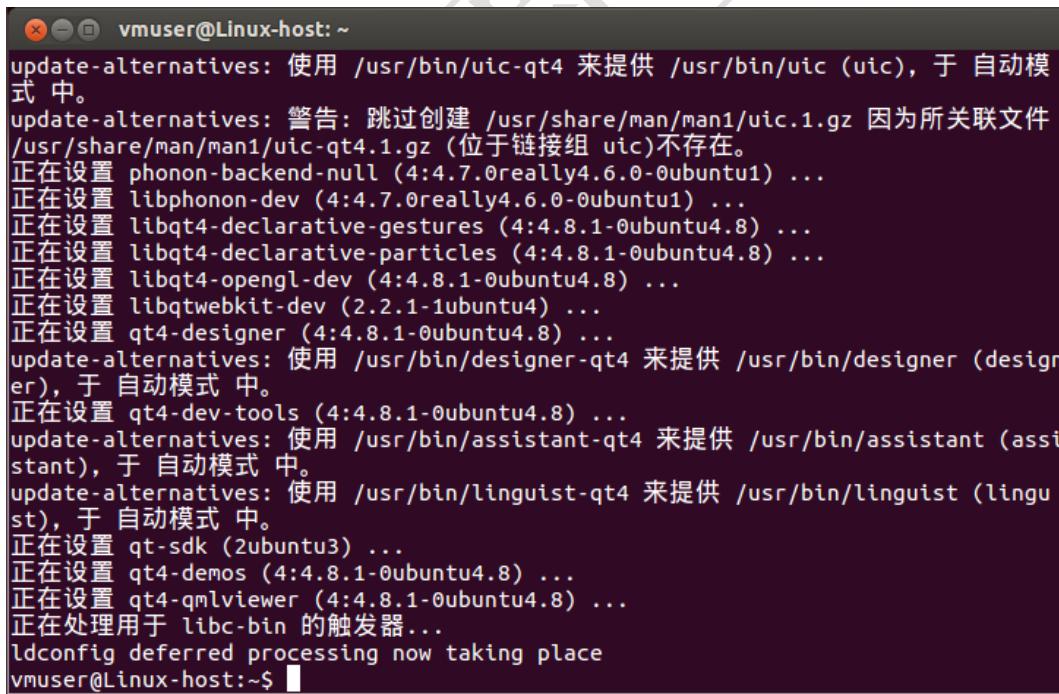


图 14.11 完成 QT SDK 的安装

14.7.3 Qt Creator 配置

QT SDK 安装完成后，通过如下命令可以启动 Qt Creator：

```
vmuser@Linux-host:~$ qtcreator
```

Qt Creator 启动后界面如图 14.8 所示。如已安装交叉编译工具链，则此时系统中将同时



存在两个 Qt 版本，一个用于桌面环境，一个用于嵌入式环境。因此首先需要设置 Qt Creator 所调用的 Qt 版本。

1. 设置 Qt 版本

把鼠标光标移动到屏幕顶端，以显示隐藏的菜单栏，如图 14.12 所示。

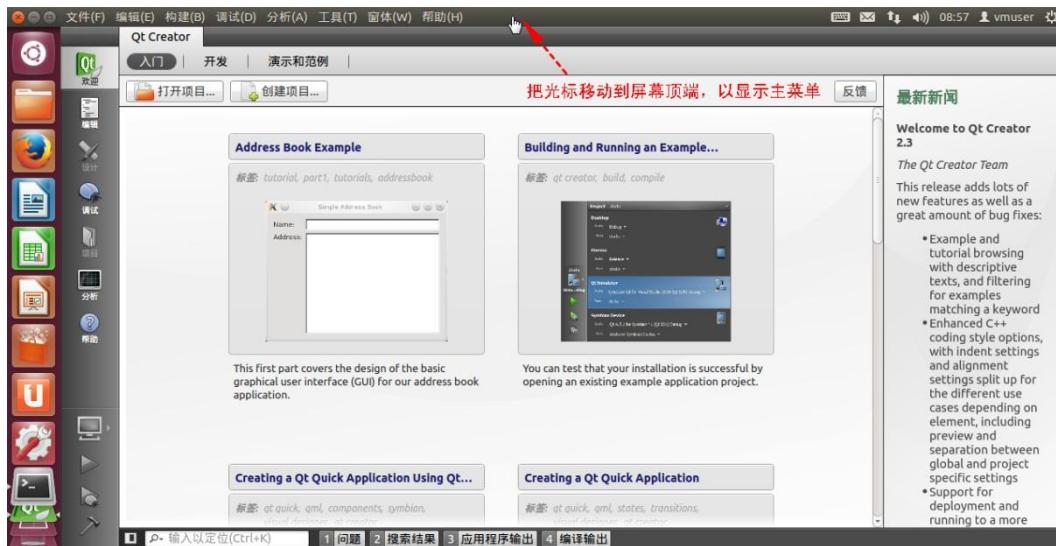


图 14.12 显示 qtcreate 的主菜单

菜单栏显示出来后，点击“工具”->“选项”菜单，打开 QT 选项配置窗口，如图 14.13 所示。

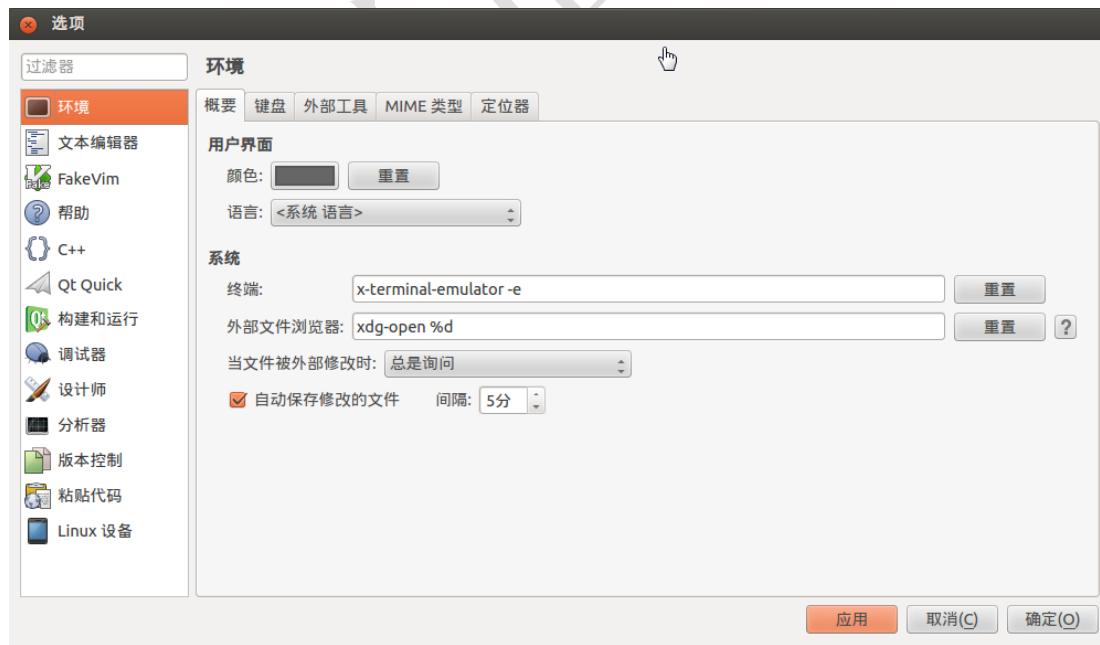


图 14.13 选项界面

在打开的“选项”窗口上，点击左边的“构建和运行”选项，然后切换至“Qt 版本”标签。在这里 Qt Creator 会根据 PATH 环境变量自动探测 Qt 版本路径，若用于桌面环境及嵌入式 Linux 开发的 Qt 版本未被自动检测到，则需要手动进行添加。



点击右侧“添加”按钮，选择“usr/bin/qmake”路径下的 qmake-qt4（若该路径下没有 qmake-qt4 文件，则选择 qmake 文件）文件以及“/opt/qt4.7.3/bin”路径下的 qmake 文件，如图 14.14 及图 14.15 所示。

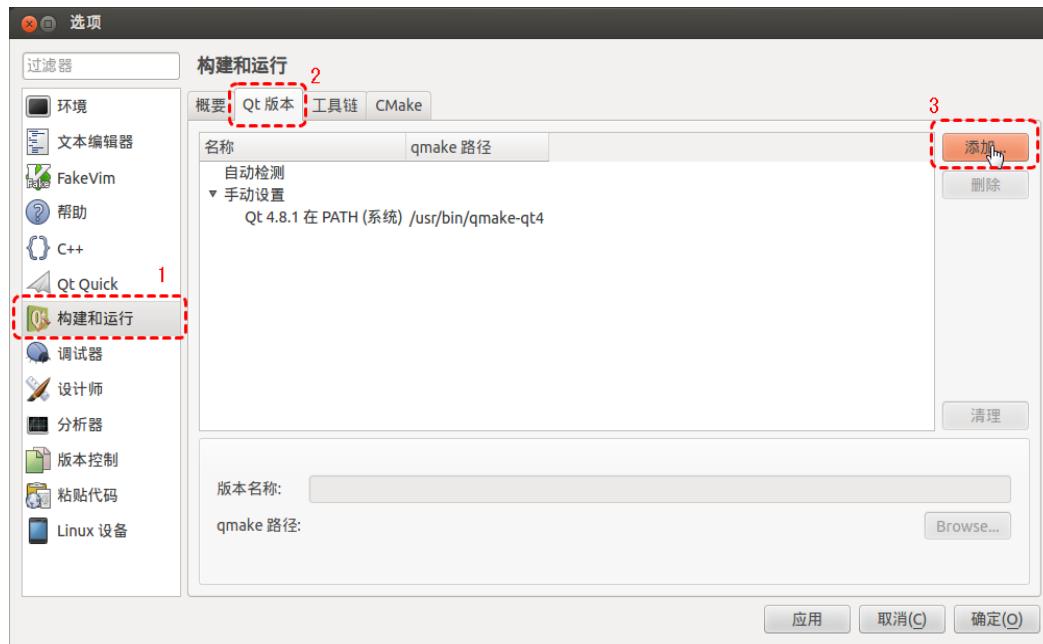


图 14.14 添加 Qt 版本

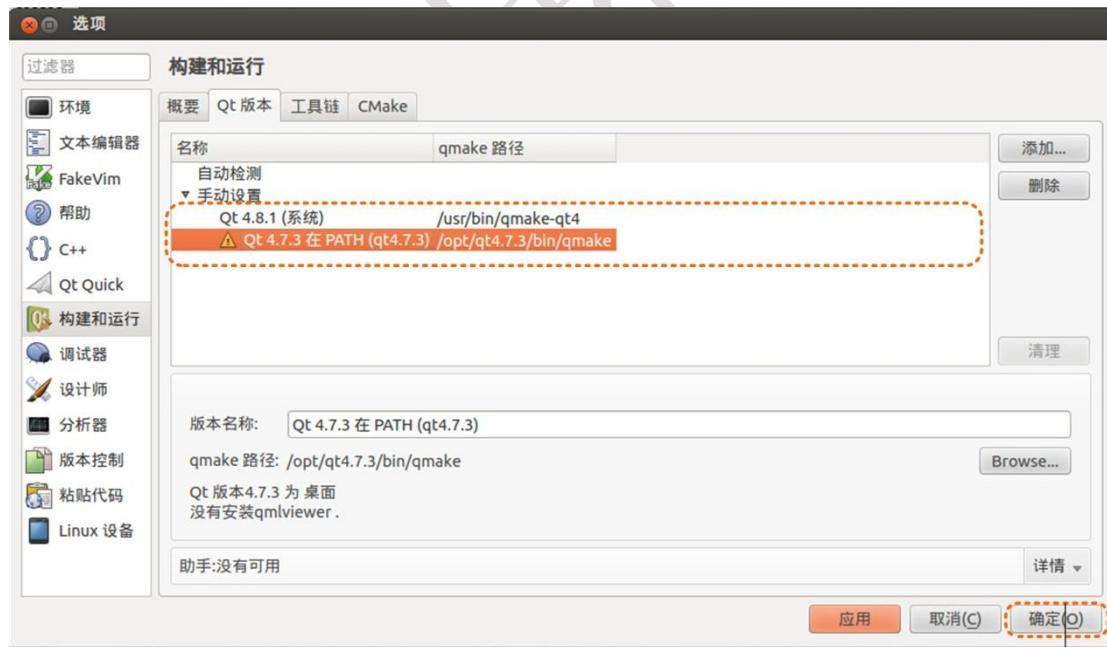


图 14.15 添加完 Qt 版本

2. 添加工具链

点击“工具链”标签，进入工具链设置。在这里已经自动检测出本地 gcc 的编译器，还需要将 EasyARM-iMX283 的交叉编译器也添加上去，点击右边的“添加”按钮，选择“GCC”菜单，如图 14.16 所示。



图 14.16 添加一个 GCC 项

再点击选中手动设置栏中的“GCC”选项，然后点击下边的“浏览”按钮，如图 14.17 所示。

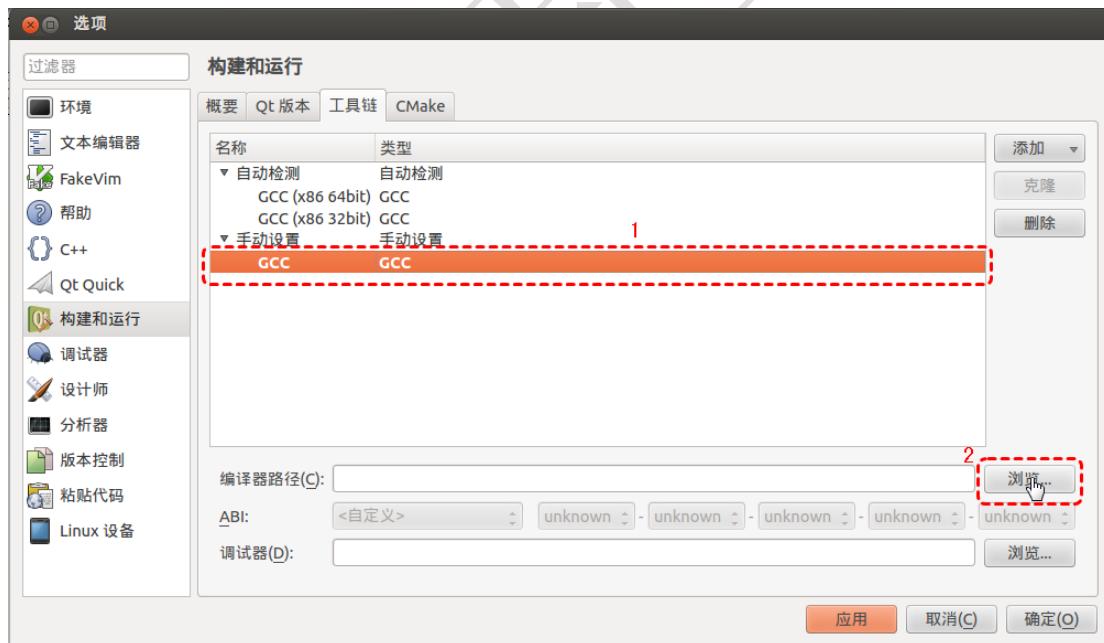


图 14.17 添加新的 GCC 项

在这里需要把“/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/”路径下的“arm-fsl-linux-gnueabi-g++”文件添加到“编译器路径”的输入框中，如图 14.18 所示。

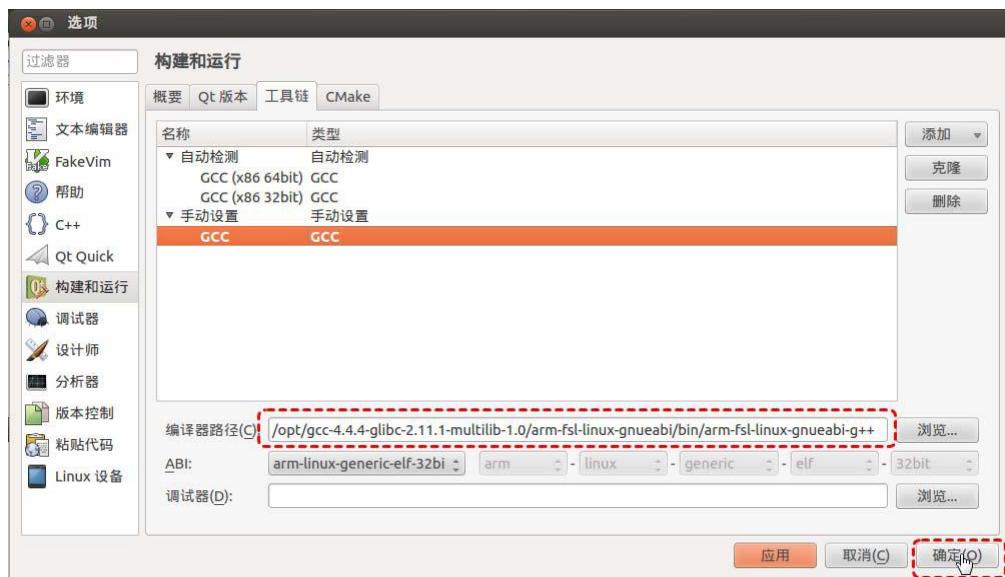


图 14.18 添加交叉编译器路径

然后点击对话框中的“确定”按钮，关闭对话框。至此，Qt Creator 开发环境已配置完成。

14.7.4 Qt Creator 使用例程

接下来将以“Hello World”项目为例，介绍如何使用 Qt Creator 开发图形界面。

1. 创建 Hello World 项目

启动 Qt Creator，单击界面中“创建项目”按钮，在弹出的“新项目”窗口上先点击左侧的“Qt 控件项目”，然后再选中右侧的“Qt Gui 应用”模板，最后点击“选择”按钮，如图 14.19 所示。

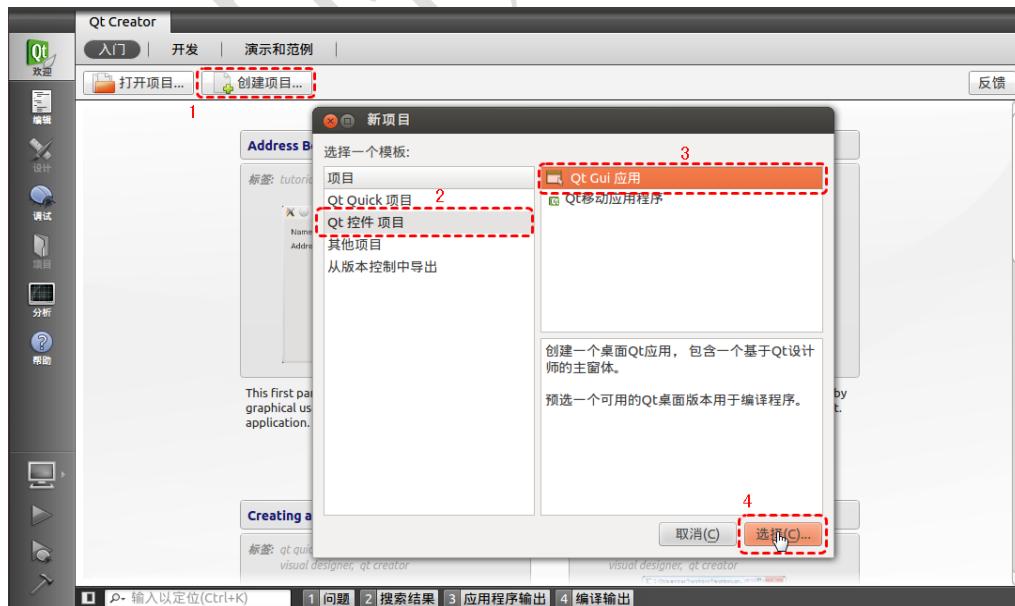


图 14.19 新建 Gui 应用项目

点击“选择”按钮后，选择项目工程存储位置及设置工程名，参考图 14.20 所示设置即可（注意：创建路径将是项目工作目录，该路径下将会自动生成一个与工程名相同的文件夹



以及编译生成的其他文件夹), 然后点击“下一步”按钮。



图 14.20 设置项目名称及创建路径

在本示例中, 创建路径为 “/home/vmuser/qt_test/” 目录。点击“下一步”按钮, 进入“目标设置”界面。这里, 选择按如图 14.21 所示的默认设置即可。



图 14.21 目标设置界面

设置完成后, 点“下一步”按钮进入“类信息”界面, 如图 14.22 所示。

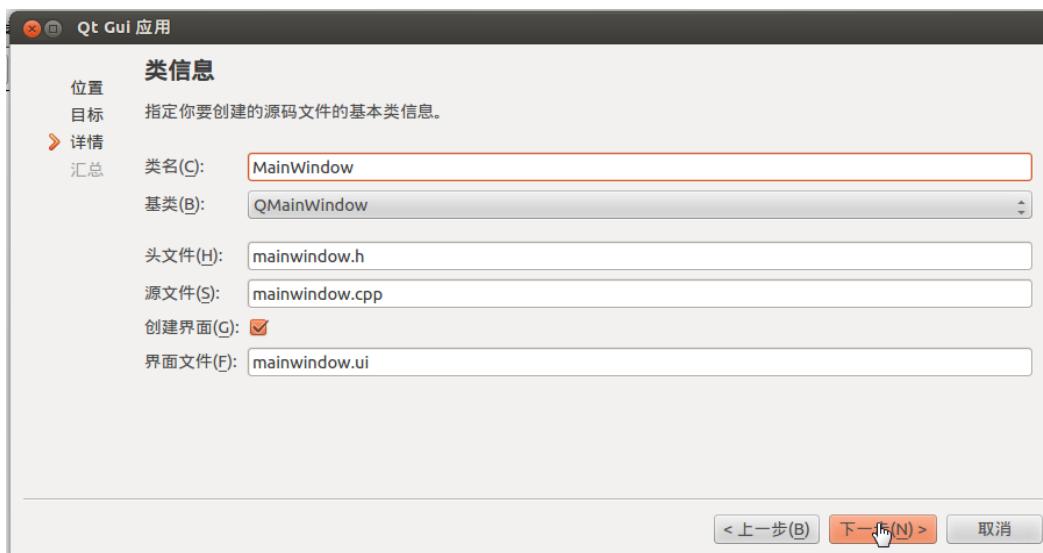


图 14.22 类信息界面

在这里按默认设置即可，直接点击“下一步”按钮，进入“项目管理”界面，如图 14.23 所示。



图 14.23 项目管理界面

单击“完成”后，可以看到创建好的 qt_test 项目，在该项目中 Qt Creator 已经自动生成 Qt 程序所需的基本代码，并已自动打开了 mainwindow.cpp 文件，如图 14.24 所示。

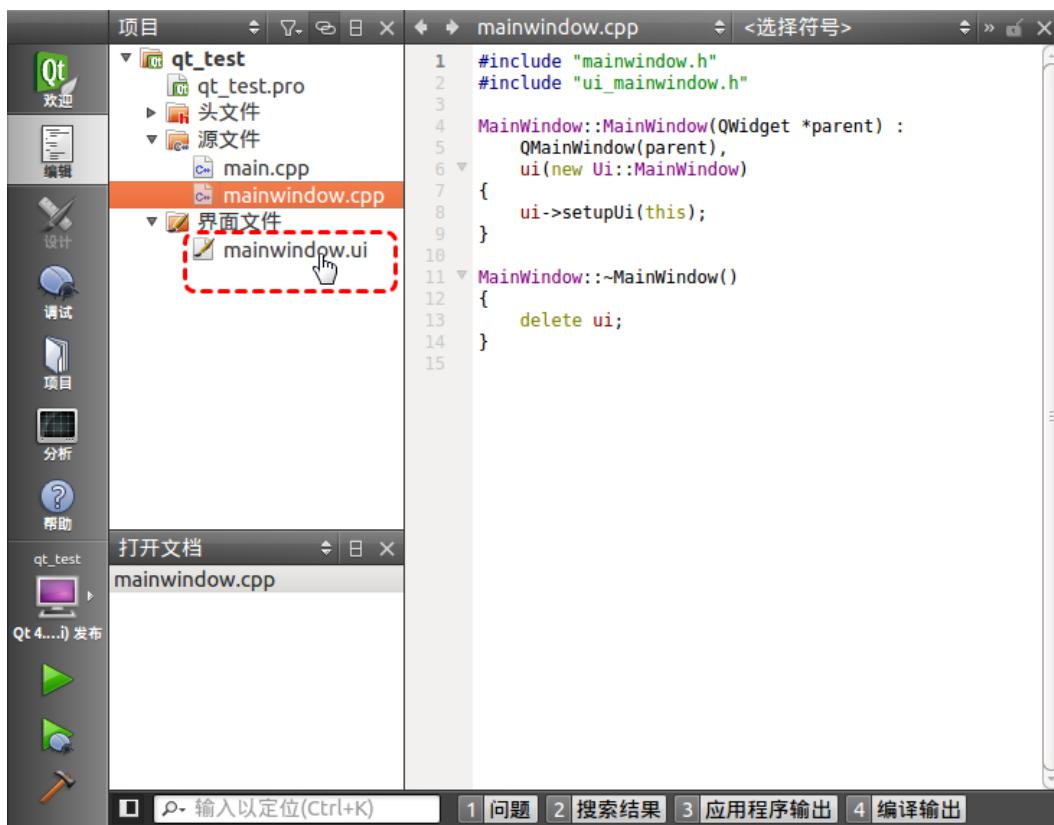


图 14.24 mainwindow.cpp 界面

双击左侧项目栏中的 mainwindow.ui 文件则可以启动可视化编辑器，如图 14.25 所示。

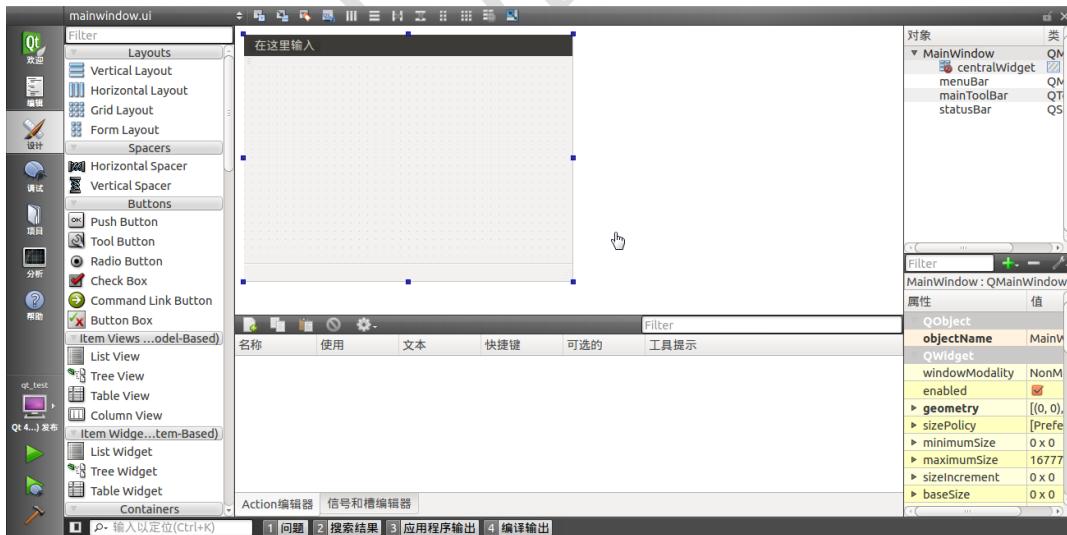


图 14.25 可视化界面编辑器

在可视化编辑器中，通过拖动左侧控件栏中控件到窗体页面中，以所见即所得的方式设计应用界面。

拖动一个 Label 控件到窗体页面，然后双击该控件，设置 Label 上文字为“Hello World!”，，然后再用鼠标调整控件大小，使文字整体可见，如图 14.26 所示。

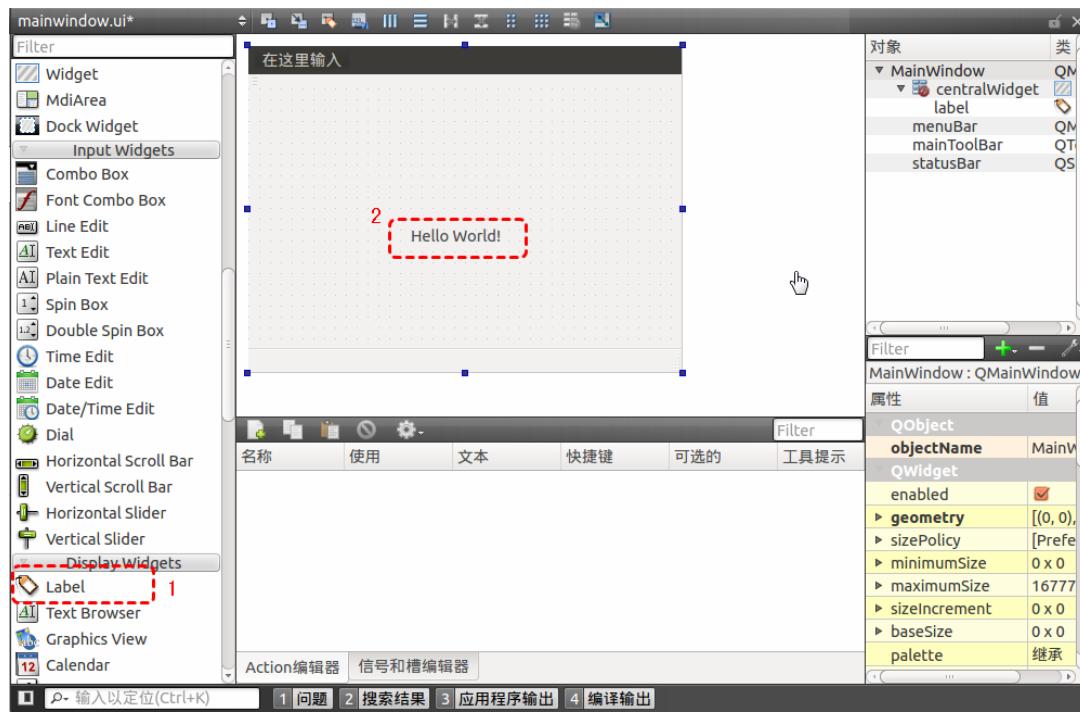


图 14.26 hello world 程序界面

至此，一个简单的 Hello World 应用界面就设计完成了，下面将介绍如何编译该 Qt 项目。

2. 本地编译 Qt 应用

点击左侧栏上的发布按钮“”，然后在“构建”一栏选择前面所设置的桌面版本的 Qt（如本例中的“Qt 4.8.1 在 PATH (系统) 发布”），如图 14.27 所示。

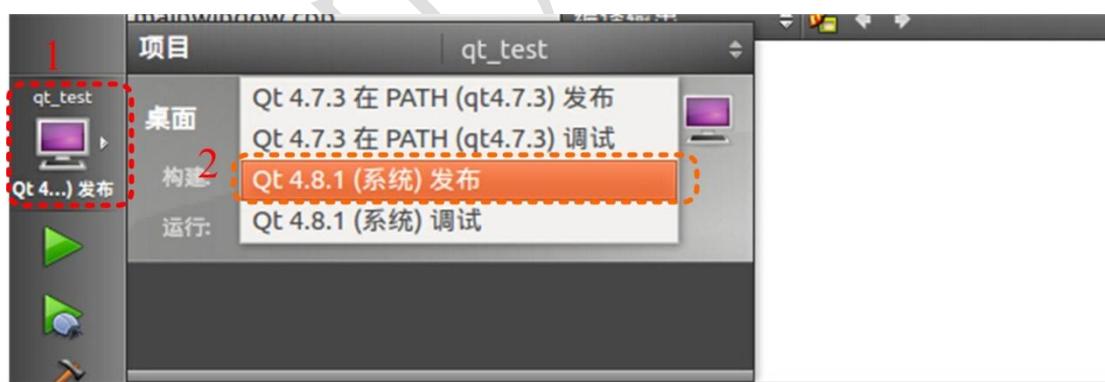


图 14.27 选择 Qt 版本

选择了“Qt 4.8.1 (系统) 发布”的本地编译器后，点击运行按钮“产品用户手册

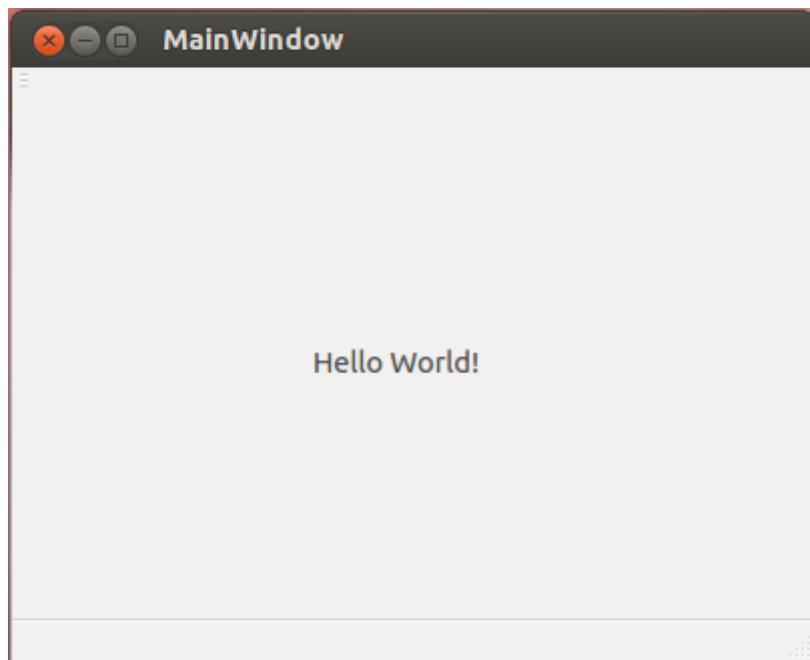


图 14.28 hello world 界面

3. 交叉编译 Qt 应用

点击左侧栏上的发布按钮“”，然后在“构建”一栏选择前面所设置的交叉编译器版本的 Qt（如本例中的“Qt 4.7.3 (arm-fsl-linux-gnueabi) 发布”），然后点击构建按钮“”进行编译，如图 14.29 的所示。

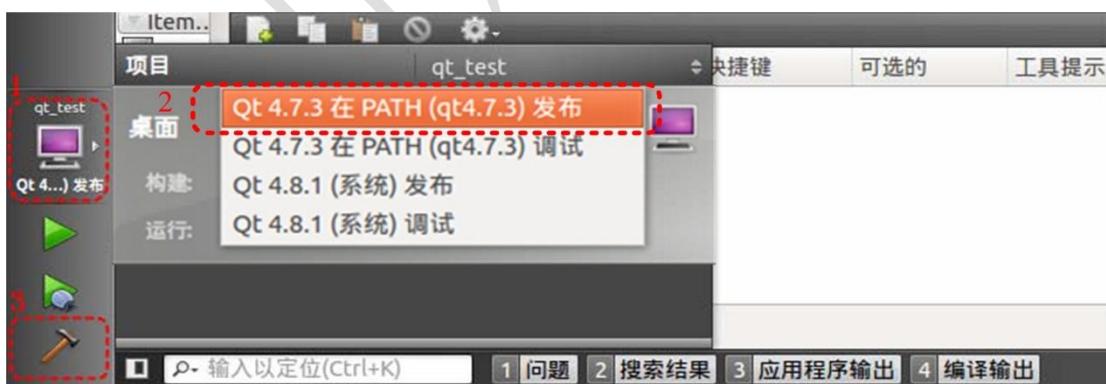


图 14.29 选择交叉编进行构建译器

注意，由于这是交叉编译，所以编译出来的程序不能在本地 PC 机上运行或调试。因此不能点击 按钮运行程序，也不能点击 按钮调试程序。

编译完成后，在项目创建目录下自动生成“`qt_test-build-desktop-Qt_4_7_3__PATH__qt4_7_3__`”目录。该目录下的内容如图 14.30 所示。



```
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_7_3__PATH__qt4_7_3__$ ls
main.o mainwindow.o Makefile moc_mainwindow.cpp moc_mainwindow.o qt_test ui_mainwindow.h
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_7_3__PATH__qt4_7_3__$ file qt_test
qt_test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.26, n
ot stripped
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_7_3__PATH__qt4_7_3__$
```

图 14.30 交叉编译后的输出内容

在该目录中有一个 `qt_test` 的文件，经文件属性查看可知，这是一个 ARM 指令架构的可执行文件。把该文件通过 nfs 或其它方式下载到 EasyARM-iMX283 的/root/目录下，然后通过串口终端登录开发套件的 Linux 系统，并通过如下指令即可启动该程序 (**J7 (DUART)** 需要通过串口线与电脑相连)。

```
root@EasyARM-iMX28x ~# ./qt_test
```

程序启动后如图 14.31 所示。

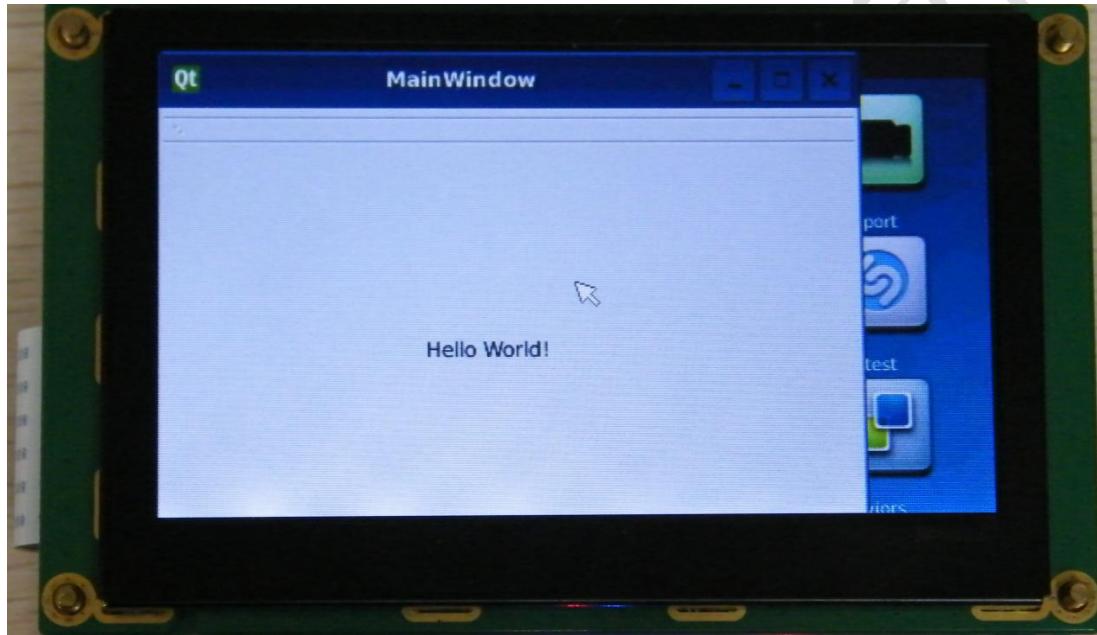


图 14.31 Hello World 程序在开发套件上运行

14.8 zylauncher 图形框架

`zylauncher` 是 Linux 上的一个简单的图形演示框架。整体界面采用当前流行的宫格界面。用户可以将自己的程序添加到演示框架中，直接从 GUI 界面启动程序。界面如图 14.32。在界面上可以放置三种类型的按钮图标。按钮图标类型如下：

- 菜单图标：点击可以切换新的宫格界面（**菜单界面**）；
- 程序图标：点击可以启动演示程序；
- 退出图标：点击将退出整个演示框架。



图 14.32 zylauncher 界面

演示框架主体是采用 qml 语言描述，用户可以通过修改 qml 文件，进行界面的配置。如果按钮图标过多，还可以新建菜单界面，以容纳更多的按钮图标。演示框架目录结构如图 14.33 所示：

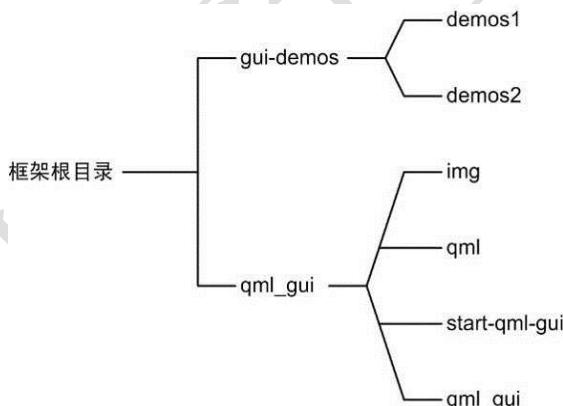


图 14.33 zylauncher 框架结构

根目录下的 `gui-demos` 中放置的是演示程序的可执行文件和相关资源文件，如果用户希望将自己的程序添加到框架中，需要把程序的可执行文件和相关资源文件放置在 `gui-demos` 目录下。

- `qml_gui` 目录下放置的是框架相关文件；
- `img` 下放置演示程序对应的按钮图标文件；
- `start-qml-gui` 与 `qml_gui` 为框架的可执行文件，其中 `start-qml-gui` 为整个框架的启动脚本，在将框架根目录拷贝至目标板文件系统后，可使用 `start-qml-gui` 启动框架；
- `qml` 目录下放置着描述框架界面的 `qml` 文件，用户可以通过修改 `qml` 下文件来对界面进行配置。

下面以 `qml` 目录下的 `MainMenu.qml` 文件来讲解如何通过修改 `qml` 文件来对界面进行配置。`MainMenu.qml` 对应的是框架的主页面，即图 14.32。`MainMenu.qml` 代码如程序清单 14.4 所示。



程序清单 14.4 MainMenu.qml 文件

```
import QtQuick 1.0
import "func.js" as Logic

Rectangle {
    width: rootloader.viewWidth
    height: rootloader.viewHeight
    color: "black"

    // 界面标题
    Text {x:rootloader.titleX; y:0; text:"Main Menu"; font.pointSize:titleFontSize; color:"white"}

    // 第一行图标
    ProButton {
        x:Logic.indexToX(0); y:Logic.indexToY(0);
        imgPath: "../img/switch.png";
        imgText : "fluidlauncher";
        execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
    }

    ProButton {
        x:Logic.indexToX(1); y:Logic.indexToY(0);
        imgPath: "../img/cpp.png";
        imgText : "syntax high";
        execText : "../gui-demos/syntaxhighlighter/start-syntaxhighlighter"
    }

    ProButton {
        x:Logic.indexToX(2); y:Logic.indexToY(0);
        imgPath: "../img/block.png";
        imgText : "move block";
        execText : "../gui-demos/moveblocks/start-moveblocks"
    }

    MenuButton{
        x:Logic.indexToX(3); y:Logic.indexToY(0);
        imgPath: "../img/cookie.png";
        imgText : "small_demos";
        menuName : "./SmallDemos.qml"
    }

    // 第二行图标
    ProButton {
        x:Logic.indexToX(0); y:Logic.indexToY(1);
        imgPath: "../img/clock.png";
        imgText : "clocks";
        execText : "../gui-demos/clocks/start-clocks"
    }

    ProButton {
        x:Logic.indexToX(1); y:Logic.indexToY(1);
        imgPath: "../img/qt.png";
    }
}
```



```
    imgText : "pixelator";
    execText : "../gui-demos/pixelator/pixelator"
}

ProButton {
    x:Logic.indexToX(2); y:Logic.indexToY(1);
    imgPath: "../img/behavior.png";
    imgText : "behaviors";
    execText : "../gui-demos/behaviors/start-behaviors"
}

MenuButton{
    x:Logic.indexToX(3); y:Logic.indexToY(1);
    imgPath: "../img/button1.png";
    imgText : "qml_demos";
    menuName : "./QmlDemos.qml"
}

// 第三行图标

ProButton {
    x:Logic.indexToX(0); y:Logic.indexToY(2);
    imgPath: "../img/stylesheet.png";
    imgText : "style sheet";
    execText : "../gui-demos/stylesheet/stylesheet"
}

ProButton {
    x:Logic.indexToX(1); y:Logic.indexToY(2);
    imgPath: "../img/sysMonitor.png";
    imgText : "sysMonitor";
    execText : "../gui-demos/sysMonitor/start-sysmonitor"
}

ProButton {
    x:Logic.indexToX(2); y:Logic.indexToY(2);
    imgPath: "../img/note.png";
    imgText : "dockWidgets";
    execText : "../gui-demos/dockwidgets/dockwidgets"
}

ExitButton{
    x:Logic.indexToX(3); y:Logic.indexToY(2);
    imgPath: "../img/exit.png";
    imgText : "exit"
}

}
```

接下来关注代码：

```
ProButton {
    x:Logic.indexToX(0);  y:Logic.indexToY(0);
    imgPath: "../img/switch.png";
```



```
    imgText : "fluidlauncher";
    execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
}
```

ProButton 指示这是一个程序按钮图标，点击此图标可以启动一个演示程序，在大括号中的是有关于此按钮的属性信息：

x,y: 指定图标在屏上的横纵坐标。zylanucher 采用的是 3 行 4 列宫格界面，可通过 Logic.indexToX(0) 与 Logic.indexToY(0) 获得 0 行 0 列宫格格子的 x, y 坐标

imgPath: 指定按钮的图标文件

imgText: 指定按钮下方的描述文字

execText: 指定按钮对应的演示程序可执行文件

其中需要注意的一点，imgPath 与 execText 都是通过相对路径指定对应的文件，但它们的相对路径的基准是不同的。

设置 imgPath 时，“相对路径”相对的是“qml 文件所在路径”(**MenuButton 中的 menuName 亦如此**)。

设置 execText 时，“相对路径”相对的是“当前工作路径”(**当使用 start-qml-gui 启动程序时，其“当前工作路径”为 start-qml-gui 文件所在目录**)。

接着关注代码：

```
MenuButton{
    x:Logic.indexToX(3); y:Logic.indexToY(1);
    imgPath: "../img/button1.png";
    imgText : "qml_demos";
    menuName : "./QmlDemos.qml"
}
```

MenuButton 是一个菜单按钮，点击此按钮，将切换到另一个菜单界面（**宫格界面**）。在大括号中的是有关于此按钮的属性信息。其属性与 ProButton 基本一致。唯一不同的是 MenuButton 没有 ProButton 中的 execText 属性，取而代之的是 menuName 属性。

menuName: 指定新菜单界面对应的 qml 文件。如上代码，指定 QmlDemos.qml 为新的菜单界面（**可以在同级目录下找到 QmlDemos.qml 文件**）。

最后关注的代码：

```
ExitButton{
    x:Logic.indexToX(3); y:Logic.indexToY(2);
    imgPath: "../img/exit.png";
    imgText : "    exit"
}
```

ExitButton 是一个退出按钮，点击此按钮，将退出演示框架。其属性参数较少，参数信息可参考 ProButton。



表格索引

- 表 2.1 快捷方式说明 42
表 3.1 常见 Linux Shell 46
表 3.2 ls 命令常用选项 47
表 3.3 Linux 下路径的表示方法 48
表 3.4 mkdir 命令支持的选项 50
表 3.5 rm 命令选项 52
表 3.6 head 和 tail 支持的选项 55
表 3.7 cat 命令常用选项 56
表 3.8 tar 常用选项 57
表 3.9 cp 命令常用选项 58
表 3.10 文件权限说明 60
表 3.11 ifconfig 命令各选项参数 61
表 3.12 mount 常用参数 62
表 3.13 insmod 命令常用选项 64
表 3.14 rmmod 常用可选项 65
表 3.15 modprobe 常用选项 66
表 3.16 Linux 常见环境变量 69
表 4.1 FHS 顶层和/usr 目录 72
表 4.2 examples.desktop 文件详细信息说明 73
表 4.3 /proc 子目录内容说明 78
表 4.4 sysfs 内部结构与外部表现 78
表 4.5 sysfs 目录结构 错误!未定义书签。
表 5.1 退出 Vi 的命令 81
表 5.2 光标快速定位 82
表 5.3 vi 的编辑命令 82
表 5.4 剪切和删除命令 83
表 5.5 配置 vi 命令 85
表 7.1 开发套件硬件资源 错误!未定义书签。
表 7.2 开发套件提供的 Linux 资源列表 108
表 7.3 开发配件 109
表 8.1 启动方式设置 111
表 9.1 NandFlash 分区信息 143
表 10.1 U-Boot 重要目录说明 161
表 12.1 FHS 顶层目录 186
表 12.2 /usr 目录 187



表 13.1 引脚和设备节点的关系	193
表 13.2 ADC 读取命令表	198
表 13.3 硬件接口与设备文件对应关系	203
表 13.4 c_cflag 部分可用选项	205
表 13.5 c_iflag 标志	206
表 13.6 c_oflag 标志	206
表 13.7 c_lflag 标志	207
表 13.8 c_cc 标志	208
表 13.9 SPI_IOC_WR_MODE 命令	221
表 13.10 SPI_IOC_RD_MODE 命令	221
表 13.11 SPI_IOC_WR_BITS_PER_WORD 命令	221
表 13.12 SPI_IOC_WR_MAX_SPEED_HZ	222
表 13.13 SPI_IOC_MESSAGE(1)命令	222
表 14.1 更新目标文件系统 Qt 库所需替换的文件或目录	240



程序清单索引

- 程序清单 5.1 vi/vim 配置文件范例 86
程序清单 6.1 Hello 程序清单 91
程序清单 6.2 /etc(exports 文件内容 101
程序清单 6.3 添加了 NFS 目录 101
程序清单 8.1 start_userapp 文件内容 126
程序清单 11.1 引脚的功能定义 175
程序清单 11.2 释放内核所占 GPIO 资源示例 176
程序清单 11.3 驱动的初始化操作 177
程序清单 11.4 gpio_miscdev 的实现 177
程序清单 11.5 gpio_fops 的实现 177
程序清单 11.6 gpio_exit 函数的实现 178
程序清单 11.7 gpio_open 函数的实现 178
程序清单 11.8 gpio_write 函数的实现 178
程序清单 11.9 gpio_ioctl 的调用 179
程序清单 11.10 gpio_release 函数的实现 179
程序清单 11.11 Makefile 文件的实现 180
程序清单 11.12 LED 测试程序代码 181
程序清单 11.13 中断检测示例代码 182
程序清单 11.14 LCD 时序列表 184
程序清单 11.15 fb_entry 的定义与初始化 184
程序清单 13.1 打开 GPIO 设备文件节点 194
程序清单 13.2 GPIO 操作命令 194
程序清单 13.3 读入电平操作示例 195
程序清单 13.4 GPIO 测试程序 195
程序清单 13.5 ADC 操作示例 198
程序清单 13.6 打开串口设备 204
程序清单 13.7 向串口设备写入数据 204
程序清单 13.8 读入串口数据 204
程序清单 13.9 termios 结构 205
程序清单 13.10 设置和获取 termios 结构属性 208
程序清单 13.11 设置串口输入/输出波特率函数 209
程序清单 13.12 设置波特率示例 209
程序清单 13.13 串口操作示例 210
程序清单 13.14 打开 I²C 设备文件 213
程序清单 13.15 ioctl 命令定义 214



- 程序清单 13.16 DS2460 测试程序 215
程序清单 13.17 打开 SPI 设备文件 220
程序清单 13.18 struct spi_ioc_transfer 结构体的定义 222
程序清单 13.19 SPI 测试代码 223
程序清单 13.20 CAN 测试程序 229
程序清单 13.21 can_frame 的定义 234
程序清单 13.22 struct sockaddr_can 结构体 234
程序清单 13.23 绑定接口 235
程序清单 13.24 接收 CAN 帧 235
程序清单 13.25 获取数据源接口信息 235
程序清单 13.26 指定输出接口的详细信息 236
程序清单 13.27 can_filter 的定义 236
程序清单 13.28 启用过滤器示例代码 236
程序清单 14.1 hellow 程序代码 241
程序清单 14.2 启动脚本代码 243
程序清单 14.3 Qt 例程代码 248
程序清单 14.4 MainMenu.qml 文件 264



参考文献

- [1] Freescale. 《IMX28CEC.pdf》, <http://www.freescale.com/>, Rev.3, 2012
- [2] Freescale. 《IMX28RM.pdf》, <http://www.freescale.com/>, Rev.1, 2010

广州周立功



免责声明

广州周立功单片机科技有限公司所提供的所有服务内容旨在协助客户加速产品的研发进度，在服务过程中所提供的任何程序、文档、测试结果、方案、支持等资料和信息，都仅供参考，客户有权不使用或自行参考修改，本公司不提供任何的完整性、可靠性等保证，若在客户使用过程中因任何原因造成的特别的、偶然的或间接的损失，本公司不承担任何责任。

广州周立功



销售与服务网络

广州周立功单片机科技有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4

邮编：510630

传真：(020)38730925

网址：www.zlgmcu.com

电话：(020)38730916 38730917 38730972 38730976 38730977



广州专卖店

地址：广州市天河区新赛格电子城 203-204 室

电话：(020)87578634 87569917

传真：(020)87578842

南京周立功

地址：南京市珠江路 280 号珠江大厦 1501 室

电话：(025)68123920 68123923 68123901

传真：(025)68123900

北京周立功

地址：北京市海淀区知春路 108 号豪景大厦 A 座 19 层

电话：(010)62536178 62536179 82628073

传真：(010)82614433

重庆周立功

地址：重庆市九龙坡区石桥铺科园一路二号大西洋国际大厦（赛格电子市场）2705 室

电话：(023)68796438 68796439

传真：(023)68796439

杭州周立功

地址：杭州市天目山路 217 号江南电子大厦 502 室

电话：(0571)89719480 89719481 89719482

89719483 89719484 89719485

传真：(0571)89719494

成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室

电话：(028)85439836 85437446

传真：(028)85437896

深圳周立功

地址：深圳市福田区深南中路 2072 号电子大厦 12 楼 1203

电话：(0755)83781788 (5 线) 83782922 83273683

传真：(0755)83793285

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）

电话：(027)87168497 87168297 87168397

传真：(027)87163755

上海周立功

地址：上海市北京东路 668 号科技京城东座 12E 室

电话：(021)53083452 53083453 53083496

传真：(021)53083491

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室

电话：(029)87881296 83063000 87881295

传真：(029)87880865

厦门办事处

E-mail：sales.xiamen@zlgmcu.com

沈阳办事处

E-mail：sales.shenyang@zlgmcu.com